Title
Enabling Efficient Persistent Memory Systems

Author
Ni, Yuanjiang

Publication Date
2022

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA
SANTA CRUZ

ENABLING EFFICIENT PERSISTENT MEMORY SYSTEMS

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

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Yuanjiang Ni

September 2022

The Dissertation of Yuanjiang Ni
is approved:

Professor Ethan L. Miller, Chair

Professor Heiner Litz

Doctor Pankaj Mehra

Peter Biehl
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Enabling Efficient Persistent Memory Systems

by

Yuanjiang Ni

The next-generation data center infrastructure must be equipped with more cost-competitive memory and storage solutions to deal with the rising I/O and memory demand. Emerging fast, byte-addressable, persistent memories (PMEM) are closing the long-standing divide between the memory and the storage, and can serve the role of both fast storage and scalable memory. However, several challenges must be addressed to fully unlock the potential of PMEM in the future infrastructure: i) traditional failure-atomicity mechanisms such as logging and shadow paging impose significant performance overhead and cause additional wear out by writing extra data into PMEM, ii) while appropriate memory selection and allocation is crucial for cloud providers and customers, existing approaches are unable to find cost-optimal configurations despite incurring significant search costs, and iii) prior work has also been limited to performance studies using simulated memories ignoring the intricate details of persistent memory devices.

The main contribution of this dissertation is a set of technologies that address these challenges. First, we address the redundant writes in failure-atomic PMEM with Shadow Sub-Paging (SSP). SSP exploits a novel cache-line-level remapping mechanism to eliminate redundant data copies in PMEM, minimizes the storage overheads using page consolidation, and removes failure-atomicity overheads from the critical path, significantly improving the perfor-

mance of PMEM systems. Our evaluation results demonstrate that SSP reduces overall write traffic by up to 1.8×, and improves transaction throughput by up to 1.6×, compared to a state-of-the-art logging design.

Next we explore methodologies to enable low-overhead configuration selection for tiered-memory systems. Our tiered memory configurator (TMC) recommends cloud configurations according to workload characteristics and resource utilization. Whereas prior work utilized extensive simulation or costly machine learning techniques, TMC profiles applications to reveal internal properties that lead to fast and accurate performance estimation. TMC's novel configuration-selection algorithm incorporates a new heuristic, packing penalty, to ensure that recommended configurations achieve good resource efficiency. We have demonstrated that TMC reduces the search cost by up to 4× over the state-of-the-art while improving resource utilization by up to 17%.

Finally, we present one of the first in-depth performance studies on the interplay of real persistent memory hardware and indexing data structures. We conduct comprehensive evaluations of various index structures leveraging diverse workloads and configurations. We first obtain important findings via a thorough investigation of the experimental results and detailed micro-architectural profiling. We then propose two novel techniques for improving the indexing data structure performance on persistent memories.

To my loving parents

Acknowledgments

I first would like to thank my advisors, Professor Ethan Miller and Professor Heiner Litz, for their guidance and encouragement. I could not have finished my research and thesis without their help. I would like to thank the members of my advancement committee and dissertation committee. Their support contributes tremendously to the completion of this thesis. I think all my colleagues in the Storage Systems Research Center (SSRC) for the valuable feedback and help they provide throughout my graduate career. My work on 3DXP memory was done during my internship at the Alibaba group. Thanks to my colleagues at Alibaba for helping me with the 3DXP work.

This research was supported by the NSF grants IIP-1266400, IIP-1841565, #1829524, #1829525, #1817077, #1823559, the industrial sponsors of the Center for Research in Storage Systems, and SRC/DARPA Center for Research on Intelligent Storage and Processing-in-memory.

Chapter 1

Introduction

Data has emerged as one of the world's most important assets, driving e-commerce platforms, social media, streaming services, and scientific research. A 2018 Forbes article reported that 2.5 exabytes of data were generated daily and that 90 percent of the world's data was generated over a two-year span [14]. The ever-rising data tide calls for a more powerful and cost-effective memory/storage solution in the data center.

On the storage side, system architects' desire to get I/O latencies as low as possible is hindered by the significant gap in the memory and storage hierarchy. Although NVMe SSDs offer microsecond scale access latencies, they are still considerably slower than DRAM. As a result, the IT industry has been long waiting for new technology to fulfill the market demands for high-performance data storage.

On the memory side, continuing growth in data center applications' main memory requirements [11, 78, 90, 118], along with the slowdown of DRAM scaling [70], renders main memory the costliest component of data center infrastructure. Meta [118] estimates the cost of

DRAM will grow and reach 33% as a fraction of server cost of ownership (TCO). Moreover, due to the strict service level objectives (SLO) imposed by data center applications, memory is often over-provisioned. Microsoft found [73] that 50% of the provisioned main memory capacity remains untouched by virtual machines (VM).

New persistent memory (PMEM) technologies are being developed to address these challenges. PMEM [80, 119] can be *any* apparatus that enables the stored data structures to be accessed with low latency at byte-granularity across the life span of the process that created or modified them. Persistent memories fit between DRAM and NAND flash from both a performance and a price point perspective, bridging the gap between the memory and the storage. Particularly, PMEM can act as either high-performance storage or as high capacity and low-cost main memory.

Enabled by the recent trends in memory technologies, fast and byte-addressable persistent memory systems are becoming a reality. Emerging byte-addressable, non-volatile memory (BNVM) technologies promise higher density and lower energy cost while only being moderately slower than DRAM [5,23,36,59,61,63,74,121,124]. On the other hand, many candidate BNVM technologies share limitations such as limited endurance and higher write latency. In addition, memory disaggregation [8, 42, 75, 118] and Compute Express Link (CXL) attached memory [4,73,78,99] enable pooled deployment of recycled, slower, previous-generation memory across a fabric. Remote memories are slower but provide significant cost reductions, as shown in prior work [8, 73]. Disaggregated, CXL-attached, remote, and persistent memories are practical, as they can be mapped into application virtual address space and accessed using conventional load-store instructions.

One class of workloads that PMEM stands to revolutionize is data-intensive workloads such as online transaction processing. Persistent memories introduce a new storage interface: applications use CPU load/store instructions to directly operate on the storage medium. This model removes the need to maintain separate data formats for memory and storage, avoiding serialization and deserialization of the in-memory data structures. PMEM's adoption into infrastructures will enable a switch to more efficient memory-style data access, resulting in substantial performance improvements [27, 52, 114].

Memory-intensive applications such as Artificial Intelligence (AI), in-memory databases [108], and data analytics [40] may experience severe performance degradation due to the spilling of the workloads to the disk [59]. The memory capacity of an application can be expanded cost-effectively by backing a part of its address space with low-cost (but slower) memory. It is possible for slower memory to result in a net cost win if the cost savings of replacing DRAM outweigh the cost penalty of reduced program performance.

The ecosystem of persistent memory systems is gradually falling into place: Intel's 3DXP memory [49] was launched in 2019, researchers and companies have built PMEM optimized file systems [29, 35, 122, 123] as well as PMEM supporting libraries [27, 52, 114], major chip manufacturers are developing and incorporating CXL solutions [9, 71, 81, 99], and tiered-memory systems are now increasingly supported by operating systems such as Linux [61, 78, 118, 124]. However, despite all the important progress that has been made, **several obstacles must be overcome to fully reap the potential benefits of the PMEM:**

- *Inefficient failure-atomic PMEM.* Since data structures in persistent memory survive a power cycle, the PMEM programming model requires mechanisms to ensure that per-

sisted data is reusable by preserving application consistency in the presence of failures. The updates within a failure-atomic section are guaranteed to be executed indivisibly (all or nothing). The following challenges must be addressed to implement efficient failure-atomic transactions in the PMEM: First, the limited write endurance [67] of PMEM compared to DRAM, second, the latency overheads introduced for guaranteeing failure-atomicity, by extra memory fences, flushes and logging and third, the inability to amortize overheads when operating on fast PMEM.

- *Resource allocation for tiered memory.* Cloud providers such as Microsoft Azure [3], Amazon AWS [1], and Google [2] offer Infrastructure as a Service (IaaS) to satisfy the computing needs of their clients. In such platforms, optimizing tiered memory systems becomes even more challenging as the optimal memory allocation now depends on multiple applications and their requirements. Selecting the right resources is beneficial for the cloud provider and client. For instance, we show that inefficient configurations increase the cost by up to 2.6× for clients, whereas resource stranding can increase the cost by 2.2× for cloud operators. However, optimizing the cloud resource configurations, such as the fast to slow memory ratio in tiered memory systems, and predicting its performance implications, is challenging due to the large search space. Utilizing a brute-force approach, an infeasible number of configurations covering all fast to slow memory ratios and hardware resources must be evaluated to devise accurate performance and TCO models.

- *Platform-specific optimizations.* Although 3DXP, one of the first high-capacity PMEM,

has been commercially available only recently, the research on PMEM based index structures started long before the arrival of real persistent memory. Due to the limited availability of PMEM, prior work resorted to DRAM-based emulation [126] or hardware simulation [92]. However, in recent empirical studies [53,125], PMEM behavior has been shown to be more nuanced than previously thought. As a result, it remains to be seen whether the proposed PMEM-based indexing approaches will actually work on real PMEM hardware. In addition, none of the prior work focuses on indexing structure from a practical perspective and explores platform-specific techniques that efficiently optimize indexes in 3DXP memory.

1.1 Contributions

We have developed a set of technologies that address the abovementioned challenges [1]. In particular, this dissertation makes the following contributions:

- *Shadow sub-paging.* We present Shadow Sub-Paging (SSP), a PMEM optimized shadow paging, to enable low overhead failure-atomic PMEM transactions. SSP leverages the following key observation to provide failure-atomicity: Instead of duplicating writes as in logging-based approaches, SSP only consistently updates a small amount of metadata in PMEM. As the metadata is small compared to the actual data, the redundant write traffic, a major concern in existing PMEM systems, is almost completely avoided. However, SSP raises two new challenges. First, metadata needs to be written in an atomically consistent

[1] We will publish the software and benchmarks used in this dissertation in `https://www.crss.us/sw-pmemhiertools.html`

way, which we address with *lightweight metadata journaling*. And second, SSP introduces memory capacity overheads, which we address via *page consolidation*. We extend the translation lookaside buffer (TLB) hardware to support SSP semantics, minimizing changes to the processor core while avoiding most address remapping overheads.

- *Tiered memory configurator.* To minimize the cost of identifying the right resource allocation in the tiered memory systems, we present the Tiered Memory Configurator (TMC), a system that only requires performance measurements of $N + 1$ hardware configurations where N represents the number of tiers in the memory system. TMC recommends near-optimal tiered-memory configurations according to the application's behavior and the data center's real-time utilization. Instead of utilizing a machine-learning based, black-box approach as in prior works, TMC devises a performance model based on the understanding of hardware performance characteristics. To trade-off execution time performance and cost, our methodology optimizes for a single metric: execution time performance per TCO. In addition, we introduce a new heuristic, packing penalty, which penalizes the configuration that exacerbates the resource stranding issue given the real-time resource utilization at the data center.

- *3DXP optimized index structures.* We conduct an in-depth performance study on the interplay of real PMEM hardware and index structures. We provide new insights on utilizing indexing data structures in Intel's 3DXP memory. We propose two write optimizations for PMEM based indexes: 1) *Group flushing*, a technique that prevents data reordering by flushing modified data in groups. An unmodified B-tree with software-directed

group flushing achieves write throughput comparable to its write-efficient counterpart while having better search and range-query performance; 2) *Persistency Optimized Log-structuring*, which translates random writes to sequential writes at the cost of additional reads, effectively addressing the larger line size deployed by 3DXP (265 Byte vs. 64 Byte).

1.2 Organization

The remainder of this dissertation is organized as follows: Chapter 2 provides background and related work helping motivate the need for our studies, Chapter 3 describes how we eliminate redundant writes in failure-atomic transactions with Shadow Sub-Paging, Chapter 4 discusses how TMC achieves fast and accurate configuration selection for tiered-memory systems, Chapter 5 presents insights and novel techniques to improve the performance of B+-Trees on the real PMEM hardware, Chapter 6 explores future work directions and Chapter 7 conclude.

Chapter 2

Background and Motivation

In this chapter, we begin by examining general characteristics of emerging memory technologies such as byte-addressable non-volatile memory (BNVM) and the prominent use cases of persistent memory. We then describe 3DXP memory, one of the first widely available BNVM, emphasizing the peculiarities of the real hardware. Next, we discuss existing techniques that have been proposed to enforce failure-atomicity in the PMEM enabled storage system. Section 2.4 reviews various key techniques that enable efficient integration of PMEM technology and indexing data structures. Finally, we explore the existing techniques in finding the right resource allocation for tiered-memory systems.

2.1 Emerging Memory Technologies

Byte-addressable, non-volatile memories (BNVM) promise to combine memory and storage characteristics by increasing cost-efficiency over DRAM, providing data persistence and byte addressability at sub-microsecond-scale latencies. Key memory manufacturers such as Mi-

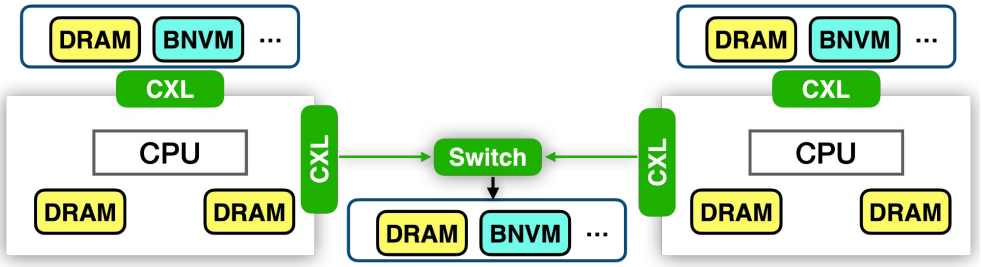


Figure 2.1: The future compute and memory architecture enabled by emerging memory technologies

cron, Intel and SK Hynix have been exploring alternative memory technologies for decades. The candidate technologies that have been developed over the past include PCM [67], ReRAM [102], 3DXP [49] and 3DVXP [107]. Although the details of these technologies vary, many share limitations such as limited write endurance and asymmetric read-write latencies where a write operation may take longer to complete [67].

In today's DIMM-based memory solutions, CPUs' memory technology support completely determines the memory subsystem. Such a rigid, hardware-defined memory subsystem limits mix-and-match of memory technologies with differing cost and performance profiles, and makes it hard to upgrade the memory capacity at a fine granularity. Compute express link (CXL) [9,71,81,99] is about to enable a new compute and memory architecture. With the advent of the new CXL standard, different types of cache-coherent memory devices can be connected to the system memory and accessed via native load/store instructions at high speed in the future servers. CXL enables us to design the memory hierarchy based on the latency, bandwidth, capacity, and persistence needs of the workloads. In this way, compute and memory resources can be scaled independently and system resources can be utilized more effectively. As shown in Figure 2.1, through CXL coherent interconnect, it is possible to expand the memory within a

server as well as pool and share memory between servers.

2.1.1 PMEM as Storage

Byte-addressable, persistent memories will enable the two-tier storage hierarchy to evolve into a single level of large and fast memory. Legacy storage interfaces such as file systems likely will continue to exist into the PMEM era [29, 35, 122, 123]. In the legacy model, system calls such as `read()` or `write()` have been used to operate on data stored in application buffers. Our work focuses on a new PMEM enabled programming model [15, 27, 46, 52, 114] in which applications directly access the storage media from userspace using processor load and store instructions.

A reliable storage system must tolerate unexpected system failures by providing *durability*. As CPU caches typically do not reside within the power-fail protected domain [101], data in the CPU cache is lost during a power cycle. Contemporary CPU provides instructions that explicitly flush the modified data to PMEM from the CPU cache *e.g.* the cache-line writeback `clwb` in x86.

Besides ensuring that modified data reaches the persistence domain, we must also guarantee that the persistent memory state is consistent after an update operation is performed by an application. Contemporary architectures generally only support eight or sixteen-byte atomic stores. As a result, for an update operation that performs a series of PMEM writes or a PMEM write that is larger than 16 bytes, a more sophisticated scheme must be devised to achieve the failure atomicity. To better understand failure consistency, consider a straightforward linked list example. In order to insert a new node into a linked list, we must

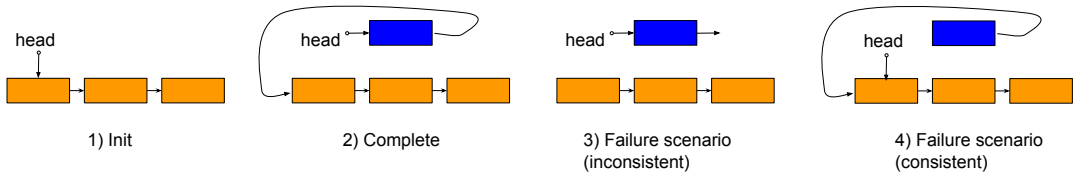


Figure 2.2: Potential states of the linked list after a crash

1. create a new node,

2. have the new node point at the node that is currently at the head,

3. update the head pointer to point at the new node.

Figure 2.2 depicts the potential states of the linked list in the PMEM after a power failure when inserting a new to the linked list: State 1 represents the case where a power failure happen before the insertion starts, state 2 represents the case where a power failure happen after the insertion completes, and state 3 and state 4 represents the potential failure scenarios where a power failure interrupts the insertion.

It is important to highlight that stores can be reordered in the CPU cache or the memory controller before they are eventually persisted in the PMEM. An inconsistent PMEM state will arise if the writes in the step 2 and step 3 are reordered, and a crash happens right after step 3 is persisted but before step 2 is persisted as shown in the state 3 of Figure 2.2. The inconsistent linked list is no longer usable as the head node may point at a wrong memory location (update in step 2 was not persisted). The failure consistency in the persistent memory can be ensured by either carefully ordering the PMEM stores or using failure-atomic transactions.

We can ensure the data is updated atomically by explicitly ordering the PMEM stores in an update operation. For instance, ensuring that the write in step 2 is persisted first guarantees

the consistency of the linked list under any failure scenarios in the linked list example. If a crash happens after step 2 is persisted but before step 3 is persisted, the persistent linked list remains consistent as if the insertion never happened (State 4 in Figure 2.2); however, the new node must be reclaimed (*e.g.* via garbage collection) to prevent a permanent memory leak. Persist ordering can be enforced with a combination of cache-line flushing and memory fencing: the cache-line flushing forces a write-back of a modified cache line, and the memory fencing ensures the completion of a persist operation.

Achieving consistency with persist memory ordering is error-prone if more complex update operations are involved [32, 47, 68, 69]. To remedy this, existing PMEM support libraries [19,27,114,128] incorporate the abstraction of failure-atomic transactions. In the linked list example, programmers now only need to specify a failure-atomic transaction comprising the two updates; the failure-atomic transactions ensure that either both updates are persisted, or none of them take effect. Section 2.3 presents details of existing failure-atomicity techniques, which can be used to implement durable transactions.

2.1.2 PMEM as Memory

Applying technologies such as persistent memory and CXL inevitably introduces multiple tiers of memory that need to be managed. Tiered memory [5, 23, 36, 59, 61, 63, 74, 121, 124] allocates performance-sensitive data in DRAM and performance-insensitive data in the second tier memory, seeking to maintain performance at a lower cost.

Researchers and companies have explored tiered memory systems: hardware-controlled and software-controlled tiered memory. In the hardware-controlled tiered memory [23, 63], the

DRAM acts as a hardware-manged cache layer on top of the slow memory tier (*e.g.* PMEM). Similar as the CPU cache, the memory controller will transparently identify performance-critical pages and place them on the DRAM cache. One major drawback of this approach is that the DRAM is not visible from the perspective of the OS, limiting the total memory capacity applications can utilize. This dissertation instead focuses on managing the tiered memory in software [5,36,59,61,121,124] where both DRAM and the second tier memory can be exposed as the main memory (volatile) and visible to the applications. The responsibility of effectively managing two or more tiers of memory falls on the OS [5, 61, 124] or applications [36, 59, 121] in software-controlled tiered memory. Although such tiered-memory systems are now increasingly supported by operating systems such as Linux [61, 78, 118, 124], it remains unclear how applications should allocate their data structures between the faster, more expensive and the slower, less-expensive tiers for maximizing performance per total cost of ownership (TCO).

Some data center operators [66, 118] have chosen to implement the slow memory tier using compressed DRAM instead of a new memory technology, such as 3DXP, as we have done in this dissertation. Linux Zswap and related mechanisms [76] compress swapped pages but are not fast enough to be load-store memory; they resemble near-I/O mechanisms such as tmpfs or RAMdisk.

2.2 3DXP Persistent Memory

Intel released 3DXP memory in 2019 as one of the first commercially available PMEM hardware. While in last section we have introduced the architecture and some prominent use

	Bandwidth (GB/s)				Idle Latency (ns)			
	Seq. Read	Rand. Read	Seq. Write	Rand. Write	Seq. Read	Rand. Read	NT Store	`clwb`
DRAM	105.9	70.4	52.3	52	81	101	86	57
3DXP	38.9	10.3	11.5	2.8	169	305	90	62

Table 2.1: The basic performance characteristics of 3DXP. Non-temporal stores (NT) and cache line writebacks (`clwb`) are followed by a memory barrier to ensure that the store reaches the ADR domain.

cases of the PMEM in general, we focus on the design of a real PMEM hardware (3DXP) in this section.

For wear-leveling and bad-block management, 3DXP coordinates access to the underlying 3DXP media via an indirection layer [53]. The Address Indirection Table (AIT) translates system addresses to device-internal addresses. The access granularity of the 3DXP media is 256 bytes [53], requiring the 3DXP controller to translate 64-byte requests from the CPU into larger 256-byte requests. The coarser 256-byte granularity corresponds to the error-correcting code (ECC) block unit of 3DXP. To avoid a $4\times$ write amplification on every write, write combining buffers are employed to aggregate cache-line-sized data into 256-byte chunks. When write combining buffers are exhausted, a hopefully full buffer is selected by the IO scheduling logic and flushed to the 3DXP media. Similarly, 256-byte buffers are allocated for incoming read requests.

The basic performance characteristics exhibited by 3DXP on an Intel Cascade Lake server are summarized in Table 2.1. A more comprehensive study of the performance characteristics of the 3DXP can be found in a prior report [53]. Overall, 3DXP provides inferior performance compared to DRAM. Furthermore, the read throughput of 3DXP is about $4\times$ higher than the write throughput and sequential workloads achieve $4\times$ higher throughput than random

Name	Low Extra Writes	Low Persistence Overhead	Low Instruction Overhead
Software redo/undo logging	✗	✗	✗
ATOM, Proteus	✗	✗	✔
DHTM	✗	✔	✔
LSNVMM	✔	✔	✗
SCSP	✗	✗	✗
SSP	✔	✔	✔

Table 2.2: Summary of existing failure-atomicity mechanisms.

workloads. The read-write asymmetric throughput can be explained by the higher write latency of the 3DXP media [67]. The performance gap between sequential and random workloads can be explained by the 4× IO amplification induced by the translation of 64-byte cache line accesses to 256-byte 3DXP accesses. In addition, the unloaded random read latency of 3DXP is about 3× higher than that of DRAMs and the random read latency is 2× higher than the sequential read latency. Finally, the unloaded latency of a non-temporal store or cache-line write back (`clwb`) is almost identical between 3DXP and DRAM. Note that this is the latency to the controller write buffer and is not sustainable as the memory access load increases.

2.3 Failure-Atomic PMEM

Storage systems traditionally ensure crash consistency by using one of three techniques: write-ahead logging [83], shadow paging [44], or log-structuring [96]. We now examine the use of these techniques in implementing failure-atomic PMEM transactions. Table 2.2 presents a summary of existing PMEM optimized failure-atomicity mechanisms, which we will discuss in more detail in the following:

- *Write-Ahead Logging.* Whenever data is overwritten, either the original data or the new

data must first be written to a logging area in PMEM. Only after persisting the log to PMEM data can be updated in place, guaranteeing that the original data can always be recovered after a failure. Prior approaches [27, 52, 56, 57, 106, 114] differ in terms of the techniques used to minimize the instruction overhead and how they reduce the impact of logging on the application performance. There are two types of logging: redo logging records the new data in the log area, whereas undo logging records the old value. Undo logging has to persist a log record (old data) first before the PM write can be made in place for each PM write, introducing excessive persist ordering and cache flushing overhead. With redo logging, log records that contain the new values can be stashed in the memory and streamed to the PMEM when a transaction completes. However, redo logging requires all memory reads to be intercepted and redirected to the redo log to obtain the latest values. As a result, it imposes significant instruction overhead for read operations. Hardware logging [56, 57, 106], therefore, has been introduced to provide high-performance logging by optimizing persistence overhead and instruction overhead. ATOM [57] and Proteus [106] (hardware undo logging) move the log update out of the critical path of the atomic update by tracking the dependency of the log update and the data updated in hardware. DHTM [56] (hardware redo logging) improves upon previous solutions by decoupling the data update from the transaction commit: the process of writing back the modified cache lines to persistent memory can overlap with the execution of the non-transactional code following the transaction. However, even under DHTM, committing the redundant writes to PMEM remains on the critical path and, as a result, may delay subsequent transactions reducing overall performance.

- *Shadow Paging.* When shadow paging performs a write, it creates a new copy of a memory page, updates the new copy, and then atomically updates the persistent virtual-to-physical address mapping to complete a failure atomic write sequence. BPFS [29] presents a redesign of traditional shadow paging for the PMEM-aware filesystem, called Short-Circuit Shadow Paging (SCSP). SCSP includes two optimizations: i) it only copies the unmodified portion of a page, and ii) it applies 8-byte atomic updates as soon as possible to avoid propagating the copy-on-write (CoW) to the root of tree. Unfortunately, SCSP cannot track the update at sub-page granularity and thus still has to CoW almost the entire page if the PM write is small (*e.g.* 8 byte). While SCSP might be suitable for file system workloads where the file data updates tend to be large, persistent memory accesses are performed on the byte-granularity, rendering the method ineffective for persistent memory systems *e.g.* CoW overhead for small updates.

- *Log-Structuring.* Log-structured stores [96] do not update data in place, but instead append newly written data to the end of a log. A continuously updated mapping table is used to map logical addresses to physical storage. The unique challenge in using log-structuring in persistent memories is that, mappings need to refer to large, fixed-size blocks of data to limit the overhead of the mapping table. Persistent memory systems, however, operate on fine-grained byte-sized granularity, which introduces fragmentation and garbage collection overheads when utilizing large block sizes. Log-Structured Non-volatile Main Memory (LSNVMM) [46] proposes a sophisticated tree-based remapping mechanism that allows the out-of-place update to be performed at varied granularities.

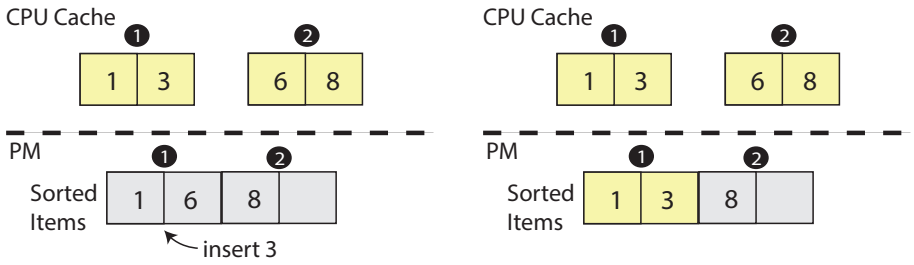
The mapping of LSNVMM is implemented as a partitioned tree (*e.g.* an array of skip lists); a node cache (*e.g.* hashtable) is employed to reduce tree traversal. Despite all its optimizations, the capacity overheads for storing the mapping tables and the instruction overhead are still significant in a system that offers data access at sub-microsecond latencies.

To sum up, all the log-based approaches share a common "write twice" problem, as PMEM writes need to be performed twice: once to the log and once to the actual data. In CoW, modified data is only written once; however, unmodified data has to be copied first, rendering this approach inefficient for small updates. Log-structuring must use a more flexible, and hence complex, mapping scheme [46] to reduce the otherwise prohibitive metadata, imposing significant instruction overhead for the read operations.

2.4 PMEM-based Index Structures

In previous sections, we have examined the characteristics of PMEM and some general issues in implementing persistent data structures in PMEM (failure-atomicity). This section explores specific techniques to enable efficient, PMEM based B+-Tree index structures.

The B-Tree [12] is a self-balancing multi-way tree structure. In a B-tree, each internal (non-leaf) node contains $k-1$ keys as separation values to divide its children into k subtrees, where k must be within the range of $\lceil \frac{b}{2} \rceil$ and b (b represents the capacity of the internal node). When data is inserted or removed from a node, internal nodes may be joined or split to maintain the pre-defined number of children. A B+-tree [28] is a modified B-tree that stores all key-values



(a) Stage 1 (b) Stage 2

Figure 2.3: Challenges in maintaining consistency for B+-Tree

in its leaves and maintains copies of keys in internal nodes. The leaves of a B+-tree are often connected via sibling pointers for fast traversal. While originally proposed to support efficient indexing on block storage, the B+-Tree is also widely used as an efficient in-memory index to alleviate the performance gap between the CPU cache and the main memory, with each node consisting of multiple cache lines [43]. Several studies investigated the design of in-DRAM B+-trees and its variants from different aspects such as cache consciousness [43, 95], scalability [17, 72, 104], the capability of memory latency hiding [20, 55, 93], and the opportunities of architecture-specific optimizations [60, 104]. With the advent of PMEM, while these techniques are still applicable, the durability and consistency requirements, as well as different media characteristics, call for new techniques in designing B+-trees. In this section, we summarize the techniques that have been proposed to adapt the index structures to PMEMs:

- *Optimizing for Writes.* While traditional B+-Tree implementation typically keeps the items in the node sorted, prior work on PMEM-based indexing proposes to leave the items unsorted in the node to deal with the higher write cost of the PMEM [21, 22, 89, 126]. In the unsorted B+-Tree, an extra *validity bitmap* can be added as the node's metadata to

encode the node slots' occupancy. Each insertion consists of 3 steps: i) identifying an empty slot by consulting the *validity bitmap*, ii) writing the item into the empty slot, and iii) updating the corresponding *validity bit*.

- *Consistency.* Ensuring consistency for B+-Tree updates is challenging as an insertion (or deletion) requires touching a large portion of a node (*e.g.* sorted node) that spans multiple cache lines. As shown in a B+-Tree example (Figure 2.3), two cache lines are touched when inserting key 3 into a sorted node. If the system crashes and only cache line ❶ is persisted, the sorted node in the PMEM becomes inconsistent as key 6 in the BTree node is essentially lost (Step 2 in Figure 2.3). The most straightforward solution is to rely on failure-atomic transactions (see Section 2.1.1) to ensure the atomicity of a B+-tree insertion. However, failure-atomic transactions are often undesirable due to the excessive logging overhead. As a result, alternative approaches have been proposed to maintain data consistency for B+-trees in the PMEM. First, unsorted B+-trees built on the *validity bitmap* [22, 89, 126] maintain the atomicity of node updates by preserving the persist order between writes to the new slot and the update to the *validity bitmap*. If the system crashes before the update to the *validity bit* is persisted, the corresponding slot remains invalid, and the insertion will be failed. Second, Hwang et al [47] propose two mechanisms—Failure-Atomic ShifT (FAST) and Failure-Atomic In-place Rebalance (FAIR)—which can be used to provide failure atomicity to the complicated B+-Tree update without resorting to expensive logging. The intuition is that if the modified cache lines can be persisted in a particular order, the inconsistent states in the PMEM caused

by an unclean shutdown can be tolerated. In the previous B+-Tree example, if we ensure that cache line ❷ is persisted before the cache-line ❶, a crash may result in a transient inconsistent state where the sorted node contains a duplicate 6. However, such inconsistency can be tolerated as it can be easily detected (*e.g.* duplicate elements) and fixed by the reader without hurting the correctness of query results.

- *Selective Persistence.* The idea of selective persistence is that the entire index can be reconstructed from a fraction of the tree. Performance can be improved by storing the recoverable part of the index in fast DRAM. Selective persistence can be applied to B+Tree [89, 126] as the leaf nodes of a B+-Tree are already linked in the sorted order, providing all the information to rebuild the internal nodes. The trade-off of this approach is between performance and the failure-recovery time.

2.5 Resource Allocation Techniques

Resource allocation in tiered memory systems has to be performed whenever an application is submitted to the cluster and depends on the current physically available resources. As a result, it is important to minimize the amount of time required to identify the optimal memory configuration of a workload. As the main goal of utilizing tiered memory systems is to reduce TCO, the runtime cost of determining a near-optimal configuration must be considerably less than running the workload itself. Ideally, to minimize the search cost, it is sufficient to execute a few test runs for establishing the performance profile of each application.

- *Machine Learning.* Machine learning techniques [79, 88, 105, 110] such as KNN, Support

Vector Machines (SVM), or regression can be used to predict application performance under a certain machine configuration. Traditional machine learnings are not conscious of the search cost and may require a large number of the samples. Ernest [110] has investigated reducing the amount of training data with experimental experiment design. However, the usage of Ernest is limited as it can only be used to predict the optimal number instances for analytic workloads.

- *Collaborative Filtering.* Collaborative filtering [34, 62] can be used to predict how an unknown application will perform across different configurations, based on sparse performance data collected by profiling training workloads. Although systems such as Selecta [62] require to profile a new, incoming application only on two configurations similar to TMC, Selecta requires extensive training on a large set of workloads and configurations for achieving ideal accuracy. Furthermore, Selecta is a black-box technique that cannot influence memory allocation and data-structure placement within memory tiers. As a result, it needs to rely on underlying OS-level technique to transparently migrate data between the memory tiers.

- *Bayesian Optimization.* Cherrypick [7] proposes to use bayesian optimization (BO) to reduce the search cost for optimizing the objective function: $C(\vec{x}) = P(\vec{x}) \times T(\vec{x})$; $\vec{x}$ represents a VM configuration, $T(\vec{x})$ represents the run time under $\vec{x}$ and $P(\vec{x})$ represents the cost per unit time for $\vec{x}$. BO treats the body of the objective function as a black box and tries to find an optimal solution that minimizes the total cost with as few samples as possible. It relies on an acquisition function to choose the best configuration to run next, and

it stops when the Estimated Improvement (EI) is smaller than a threshold. Although BO significantly reduces the search cost as compared to exhaustive search, it still requires 6 to 12 samples for identifying a cost-optimal configuration. Cherrypick reduces the search overhead by learning a black-box model that predicts cost-efficient cloud configurations based on a limited number of training samples. However, as it lacks an actual performance model, it cannot handle tasks such as finding all configurations whose execution costs is within a certain range or selecting optimal configurations based on real-time resource availability.

2.6 Chapter Summary

This dissertation is motivated by the need for new techniques to facilitate the efficient incorporation of the emerging memory technologies into the system infrastructure. In this chapter, we have provided detailed background and motivation for the dissertation. We have described the nuanced behavior of 3DXP persistent memory and shown that there is lack of platform specific insights and optimizations. We have introduced the existing PMEM optimized failure-atomicity mechanisms and pointed out their inefficiencies. We have also discussed techniques that can be used to achieve near-optimal resource allocation for tiered-memory systems.

Chapter 3

Eliminating redundant writes in failure-atomic PMEM

Computer systems that deploy persistent memories (PMEM) can leverage their non-volatility on the main memory level, however need to take into account that processor caches may remain volatile. As a result, the integrity of persistent data structures after an unclean shut-down remains a major concern. Existing failure-consistency mechanisms such as logging and shadow paging, which are designed for the traditional disk-based I/O model, impose significant performance and energy overheads by writing extra data into the PMEM.

We introduce a hardware-friendly remapping mechanism, based on shadow paging, that can i) reduce the number of extra NVRAM writes over logging, and ii) avoid extra data copying within the critical path. We propose Shadow Sub-Paging (SSP), that supports cache-line-level remapping with low metadata overhead: our approach requires only three bits for each cache line in pages that are being actively updated. The process of atomic updates and transac-

tion commit only involves updating the per-page metadata using simple bitwise operations and no extra data movement is required in the critical path.

In a nutshell, our approach works as follows: For each virtual PMEM memory page in the TLB, SSP maintains two physical page mappings. Persistent writes are applied to the two pages alternatively and SSP switches the page mapping on each failure-atomic transaction. Instead of performing CoW on a per page granularity, SSP maintains additional meta information that tracks state on a cache line basis within each page. Finally, when a page is evicted from the TLB, SSP performs page consolidation to merge the two physical pages into one. Page consolidation is the only point where SSP introduces redundant writes. Our key observation, however, is that the number of transactions is much higher than the number of TLB evictions for most applications. This allows SSP to batch redundant writes to PMEM resulting in a significant decrease of overall writes. As page consolidation can be performed in the background, SSP removes overheads required for failure-atomicity from the performance critical path.

3.1 SSP Design

In this section we introduce the design of SSP. First, we introduce the programming model and then we discuss the basic concept of SSP. We then introduce two key techniques: i) metadata journaling and i) page consolidation. We conclude this section with a discussion.

3.1.1 Programming Model and ISA Extension

We adopt a programming model proposed by Mnemosyne [114], in which programmers use the language construct `atomic{...}` to define a failure atomic section (*e.g.* updates inside are persist in a all or nothing fashion), and use the `persistent` keyword to annotate pointers to persistent data. Furthermore, we extend the ISA with a pair of new instructions—`ATOMIC_BEGIN` and `ATOMIC_END`—to define the begin and the end of a failure-atomic section and a new instruction called `ATOMIC_STORE` to hint a store must be conducted in an atomic fashion. `ATOMIC_BEGIN` and `ATOMIC_END` act as a full memory barriers. The `ATOMIC_STORE` instruction adds the store address to the transaction's write set so that it can be flushed to PMEM during commit. The compiler can be modified to translate these software interfaces (*e.g.* the `atomic` block and the `persistent` pointers) to the ISA instructions.

Note that our interface resembles the interface of Intel TSX [51, 116] where `XBEGIN` and `XEND` are used to indicate the beginning and the end of a transaction. Intel TSX is designed to replace locking as a new way to ensure thread synchronization, and provides no guarantee on durability. In future work, we will investigate integrating SSP with hardware transactional memory through which we can provide an unified interface for supporting ACID transactions. In this work, we assume the isolation of threads are guaranteed using locks: by grabbing locks before the operation is performed, other threads are prevented from observing intermediate states.

3.1.2 Shadow Sub-Paging

Conventional shadow paging suffers from the problem that it operates on full pages which means a cache line write requires CoW of an entire page. To address this challenge, we propose Shadow Sub-Paging, a technique that can track updates on a much finer granularity: cache lines. Shadow Sub-Paging draws inspirations from the PTM-select technique [24], with major extensions to support failure-atomicity for persistent memory.

SSP Abstraction. Shadow Sub-Paging (SSP) is a persistent-memory-optimized version of shadow paging. When SSP is used to perform atomic updates, each active virtual page is associated with a second physical page. A page is active as long as it is in the TLB; for inactive pages the two physical pages are consolidated into one for space efficiency. We refer to the original physical page as "P0" and the extra physical page as "P1". Besides a second physical page, SSP requires three bitmaps for pages that are being actively updated. Specifically, the state of each cache line in the virtual page is represented by a single *current bit*, single *updated bit* and a single *committed bit*; each bit in these bitmaps refers to the cache line of the same offset. As bitmap access is performed on the critical path, they are cached in the TLB as we will explain in the metadata section. The *current bit* defines whether the most recent version of some data referred to by a virtual address is currently mapped to physical page P0 or P1. The *updated bit* is set to one whenever the cache line is written, and is reset as part of the commit process. The *updated bitmap* represents the write set of a transaction. The *Committed bit* defines whether P0 or P1 currently contains the committed (old) version of the cache line.

As a transaction is being processed, reads are directed to the page determined by the

current bit. When the cache line is written to for the first time in a transaction SSP performs three tasks atomically. First, the corresponding *updated bit* is set to track the line as part of the transaction's write set. Second, the write is applied to the cache line that resides on the "other" page, for instance, to P1 if the committed cache line is part contained in P0. Third, the *current bit* is inverted such that the most recent (but still transient) version of the cache line now points to the new page. Note that in SSP it is possible that for a specific virtual page, some cache lines are currently stored on P0 and some on P1. Whenever a write targets a cache line that is already in the write set, it simply updates the current cache line. To commit a failure-atomic transaction, SSP persists all cache lines in the write set by flushing them to PMEM. Note that cache lines might have already been persisted during the transaction in case they were evicted from the cache. This is not a problem in SSP, even in the case of a power failure, as writes never overwrite committed data in place. Furthermore, as part of the commit sequence, the *updated bitmap* is cleared to atomically commit the speculative updates; Lastly, the *committed bitmap* is persisted so that in case of a system failure, the *current bitmaps* can be recovered.

The approach explained above suffers from the following problem. Consider a cache line for a virtual address that is currently mapped to P0. If the cache line is transactionally written, the update needs to be applied to P1, requiring a copy-on-write of P0 into P1 which is costly. The other option would be to cache both the P0 and the P1 cache line. However, this would virtually reduce the cache size by $2\times$. We address this issue by the following technique. Instead of performing CoW, we directly apply the write to the cache line, however, we atomically change the tag so that the line now maps to the "other" page. As the "old" line has already been flushed to PMEM as part of a previous commit, this approach is safe.

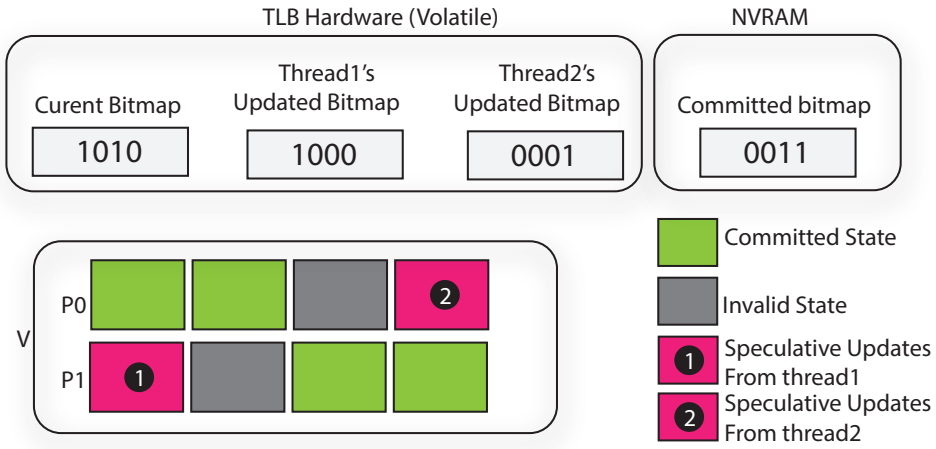


Figure 3.1: The metadata of SSP: each thread has to track their own write set with private *updated bitmaps*; all threads should agree on a *current bitmap* for a virtual page; the per-page *committed bitmap* is used to preserve the consistent state of a page and should be stored durably.

Metadata Storage. The per-page bitmaps need to be checked (and updated) in the critical path. As in prior work [103], we extend the TLB hardware to cache extra metadata required by SSP. Specifically, we store the second physical page number, the *updated bitmap*, and the *current bitmap* in the TLB hardware. As shown in Figure 3.1, threads (cores) are required to use their own set of *updated bitmaps* to track the write-set of the on-going transaction so that they can commit (or abort) their modifications in isolation. To ensure a single view of shared memory, all threads share a *current bitmap* for a given virtual page. We will discuss how to ensure the coherence of *current bitmap* among cores in section 3.2.2. Our system must guarantee that data from previously committed transactions can always be retrieved after a power cycle. The per-page *committed bitmap* is durably stored in the PMEM and is updated as part of the commit process.

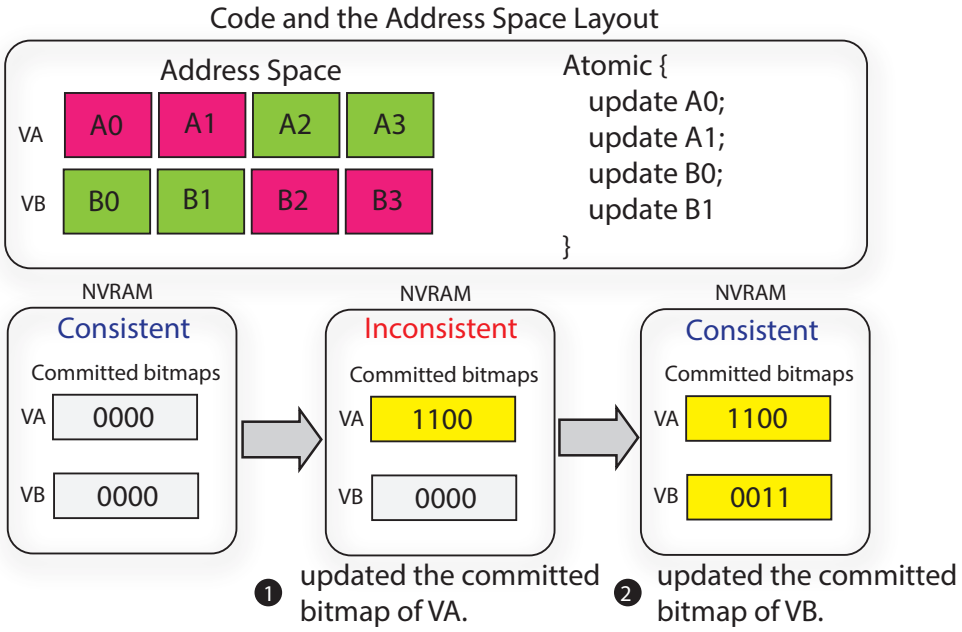


Figure 3.2: The consistency of SSP: the commit process might consist of updating multiple *committed bitmap* in the PMEM; power failure in between will leave the system in an inconsistent state.

3.1.3 Metadata Journaling

To preserve the atomicity of data updates in the case of transactions spanning multiple pages, we must update all *committed bitmaps* **atomically** during transaction commit. An example describing this scenario is shown in Figure 3.2 where a code section specifies four cache lines need to be updated atomically. The commit process involves updating the *committed bitmap* of `VA` (from "0000" to "1100") and that of `VB` (from "0000" to "0011"). However, it might take two separate steps to update these *committed bitmaps* from the perspective of the memory controller. If the system crashes in between, only updates on `VA` (*e.g.* `A0` and `A1`) will be visible after recovery, which violates atomicity. We use metadata journaling to ensure the atomicity of updates on the metadata of SSP. Our metadata journaling approach can be con-

sidered as a redo logging, however, only for the SSP metadata and not for the data itself as in conventional redo logging. It works as follows: every update to the per-page metadata, appends an entry (operation) to the log where an entry contains the page ID and the committed bitmap. Only after persisting the meta log for a transaction to PMEM, SSP updates the per page committed bitmaps in the metadata area. More details on the implementation of metadata journaling are provided in section 3.2.1. As compared to data journaling (*e.g.* redo/undo logging), which requires to log every modified data block (*e.g.* typically 64 Byte), SSP journaling is lightweight as it only needs to record 128 bits of metadata for each modified page.

3.1.4 Page Consolidation

Associating each virtual page in the system with two physical pages represents a 2× capacity overhead. To address this problem, SSP consolidates physical pages into a single page whenever a virtual page is not actively used, and thus not contained in the TLB. As SSP requires pages to be resident in the TLB if they are written as part of a transaction, it is safe to consolidate a page even if some lines are still cached because a line without TLB mapping must either be committed or invalid.

At the time of consolidation, the valid data of a virtual page is likely to be distributed across the two associated physical pages. To minimize the data copying overhead, we identify the physical page (*e.g.* P0 or P1) which contains fewer valid cache lines and copy its valid data to the other physical page (*e.g.* P1 or P0). Note that we can easily compute the number of valid cache lines in P0 or P1 by counting the number of ‘0” or “1” in the corresponding *committed bitmap*. Finally, we update the virtual-to-physical mapping table so that the virtual page refers

to the physical page with all the valid data.

Another issue that needs to be addressed is the accurate identification of inactive virtual pages. It is important, as premature consolidations of pages that are still being actively updated will result in unnecessary data copying overhead. We reuse the TLB hotness tracking: when a virtual page is not referenced by any TLB entry, we consider it as inactive. These inactive pages could be consolidated eagerly (*e.g.* immediately after being detected) or lazily (*e.g.* when the demands on the memory resources are high). Our current implementation consolidate inactive pages eagerly and we plan to investigate lazy consolidation in the future.

3.1.5 Discussion

Virtually-Indexed cache: SSP can work seamlessly with physical, or virtually-indexed physically-tagged caches. To support SSP on a virtually-indexed cache, we extend the tag with one *TX bit* to indicate whether a cache line has been modified by the current transaction. When a modified cache line is written back to memory, the *TX* bit allows Shadow Sub-Paging to distinguish transactional cache lines from regular cache lines. For a transactional cache line, we leverage SSP remapping to prevent overwriting the committed data. A read miss will also require to access the extended TLB. In this case, SSP locates the current mapping (P0 or P1) of a cache line.

Superpages: Superpages [31, 65, 85, 109] are commonly used to increase the coverage of the TLB. Supporting superpages in SSP is challenging due to the large per-page metadata overhead. For instance, a 2 MiB page has 32,768 cache lines and thus, the required bitmap size is 262,144 bytes. It is unpractical to scale TLB entries to support such large bitmaps. SSP currently only

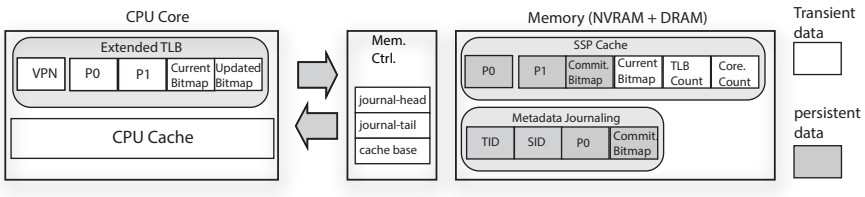


Figure 3.3: The architecture of SSP.

supports 4 KiB base pages. However, techniques such as transparent superpages and page clustering as supported by Linux can be extended to support SSP. In particular, coexistence of small and superpages is possible by automatically demoting superpages when they are updated and promoting pages to superpages when they become "inactive". With this approach, superpages can be used for read-only data. As for the design of TLB hardware, support for superpages and that for SSP can coexist because most processor vendors use split TLBs [31]. SSP only requires TLB extensions for the 4 KiB base pages.

Limitation and Fall-back path. The SSP design has limitations in terms of the size of a transaction it can support. If a transaction updates more pages than the TLB can hold, SSP needs to abort the transaction and revert to a fall-back path. The fall-back path transfers control to a programmer-defined handler which can implement any kind of unbounded software redo or undo logging to ensure atomicity. SSP is designed to handle small and medium sized transactions efficiently, in line with to existing commercial HTMs such Intel's RTM [51, 116].

3.2 SSP Architecture

Figure 3.3 depicts the architectural details of SSP. The per-page metadata of SSP, which contains persistent fields such as the *committed bitmap* and volatile fields such as the

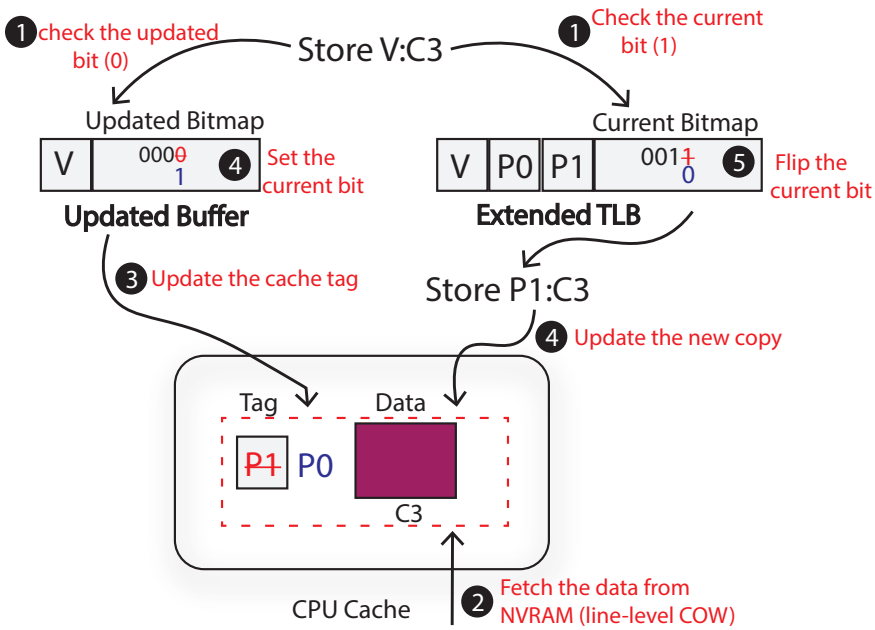


Figure 3.4: The atomic update process of SSP

current bitmap, is managed by the memory controller in the form of the SSP cache. We extend the TLB hardware to cache the *current bitmap*, the *updated bitmap* and the second physical page number. The core is extended to handle atomic updates of cache lines so that on each write the cache tag, the *updated bit* and the *current bit* is updated atomically. To support transaction commit, the core needs to i) write back the cache lines that have been modified and ii) issue a metadata update instruction for each modified page to the memory controller. The memory controller performs metadata journaling to ensure the atomicity of metadata updates. Furthermore, the memory controller tracks the status of each page using the SSP cache and it also conducts page consolidation.

3.2.1 Extensions on CPU hardware

We propose a set of architectural extensions to facilitate SSP. In particular, we adopt a wider TLB entry in which we can cache the second physical page number, the *current bitmap* and the *updated bitmap* of a page that is being modified. In the event of TLB miss, the core will conduct a page table walk as usual to obtain the original physical page number (*e.g.* P0) for the missing page. Afterwards, it interacts with the memory controller to fetch the SSP-specific metadata (using P0 as index), that is the second physical page number (*e.g.* P1) and the *current bitmap*; the updated bitmap is initialized with all zeros.

Memory Read and Write. During address translation, the virtual address is translated into either P0 or P1, depending on the corresponding *current bit* for the accessed cache line. The remainder of the memory access path remains unmodified.

Atomic Update. Figure 3.4 shows how SSP handles atomic updates. We assume a write-back cache with a write allocate policy. An atomic update process is described as follows: i) Shadow Sub-Paging checks the *current bit* to determine which page to write; ii) a copy of the data is loaded into the cache if it is not present; iii) the cache line is remapped by changing the tag, so the "other" cache line can be updated; iv) the new cache is updated with the written data; v) the *current bit* is flipped. Note that iii) and v) are only necessary if the line was written for the first time during the transaction. Since this modified cache line now is associated with another page, it is safe to evict it from the cache anytime without worrying about overwriting the committed state in PMEM.

To conduct the remapping, the *current bitmap* for a page needs to be changed. To keep

the current state of shared pages coherent across cores (and the memory controller), the most straightforward solution is to perform a TLB shootdown. However, TLB shootdowns incur significant overheads [10, 113]. We instead adopt the approach proposed by page-overlays [103] which exploits the cache coherence network to guarantee coherency of the TLB entries, including the *current bitmap*. The cache coherence network is extended with a new message called *flip-current-bit*. When a cache line is updated for the first time in a transaction (*current bit* is zero), a *flip-current-bit* message is broadcast via the cache coherence network to notify other cores as well as memory controllers to flip the *current bit* for the corresponding cache line. Note that we may piggy back the *flip-current-bit* on the invalidation message. The approach can be trivially extended to support directory based cache coherency protocols. We believe the broadcasting will only impose minimal overhead on the system overall. As shown in previous work [84], in typical PM workloads, only a small portion ($< 4\%$) of accesses are to PM; the majority to DRAM. Moreover, our design only requires a broadcast operation for a fraction of stores to PM (e.g. modifying a cacheline for the first time in a transaction).

Transaction Commit. Durable transactions must guarantee data persistence after commit requests are acknowledged. The commit process of SSP involves two steps i) data persistence and ii) metadata update. Atomic updates themselves do not ensure data persistence—some updates might still be in the cache at the transaction commit. Here we use a write-back instruction such as *clwb* to write back the cache lines modified by the committing transaction. The write-set of the committing transaction is tracked by the *updated bitmaps* stored in the extended TLB. The commit process also needs to update the *committed bitmaps* stored in PMEM. We extend the write interface of the memory controller with a special metadata update instruction. For each

updated page (identified by the update buffer), we pass information such as the page ID (*e.g.* P0) and the *updated bitmap* to the memory controller using the metadata update instruction. The memory controller will perform journaling to ensure the atomicity of the metadata updates. The metadata update instructions are passed to the memory controller without caching. Note that we must ensure the ordering between the data persistence and the metadata update. If the system crashes before the atomic metadata update is complete, all speculative updates will be discarded, recovering into an consistent state.

3.2.2 Memory Controller Extensions

In the SSP architecture, the memory controller provides centralized storage for metadata and it performs page consolidation and metadata journaling. Furthermore, it is responsible for managing a set of pages from which the second physical pages can be allocated. Note that the number of pages is bounded by the number of TLB entries and that page consolidation ensures that pages will be freed after they become inactive.

Metadata Storage. SSP associates each virtual page with additional metadata. Since we only need these additional fields when pages are active, we do not require extension of the page table entries. Instead, the memory controller maintains a SSP cache separately to store the SSP-related metadata. An SSP cache entry contains the following details regarding a page that is being actively updated: the original/second physical page number (PPN0/PPN1); the consistent state (*committed bitmap*); the current state (*current bitmap*); the number of TLBs that have cached the translation for this page (TLB reference count); the number of cores that are updating this page (*core reference count*). Among this information, the physical page numbers and the

committed bitmap must be stored durably. Fields such as the *current bitmap*, the core/TLB reference count are transient and are not necessary for the recovery.

Whenever a SSP cache entry is accessed (*e.g.* after a TLB miss), the SSP cache is consulted with the original physical page number (P0). In case of a miss, the memory controller inserts a new entry into the SSP cache. The replacement algorithm of the SSP cache is straightforward. The memory controller may evict any entry that contains a page that is i) already consolidated (*e.g. committed bitmap* is zero) and ii) not referenced by any TLB (*e.g.* TLB reference count is zero). The SSP cache can be sized according to the TLB and the number of cores. For instance, in a system with N cores and T-entry TLBs, the size of the SSP cache is set to $N \times T + O$. Here O is the overprovisioning factor used to accommodate pages that are being consolidated. The rationale behind this is that i) the maximum number of concurrent transactions is N, ii) each transaction can touch no more than T pages, and iii) O entries are overprovisioned so that we do not have to wait for pages to be consolidated in order to make room for new TLB-fill requests. If under rare conditions, we find that the cache entries we reserve are not enough, we can resize the SSP cache and request more pages from the OS.

Free Space Management. At system initialization, the OS will reserve a small mount of continuous PMEM physical pages and pass the base address to the memory controller by setting one of its registers. The memory controller will associate each entry of the SSP cache with an extra physical page up front. The extra physical page is utilized by the virtual page assigned to an entry and can be reused when the entry is assigned to a new virtual page as all data stored in the extra page is persisted during consolidation. To overcome uneven wear out, the memory controller may exchange the per-slot extra physical pages with fresh pages from time to time.

Page Consolidation. SSP decides whether a page is eligible for consolidation according to the following information: is there any TLB that has cached the SSP cache entry for this page? Specifically, the TLB reference count is used to decide when to consolidate a page. The TLB reference count is increased by one if a core fetches the SSP cache entry from the memory controller and is decreased by one if a core evicts the SSP cache entry from its TLB. When the memory controller detects that the reference counter for a page drops to zero, an entry, which includes the two physical page numbers and the *committed bitmap*, is inserted into a consolidation queue. An OS thread conducts page consolidation in the background, allowing the new TLB entry to be inserted with minimal delay. When a page has been consolidated, the consolidation thread inserts an entry into a finish queue to notify the controller. We reserve several bits in the SSP cache entry to track the status of a page (e.g. whether it is being consolidated). In the rare case a page is requested by the TLB during consolidation, the response is delayed until after the consolidation for that page has been completed.

Metadata Journaling. A multi-page transaction requires multiple updates to the metadata area that stores the pages' *committed bitmaps*. As shown in Figure 3.3, each metadata journaling record, which represents the intention to update the SSP cache, has four fields—the Transaction ID (TID), the ID of the cache slot that that is being modified (SID), the new value of original physical page number and the new value of the *committed bitmap*. The TID is assigned by the memory controller to uniquely identify the metadata updates from the same transaction. The SID is used to compute the physical address of the slot given the base address of the SSP cache. Upon receiving a metadata update instruction, the memory controller generates a record and appends it to the metadata journal. Note that journaling records are written back to PMEM, at

cache level granularity, only when the log buffer is full or an explicit request is made to flush the buffer.

Checkpointing. To limit the growth of the journaling space and also to bound the recovery time, the memory controller needs to perform checkpointing, which updates the state of the persistent SSP cache to the most recent consistent snapshot and then needs to clear the journaling space. A background OS thread is used for checkpointing and takes three steps: i) it records the current head pointer—where appends happen—of the journal, ii) it applies the log records to the persistent SSP cache and iii) it advances the tail pointer of the journal. Note that the checkpointing thread will capture the final state of a modified cache entry and only write it back to the persistent cache.

3.2.3 Architecture Details

We here discuss several optional details that improve the efficiency of the implementation of SSP.

Write-set Buffer. Storing the *updated bitmap* in the TLB entry along with the physical page numbers and the *current bitmap*, albeit straightforward, entails a problem—the burst of non-transactional accesses may cause an in-transaction TLB entry (*e.g. updated bitmap is non-zero*) to be evicted, making it impossible to track the write-set of the transaction. To address this issue, a separate write-set buffer can be added to store the *updated bitmaps*. The write-set buffer is cleared once the ongoing transaction is committed. By decoupling the *updated bitmap* from the TLB, a page might be evicted from the TLB while it is written as part of a ongoing transaction. We deal with this corner case with a per-page core reference count. The per-page reference

count is increased upon receiving a *flip-current-bit* from a specific core, and is cleared upon receiving a metadata update instruction. A page with non-zero core reference count will not be considered for consolidation or cache eviction.

SSP Cache Organization. Transient runtime information such as the reference count is updated frequently. Placing the SSP cache in PMEM will cause unnecessary wear out. The SSP cache is organized as a transient SSP cache (stored in DRAM) and a persistent SSP cache (stored in PMEM): the transient cache is employed to serve the requests from the cores; the persistent cache serves as a backup and is used only during recovery. We only store persistent metadata such as the physical page numbers and the *committed bitmap* in the persistent cache.

To leverage faster memory in the hierarchy, we use a small portion of the L3 to as a "cache" for the SSP cache [30]. Only 1% of a 12-megabyte L3 cache could be used to cache about 4K SSP cache entries. We will study the sensitivity of the access latency of the SSP cache in the evaluation.

3.2.4 Hardware Cost and Complexity Trade-off

There are two main hardware overheads in our design: the extended TLB entries and the write-set buffer. For a typical 4 KiB page and 64 byte cache line, there are 64 cache lines per page, so each bitmap has 64 bits, adding 64 bits and a second physical page number (*e.g.* 40 bits) to each TLB entry. Across the 64-entry L1 TLB, the overall cost to expand the TLB is 832 bytes. If we take the L2 TLB into consideration, a 1024-entry L2 TLB will add another 13 kilobytes. Each write-set buffer entry includes a 36-bit tag and a 64-bit bitmap. The size of a 64-entry write-set buffer is therefore 800 bytes. Thus, the overall hardware cost is 14.6

kilobytes.

We now identify opportunities to address the hardware overhead. In our original design, we conservatively assume the ideal granularity for ensuring persistence is 64 bytes (e.g. cache line granularity). However, as disclosed by a recent work [53], the preferable granularity for persisting data to the Intel's Optane DC Persistent Memory is 256 bytes. Utilizing 4× larger sub-pages, the size of the bitmap could be reduced to 16 bits, significantly reducing state overhead of the TLB entries. Furthermore, recent Intel, IBM and ARM processors provide HTM support. HTM uses one transactional bit per cache line to track the speculative updates. By reusing the transactional bit, we might be able to eliminate the need for using the *updated bitmaps*.

Our current design trades-off complexity for higher performance. To reduce hardware complexity, we may drop the modifications on TLB hardware by implementing SSP mappings in userspace. However, this imposes significant instruction overheads as now every load/store must be intercepted similarly to software transactional memory systems. Second, we can avoid the changes on the TLB coherence network by using TLB shootdowns instead. However, the TLB shootdown procedure involves trapping into the OS, issuing inter-process interrupts, imposing a significant performance overhead. Note that our current implementation only serves as a baseline for exploring the viability of SSP. Other alternatives might be considered in practice.

3.2.5 Recovery

Upon restart from an unclean shutdown, SSP performs the following two steps for recovery. First, it rebuilds the transient SSP cache with the persistent metadata stored in the

Processor	4 OoO Cores, 3.7 GHz, 5-wide issue, 4-wide retire, 128 ROB entries, Load/Store Queue: 48/32, 64 DTLB entries
L1I and L1D	32 KiB, 64-byte lines, 8-way, 4 cycles
L2	256 KiB, 64-byte lines, 8-way, 6 cycles
L3	12 MiB, 64-byte lines, 16-way, 27 cycles
DRAM	8 GiB, 1 channel, 64 banks per rank, 1 KiB row-buffer, read/write 50 ns
PMEM	8 GiB, 1 channel, 32 banks per rank, 2 KiB row-buffer, read/write 50/200 ns

Table 3.1: System Parameters

Name	Write Set	Description
RBTree-Rand	12/3/13	Insert/delete nodes in a red-black tree; Random workloads
BTree-Rand	10/6/21	Insert/delete nodes in a B+-Tree; Random workloads
Hash-Rand	3/3/4	Insert/delete nodes in a hashtable; Random workloads
SPS	2/2/2	Swap elements in an array
RBTree-Zipf	5/2/6	Insert/delete nodes in a red-black tree; Zipfian workloads.
BTree-Zipf	6/4/15	Insert/delete nodes in a B+-Tree; Zipfian workloads.
Hash-Zipf	3/3/4	Insert/delete nodes in a hashtable; Zipfian workloads
Memcached	3/2/35	Memslap as workload generator; Four clients; 90% SET
Vacation	4/3/9	Four clients; 16 million tuples

Table 3.2: A list of evaluated microbenchmarks showing the write set size (average number of cache lines modified / average number of pages modified / maximum number of pages modified). The write set consists of atomic updates within a transaction.

persistent cache. Specifically, fields such as the two physical page numbers and the *committed bitmap* are reloaded directly from the persistent cache. Then the *current bitmap* is initialized with the value of the *committed bitmap* all other transient fields (*e.g.* reference counters) are initialized as zeros. Furthermore, the state of the transient cache needs to be updated to the most recent consistent snapshot, replaying the records in the metadata journal whereas the entries of aborted transactions are skipped.

3.3 Evaluation

3.3.1 Experimental Setup

We implemented SSP on `MarssX86` [91], which is a cycle-accurate full system simulator for the x86-64 architecture. We integrated `DRAMSim2` [97] into `MarssX86` for a more detailed memory simulation. The `DRAMSim2` is extended to model a hybrid memory system with both DRAM and PMEM connected to the same memory-bus. Table 3.1 shows the main parameters of the system. Our simulated machine supports out-of-order execution, and includes a 64-entry L1 DTLB, 3 levels of cache, and an NVDIMM.

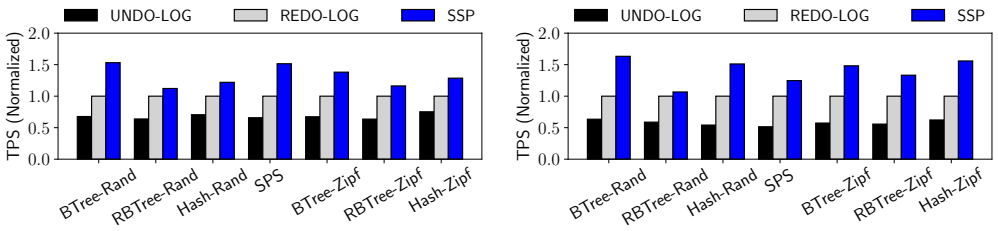
Simulation Methodology. We simulate the impact of logging, page consolidation as well as data persistence using the `MarssX86` and `DRAMSIM2`. To model the impact of SSP on the TLB and on cache coherency, we measure the number of TLB misses and the number of *flip-current-bit* messages. Note that we only count the TLB misses caused by accessing the persistent heap. The latency of accessing SSP cache is modeled for a given workload according to the L3 SSP cache miss ratio, L3 latency (*e.g.* 27 cycles) and DRAM latency (*e.g.* 185 cycles). The extra cycles are then added to the total cycles. To better understand the impact of the latency of SSP cache access, we conduct a sensitivity study in Section 3.3.3. In our experiment, we reserve 0.3% of the L3 cache to be used to store the SSP cache (*e.g.* about 1K SSP cache entries).

Benchmarks. We evaluate both microbenchmarks and real workloads in our experiments. The microbenchmarks cover commonly used data structures such as the B+-Tree (*BTree*), as described in Table 3.2. The elements, keys or values used in these workloads are all 8-byte integers. Each data structure update (*e.g.* insert, delete, or swap) is wrapped inside a durable transaction. Benchmarks *BTree*, *RBTree* and *Hash* first search for a key and then either delete

(key found) or insert (key absent) a key/value pair. We vary the access patterns for these workloads by changing the key distribution. We use suffix "-Rand" and "-Zipf" to denote random workloads and zipfian workloads. For zipfian workloads, 80% of the updates are applied to 15% of the keys. The key/value pairs are generated prior to each run. We evaluate two real applications: Memcached [37] is a well-known in-memory Key/Value cache and Vacation [82] which emulates an OLTP system. Prior work [84] has published the persistent-memory-aware version of these applications; we merely replace their durable interfaces with ours. The characterization of the benchmarks is shown in Figure 3.2. As none of the evaluated applications writes to more than 64 pages during a transaction, a 64-entry write-set buffer is sufficient to accommodate all of the workloads. As a result, none of our evaluated applications requires the unbounded fall-back path.

Evaluated Designs. We compare SSP with two other designs for which we use tuned, optimal parameters (*e.g.* size of the log buffer). We do not compare with conventional shadow paging. As shown in Table 3.2, transactions only touch 2-6 cache lines on average. Conventional shadow paging degrades performance by writing up to 64× more cache lines.

- UNDO-LOG represents a naive hardware undo logging mechanism. Each atomic store will generate a log entry. The store then will be blocked until the log entry reaches persistent memory. Under undo logging, if a value is repeatedly updated multiple times, we only need to generate a log entry for the first update. We employ a log buffer to avoid writing redundant log entries.
- REDO-LOG [56] is a state-of-the-art hardware redo logging. It allows overlapping data persistence with the non-transactional code following the transaction commit. Besides, it



(a) One thread.

(b) Four Threads.

Figure 3.5: Performance of micro-benchmarks (higher is better).

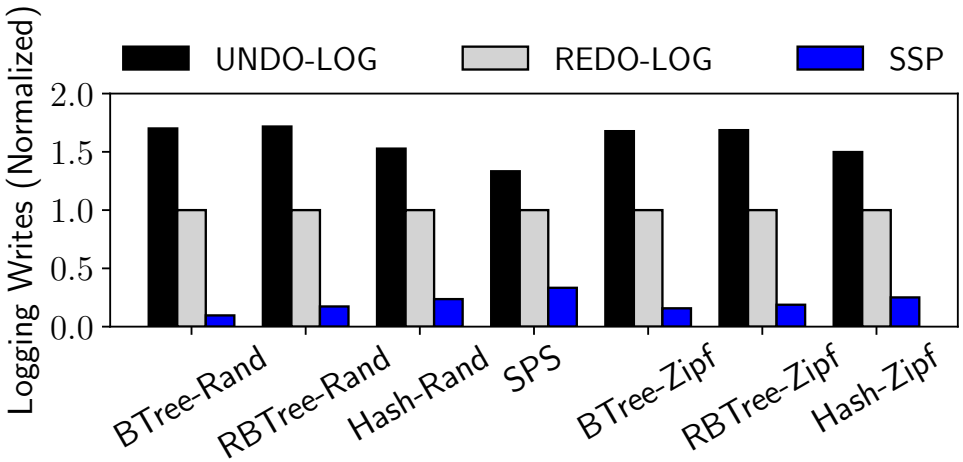
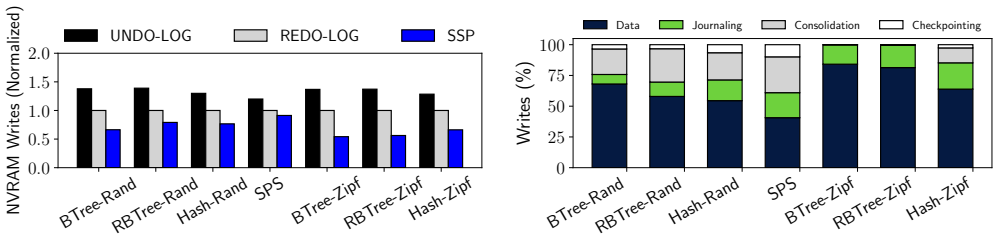


Figure 3.6: Comparison of logging writes (lower is better).

also employs a log buffer to predict the final state of a cache line and thus avoids redundant log entries.

3.3.2 Mirobenchmark Results

Transactional throughput. We show performance results of the four designs running the microbenchmarks. From Figure 3.5a, we observe that SSP outperforms UNDO-LOG and REDO-LOG by 1.9× and 1.3× on average under single threaded workloads. The improvement mainly comes from the ability of SSP to reduce the logging overhead. As shown in Figure 3.6, SSP can



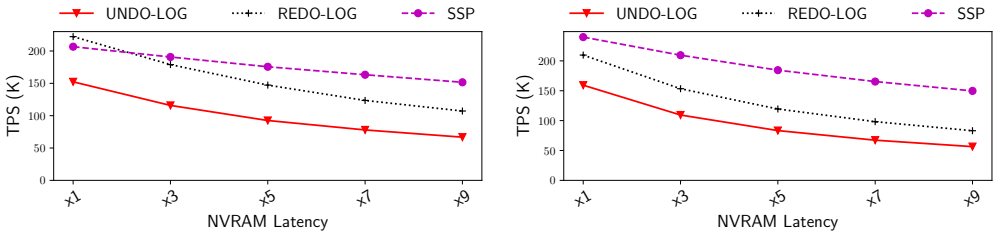
(a) Comparison of PMEM writes.

(b) Breakdown of PMEM writes for SSP.

Figure 3.7: PMEM writes.

decrease the write traffic caused by logging by 7.6× and respectively 4.7× compared to UNDO-LOG and UNDO-LOG. In particular, under the *BTree-Rand* workload, SSP nearly eliminates the logging writes, which results in a 1.5× improvement in transactional throughput. As we can see in Table 3.2, the *BTree* benchmark exhibits great spatial locality (*e.g.* several cache lines modified in a single page), which minimizes the writes introduced by metadata journaling. Figure 3.5a shows the performance under four threads. We can see SSP scales well. SSP can improve the performance by 2.4× and 1.4× over the UNDO-LOG and REDO-LOG on average, respectively.

PMEM Writes. Figure 3.7a compares SSP to the baseline designs in terms of the number of PMEM writes. We make two observations. First, SSP can save 45% and 28% write traffic as compared to UNDO-LOG and REDO-LOG on average, as the Undo/Redo logging designs essentially require data to be written twice. Although page consolidation also causes extra writes in SSP, it is not a per-transaction operation: modified data does not require an immediate copy operation during the transaction commit but instead additional writes are only required when an active page turns inactive. As the transaction frequency is much higher than the frequency of pages becoming inactive, SSP can effectively "batch" the additional writes required for failure-



(a) RBTree. (b) BTree.

Figure 3.8: Sensitivity to the Latency of PMEM: the x-axis shows the PMEM latency in multiple of DRAM latency.

atomicity. Figure 3.7b shows the breakdown of PMEM writes in our SSP design. As we can see, the number of writes caused by page consolidation is less than the data writes under most of the workloads except for *SPS*. Second, the locality of the workloads affects the number of PMEM writes. Under benchmarks with zipfian access pattern (*e.g. BTree-Zipf*, *RBTree-Zipf* and *Hash-Zipf*), SSP on average can reduce the write traffic by 56% and 42% over UNDO-LOG and REDO-LOG. In contrast, for random workloads using a unified distribution SSP can only save 43% and 23% write traffic over UNDO-LOG and REDO-LOG. As shown in Figure 3.7b, under workloads with locality, extra writes caused by page consolidation are negligible. It demonstrates the SSP design can efficiently prevent the premature consolidation of hot pages and thus minimize page consolidation overhead for zipfian workloads.

3.3.3 Sensitivity Study

Latency of PMEM. Figure 4.9 shows the transaction throughput with varying memory latency. Overall, it can be seen that the performance of SSP and the baseline designs degrade when the PMEM latency increases. However, the gap between SSP and other designs is increasing

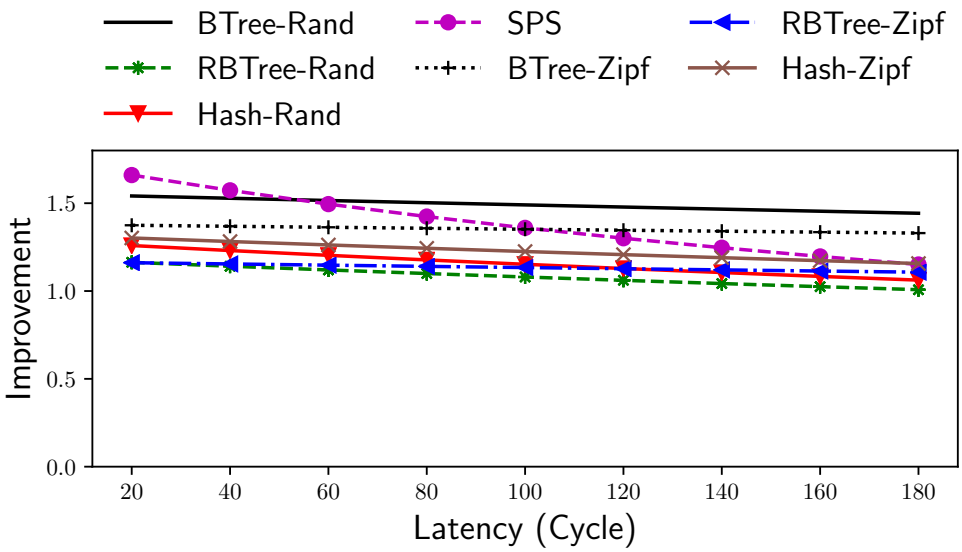


Figure 3.9: Sensitivity to the latency of SSP Cache: y-axis shows the speedup over REDO-LOG.

as well. In particular, the speedup over REDO-LOG increases from 1.1× to 1.8× under the benchmark *BTree* (Figure 3.8b). SSP minimizes the logging writes and thus is less sensitive to the change of the PMEM latency. We observe that when the PMEM is as fast as DRAM, REDO-LOG outperforms SSP by 8% under the benchmark *RBTree* (Figure 3.8a). The reason behind this is that when the persistency overhead is low (*e.g.* DRAM latency), REDO-LOG can hide the most of the delay caused by persisting data.

Latency of the SSP Cache. Figure 3.9 shows the impact of the latency of the SSP cache on the performance of our SSP design. For all workloads besides *SPS*, the cache latency has limited impact on SSP performance, showing only a moderate linear performance decrease with increased latency. However, the latency of SSP cache is still critical for benchmarks such as *SPS* and *Hash-Rand*. This is because their poor locality leads to frequent TLB misses, which in turn increase the frequency of accessing the SSP cache. We observe that the zipfian workloads are

	UNDO-LOG	REDO-LOG
Memcached	75%	35%
Vacation	27%	13%

Table 3.3: The performance improvement over other designs for Benchmarks *Memcached* and *Vacation*.

	UNDO-LOG	REDO-LOG
Memcached	49%	46%
Vacation	38%	17%

Table 3.4: The saving of write traffic over other designs for Benchmarks *Memcached* and *Vacation*.

less sensitive to the latency of the SSP cache than these random ones. This can also be explained by the difference in locality exposed by these workloads.

3.3.4 Performance of Real Workloads

Table 3.3 shows the performance improvement of SSP over other designs for the *Memcached* and *Vacation* benchmarks. For the *Memcached* benchmark, SSP provides a 74% throughput improvement over UNDO-LOG and a 35% higher throughput compared to REDO-LOG. For the *Vacation* Benchmark, SSP provides a 27% improvement over UNDO-LOG and 13% higher throughput over REDO-LOG. The improvement comes from the reduction in logging overhead. Specifically, SSP saves 86% and 82% logging writes over UNDO-LOG and REDO-LOG under the real workloads on average. In the *Vacation* benchmark, SSP generates less improvement over REDO-LOG. This is because the volatile execution contributes to most of the overhead of the *Vacation* benchmark.

Table 3.4 shows the reduction of PMEM writes. As we can see, SSP continues to save write traffic to PMEM: 49% and 46% reduction over UNDO-LOG and REDO-LOG under the

Memcached Workload and 38% and 17% reduction over UNDO-LOG and REDO-LOG under the *Memcached* Workload. The extra write traffic caused by page consolidation is only 15% and 31% of the total write traffic for the *Memcached* and *Vacation* workloads.

3.4 Chapter Summary

We proposed SSP, a novel shadow paging scheme that leverages fine grain cache line level remapping, to enable efficient, failure-atomic transactions. In particular, SSP eliminates most of the redundant writes introduced by prior log-based techniques. Our key idea is that we can delay the application of redundant writes via address remapping, enabling write batching to reduce the overall number of writes to NVRAM. By introducing cache line remapping, our technique successfully eliminates the copy-on-write overhead that made prior shadow mapping schemes unfeasible, while only requiring moderate changes to the TLB hardware. In addition to improving endurance, SSP removes redundant writes from the critical path improving transactional performance. In particular, our experimental results show that SSP can reduce overall NVRAM writes by up to 1.8×, and improve performance by up to 1.6×, as compared to a state-of-the-art hardware logging.

Chapter 4

Near-Optimal Resource Allocation for Tiered-Memory Systems

Allocation policies of the tiered-memory system [5, 23, 36, 59, 61, 63, 74, 121, 124] have been devised to determine the best memory type for a given data item given a fixed fast to slow memory ratio. We claim that these prior works on allocation policies are insufficient in the IaaS cloud setting, where operators and customers desire improved cost efficiency in addition to raw performance.

Selecting the right resources, such as the optimal fast to slow memory ratio, is critical for both the cloud provider and client. For instance, a wrong configuration (*e.g.* a wrong fast to slow memory ratio) increases the TCO by up to 2.6× for the cloud customer. In addition, the optimization of cloud configurations also determines the overall resource efficiency of a data center. In particular, to reduce resource stranding, an efficient tiered-memory configuration policy also needs to consider the actual available amount of memory in the different tiers.

Existing solutions utilizing simulation [36] or machine learning techniques [79, 88, 105, 110] are insufficient as they incur high cost when being used to find a cost-optimal configuration for an application. Tiered memory configurator (TMC) profiles applications to reveal internal properties, which then can be used to enable fast performance estimation. We first devise a model to optimize the memory and data structure placement into different memory tiers for a single application and then generalize the technique to optimize allocation across applications running in a data center. TMC's model only contains a minimal amount of workload specific variables which can be filled in with the data of only three profile runs of an application. Moreover, we proposed to use a new heuristic, namely packing penalty, to quantify the impact of a configuration on the resource efficiency in the cloud.

4.1 Motivation

Public IaaS cloud providers such as Amazon's EC2 [1] offer their customers a limited number of predefined VM instance types and charge them on a per-hour basis. On the other hand, a few IaaS cloud providers, such as Google's Compute Engine, further allow cloud users to create a VM instance with a customized number of vCPUs and amount of memory [2]. Prior work [131] has shown that making VM customizable is beneficial for both the provider and the user. This work targets a future IaaS cloud incorporating both traditional DRAM and a second, slower memory tier. We expect, that in future clouds, customers will be able to configure the number of vCPU, the amount of local DRAM, the amount of second tier memory, and the capacity of the last-level cache (LLC) of their vCPUs in a VM. The IaaS cloud charges the

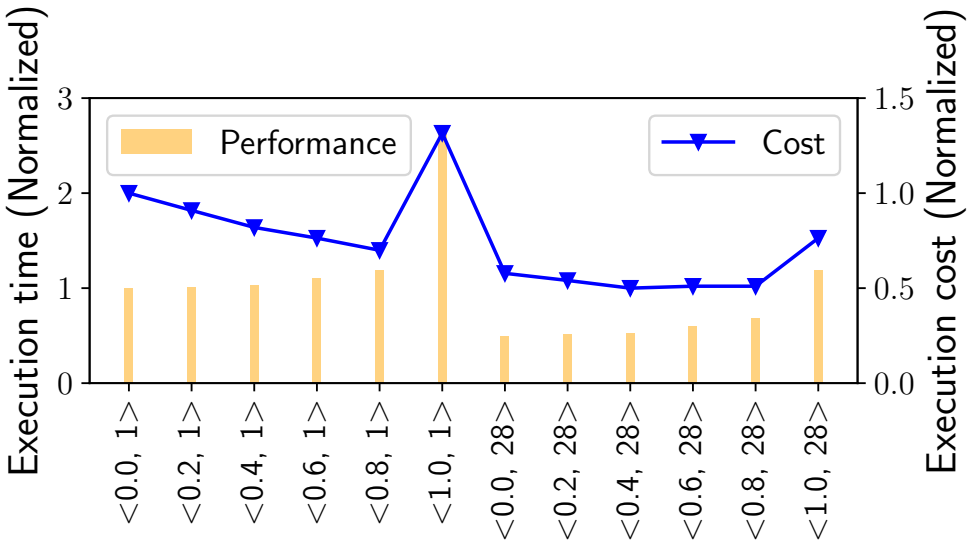


Figure 4.1: Execution time and cost for 12 ⟨Slow memory ratio, LLC capacity⟩ configurations (*graph500*)

users based on Equation 4.1.

$$Total\,cost = VM\,cost \times Execution\,time \tag{4.1}$$

Scaling down a VM configuration reduces its hourly VM cost, however, also increases the execution time of applications. In order to minimize the total cost, one must choose a proper configuration that optimizes for both. Figure 4.1 shows the execution time and cost for *graph500* utilizing different ratios of slow and fast memory [1] and LLC sizes. As the ratio of slow memory increases, the total cost is reduced, though the run time increases. This is because the saving in hourly cost outweighs the performance penalty of scaling down the VM. However, the total cost eventually goes up as the increase in run time overwhelms the saving

[1] Slow memory ratio can be represented as $\frac{slow\_mem\_size}{slow\_mem\_size+DRAM\_size}$

Applications	Config 1	Config 2	Config 3
	2× slower, 0.7× cost	3× slower, 0.4× cost	4× slower 0.3× cost
cactusBSSN	⟨ 0.1, 28 ⟩	⟨ 0.1, 28 ⟩	⟨ 1.0, 28 ⟩
graph500	⟨ 0.4, 28 ⟩	⟨ 0.4, 28 ⟩	⟨ 0.9, 20 ⟩
memcached	⟨ 0.0, 2 ⟩	⟨ 0.9, 2 ⟩	⟨ 0.9, 2 ⟩
xsbench	⟨ 0.9, 4 ⟩	⟨ 0.9, 3 ⟩	⟨ 0.9, 3 ⟩
canneal	⟨ 0.6, 2 ⟩	⟨ 0.6, 1 ⟩	⟨ 0.6, 1 ⟩
xhpcg	⟨ 0.0, 1 ⟩	⟨ 0.0, 1 ⟩	⟨ 0.0, 1 ⟩

Table 4.1: Diversity of optimal configurations: Each column represents the optimal memory configurations for a specific tiered-memory configuration.

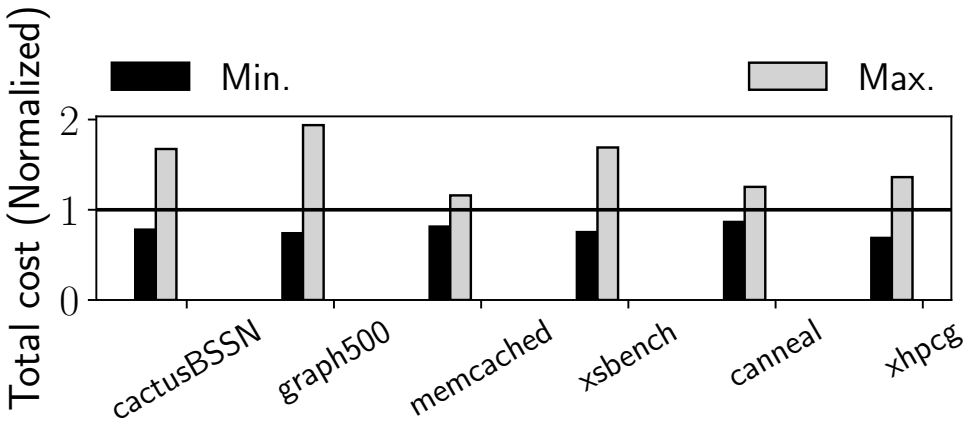


Figure 4.2: Costs for best/worst/average configuration, normalized to the average cost of all configurations.

in hourly cost. Note that in this experiment, we translate a performance slowdown into cost by computing the additional total number of VMs required to offset the performance degradation. This assumes that applications are throughput-bound, ie an increase in execution time can be offset with additional hardware resources. This assumption is typical for data centers that scale user request-level throughput with hardware resources or deploy high fan-out architectures (e.g. Google Websearch) to distribute the execution time of a request across servers.

Table 4.1 shows the the optimal tiered memory ratio and LLC configuration across six different workloads and three memory technologies. Each column represents a specific tiered

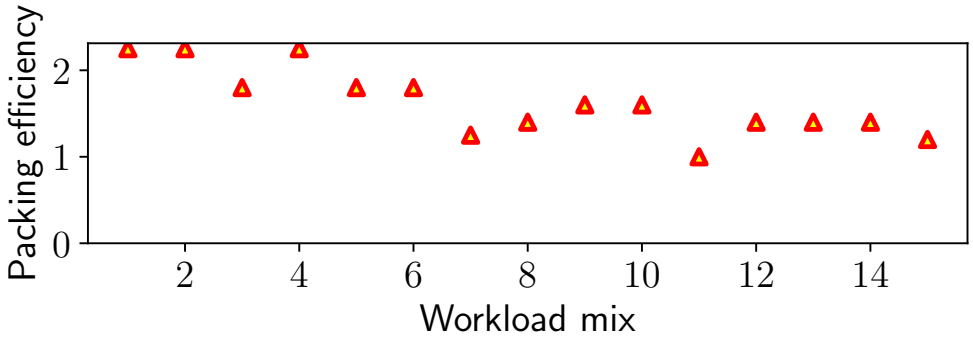


Figure 4.3: The packing efficiency improvement achieved by a resource-optimal policy over the naive policy.

memory technology where the slow memory is s times slower and p times cheaper than DRAM. Performance per TCO optimal configurations for each workload are shown in the form of ⟨slow memory ratio, LLC size⟩. For example, if the the slower memory tier has a 3× higher read latency but 0.4× lower cost, the cost-optimal configuration for the workload *graph500* has a ⟨ slow memory ratio = 0.4, LLC size = 28 ways ⟩. As we can see, there exists no configuration that is uniformly best for all workloads or memory technologies. Figure 4.2 further shows the minimum and maximum cost for the different hardware configurations and workloads (normalized to average cost of all configurations). As can be seen, customers spend 1.2–1.5× less for the optimal configuration compared to the average configuration and 1.4 - 2.6× less compared to the worst configuration.

Cloud customers request VM instances of a specific hardware configuration. The cloud's VM scheduler is responsible for selecting a server that can hold the new VM according to the hardware requirements and the current availability of machines in the cluster. One important aspect for optimizing the cost efficiency of such clouds is to optimize the packing den-

sity [111]. If VMs can be packed into fewer machines at a given time, idle machines can be powered down to save energy and cost, or they can be used to run low-priority batch jobs. Packing inefficiency leads to resource stranding where one of the resources (e.g. vCPUs) becomes fully utilized while others (e.g. memory) are not. Existing cluster schedulers [39, 41, 112] consider packing efficiency during job scheduling to maximize cloud resource utilization. However, they fail to be effective if the resource demand of the VM workload is fundamentally unbalanced. For example, if all workloads at a given moment request disproportionately large amounts of DRAM, a large amount of second tier memory can be left unused. VM configurations need to adopt based on the real-time resource utilization of the cloud.

We thus investigate the potential upper-bound benefit of considering packing efficiency when choosing a tiered memory configuration. In particular, we determine the optimal slow to fast memory ratio based on resource availability. We evaluate 15 different workload mixes consisting of 4 applications each, and compute the benefit provided by the resource-optimal policy considering packing efficiency over a naive policy. The naive policy requests optimal tiered memory allocations for each individual application in isolation, whereas the resource-optimal policy considers the availability of physical hardware resources. For instance, if a machine contains 3× more slow than fast memory, the policy reserves 0.5× fast and 1.5× slow memory regardless of what the actual optimal slow to fast memory ratio of an application is. As we can see from Figure 4.3, the resource-optimal policy achieves up to 2.2× higher packing efficiency than the naive cost-optimal configuration, motivating the consideration of both configuration cost and packing efficiency.

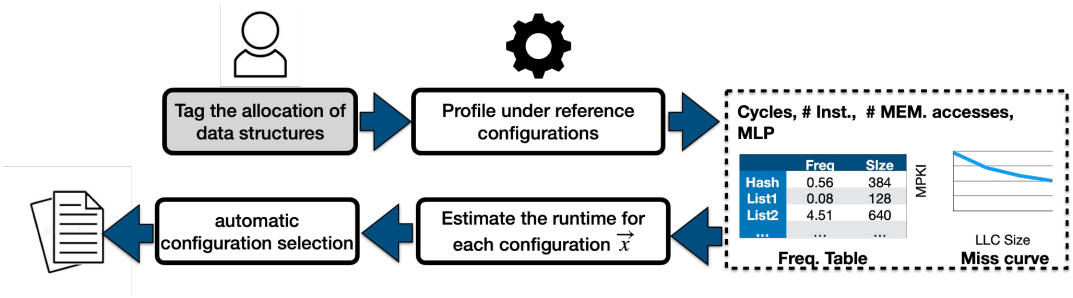


Figure 4.4: workflow of TMC

4.2 TMC Design

In this section we describe how TMC produces tiered memory and LLC configurations optimizing cost efficiency for both the cloud provider and user.

4.2.1 Overview

TMC requires 3 inputs: i) the workload submitted by a cloud customer, ii) the cost model stating the price of slow and fast memory as well as the cost of an LLC way, and iii) the latest resource utilization info of the available hardware. To enable its performance predictions, TMC monitors the memory consumption of an application during a profiling stage. TMC then automatically determines two VM parameters: the near-optimal memory ratio of local DRAM and slow memory, and the number of LLC-ways allocated to the VM. In line with prior work [36, 115], we observe that the characteristics above are almost always data-structure specific. As a result, our proposed methodology determines application properties such as memory access rate on a per data-structure level and not on raw page data. Every memory allocation is assigned with a tag, either by the user, compiler, or allocator (by examining the call stack).

Symbol	Definition
CPI	Cycle Per Instructions
CPI_{cache}	CPI in a system with perfect LLC cache
CPI_{dram}	CPI in a DRAM-only configuration
MPI	LLC misses per instruction
MPI_{dram}	LLC misses to DRAM per instruction
MPI_{slow}	LLC misses to slow memory per instruction
MLP	Memory level parallelism
L_{dram}	DRAM latency
L_{slow}	Latency of slow memory
ΔL	Latency increase in slow memory over DRAM

Table 4.2: List of symbols and their definitions

Figure 4.4 outlines the workflow of TMC. After receiving the workload, TMC profiles it using three reference configurations: $\langle$slow memory = 0%, LLC = MIN$\rangle$, $\langle$slow memory = 0%, LLC = MAX$\rangle$, $\langle$slow memory = 100%, LLC = MAX$\rangle$. Important application properties, such as average memory latency and MLP are collected in the profiling step. The application properties are then used to enable performance prediction (Section 4.2.3) and to guide data placement (Section 4.2.4). Finally, our configuration selection algorithm searches for a configuration that satisfies both the customer's and cloud provider's needs (Section 4.2.5).

As an optimization, TMC can retain the performance profiles for workloads it has profiled recently. TMC can simply use the command and the user's input (*e.g.* number of iterations for a scientific application) to uniquely identify a workload. Prior work [64] has investigated automatically identifying parameters that significantly impact the application behavior.

4.2.2 Analyzing Application Properties

In this section, we analyze prior work on application performance modeling and then propose an improved model that can handle tiered memory systems. Prior work [26] proposed

Eq. 4.2 to quantify the relationship between off-chip memory accesses and the application performance in a homogeneous system (*e.g.* DRAM-only). Table 4.2 shows the performance metrics used by the devised models.

$$CPI_{dram} = CPI_{cache} + \frac{MPI \times L_{dram}}{MLP} \tag{4.2}$$

We extend Eq. 4.2 to model systems with a tiered memory hierarchy as shown in Eq. 4.3

$$\begin{aligned} CPI &= CPI_{cache} + \frac{MPI_{dram} \times L_{dram} + MPI_{slow} \times L_{slow}}{MLP} \\ &= CPI_{cache} + \frac{(MPI - MPI_{slow}) \times L_{dram} + MPI_{slow} \times L_{slow}}{MLP} \\ &= CPI_{cache} + \frac{MPI \times L_{dram} + MPI_{slow} \times (L_{slow} - L_{dram})}{MLP} \\ &= CPI_{dram} + \frac{MPI_{slow} \times \Delta L}{MLP} \end{aligned} \tag{4.3}$$

The performance penalty introduced by a second-tier memory technology is determined by the application's memory access rate, its latency-sensitivity, and the availability of memory level parallelism (MLP). As a result, the performance prediction mainly relies on three types of application properties:

Access rate to the slow memory. To devise a performance model, TMC needs to estimate how the slow-memory access rate changes with the size of the slow memory tier and that of the LLC. As a result, our profiling mechanism measures the access rate (*i.e.* access per

instruction) of each data structure.

Memory latency. The effective memory latency can be highly dependent on the access pattern of the application [53, 74, 125]. For instance, the queuing delay within the memory subsystem depends on the memory bandwidth, while the access patterns affect the probability of DRAM row hits. Thus, we assess the average end-to-end memory latency for each application individually. In particular, for each memory tier, we measure the average memory latency when all of the application's data is placed in that memory tier. We assume that memory latency is similar across machines; if machine types differ significantly, profiling per machine type is required.

Memory level parallelism. Memory accesses are costly and introduce long CPU stalls. To hide some of this latency, contemporary Out-of-Order CPUs execute multiple memory accesses in parallel. To consider MLP, TMC profiles the average number of outstanding memory accesses to an application. Our approach assumes that MLP is an application specific property unaffected by the memory technology, which has shown to hold in practice. While there does not exist a PMU counter to directly measure MLP, we can derive it from measuring back-end stalls. We provide a detailed description of its implementation in Section 4.4.

4.2.3 Inferring Tiered-Memory Performance

As described in Section 4.2.1, TMC only performs 3 profile runs to determine application specific performance models. However, due to spatial and temporal locality within memory accesses, memory tiers affect performance non-linearly. In particular, there exist performance *cliffs* [13, 25], for instance, when the hot working set of an application exceeds the

LLC size. We utilize *cache miss curves* to estimate the memory access rate depending on the LLC-size configuration as proposed by Qureshi [94]. The technique utilizes the LRU stack property to measure the number of hits in a small number of cache sets and then derives a miss curve from the per-set hit counters. In addition, we analyze application memory traces collected throughout the profiling step to build a *frequency table*, tracking the access rate and memory region of each data structure. During offline analysis, we iterate through each memory access, map it back to the data structure according to its address, and then update the access count of that data structure. By sorting data structures by access frequency, we can estimate the slow-memory access rate based on a particular size of slow memory and its contained data structures. Data structure access frequencies can be obtained via Intel's Precise Event-Based Sampling (PEBS) and other similar hardware sampling techniques as it enables the recording of LLC-miss addresses directly. We will investigate the implication of the PEBS sampling rate on accuracy and performance overheads in Section 4.4.

Profiling the first and second reference configuration, as shown in Section 4.2.1, only provides us with the CPI_{dram} for two special DRAM-only configurations, where LLC = MIN, respectively LLC = MAX. In the following, we describe how to estimate CPI_{dram} for all other LLC configurations. First, we compute CPI_{cache} for $LLC = MIN$ using Eq. 4.4 where $CPI_{dram}(LLC = MIN)$ and $MPI(LLC = MIN)$ are the CPI and MPI measured in the reference configuration $\langle$slow memory = 0%, LLC = MIN$\rangle$. We then estimate the CPIs for all other LLC configurations using (Eq. 4.5) where $MPI(LLC = x)$ represents the *cache miss curve*. We can utilize the cache miss curve to estimate the memory access rate given a certain number of

LLC-ways x.

$$CPI_{cache} = CPI_{dram}(LLC = MIN) - \frac{MPI(LLC = MIN) \times L_{dram}}{MLP} \tag{4.4}$$

$$CPI_{dram}(LLC = x) = CPI_{cache} + \frac{MPI(LLC = x) \times L_{dram}}{MLP} \tag{4.5}$$

Above we explained how we can compute the CPI for all LLC configurations for the DRAM-only system. In the following, we discuss how to compute the performance of a tiered memory system. In particular, we estimate the performance loss induced by accessing slow memory, depending on the slow to fast memory ratio and application characteristics. To estimate the slow-memory access rate, we again utilize the *cache miss curve* as well as the *frequency table*. The *frequency table* is used to estimate the relative access rate to different data structures. We estimate the slow-memory access rate to be the cumulative access rate of data structures that will be placed in slow memory. An example of the *frequency table* is shown in Figure 4.4. Data structures in this example have already been ranked according to their hotness. Let us assume the working-set size of the application is 2.5 GiB in the frequency table example. When the slow memory ratio is 20% (512 MiB), the entire data structure of `Hash` (384 MiB) and `List1` (128 MiB) will be placed into slow memory, so that the access rate to the second-tier memory can be estimated to be the accumulated access rate of Hash and `List1` (*i.e.* 0.8.). The *frequency table* only allows us to estimate the slow-memory access rate for the configurations where LLC = MIN or LLC = MAX. Analogous for computing performance for all LLC-ways, we now again leverage the *cache miss curve* to estimate the slow-memory access rate for the

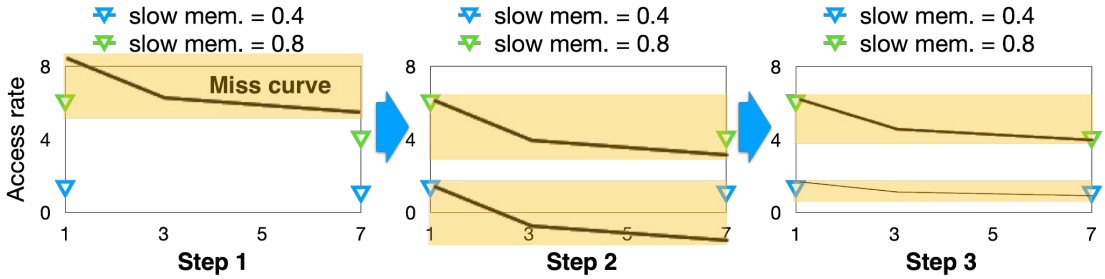


Figure 4.5: Estimating the rest of configurations with the *miss curve*

other configurations, i.e. configurations with other LLC sizes than MIN or MAX. The process is detailed in Figure 4.5. First, for each ratio, the access rate to the second-tier memory is only known for two LLC sizes (MIN and MAX), constituting the start and end point of a *slow-memory miss curve* . The *slow-memory miss curve* depicts how the slow-memory access rate changes over the LLC size. We can approximate the complete miss curve for the target ratio by first moving the *cache miss curve* to align with the start point of a *slow-memory miss curve* (step 2) and scale it vertically to fit the end point (step 3).

4.2.4 Data Placement

The presented methodology above enables TMC to determine a near-optimal slow to fast memory ratio and number of LLC ways. While Section 4.2.3 will show how to compute these exact ratios and ways, we first need to devise a policy to decide which data structures should be placed in fast respectively slow memory. TMC utilizes a policy based on the access count per MiB as determined by $\frac{Access\ Count}{Size}$ to represent the hotness of a data structure. While prior work such as X-MEM [36] have proposed more complex policies that consider MLP for data structure placement, we observed that such an approach only provides little performance

Algorithm 1 Configuration Selection Algorithm

$conf_i$ ▷ i'th candidate configuration
C_i ▷ Total cost for $conf_i$
R ▷ Total resource capacity in a machine
U ▷ Overall resource utilization in a cloud

procedure OPTIMIZATION(...)
▷ First round: optimize the cost for the customers
for each $i \in \{1...N\}$ **do** ▷ N is the size of $conf$
if $\frac{C_i}{C_{min}} - 1 \leq T$ **then**
opt.insert($conf_i$)
end if
end for

▷ Second round: optimize the resource efficiency
for each $i \in \{1...S\}$ **do** ▷ S is the size of opt
$D_i \leftarrow \{\frac{opt_{i,cpu}}{R_{cpu}}, \frac{opt_{i,dram}}{R_{dram}}, \frac{opt_{i,slow}}{R_{slow}}, \frac{opt_{i,llc}}{R_{llc}}\}$
$penalty_i = D_i \cdot U$
end for
Sort the opt according the $penalty$ (increase)
return opt_0
end procedure

improvements over a policy based on access count. In addition, X-MEM introduces a 40× slowdown as it must use expensive application instrumentation to analyze continuous memory access traces in order to classify memory accesses as pointer chasing, sequential or random.

4.2.5 Optimizing Packing Efficiency

Now that we can estimate the run time, we calculate the total cost for all candidate configurations. Algorithm 1 shows the pseudo-code for our configuration selection. Our selection algorithm consists of two rounds. In the first round, we identify all configurations that satisfy the cost-performance objective of the customers. If the estimated cost of a configuration is within a threshold `T` of the estimated optimal cost, we consider it as a cost-optimal configu-

ration. In the second round, we pick a configuration that maximizes the resource efficiency for the cloud provider. We propose a new heuristic, *packing penalty*, to evaluate the impact of a configuration on the resource efficiency at the cloud: a configuration with lower packing penalty makes more efficient usage of cloud resources and vice versa. The packing penalty is the dot product of two vectors U and D whose dimension is the number of valid configurations. U represents the overall resource utilization of each configuration, while D represents the resource demand of each configuration. The rationale behind this is that the configurations utilizing a large amount of scarce resources should be penalized. Resource demand of a configuration will be normalized first using the total resource capacity per-machine. Note that when the slow memory tier is provisioned across machines as in CXL memory pools, the slow-memory capacity per machine is statically determined by the total capacity of CXL-memory across machines.

4.2.6 Discussion

Bandwidth. Our prediction model assumes that the end-to-end memory latency of an application is relatively stable across different configurations. We found that this assumption holds as long as the memory bandwidth of the system is not overly saturated, particularly, less than 80% of peak. As demonstrated in prior studies [38, 53, 54, 125], there is a "knee" in the bandwidth-latency curve at around 80% of the maximal bandwidth. For most of the operating range, memory latency is relatively flat, however, increases exponentially after the "knee". As reported in [53], the "knee" for 3DXP when dealing with random read workload is at around 10 GiB/s where the latency increases from 300ns to 400ns. Our technique assumes that the cloud provider enforces workload mixes via scheduling that consume at most 80% of a machines

DRAM bandwidth. This has been naturally the case for all workload mixes evaluated in this paper. Google has reported [58] that data center applications are almost exclusively DRAM latency and not bandwidth limited. If higher bandwidth utilization is desirable, our model can be extended easily.

Application specific MLP. Prior work has proposed measuring data structure specific MLP by extending the memory controller [74] or by adding expensive instrumentation [36]. However, our study indicates that replacing application specific MLP with data structure specific MLP provides little improvement over the prediction accuracy (less than 1%). Hence, our work measures the MLP of the application, assuming MLP is identical across different memory configurations. Our approach has shown to be accurate when applied in real systems for predicting application performance (Section 4.4).

Noisy profiles. If the profiling runs of an application happen to be scheduled on a machine that is under abnormal conditions *e.g.* overloaded, TMC might produce a performance model that is not representative of the machines in the cluster. To overcome this problem, we can design a micro-benchmark that performs operations exercising different resources in the system, have it run periodically in the machines of the cluster, and report back the representative metrics. We will schedule the profiling runs on a machine whose condition is consistent with most of machine in the cluster.

	Description
CactusBSSN	Model a vacuum flat space-time
Graph500	BFS search on an undirected graph
Memcached	Workload C in YCSB
XSBench	Monte Carlo neutron transport
Canneal	Optimize routing cost of a chip design
XHPCG	Preconditioned conjugate gradient

Table 4.3: Description of the benchmarks

4.3 Evaluation in Simulation

In this section, we first present our experiment methodology. We then analyze the effectiveness of TMC.

4.3.1 Experimental setup

Cloud VM Simulation. We simulate physical hardware in the cloud using the cycle-accurate simulator Scarab [45]. We modify Scarab to implement the required performance counters for capturing cache miss curves and application-specific properties (see Section 4.2.3) required by TMC. The simulated machines in our study contain 12 cores, 12 MiB 48-way associative LLC and 24 GB DRAM. We simulate a fast memory tier (DRAM) and a CXL-attached memory pool as the slow tier. Our CXL-based disaggregated system provides a shared memory pool for each 8-node rack. Similar to prior work [73], we add an additional latency of 85 ns to each CXL access *i.e.* the end-to-end memory latency is comprised of the CXL delay and the access latency of the second tier memory. Although our work makes no assumption on the type of media that will be used in the slow memory tier, we simulate the performance characteristics of 3DXP memory by default *i.e.* $3\times$ [53, 125] the DRAM latency (100 ns). In our experiment, the slower memory tier is provisioned by attaching a 128 GiB 3DXP DIMM to a server, except for

Section 4.3.5 where we explore the effectiveness of TMC in other memory tiering architectures. New allocated VMs are added to a queue in order of their arrival. Every time a new virtual machine is created, the scheduler checks all machines to find one with sufficient resources. TMC analyzes the machines in random order, and places the job on the first one that has the required available resources.

IaaS cloud. We assume that customers can choose a slow memory ratio out of 11 slow memory ratios (0%, 10%, 20%, ... , 100%) and an LLC size out of seven configurations (1, 2, 3, 4, 12, 20, 28). We assume that the cloud providers utilize a technology such as Intel's cache allocation technology (CAT) [48, 100] to assign ways to applications. In total, there exist 77 candidate configurations. We obtain the hourly cost for a single vCPU and 1 GB of DRAM by using the least square method to solve a system of equations derived from all VM instances in the Msv2-series of Microsoft Azure. We assume the cost of 3DXP memory is 0.4× that of DRAM. Following the methodology described in [130], we obtain the hourly cost for a unit of LLC capacity according to the estimated area percentage of the LLC in a CPU chip.

Workloads. The workloads used in our experiments cover a broad spectrum of applications (Table 4.3). We modify the applications to utilize our custom memory allocator that assigns each data structure with a tag. For each application, we execute 2 billion representative instructions. We pick four out of the six workloads to form a workload mix resulting in 15 workload mixes in total. A workload mix represents the workloads that are being submitted to the cluster at a given period.

Baselines. We compare TMC with the following strategies: i) Exhaustive search (ES) finds cost-optimal configurations by running all the configurations. It provides an upper bound

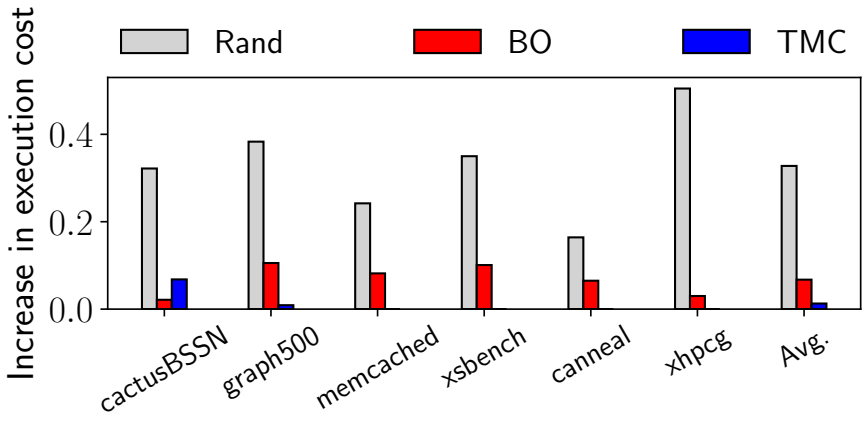


Figure 4.6: Execution cost increase over exhaustive search

on the overall performance. ii) Random (Rand) selects a configuration randomly from a set of candidate configurations without any test runs. iii) Bayesian Optimization (BO) is a state-of-the-art solution that has been used in prior work [7] to reduce the number of samples to reach a cost-optimal configuration. In our experiment, we use EI = 5% and three initial samples.

4.3.2 Execution and Search Cost

In this work we aim to learn cost-optimal tiered-memory configurations with minimal search overheads. Figure 4.6 shows the execution cost of the VM, LLC, fast and slow memory configuration determined by TMC and the baseline techniques. As can be seen, TMC reduces the execution cost by 1.3× over Rand and by 1.05× over BO in average. TMC only incurs a 2% higher TCO per performance than the ES's cost-optimal configuration. While ES provides the best performance in terms of cost minimization, it also incurs the highest search cost of all approaches as shown in Figure 4.7. In particular, TMC provides 26× lower search cost than ES and 3× lower search cost than BO. Rand imposes no search overhead by simply choosing a

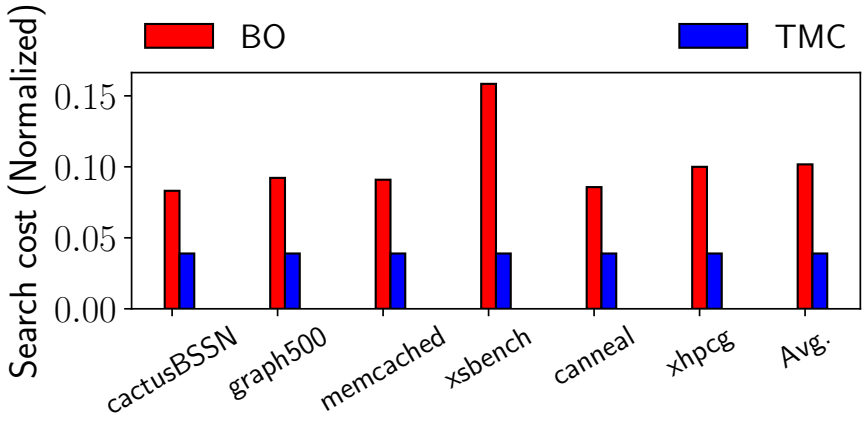
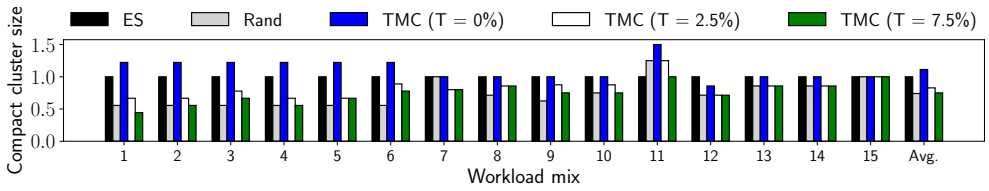


Figure 4.7: Search cost of TMC and BO, normalized to the search cost of exhaustive search (ES). ES and Rand are omitted as they are one and zero.

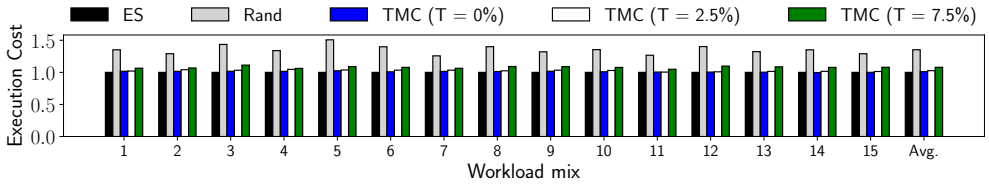
configuration randomly, however, it has no way of controlling the quality of the recommended configuration, and, as a result, it increases TCO by up to 50% and 33% on average compared to ES.

4.3.3 Improving Packing Efficiency

The second goal of TMC is to increase the packing efficiency without introducing a significant cost penalty for the customer. For the cost penalty, we set the threshold `T` to 2.5%, so that the recommended configuration can be at most 2.5% more costly than the optimal configuration. We use compact cluster size [111] as the metric to measure the packing efficiency of the cloud. Figure 4.8 compares the compact cluster size and the average running cost achieved by the evaluated schemes. Exhaustive Search recommends actual cost optimal configurations, but entails significant search overheads as seen in Section 4.3.2. As compared to ES, TMC reduces the compact cluster size by 17% and introduces only a 1.5% higher cost. In addition, we also observe that TMC can indeed control the quality of the recommended configurations and



(a) compact cluster size



(b) execution cost

Figure 4.8: Efficiency of TMC's configuration selection. TMC increases the efficiency while minimizing the cost penalty for the customer.

achieve a significantly lower execution cost compared to a randomly selected configuration.

4.3.4 Threshold Sensitivity Study

We next study the impact of the threshold `T`. A larger threshold `T` allows more configurations to be deemed cost-optimal, enabling TMC to further increase resource efficiency. For example, when the `T` increases from 0% to 2.5% (resp., 7.5%), TMC can boost resource efficiency by reducing the compact cluster size by 25% (resp., 32%). On the other hand, increasing `T` also lowers the standard of cost-optimal configurations, which in turn causes the execution cost of the recommended configurations to rise. For example, the average cost execution of the configurations recommended by the TMC is 1.1%, 2.6% and 7.9% higher than the optimal cost when the threshold `T` is 0%, 2.5% and 7.5% respectively.

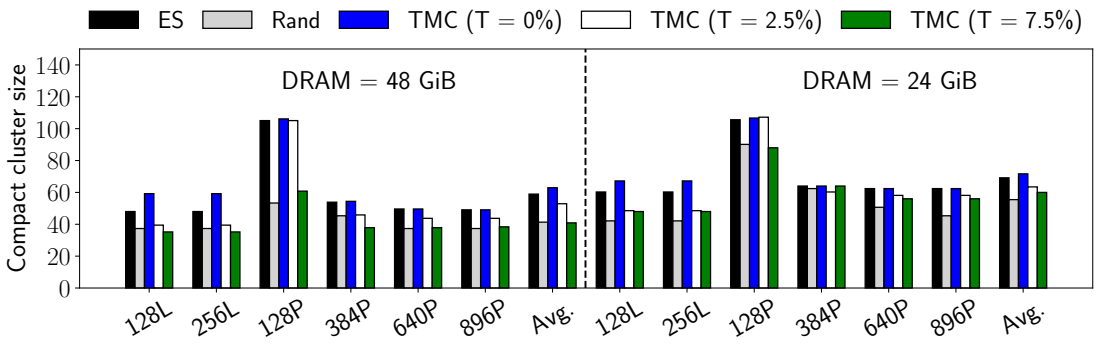


Figure 4.9: Effectiveness of TMC under various configurations of tiered memory.

4.3.5 Memory Tiering Sensitivity Analysis

This experiment investigates the effectiveness of TMC under various tiered memory architectures. The two *Local* configurations deploy slow memory locally via DIMMs: *128L* attaches one 128 GiB 3DXP module and *256L* attaches two 128 GiB modules to each node. The current 3DXP modules only come in capacities of 128GB, 256GB and 512GB. As a result, *Local* can only increase the memory capacity at a very coarse granularity. The four *Pool* configurations explore rack-scale pooled deployment enabled by the emerging CXL technology to connect slow memories to 8 nodes via a CXL fabric: *128P* / *384P* / *640P* / *896P* respectively provide 128 / 384 / 640 / 896 GiB of 3DXP memory in the pool. For each such *far memory* configuration, we evaluate two configurations of local DRAM with either 24 GiB or 48 GiB per machine.

We run the 15 workload mixes under various tiered memory configurations to observe the average compact cluster size of each configuration shown in Figure 4.9. First, we demonstrate that TMC improves resource efficiency in nearly all architectures as compared to other approaches. For example, TMC reduces the compact cluster size on average by 30%

(resp. 13%) as compared to ES when the DRAM size per machine is 48 GiB (resp. 24 GiB). Second, we observe that the *384P* and *640P* already allow TMC to achieve optimal resource efficiency when the size of the local DRAM is 24 GiB and 48 GiB respectively. *384P* and *640P* effectively provision 48 GiB and 80 GiB far memory per node. On the other hand, *Local* only allows the memory to be expanded at the coarser 128 GiB granularity, potentially leading to the over-provisioning and stranding of slow memory. Third, we show that TMC delivers smaller improvements in resource efficiency when managing CXL-pooled slow memory, compared to local-attached slow memory. For instance, TMC ($\mathtt{T}$ = 7.5%) delivers just 10% resource efficiency improvement under the *640P* compared to 28% under the *128L* with 24 GiB local DRAM. When DRAM is under high pressure, TMC improves resource efficiency by finding cost-optimal configurations with larger slow memory ratios. However, CXL dilates execution times due to its longer access latency, thus increasing the execution cost of VM configurations with larger slow memory ratios, making it harder for TMC to improve resource efficiency with alternative configurations. Finally, data center operators must comprehensively consider both platform cost and resource efficiency in order to find an optimal server configuration. We particularly note two interesting examples: i) Doubling the local DRAM from 24 GiB to 48 GiB reduces the compact cluster size for all approaches; However, it also introduces significantly higher per-machine costs, ii) *640P* reduces platform cost by requiring 37.5% less 3DXP memory as compared to *128L*; however, TMC ($\mathtt{T}$=7.5%) achieves 14% lower resource efficiency in a *640P*.

4.4 Real System Experiments

Simulation allows us to easily observe any application properties and thus enables us to quickly verify our proposed techniques on the performance estimation and the configuration selection. In this section, we introduce a proof-of-concept implementation and partly verify the applicability of using TMC in real systems. In particular, we build a working prototype that can predict how the run time changes with the size of the slow memory. As hardware monitors for learning the cache miss curve [94] become available, TMC can be completely implemented in software. In the following section, we consider the following three main questions:

- What is the overhead of PEBS sampling?
- Can we accurately estimate the access frequencies?
- Can we accurately estimate the performance impact?

We utilize precise event-based sampling (PEBS) to intercept samples of memory accesses to estimate the access frequency of data structures. PEBS captures a snapshot of the processor state upon certain configurable hardware events. We program PEBS to monitor `MEM_LOAD_RETIRED.L3_MISS` events. PEBS can be configured with a sampling interval (SI). For a sampling interval of n, PEBS captures every n^{th} event into a buffer. When the PEBS buffer is full, an interrupt is triggered, during which TMC records the CPU state in a software-accessible buffer. We only record the virtual address accessed by CPU misses. The sampled memory accesses are then written to a file in a separate thread. As discussed in Section 4.2.3, we count the sampled accesses to different data structures in the offline analysis. We multiply the number of sampled memory accesses by the sampling interval (n) to estimate the actual

access frequencies.

Due to the lack of general, well-documented performance counters that allow us to measure the MLP as well as the average memory latency in contemporary Intel CPUs, our current implementation measures MLP and the latency sensitivity of an application indirectly. In particular, we measure the amortized performance penalty introduced by accessing the slow memory. The amortized performance penalty takes into consideration both the latency-sensitivity as well as the effect of memory level parallelism on an application. We measure the CPI of an application in the DRAM-only configuration (CPI_{dram}), CPI in the configuration where all data is placed in the slow memory (CPI_{slow}), as well as the number of accesses to the slow memory (MPI_{slow}). The amortized performance penalty can then be computed via Eq. 4.6. The performance impact associated with placing data to the slow memory can be estimated by simply multiplying the access rate to the slow memory (estimated) and the amortized performance penalty.

$$Perf\ penalty = \frac{\Delta L}{MLP} = \frac{CPI_{slow} - CPI_{dram}}{MPI_{slow}} \tag{4.6}$$

4.4.1 Evaluation

Setup. To evaluate the proposed scheme, we use a server equipped with a Xeon Gold 5218 processor and a tiered memory hierarchy with six 32 GiB DRAM DIMMs (192 GiB in total) and a 128 GiB intel DC Persistent Memory.

Overhead. We first study the overhead introduced by PEBS which is important as it impacts the search cost of TMC. Figure 4.10 shows the sampling overhead for different SIs

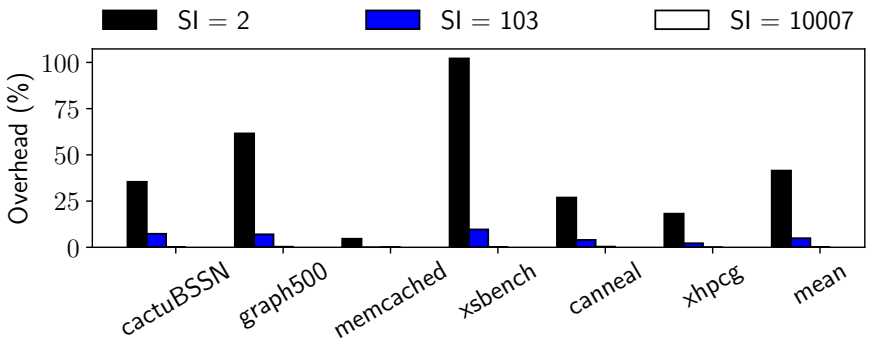
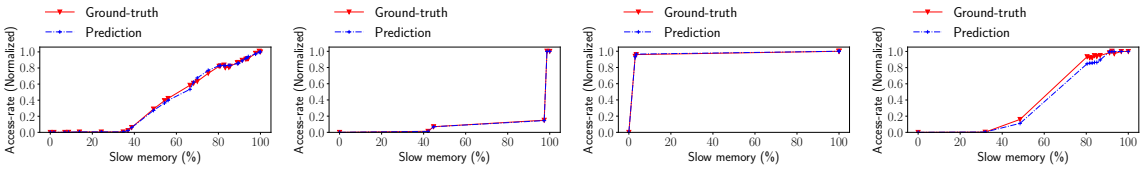


Figure 4.10: PEBS sampling overhead. We choose prime SIs to avoid bias from periodicities like prior work [77].

compared to the application execution time without PEBS monitoring. The PEBS sampling overhead is comprised of induced pipeline flushes due to the PEBS assist, and the overhead for handling extra interrupts [6]. High sampling rates results in substantial performance overhead. With a SI of two, the performance overhead can be as high as 102.1% (41.4% on average). We configure PEBS to use a large sampling interval (10007). With such a large SI, we observe virtually no overhead ($< 1\%$) due to PEBS sampling across all workloads.

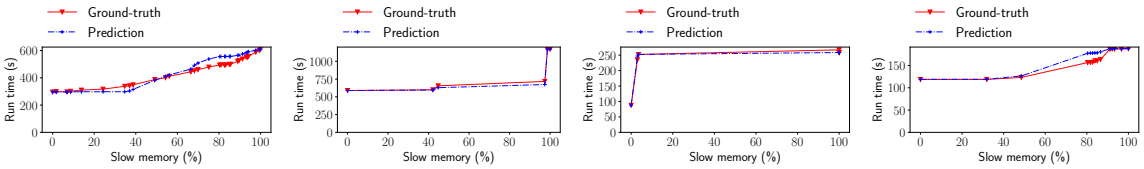
Access-rate estimation. In our experiment, we move data structures of an application to the slow memory tier one by one in a random order and then measure the ground-truth number of accesses to the slow memory and the ground-truth run-time. Figure 4.11 shows the number of accesses to slow memory depending on the amount of data allocated in slow memory. We show the ground-truth and estimated number of accesses to the slow memory. As we can see, we achieve a high accuracy in estimating the access frequencies even at a relatively low sampling rate. The inaccuracies in the workload *xhpcg* is likely caused by the shadow effect of PEBS [127].



(a) cactusBSSN (b) graph500 (c) xsbench (d) xhpcg

Figure 4.11: Accuracy of estimating the slow memory access frequency in a real system using profiling.

Performance estimation. Figure 4.12 shows the ground-truth and estimated run-time depending on the amount of application data allocated in slow memory. Overall, TMC achieves high accuracy in estimating the amortized performance penalty of different applications and, as a result, is capable of producing accurate predictions on the execution time of the application. This observation demonstrates that our assumption on the MLP and the memory latency as stated in Section 4.2.2 holds in most of the cases. However, we also observe that there is a relatively high prediction error (7%) on the performance although TMC achieves high accuracy in estimating the access rate for the *cactusBSSN* workload as shown in Figure 4.11a. The *cactusBSSN* workload presents an interesting example where the assumption that MLP / memory latency is identical across different memory configurations might not hold. However, we argue that adding new hardware [74] or using costly instrumentation [36] (40× slowdown) to capture the data structure specific MLP just for improving the accuracy for a few applications is not justified.



(a) cactusBSSN (b) graph500 (c) xsbench (d) xhpcg

Figure 4.12: Accuracy of estimating the execution performance in a real system using profiling.

4.5 Chapter Summary

This work investigates how to quickly identify ideal memory configurations for applications in tiered-memory cloud systems. TMC captures application-specific properties with existing performance monitoring hardware and uses them for accurate performance prediction. We demonstrate that TMC reduces the search cost by up to 4× while recommending high-quality configurations. Our approach additionally improves resource efficiency by 17% on average versus a naive policy that requests optimal allocations for each application in isolation. As a result, TMC provides the tools to efficiently support emerging tiered memory systems and to reap both performance and cost benefits.

Chapter 5

Performance optimized indexing for 3DXP memories

Prior research on PMEM based data structures has leveraged simulation or emulation methodologies, ignoring the intricate details and performance pathologies of persistent memories such as Intel's 3DXP. This work focuses on the B+-tree and its variants, exploring the common design issues of sorted index structures in a real system utilizing persistent memory. We demonstrate how to achieve high performance on a real-world persistent memory platform, combining several different techniques and providing an in-depth analysis of their micro architectural performance characteristics. Furthermore, we present two novel techniques *group flushing* and *log structuring* for PMEM.

5.1 Experimental Setup

We now describe our methodology to evaluate the performance characteristics of different B+-Tree implementations on 3DXP.

5.1.1 Methodology

In this work, we compare six B+-Tree solutions: 1) *btree*: A conventional in-memory B+-Tree with sorted keys in each node; 2) *unsorted leaf*: A write-optimized B+-Tree with unsorted leaf nodes and sorted keys in internal nodes; 3) *btree-WAL:* A B+-Tree utilizing WAL for consistency; 4) *FAST/FAIR:* A B+-Tree with FAST/FAIR [47] for consistency (details on FAST/FAIR can be found in Section 2.4); 5) *persistent unsorted:* A B+-Tree with *unsorted leafs* utilizing native atomic in-place updates to ensure the consistency of leaf updates and WAL for supporting the rare case of structural modifications; 6) *FAST/FAIR SP*: A FAST/FAIR B+-Tree that employs selective persistence. All of these six solutions employ the same fine-grained, optimistic concurrency control mechanism [18] to enable multi-core scalability. Note that we employ linear search for all our implementations as prior work [47] has shown that binary search performs worse when the node size is smaller than 4 KB due to branch mispredictions.

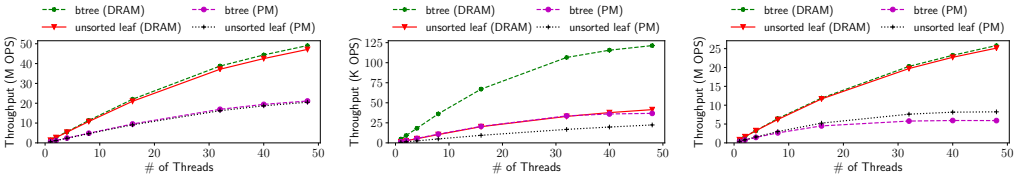
We profile the executions of the different B+-Tree implementations using multiple hardware performance counters, including the total number of instructions, instruction per cycles (IPC), cache miss stalls and resource related stalls, to interpret the results. The performance counters are obtained via `perf` [33]. The resource related stalls include stalls caused by the limited size of hardware resources such as the load/store queue as well as stalls induced by the

execution of memory fences. We also collect the performance counters from the 3DXP, including the amount data that is read and written from and to the 3DXP controller as well as the amount of data that is read and written from and to the 3DXP storage media.

5.1.2 Experimental Environments

All of our experiments are conducted on a 2-socket, 56-core machine with 32KB/1024KB/38MB L1/L2/L3 Caches. The 12 memory channels (2 sockets $\times$ 6 channels/socket) are fully populated using DRAM and 3DXP modules. In particular, we deploy 96GB of DRAM (6 $\times$ 16 GB/DIMM) and 1.5TB of 3DXP (6 $\times$ 256 GB/DIMM). Note that the per-socket channels are evenly split between DRAM and 3DXP. We use the ext4-DAX file system on the Fedora distribution (kernel 5.0.9). All our experiments are executed on a single socket by pinning threads and restricting memory allocation to the same NUMA node. Our code is written in C/C++ and compiled with `clang 8.0.0` with `-O3` flag.

We design four micro-benchmarks for evaluation. Unless otherwise stated, the system is warmed up by loading a tree with 160 million random entries and then one of the four operations is performed: *Fill Random* randomly inserts 80 million additional key/value pairs; *Read Random* retrieves 320 million random keys; *Range Query* performs range query requests with a selection ratio of 0.001%; *Read/Write* simulates a mixed read-write workload. Both keys and values in these workloads are 16 bytes in size. We perform an exhaustive search of the B+-Tree node size as one of the experiments. For all other experiments we utilize a node size of 512 bytes which provides good performance in all configurations (Section 5.3.2).



(a) *Read Random* (b) *Range Query* (c) *Fill Random*

Figure 5.1: Throughput under three different workloads (higher is better)

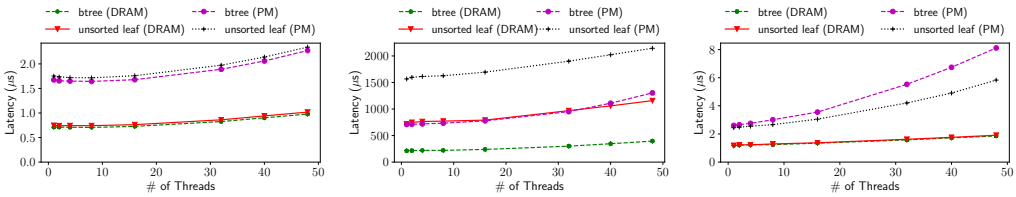
5.2 PMEM Indexing Techniques

In this section, we conduct experiments to measure the efficiency of PMEM techniques for addressing the read-write asymmetry and for providing storage consistency and durability. Furthermore, we analyze how B+-Trees can leverage a combination of DRAM and PMEM devices to optimize performance.

5.2.1 Write-optimized Indexing Structures

In the first experiment we analyze the benefit of write optimized data structures. As described in Section 2.4 the *unsorted leaf* index reduces writes while increasing computational complexity by maintaining the bitmaps. We compare *unsorted leaf* against a conventional *btree* baseline showing the achieved throughput in Figure 5.1 and latency in Figure 5.2 for three different workloads. We compare the in-DRAM Performance (prefixed by DRAM) and in-PMEM Performance (prefixed by PMEM).

Unsorted leaf reduces costly PMEM writes by avoiding sorting the entries, and thus can improve the update performance by up to 1.40× in the 3DXP (Figure 5.1c and Figure 5.2c). In DRAM, reads and writes have almost the same cost, so *unsorted leaf* only improves the



(a) *Read Random* (b) *Range Query* (c) *Fill Random*

Figure 5.2: Latency under three different workloads (lower is better)

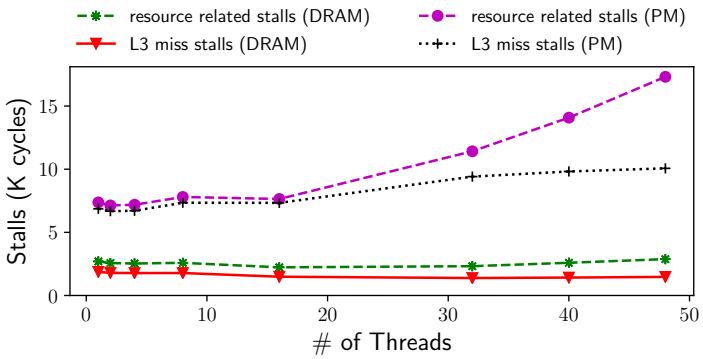


Figure 5.3: Profiling of *btree* under the *Fill Random*

update performance by 1.06×. The impact of searching values within unsorted nodes requires that all valid slots must be checked. In contrast, on average, only half of the entries need to be checked in a sorted node for an existing key. However, as *unsorted leaf* only suffers from additional overhead while accessing leaf nodes, the read performance of *unsorted leaf* is comparable to that of the *btree* (Figure 5.1a and Figure 5.2a). As the *unsorted leaf* incurs significant instruction overhead to sort the keys in a range query, it exhibits up to 3× performance degradation under the DRAM and 2× degradation under 3DXP (Figure 5.1a and Figure 5.2b). *Unsorted leaf* suffers from a smaller performance hit when performing a range query in PMEM as the software overhead becomes less significant compared to the cost of device accesses.

The read performance of in-PMEM indexes is lower than that of the in-DRAM in-

dexes by a constant ratio (Figure 5.1a and Figure 5.2a). For write workloads, the slowdown for *btree* ranges from 2.2× to 4.4× whereas the slowdown for *unsorted leaf* only ranges from 2.0× to 3.0× (Figure 5.1c and Figure 5.2c) as the number of threads increases from 1 to 48. This observation is in line with 3D XPoint's asymmetric read/write performance. When the load of the system is low (1 thread), the performance gap between PMEM and DRAM is between 2–3× as the node traversal dominates the total runtime with memory writes being mostly performed out of the critical path. As the number of execution threads increases, the 3DXP approaches its maximum I/O capacity. The impact on performance is twofold. First, increasing the PMEM operations creates back pressure, eventually filling up the hardware resources (reorder buffer, store queue) and stalling the processor pipeline. Second, the amortized penalty of L3 cache misses increases as it takes more time to process the read requests in a fully loaded 3DXP. Figure 5.3 shows the performance profile for *btree* on both DRAM and PMEM. Compared to the in-DRAM *btree*, the in-PMEM one experiences a drastic increase in L3 miss stalls and, furthermore, resource related stalls increase as the number of execution threads scales beyond 16, indicating that the bandwidth of PMEM is indeed becoming a bottleneck.

Finding 1: The gap between the insertion performance of in-DRAM indexes and that of the in-PMEM indexes widens as the number of execution threads increases.

Implication: The PMEM bandwidth can be a limiting factor for indexing structures to handle requests at large scale. First, the indexing structures should be designed to carefully avoid wasting PMEM bandwidth. Second, more hardware resources should be added to the system to mitigate the back pressure from the PMEM.

Our evaluation shows that the actual improvement of *unsorted leaf* on insertion performance is relatively small (e.g. 6%) in a single-thread workload, which is significantly less

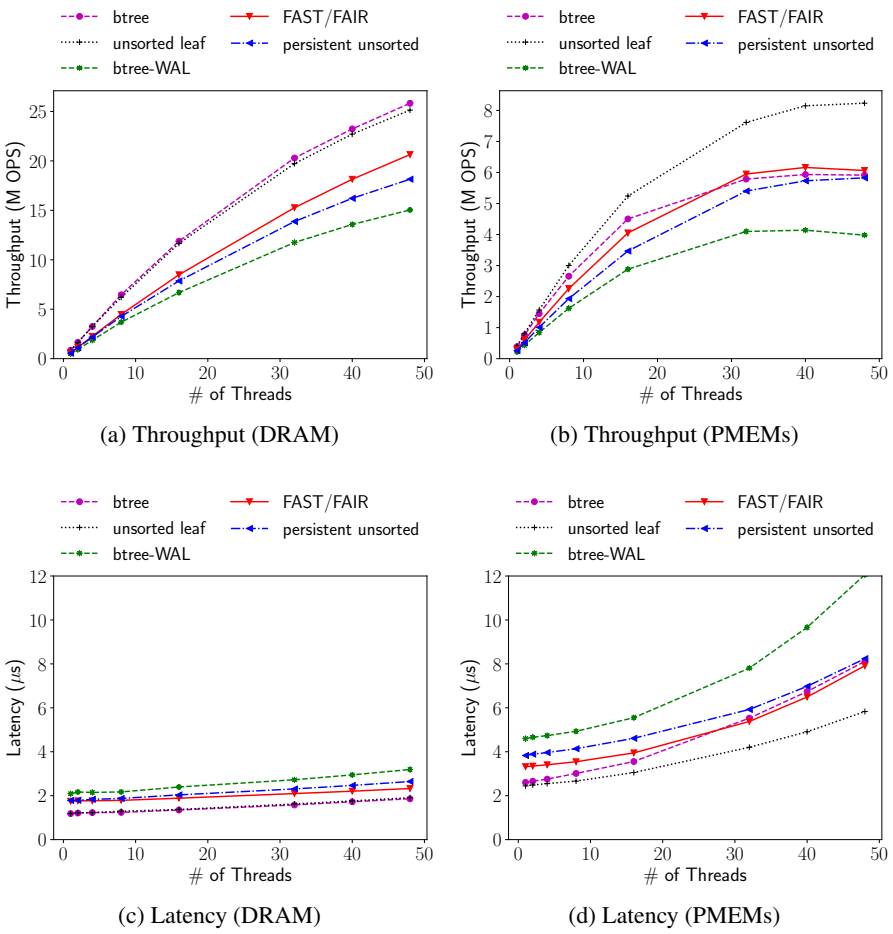
than what prior work [21], utilizing a simulation based methodology, suggested. Actual PMEM devices are better at hiding high cost of writes than expected. In a real-world system, PMEM writes can be queued at multiple layers (store buffer, cache, iMC, 3DXP controller) and slowly drained to PMEM. Finding 1 indicates that the 3DXP becomes bandwidth-bound as the number of execution threads increases beyond 16. As we can see, the impact of write reduction in the *unsorted leaf* on performance increases with I/O pressure on the 3DXP.

Finding 2: The real-world efficiency of write-optimized indexing structures varies significantly based on the intensity of the workloads.
Implication: To enable write-optimized indexing structures B+-trees have to be restructured, increasing the search overhead. As a result, for workloads with a low insertion rate conventional B+-trees are preferable over write-optimized implementations in PMEMs.

5.2.2 Storage Consistency

Figure 5.4 compares the insertion performance of persistent B-Trees with various consistency mechanisms. The persistent trees (*btree-WAL*, *FAST/FAIR* and *persistent unsorted*) pay extra costs to ensure storage consistency. To evaluate the persistence overhead, we compare them with their volatile counterparts (*e.g. btree* and *unsorted*).

Several general observations are made. First, as expected, the consistency mechanisms introduce runtime overheads, in particular, the insertion latencies of *btree-WAL* and *persistent unsorted* are up to 1.77× and 1.56× higher than the latency of the volatile B+-Tree implementations (Figure 5.4d). The persistence overhead includes i) execution of additional instructions to implement WAL and ii) introduction of execution stalls caused by memory barriers. Table 5.1 shows a detailed performance profile. It confirms that *btree-WAL*, *FAST/FAIR* and



(a) Throughput (DRAM)

(b) Throughput (PMEMs)

(c) Latency (DRAM)

(d) Latency (PMEMs)

Figure 5.4: The cost of persistence

persistent unsorted indeed increase the number of instructions and experience more resource related stalls. Second, *btree-WAL* and *persistent unsorted* cannot achieve comparable throughput to their volatile counterparts (*e.g. btree* and *unsorted*) in the PMEM (Figure 5.4b). This is because the *persistent unsorted* and *btree-WAL* rely on write-ahead logging and, therefore, incur extra PMEM write traffic. In contrast, the WAL-free *FAST/FAIR* achieves comparable throughput as *btree*. Third, *btree-WAL* exhibits the lowest performance among the three persistent trees as employing write-ahead logging for the sorted node entails considerably higher overheads.

	CPU				Memory			
	Instruction	IPC	Resource Stalls	Per. Instr.	Read	Write	RA	WA
btree	2107	0.22	7740	0/0	1626	400	1.6	3.2
unsorted	2381	0.26	6532	0/0	1651	236	1.5	3.5
btree-WAL	4620	0.26	11240	12/4	2001	937	1.6	1.8
FAST/FAIR	2548	0.19	9982	6/10	2119	474	1.5	2.2
persistent unsorted	3137	0.22	9244	5/5	1973	411	1.4	2.6

Table 5.1: Profiling of the evaluated designs under the *FillRandom* workload (PMEM, 1 thread). It shows Instructions Per Cycles (IPC), the number of instructions, resource related stalls and persistence instructions (`clwb`/`sfence`), read traffic to PMEM (bytes), write traffic to PMEM (bytes), Read Amplification and Write Amplification per operation

Compared to the latency in DRAM (Figure 5.4c), the latency of the persistent trees in PMEM increases drastically as the number of the execution threads increases (Figure 5.4d). In particular, the latency of *FAST/FAIR* increases by 2.40× in PMEM but only 1.35× when residing in DRAM. As we can see in Table 5.1, the 3DXP-level write-amplification introduced by the persistent trees remains high, indicating the full I/O capacity is still underutilized. Note that the write-amplification of the *btree-WAL* is relatively low. This is because it generates redundant writes written to the in-PMEM log sequentially.

Finding 3: The restricted I/O bandwidth of 3DXP limits the insertion rate of the persistent trees while due to random access patterns the PMEM bandwidth cannot be not fully utilized.

Implication: In conventional B+-Trees, random updates are unavoidable increasing the write-amplification for PMEMs. Given the constrained I/O capacity of PMEMs, the insertion performance can be further improved by reshaping the I/O pattern of the B+-Tree via log-structuring.

Despite the common belief that persistence introduces performance overheads Figure 5.4d shows an interesting counterexample. For the *Fill Random* workload, *FAST/FAIR* delivers 3% higher throughput than the volatile *btree* on 3DXP, when the number of execution threads is 48. This performance increase is provided by the explicit cache-line flushing mecha-

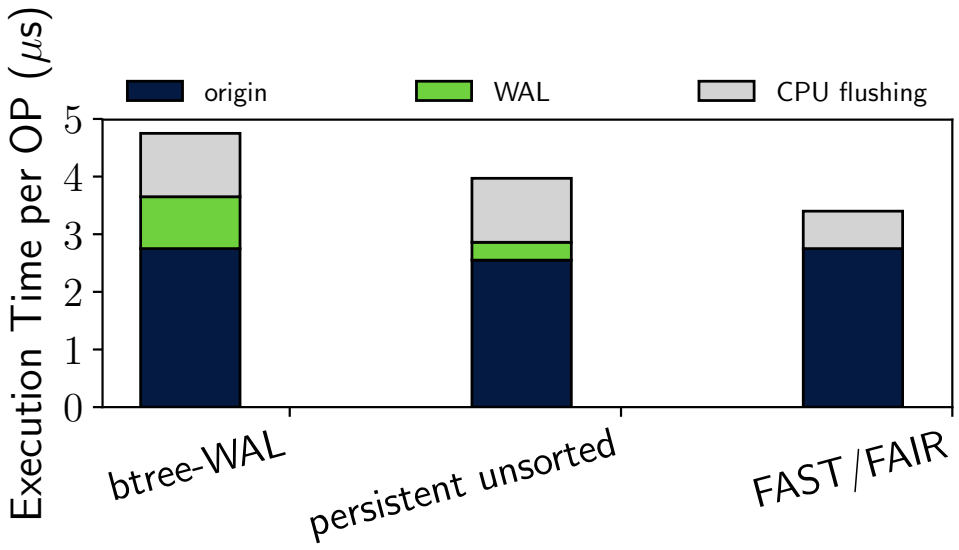


Figure 5.5: Persistence cost decomposition (4 threads)

nism required for ensuring consistency which, as a side-effect, forces cache-lines to be written back to the 3DXP controller sequentially. In contrast, the timing of cache-lines being written back in *btree* is implicitly controlled by the CPU cache. This observation has inspired us to develop a new optimization described in Section 5.4.2.

Finding 4: With conventional B-Trees, sequentially modified cache lines are often written back out of order by the cache controller, resulting in sub-optimal bandwidth utilization in PMEM.

Implication: We can improve the performance of the conventional indexes by preserving the spatial locality of updates for the 3DXP controller.

Figure 5.5 breaks down the runtime overhead: For each implementation, we list the execution time induced by the *btree* implementation itself as well as the time induced by WAL, memory fences and cache-line flushes. We observe the CPU flushing overhead of *btree-WAL* is actually low (Figure 5.5) and is close to that of *persistent unsorted* although it requires 2× more cache-line flush instructions (Table 5.1). This is because the frequency of memory barri-

ers is lower in WAL (1 memory barrier for every 3 cache-line flushes) which allows multiple cache flush operations to proceed in parallel. As a result, the software overhead of WAL of 0.9 µs almost outweighs the overhead for persisting data of 1.1 µs in *btree-WAL*. In contrast, the software overhead of WAL in the case of *persistent unsorted* is minimal (0.3 µs) as it is only required to implement rarely occurring structural modifications of the tree.

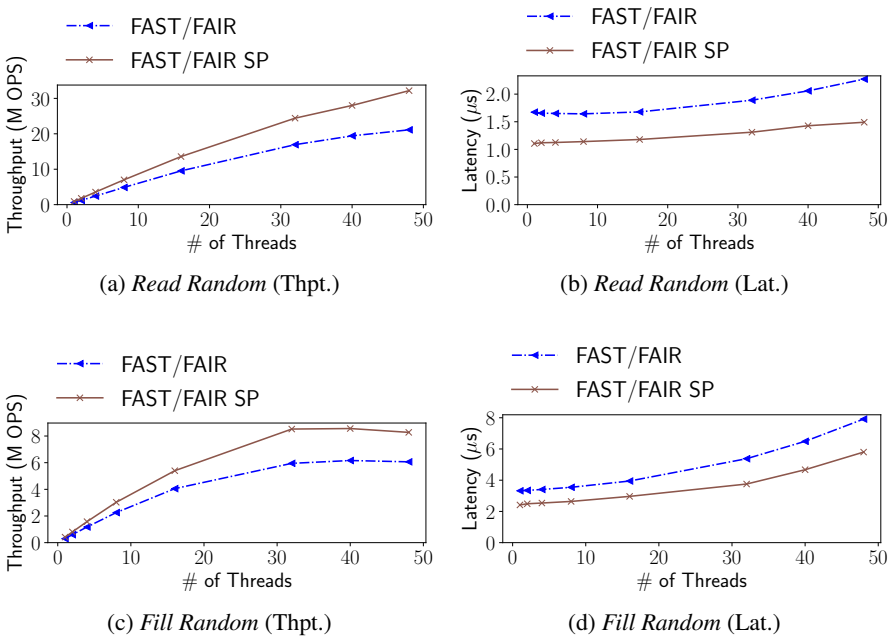
Finding 5: WAL provides a straightforward approach to ensure the consistency for complicated operations, however, incurs significant software overhead.

As shown in Table 5.1, *FAST/FAIR* requires more flush operations than the *persistent unsorted* B+-Tree. Intuitively, *FAST/FAIR* incurs more CPU flushing overheads and exhibits higher latency than the *persistent unsorted* implementation in the latency-bound workloads when the number of the execution threads is small. Figure 5.5, however, shows the opposite trend: The CPU flushing overhead of *FAST/FAIR* is 1.7× lower than that of the *persistent unsorted* tree. In contrast, the latency of *FAST/FAIR* and that of *persistent unsorted* is comparable when placed in DRAM under the same workload. This is because *FAST/FAIR* flushes cache lines continuously inducing smaller overheads for 3DXP.

Finding 6: *FAST/FAIR* in the 3DXP benefits from flushing cache-line continuously, reducing latency when the load of the system is low. When utilizing PMEM, the latency is only to 1.27× higher than that of the *btree*.

5.2.3 Selective Persistence

We examine the efficiency of the selective persistence technique described in Section 2.4. We store the internal nodes in fast DRAM as they can be rebuilt from the linked leaf nodes. First, as frequently accessed index nodes are placed in DRAM, the search performance greatly benefits from selective persistence. As shown in Figure 5.6a and Figure 5.6b, storing



(a) *Read Random* (Thpt.)

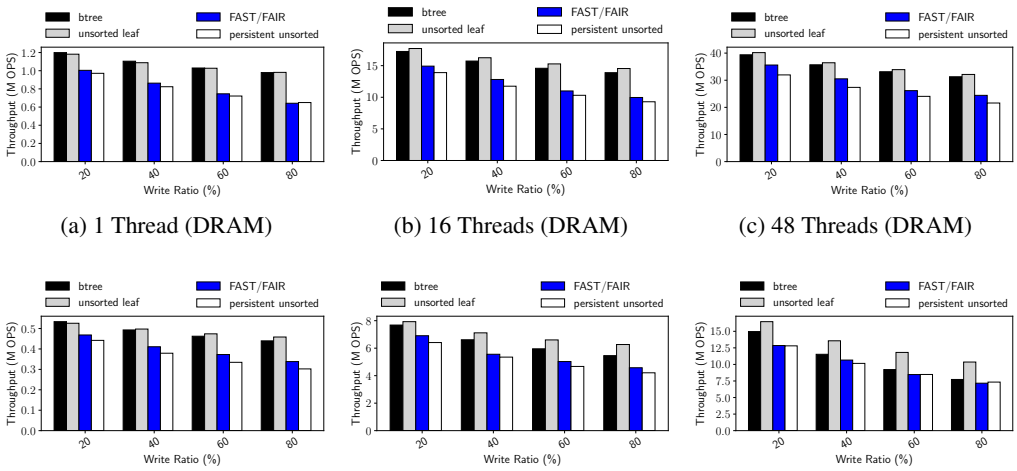
(b) *Read Random* (Lat.)

(c) *Fill Random* (Thpt.)

(d) *Fill Random* (Lat.)

Figure 5.6: The efficiency of selective persistence

index nodes in DRAM indeed improves the read performance by about 1.5× closing the gap between in-DRAM indexes and in-PMEM indexes. Second, the selective persistence also improves the update performance as it i) boosts the process of tree traversal and ii) avoids the PMEM writes to update the internal nodes stored in DRAM. As shown in Figure 5.6c and Figure 5.6d, the insertion performance is improved by up to 1.4× using the selective persistence technique. One drawback of the selective persistence approach is that it increases the recovery time. We have measured the time that it takes to rebuild the index nodes: as we increase the number of inserted entries from 10^4 to 10^8, the recovery time increases from 10 milliseconds to 10 seconds almost linearly.



(a) 1 Thread (DRAM) (b) 16 Threads (DRAM) (c) 48 Threads (DRAM)

(d) 1 Thread (PMEM) (e) 16 Threads (PMEM) (f) 48 Threads (PMEM)

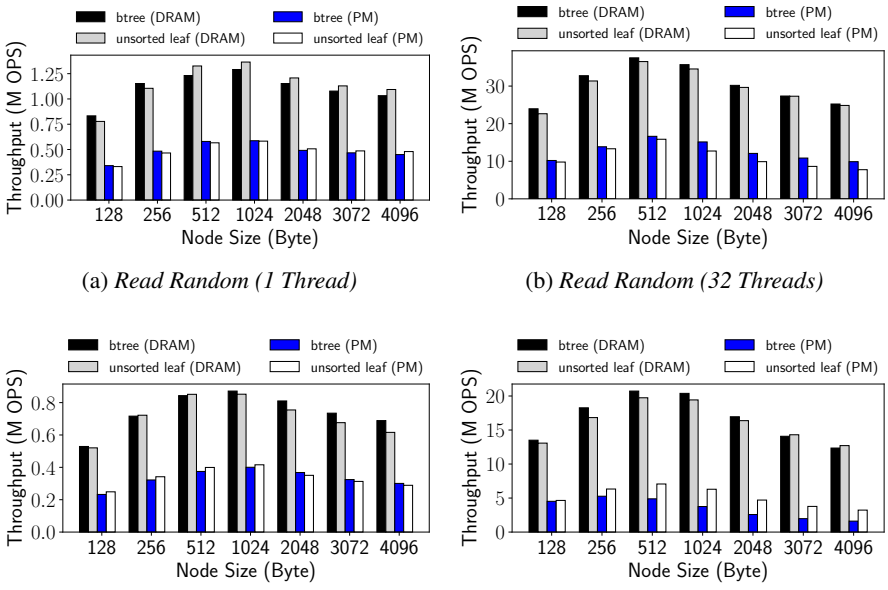
Figure 5.7: The performance of mixed read-write workloads

5.3 Workload Performance

In this section, we extend our experiments to more diverse workloads and configurations.

5.3.1 Mixed Workloads

Figure 5.7 shows the index performance under workloads with varying ratios of PUT and GET operations and different number of execution threads. The first row in Figure 5.7 shows the performance of different index structures in DRAM and the second row shows the performance in PMEM. Two observations are made. First, the ratio of PUT/GET operations has a greater impact on the performance of the evaluated index structures in the 3DXP platform. In particular, the throughput of the *btree* degrades by 2.3× in the PMEM and only by 1.3× in the DRAM as the write ratio increases from 20% to 80% with 48 execution threads.



(a) *Read Random (1 Thread)* (b) *Read Random (32 Threads)*

(c) *Fill Random (1 Thread)* (d) *Fill Random (32 Threads)*

Figure 5.8: Performance sensitivity to the size of the node

This observation further demonstrates the read/write asymmetry issue of the 3DXP. Second, in-memory indexes do not benefit much from the improved write efficiency of unsorted B-Trees when dealing with read-intensive workloads or when the load of the system is low. For instance, in a workload with 16 execution threads and 40% PUT ratio, the *unsorted leaf* implementation only achieves a 2% higher throughput.

5.3.2 Sensitivity to the Node Size

Figure 5.8 shows the effect of the node size on performance. We make two observations. First, the performance of all evaluated B-Tree indexes peaks at the node size of around 512 Bytes in both *Fill Random* and *Read Random* benchmarks. Figure 5.9 shows the profiling of *btree* under the *Read Random* benchmark. As we can see, when the size of the node

increases from 128 bytes to 512 bytes, the L3 miss penalty (L3-miss stall cycles) is significantly reduced, thus increasing overall performance. This can be explained by the improved efficiency of the hardware prefetcher for large nodes spanning consecutive cache lines. Note that the total amount of data that needs to be read from DRAM on a traversal is similar as trees with smaller nodes contain more levels. When the node size increases beyond 512 bytes, the L3 miss penalty no longer decreases, however, the key-scanning overhead continues to increase, as larger nodes need to be searched, leading to performance degradation. Second, we find the insertion performance of in-PMEM *btree* and in-DRAM *btree* peaks at different node sizes when the number of update threads is 32 as shown in Figure 5.8d. In particular, when the size of the node increases from 256 bytes to 512 bytes, the performance of the in-DRAM *btree* increases by 1.13× whereas that of the in-PMEM *btree* degrades by 1.08×. This is because in a write intensive workload, the overhead of writing more data in PMEM outweighs the reduced miss penalty when the node size is increased to 512 bytes.

Finding 7: In order to maximize the performance, parameters such as node size for in-DRAM indexes need to be re-tuned for in-PMEM indexes.

5.4 Optimizations

In this section, we conduct three studies to show the potential optimizations enabled by our findings. We focus on showcasing the key ideas and their impact on performance.

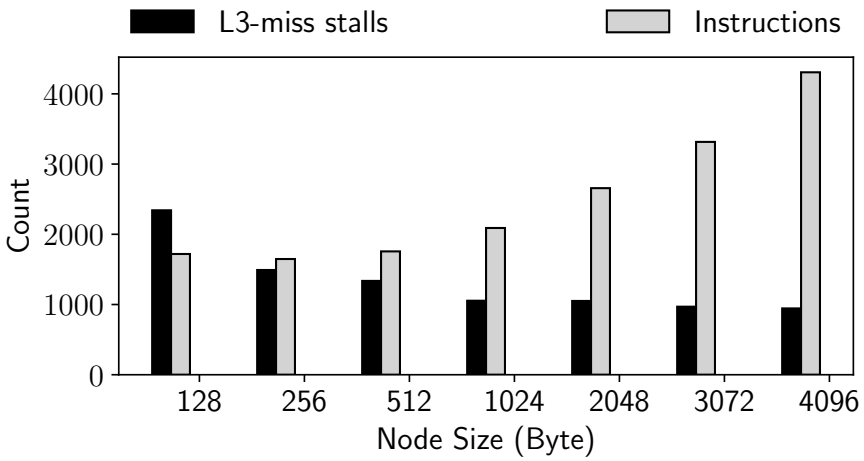


Figure 5.9: Profiling for *btree* under the *Read Random* application

5.4.1 Interleaving Operations

Major in-memory B+-tree operations such as query, insertion, and removal are implemented by traversing the tree from the root to leafs, resulting in the *pointer chasing* problem: a node cannot be accessed before the previous node's pointer is resolved. If the accessed node is not already in the CPU cache, the CPU stalls waiting for data from the main memory. As shown in Table 2.1, 3DXP's longer read latency compared to DRAM exacerbates the pointer chasing issue. There are several techniques proposed to hide memory latencies [20, 55, 93] of B+-tree variants. One effective approach is to group multiple operations against a data structure and then issue them as a batch, thus increasing memory level parallelism [55, 93]. In this section, we examine the efficiency of interleaving the execution of multiple GET requests. The multi-GET performance of the *btree* with varying node size and batch size is shown in Figure 5.10, where throughput is computed as batch size times the number of multi-GET operations / second. The efficiency of interleaving decreases as the node size grows. For instance, with DRAM,

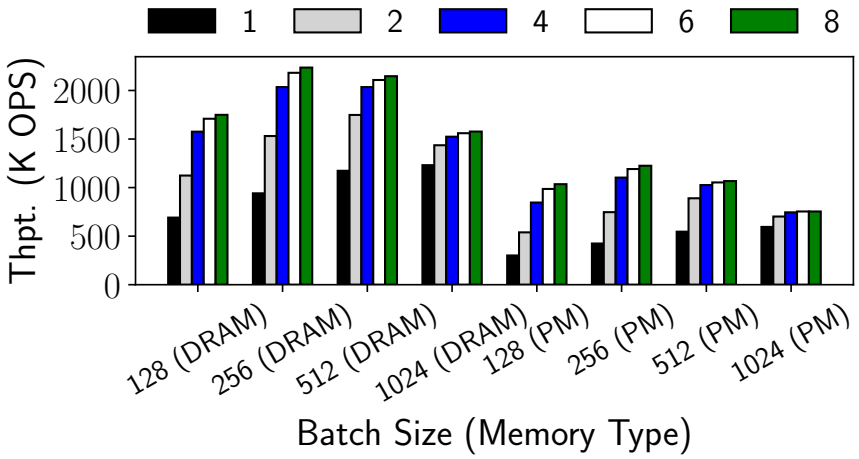
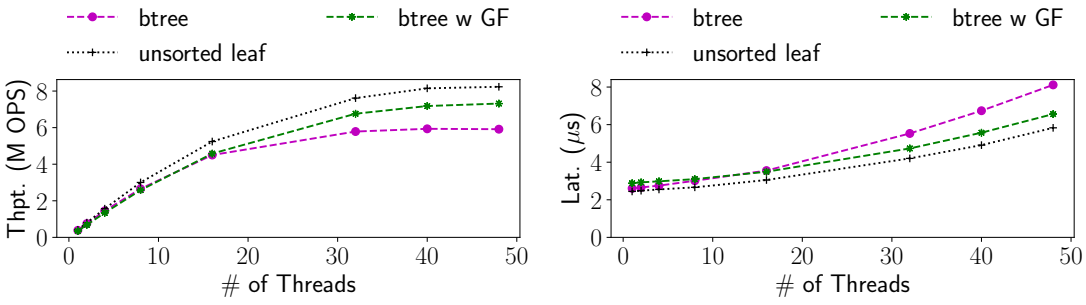


Figure 5.10: Interleaving efficiency

when the batch size is increased from 1 to 4, the GET throughput increases by 2.1× with 256-byte nodes but only by 1.2× with 1024-byte nodes. This is due to the fact that modern CPU cores have a limited number (10 20) of line fill buffers (LSB) to support memory parallelism. With a smaller node size, fewer cache lines need to be prefetched at once for a single GET operation, hence more GET operations can be performed in parallel. Due to larger access latencies in-PMEM B+-tree benefits more from interleaving. For instance, with 128-byte nodes the multi-GET throughput of the in-PMEM *btree* increases by 3.4× compared to the conventional implementation, whereas with the same configuration the in-DRAM *btree* only increases performance by 2.5×.

Lesson 1: In-PMEM indexes benefit more from interleaving compared to its in-DRAM counterparts as long as adequate hardware resources (LSBs) are provided. System designers should consider increasing hardware resources on future CPUs to enable higher memory parallelism.



(a) Throughput (b) Latency

Figure 5.11: Benefits of preserving sequentiality in software.

5.4.2 Group Flushing

As described in **Finding 3**, when PMEM is used as memory and no explicit cache-line flush is used to ensure persistence, the order of the modified cache lines being written back to the PMEM controller is implicitly decided by the cache replacement algorithm. Since the data in the CPU cache is managed at the cache line granularity (typically 64 bytes), sequential updates on the software level may be translated into small random accesses by the PMEM controller, resulting in sub-optimal bandwidth utilization. Intuitively, the leaf and internal node updates in the *b-tree* largely consist of sequential accesses and should induce low write amplification within the device. However, Table 5.1 shows that the device write amplification for the *btree* with the *Fill Random* workload is close to 4, indicating little sequentiality is preserved when accessing the PMEM.

The most efficient solution to this problem is to match the unit of the cache replacement policy to the access granularity of the PMEM controller. However, this would require significant modifications to the CPU architecture. We investigate the potential improvement of preserving sequentiality in PMEM via a simple software-level solution—*group flushing*. The

key idea of *group flushing* is to explicitly flush modified cache lines in contiguous groups via cache-line flush instructions. We first identify places in B+-Tree where updates are likely to be performed in large blocks such as node updates and node initializations. We then emit `clwb` instructions following these updates. Note that the number of lines affected by an update is determined dynamically at runtime and, hence, additional logic is required to avoid flushing when an update does not span more than one cache line.

We denote a B+Tree optimized with *group flushing* as *btree w GF*. Figure 5.11 compares the *btree w GF* to the original B+Tree (*btree*) as well as the unsorted B+Tree (*unsorted leaf*). As we can see, *btree w GF* achieves 24% higher insertion throughput compared to the *btree*. The *unsorted leaf* still achieves a slightly higher update performance, however, it sacrifices range search performance by up to 50% as shown in Figure 5.2b and Figure 5.1b. We also observe that the latency of *btree w GF* is higher than that of the *btree* when the thread number is less than 16. This is because, in the case of a low system load, writes to PMEM do not represent the main performance issue and the group flushing technique incurs extra instruction overheads due to additional `clwb` instructions. With **Finding 2** showing that *btree* and *unsorted leaf* have comparable performance under low load, in practice *btree* can be used by only enabling *group flushing* when the system load is high.

Lesson 2: Prior research focuses on addressing the write performance bottleneck by reducing writes at the software level, for instance, by leaving nodes unsorted. *Group flushing* provides a new direction by improving the I/O efficiency of PMEM by addressing the granularity disparity between the CPU and the PMEM. We envision *group flushing* to encourage hardware designers investigating PMEM-aware CPU cache replacement policies to further avoid the software overheads.

5.4.3 Log-structuring

Finding 3 and **Finding 6** motivate us to employ log-structuring for improving the update performance of persistent B+-Trees. As shown in Figure 5.12, random update operations on a conventional B+-Tree are translated to small random in-place writes in the PMEM address space. Due to the access disparity between CPU caches and the PMEM (64 vs 256 bytes), this random write pattern results in a significant write amplification of 3.2× as shown in Table 5.1. To confirm this issue, Table 5.1 shows that the write amplification of the *unsorted btree* is almost 3, indicating that two thirds of the write bandwidth is wasted, due to random in-place updates in the PMEM address space.

By incorporating a log-structured layout [96, 98], random writes can be batched into large sequential runs to reduce write amplification and thus better utilize the limited PMEM write bandwidth. However, this approach requires an efficient way to locate data in the log space. Virtualized B-Trees [72, 120] decouple the physical representation of the tree node from the logical representation. An indirection layer maps a logical identifier of a tree node to a chain of delta records in the log space, each record in the chain representing an update to the

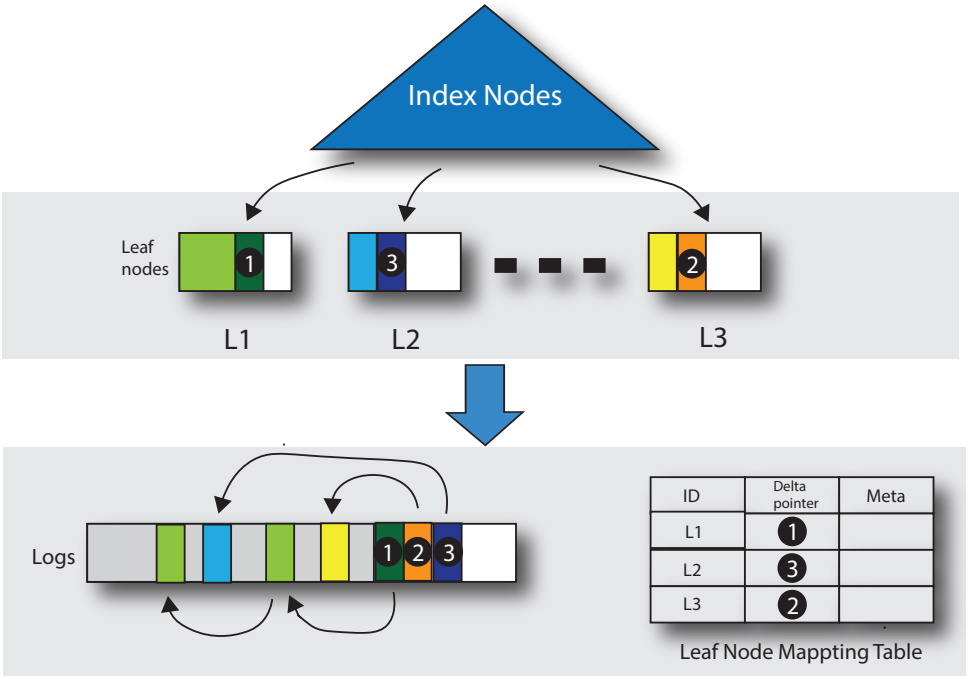


Figure 5.12: Design considerations.

corresponding node. The effectiveness of log structuring, to a large degree, depends on the efficiency of garbage collection (GC) and failure recovery.

5.4.3.1 Design Considerations

Employ indirection only for leaf nodes. The virtualized B+-Tree in prior works [72, 120] fully separates the physical representation from the logical view. However, Wang et al. [117] demonstrates that the indirection overhead of the BwTree [72] can be as high as 18%. We propose a novel hybrid layout: indirection is only used when accessing leaves that are write-heavy. This approach represents a good trade-off between read and write performance. Figure 5.12 shows the organization of our proposed tree where each leaf node is identified by its leaf node ID (LNID) and a leaf node mapping table is used to translate the LNID to the physical pointer to

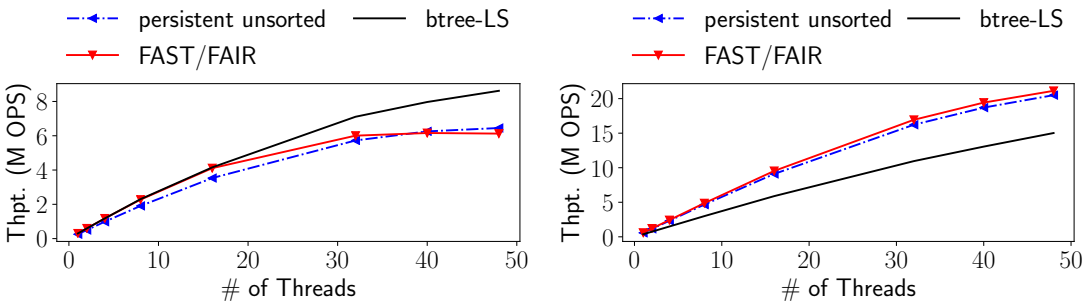
the head of the delta (update) chain. For reads, a list traversal is performed on the delta chain to identify the search key. For an update, a delta record is prepared, the delta record is flushed to the 3DXP and the newly prepared delta is prepended to the delta chain. If the number of the items in the delta chain exceeds the pre-defined node size, a split operation is performed. The valid data on the chain is then distributed into two new nodes of equal size.

Supporting multiple logs. A single log used by all threads is the most straightforward log space organization, however, this approach severely limits the concurrency of updates and recovery operations. At the other extreme, a per-node log in *persistent unsorted* trees avoids the scalability issues at the cost of random writes and higher write amplification. We choose the middle ground by adopting a multi-log layout where updates are dispatched to a circular buffer-backed log based on the hash of its LNID. When the garbage of a log exceeds a pre-determined threshold, a GC task is initiated by one of the multiple GC threads to start at the head of each circular buffer copying the valid data to its tail. The validity of a delta record is determined by its availability in the mapping table. During GC, valid delta records of the same chain are consolidated by the garbage collector into a new record and the address is updated in the mapping table.

Avoiding random PMEM writes to the mapping table. Each node update requires modifying the corresponding chain head pointer in the leaf node mapping table, producing undesirable random writes. We recognize the fact that all the data required for the mapping table during recovery is always persisted in logs enabling us to keep the mapping table in DRAM and rebuild it upon recovery. When a failure is detected, multiple log replay threads are initiated each one assigned with a number of logs. The threads work in parallel by scanning each log in the

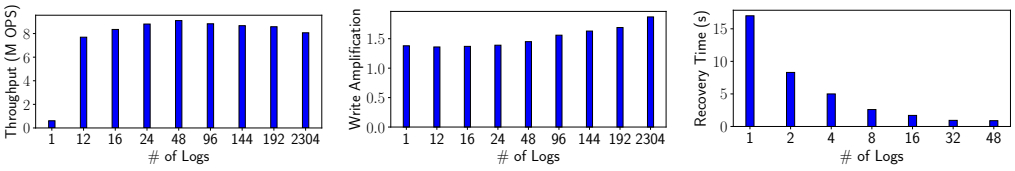
chronological order rebuilding the delta chains in the mapping table.

Hiding access latencies with prefetching. The delta chain reduces the spatial locality within a leaf nod degrading search performance. For instance, two consecutive items in a conventional B-Tree may be separated in the log space of a virtualized B+-Tree, resulting in an additional cache miss when accessed together. Prefetching is used to mitigate the performance degradation for reads. In particular, for each leaf node, we cache the pointers to the most recent delta records, and prefetch them whenever a chain traversal is initiated. The number of delta records cached, represents a trade-off between read performance and the space consumption of the mapping table.



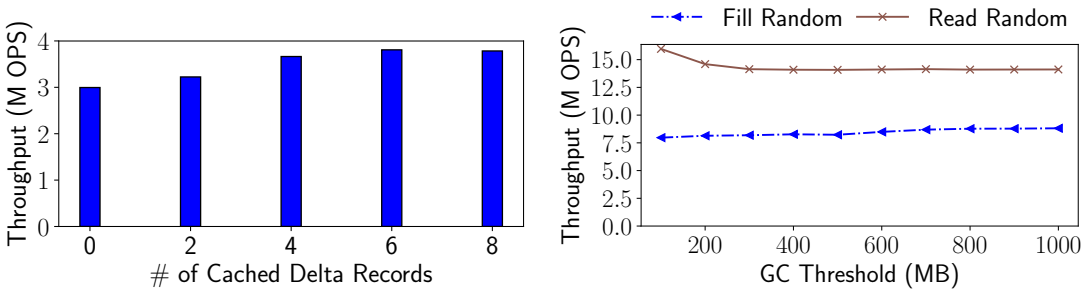
(a) Random Fill (b) Random Read

Figure 5.13: Performance comparison between *btree-LS*, *persistent unsorted* and *FAST/FAIR*.



(a) IOPS (b) WA (c) Recovery Time

Figure 5.14: The impact of the number of logs.



(a) Prefetching (8 threads)

(b) GC (48 threads)

Figure 5.15: The impact of GC and delta prefetching.

5.4.3.2 Evaluation

The results of the log-structured B+-tree implementation (denoted as *btree-LS*) are compared with those of *persistent unsorted* and *FAST/FAIR* in Figure 5.13, with GC disabled. One can see in Figure 5.13a that *btree-LS* achieves 41% higher insertion performance as compared to *FAST/FAIR*, from efficiently reducing the PMEM-level write amplification with a high system load. A 37% degradation in read performance is also observed (Figure 5.13b), mainly because of the reduced spatial locality in cache.

Figure 5.14 demonstrates the impact of log count selection by running *Fill Random* with 48 threads. Figure 5.14a shows that the update performance first increases drastically with the number of logs, as the contention for log updates is removed. After the number of logs goes beyond 48, the performance gradually drops as write sequentiality diminishes. Figure 5.14b confirms that write amplification inside the 3DXP increases from 1.4 to 1.7 as the number of logs increases. Figure 5.14c shows the recovery time of 160M entries by varying log count, with the replayer count equal to the log count and GC threshold set at 50MB. By effectively exploiting parallelism of the recovery process, with 48 replayer threads (and 48 logs) the whole

index can be recovered in 0.89 secs and the speedup over the single-thread implementation is 21×.

We show the impact of garbage collection and delta prefetching in Figure 5.15. Figure 5.15b shows the impact of garbage collection. The background GC workers attempt to keep the amount of garbage data within a tunable threshold. The update performance degrades by only up to 7% as the GC threshold decreases from 1000MB to 100MB. Meanwhile the read performance increases by 20%. This is because GC is also responsible for consolidating small delta records and thus a more aggressive GC setting better preserves spatial locality. Figure 5.15a shows the efficiency of delta prefetching. As the number of cached delta entries is increased from 1 to 8, the read performance is improved by 26%. Delta prefetching can barely improve performance when the number cached delta entries goes beyond 4. We suspect this is because the maximum level of memory parallelism supported by the hardware is reached.

Lesson 3: Log-structuring efficiently utilizes the limited write bandwidth by significantly reducing device write amplification, at the cost of search performance and additional implementation complexities. By placing the indirection layer in DRAM, exploiting parallelism within the log space and judiciously selecting software parameters such as the garbage threshold, we can make log-structuring practical for PMEM indexes.

5.5 Chapter Summary

This work work studies the common design issues of sorted index structures in a real system utilizing persistent memory. We present two novel techniques group flushing and log structuring for PM. Group flushing improves performance by 24% by addressing the granularity disparity between CPU caches and the DCPMM controller. In addition, our study revisits the

log structuring technique in the context of persistent memories improving performance by 41% as compared to state-of-the art persistent B+-Tree.

Chapter 6

Future Directions

System use of memory is changing as memory technologies advance rapidly and new interconnect standards emerge. We must also evolve our ideas to take advantage of future memories. The facts that compute express link (CXL) [4] enables a fast, coherent interconnect that speaks load/store memory semantics and that CXL can connect memory media with differing cost-per-GB and performance profiles are bound to have major repercussions on future system designs. In traditional architectures, the type of memory technology (*e.g.* DDR4), memory speed, and maximum memory capacity depend entirely on the CPU architecture. CXL will enable the transition from a rigid, hardware-defined memory hierarchy into a flexible, software-defined memory subsystem with desired capacity, bandwidth, and cost-per-GB based on the workload demands.

While our work has considered the allocation and placement of volatile data objects, the management of persistent data must also be studied in the CXL-enabled infrastructure. It should be common to have different non-volatile memory technologies deployed across the

CXL fabric in the future memory hierarchy. These non-volatile memory technologies may differ in terms of the cost-per-GB, performance, and endurance level. For instance, battery-backed DRAM achieves DRAM-like latency and endurance level, but it is much more expensive than other non-volatile memories such as PCM. Placing the persistent data in such a tiered memory presents new challenges and calls for additional research *e.g.* what are the performance and endurance implications of placing persistent objects in a particular memory tier?

The idea of memory selection and allocation must also be revisited in the context of the next-generation, data-centric OS [16] that is specifically designed for future memories. In a data-centric OS such as Twizzler [16], data is organized into objects, which are uniquely identified by a global object ID. In such an OS, it is important to have a memory configurator that ensures the data objects are placed on the right server with appropriate memory selection and allocation based on the capacity, latency, bandwidth, and persistence needs. The memory configurator must employ mechanisms to learn the characteristics such as access and write rate of the data structures and devise algorithms to orchestrate the data allocation and placement.

We envision fast, remote persistent memories enabled by CXL will be an attractive option in the future as they provide sufficient fault isolation between PMEMs and the CPU and between mirrored PMEMs [80]. Much prior work on PMEM optimized failure-atomicity mechanisms such as our SSP [86, 87] has been designed and tested assuming that the PMEMs are directly connected to the CPU. Adapting the failure-atomicity methodologies to the CXL-attached persistent memories remains future work.

Chapter 7

Conclusions

Three obstacles must be overcome in order to fully unlock the potential of emerging memory technologies such as PMEM in the future infrastructure: costly failure-atomic PMEM, overhead in finding the right resource allocation in tiered memory, and lack of platform-specific insights and optimizations. This thesis presents my work on two approaches, SSP and TMC, which improve the performance and cost efficiency of PMEM-enabled storage and tiered memory. In addition, we conduct an in-depth performance study on the interplay of real PMEM hardware and index structures. Our work leads us to draw the following implications for the future design of the processor, memory as well as software:

- *Processor design.* While SSP can improve the performance by reducing the persistence overhead introduced by extra PMEM writes, a significant performance gap still exists between the persistent and non-persistent versions of an application. To further eliminate the performance overhead introduced by flushing the cache line in the future, cost-effective solutions that expand the persistence domain would be critical in the future. For instance,

low-latency nonvolatile memory such as STT-RAM can be used to replace volatile SRAM as the last level cache [129], or techniques such as eADR [50] can be used to flush the updates in the CPU cache to the PMEM media on power failure. If the granularity disparity between CPU caches and PMEM remains in the future generation of the 3DXP modules, we could consider extending the processor to improve the I/O efficiency of 3DXP. For instance, the cache replacement logic can be made aware of the underlying 3DXP media *e.g.* prefer writing back modified cache lines in 256-byte blocks. Furthermore, systems such as our TMC benefit greatly from better observability of the workload. In the future, new PMU counters can be added to improve the observability *e.g.* counters can be added to measure the MLP of a workload.

- *Memory subsystem design.* Prior work [53] discloses that 3DXP memory embeds an address indirection table to achieve wear leveling and bad-block management. Such an indirection layer offers opportunities to incorporate the functionalities of SSP entirely inside the PM controller, significantly reducing the complexity without sacrificing much performance. Furthermore, the operations on a index structure can be interleaved to hide the latency of memory accesses [20, 55, 93]. As the latency of PMEM is several times higher than that of DRAM, it becomes even more critical to be able to hide latency of PMEM accesses. However, our work demonstrated that the lack of hardware resources such as line fill buffers limits the memory parallelism, which makes it ineffective to hide the PMEM latency.

- *Software and OS design.* SSP can be implemented in software leveraging a software

mapping layer in userspace similar to LSNVMM [46]. However, an extra indirection may result in non-trivial software overhead, which calls for optimizing the SSP mapping in the userspace. The write bandwidth of 3DXP is rather limited. We can improve the write performance of index structures by leveraging techniques that reshape the access pattern, such as log-structuring.

We hope that ideas presented in this work and other emerging ideas (*e.g.* Software Defined Memory [66, 78, 118]) will be combined and improved further through continued research on cost- and capacity-efficient high performance memory systems.

Bibliography

[1] Amazon elastic compute cloud. https://aws.amazon.com/ec2.

[2] Create a VM with a custom machine type. https://cloud.google.com/compute/docs/instances/creating-instance-with-custom-machine-type.

[3] Microsoft azure: Cloud computing services. https://azure.microsoft.com/.

[4] Compute express link: The breakthrough cpu-to-device interconnect. https://www.computeexpresslink.org/, 2020.

[5] Neha Agarwal and Thomas F Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the 2017 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 631–644, 2017.

[6] Soramichi Akiyama and Takahiro Hirofuchi. Quantitative evaluation of intel pebs overhead for online system-noise analysis. In *Proceedings of the 7th International Workshop on Runtime and Operating Systems for Supercomputers ROSS 2017*, pages 1–8, 2017.

[7] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. {CherryPick}: Adaptively unearthing the best cloud configurations for big data analytics. In *Proceedings of the 13th Symposium on Networked Systems Design and Implementation (NSDI '17)*, pages 469–482, 2017.

[8] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *Proceedings of the 15th European Conference on Computer Systems (EuroSys '20)*, pages 1–16, 2020.

[9] AMD. Amd joins consortia to advance cxl, a new high-speed interconnect for breakthrough performance. https://community.amd.com/t5/business/amd-joins-consortia-to-advance-cxl-a-new-high-speed-interconnect/ba-p/418202, 2019.

[10] Nadav Amit. Optimizing the TLB shootdown algorithm with page access tracking. In *Proceedings of the 2017 USENIX Annual Technical Conference*, pages 27–39, 2017.

[11] Andy Patrizio. Facebook and amazon are causing a memory shortage. https://www.networkworld.com/article/3247775/facebook-andamazon-are-causing-a-memory-shortage.html, 2018.

[12] R. Bayer and E. M. Mccreight. Organization and maintenance of large ordered indexes. *Acta Inf.*, 1(3), September 1972.

[13] Nathan Beckmann and Daniel Sanchez. Talus: A simple way to remove cliffs in cache performance. In *Proceedings of the 21th Int'l Symposium on High-Performance Computer Architecture (HPCA-21)*, pages 64–75. IEEE, 2015.

[14] Bernard Marr. How much data do we create every day? the mind-blowing stats everyone should read. https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/?sh=323acd8760ba, 2018.

[15] Daniel Bittman, Peter Alvaro, Darrell D. E. Long, and Ethan L. Miller. A tale of two abstractions: The case for object space. In *Proceedings of the 11th Workshop on Hot Topics in Storage and File Systems (HotStorage '19)*, July 2019.

[16] Daniel Bittman, Peter Alvaro, Pankaj Mehra, Darrell DE Long, and Ethan L Miller. Twizzler: a data-centric os for non-volatile memory. *ACM Transactions on Storage (TOS)*, 17(2):1–31, 2021.

[17] Anastasia Braginsky and Erez Petrank. A lock-free b+tree. In *Proceedings of the 24th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 58–67, 2012.

[18] Sang Kyun Cha, Sangyong Hwang, Kihong Kim, and Keunjoo Kwon. Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems. In *Proceedings of the VLDB Endowment*, volume 1, pages 181–190, 2001.

[19] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 433–452, 2014.

[20] Shimin Chen, Phillip B. Gibbons, and Todd C. Mowry. Improving index performance through prefetching. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, SIGMOD '01, pages 235–246, 2001.

[21] Shimin Chen, Phillip B. Gibbons, and Suman Nath. Rethinking database algorithms for phase change memory. In *CIDR'11: 5th Biennial Conference on Innovative Data Systems Research*, January 2011.

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

[23] Chia Chen Chou, Aamer Jaleel, and Moinuddin K Qureshi. Cameo: A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1–12. IEEE, 2014.

[24] Weihaw Chuang, Satish Narayanasamy, Ganesh Venkatesh, Jack Sampson, Michael Van Biesbrouck, Gilles Pokam, Brad Calder, and Osvaldo Colavin. Unbounded page-based transactional memory. *ACM Sigplan Notices*, 41(11):347–358, 2006.

[25] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI '16)*, pages 379–392, 2016.

[26] Russell Clapp, Martin Dimitrov, Karthik Kumar, Vish Viswanathan, and Thomas Willhalm. Quantifying the performance impact of memory latency and bandwidth for big data workloads. In *2015 IEEE International Symposium on Workload Characterization*, pages 213–224. IEEE, 2015.

[27] 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 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*, pages 105–118, March 2011.

[28] Douglas Comer. Ubiquitous b-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979.

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

[30] Pat Conway, Nathan Kalyanasundharam, Gregg Donley, Kevin Lepak, and Bill Hughes. Cache hierarchy and memory subsystem of the AMD Opteron processor. *IEEE micro*, 30(2):16–29, 2010.

[31] Guilherme Cox and Abhishek Bhattacharjee. Efficient address translation for architectures with multiple page sizes. In *Proceedings of the 2017 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 435–448, New York, NY, USA, 2017. ACM.

[32] Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. Log-free concurrent data structures. In *Proceedings of the 2018 USENIX Annual Technical Conference*, pages 373–386, 2018.

[33] Arnaldo Carvalho De Melo. The new linux'perf'tools. In *Slides from Linux Kongress*, volume 18, pages 1–42, 2010.

[34] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. *ACM SIGPLAN Notices*, 48(4):77–88, 2013.

[35] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys '14)*, April 2014.

[36] Subramanya R Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data tiering in heterogeneous memory systems. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys '16)*, pages 1–16, 2016.

[37] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[38] Sadagopan Srinivasan Li Zhao Brinda Ganesh, Bruce Jacob, and Mike Espig Ravi Iyer. CMP memory modeling: How much does accuracy matter? In *Fifth Annual Workshop on Modeling, Benchmarking and Simulation*, pages 24–33.

[39] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert NM Watson, and Steven Hand. Firmament: Fast, centralized cluster scheduling at scale. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI '16)*, pages 99–115, 2016.

[40] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI '12)*, pages 17–30, 2012.

[41] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review*, 44(4):455–466, 2014.

[42] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. Efficient memory disaggregation with infiniswap. In *Proceedings of the 13th Symposium on Networked Systems Design and Implementation (NSDI '17)*, pages 649–667, 2017.

[43] Richard A. Hankins and Jignesh M. Patel. Effect of node size on the performance of cache-conscious b+-trees. In *Proceedings of the 2003 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '03, pages 283–294, 2003.

[44] Dave Hitz, James Lau, and Michael Malcom. File system design for an NFS file server appliance. In *Proceedings of the Winter 1994 USENIX Technical Conference*, pages 235–246, San Francisco, CA, January 1994.

[45] HPS. scarab. `https://github.com/hpsresearchgroup/scarab`, [n.d.].

[46] Qingda Hu, Jinglei Ren, Anirudh Badam, and Thomas Moscibrod. Log-structured non-volatile main memory. In *Proceedings of the 2017 USENIX Annual Technical Conference*, pages 703–717, Santa Clara, CA, June 2017.

[47] 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 '18)*, page 187, 2018.

[48] Intel. Introduction to cache allocation technology in the intel xeon processor e5 v4 family. `https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-cache-allocation-technology.html`, 2016.

[49] Intel. Intel optane persistent memory. `https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html`, 2019.

[50] Intel. eADR: New opportunities for persistent memory applications. `https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html`, 2021.

[51] Intel Corporation. Architecture instruction set extensions programming reference, 2012.

[52] Intel Corporation. Persistent memory programming. `http://http://pmem.io/`, 2015.

[53] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. Basic performance measurements of the intel optane dc persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.

[54] Bruce Jacob. The memory system: you can't avoid it, you can't ignore it, you can't fake it. *Synthesis Lectures on Computer Architecture*, 4(1):1–77, 2009.

[55] Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin Levandoski, and Gor Nishanov. Exploiting coroutines to attack the killer nanoseconds. *Proc. VLDB Endow.*, 11(11):1702–1714, July 2018.

[56] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. DHTM: Durable hardware transactional memory. In *Proceedings of the 45th Int'l Symposium on Computer Architecture*, 2018.

[57] Arpit Joshi, Vijay Nagarajan, Stratis Viglas, and Marcelo Cintra. ATOM: Atomic durability in non-volatile memory through hardware logging. In *Proceedings of the 23th Int'l Symposium on High-Performance Computer Architecture (HPCA-23)*, pages 361–372. IEEE, 2017.

[58] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42th Int'l Symposium on Computer Architecture*, pages 158–169, 2015.

[59] Hiwot Tadese Kassa, Jason Akers, Mrinmoy Ghosh, Zhichao Cao, Vaibhav Gogte, and Ronald Dreslinski. Improving performance of flash based {Key-Value} stores using storage class memory as a volatile memory extension. In *Proceedings of the 2021 USENIX Annual Technical Conference*, pages 821–837, 2021.

[60] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D Nguyen, Tim Kaldewey, Victor W Lee, Scott A Brandt, and Pradeep Dubey. FAST: fast architecture sensitive tree search on modern cpus and gpus. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 339–350. ACM, 2010.

[61] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. Exploring the design space of page management for multi-tiered memory systems. In *Proceedings of the 2021 USENIX Annual Technical Conference*, pages 715–728, 2021.

[62] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Selecta: Heterogeneous cloud storage configuration for data analytics. In *Proceedings of the 2018 USENIX Annual Technical Conference*, pages 759–773, 2018.

[63] Apostolos Kokolis, Dimitrios Skarlatos, and Josep Torrellas. Pageseer: Using page walks to trigger page swaps in hybrid memory systems. In *Proceedings of the 25th Int'l Symposium on High-Performance Computer Architecture (HPCA-25)*, pages 596–608. IEEE, 2019.

[64] Shonali Krishnaswamy, Seng Wai Loke, and Arkady Zaslavsky. Estimating computation times of data-intensive applications. *IEEE Distributed Systems Online*, 5(4), 2004.

[65] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J Rossbach, and Emmett Witchel. Coordinated and efficient huge page management with Ingens. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI '16)*, pages 705–721, 2016.

[66] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, et al. Software-defined far memory in warehouse-scale computers. In *Proceedings of the 2019 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 317–330, 2019.

[67] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable DRAM alternative. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA '09)*, ISCA '09, pages 2–13, New York, NY, USA, 2009. ACM.

[68] 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 '17)*, pages 257–270, 2017.

[69] 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 28th ACM Symposium on Operating Systems Principles (SOSP '19)*, pages 462–477, 2019.

[70] Seok-Hee Lee. Technology scaling challenges and opportunities of memory devices. In *2016 IEEE International Electron Devices Meeting (IEDM)*, pages 1–1. IEEE, 2016.

[71] Lenovo. Cxl and the tiered-memory future of servers. `https://www.lenovoxperience.com/newsDetail/283yi044hzgcdv7snkrmmx9ovpq6aesmy9u9k7ai2648j7or`, 2021.

[72] 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 '13)*, pages 302–313. IEEE, 2013.

[73] Huaicheng Li, Daniel S Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Ishwar Agarwal, Mark Hill, Marcus Fontoura, et al. First-generation memory disaggregation for cloud platforms. *arXiv preprint arXiv:2203.00241*, 2022.

[74] Yang Li, Saugata Ghose, Jongmoo Choi, Jin Sun, Hui Wang, and Onur Mutlu. Utility-based hybrid memory management. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 152–165. IEEE, 2017.

[75] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K Reinhardt, and Thomas F Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th annual international symposium on Computer architecture*, pages 267–278, 2009.

[76] Linux Kernel Organization. Linux zswap. `https://www.kernel.org/doc/Documentation/vm/zswap.txt`, [n.d.].

[77] Liang Luo, Akshitha Sriraman, Brooke Fugate, Shiliang Hu, Gilles Pokam, Chris J Newburn, and Joseph Devietti. Laser: Light, accurate sharing detection and repair. In *Proceedings of the 22th Int'l Symposium on High-Performance Computer Architecture (HPCA-22)*, pages 261–273. IEEE, 2016.

[78] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. TPP: Transparent page placement for cxl-enabled tiered memory. *arXiv preprint arXiv:2206.02878*, 2022.

[79] Andréa Matsunaga and José AB Fortes. On the use of machine learning to predict the time and resources consumed by applications. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 495–504. IEEE, 2010.

[80] Pankaj Mehra and Samuel Fineberg. Fast and flexible persistence: the magic potion for fault-tolerance, scalability and performance in online data stores. In *Proceedings of the 18st International Parallel & Distributed Processing Symposium (IPDPS 2004)*, page 206. IEEE, 2004.

[81] Micron. Micron exits 3D xpoint market, eyes CXL opportunities. `https://www.eetimes.com/micron-exits-3d-xpoint-market-eyes-cxl-opportunities/`, 2021.

[82] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IEEE International Symposium on Workload Characterization (IISWC '08)*, pages 35–46. Citeseer, 2008.

[83] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, 1992.

[84] 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 2017 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ASPLOS '17, pages 135–148, New York, NY, USA, 2017. ACM.

[85] Juan Navarro, Sitararn Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. *ACM SIGOPS Operating Systems Review*, 36(SI):89–104, 2002.

[86] Yuanjiang Ni, Jishen Zhao, Daniel Bittman, and Ethan Miller. Reducing NVM writes with optimized shadow paging. In *Proceedings of the 10th Workshop on Hot Topics in Storage and File Systems (HotStorage '18)*, July 2018.

[87] Yuanjiang Ni, Jishen Zhao, Heiner Litz, Daniel Bittman, and Ethan L Miller. Ssp: Eliminating redundant writes in failure-atomic nvrams via shadow sub-paging. In *Proceedings of the 52th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 836–848, 2019.

[88] Oliver Niehorster, Alexander Krieger, Jens Simon, and Andre Brinkmann. Autonomic resource management with support vector machines. In *2011 IEEE/ACM 12th International Conference on Grid Computing*, pages 157–164. IEEE, 2011.

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

[90] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for RAMClouds: Scalable high-performance storage entirely in DRAM. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, December 2009.

[91] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. Marss: a full system simulator for multicore x86 cpus. In *2011 48th Design Automation Conference (DAC)*, pages 1050–1055. IEEE, 2011.

[92] M. Poremba, S. Mittal, D. Li, J. S. Vetter, and Y. Xie. Destiny: A tool for modeling emerging 3d nvm and edram caches. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2015.

[93] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. Interleaving with coroutines: A practical approach for robust index joins. *Proc. VLDB Endow.*, 11(2):230–242, October 2017.

[94] Moinuddin K Qureshi and Yale N Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 423–432. IEEE, 2006.

[95] Jun Rao and Kenneth A. Ross. Making B+-trees cache conscious in main memory. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 475–486, Dallas, TX, May 2000.

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

[97] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. DRAMSim2: A cycle accurate memory system simulator. *IEEE Computer Architecture Letters*, 10(1):16–19, 2011.

[98] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. Log-structured memory for DRAM-based storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST '14)*, February 2014.

[99] Samsung. Samsung unveils industry-first memory module incorporating new cxl interconnect standard. `https://news.samsung.com/global/samsung-unveils-industry-first-memory-module-incorporating-new-cxl-interconnect-standard`, 2015.

[100] Daniel Sanchez and Christos Kozyrakis. Vantage: Scalable and efficient fine-grain cache partitioning. In *Proceedings of the 38th Int'l Symposium on Computer Architecture*, pages 57–68, 2011.

[101] Steve Scargall. *Programming persistent memory: A comprehensive guide for developers*. Springer Nature, 2020.

[102] Fujitsu Semiconductor. Fujitsu semiconductor releases world's largest density 8mbit reram product from september. https://www.fujitsu.com/global/products/devices/semiconductor/memory/reram/spi-8m-mb85as8mt.html, 2019.

[103] Vivek Seshadri, Gennady Pekhimenko, Olatunji Ruwase, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, Todd C Mowry, and Trishul Chilimbi. Page overlays: An enhanced virtual memory framework to enable fine-grained memory management. In *Proceedings of the 42th Int'l Symposium on Computer Architecture*, pages 79–91. IEEE, 2015.

[104] Jason Sewall, Jatin Chhugani, Changkyu Kim, Nadathur Satish, and Pradeep Dubey. PALM: Parallel architecture-friendly latch-free modifications to b+ trees on many-core processors. *Proc. VLDB Endowment*, 4(11):795–806, 2011.

[105] Akbar Sharifi, Shekhar Srikantaiah, Asit K Mishra, Mahmut Kandemir, and Chita R Das. METE: meeting end-to-end qos in multicores through system-wide resource management. In *Proceedings of the 2011 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 13–24, 2011.

[106] 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*, pages 178–190. ACM, 2017.

[107] SK hynix. tructural and device considerations for vertical cross point memory with single-stack memory toward CXL memory beyond 1xnm 3DXP. https://research.skhynix.com/blog/detail?seq=147, 2022.

[108] Michael Stonebraker and Ariel Weisberg. The voltdb main memory dbms. *Data Engineering*, page 21, 2013.

[109] Madhusudhan Talluri and Mark D. Hill. Surpassing the tlb performance of superpages with less operating system support. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '94)*, pages 171–182, New York, NY, USA, 1994. ACM.

[110] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient performance prediction for {Large-Scale} advanced analytics. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI '16)*, pages 363–378, 2016.

[111] Abhishek Verma, Madhukar Korupolu, and John Wilkes. Evaluating job packing in warehouse-scale computing. In *Proceedings of the 2014 IEEE International Conference on Cluster Computing*, pages 48–56. IEEE, 2014.

[112] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys '15)*, pages 1–17, 2015.

[113] Carlos Villavieja, Vasileios Karakostas, Lluis Vilanova, Yoav Etsion, Alex Ramirez, Avi Mendelson, Nacho Navarro, Adrian Cristal, and Osman S Unsal. DiDi: Mitigating the performance impact of TLB shootdowns using a shared TLB directory. In *2011 International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 340–349. IEEE, 2011.

[114] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*, March 2011.

[115] Chenxi Wang, Huimin Cui, Ting Cao, John Zigman, Haris Volos, Onur Mutlu, Fang Lv, Xiaobing Feng, and Guoqing Harry Xu. Panthera: Holistic memory management for big data processing over hybrid memories. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 347–362, 2019.

[116] Zhaoguo Wang, Hao Qian, Jinyang Li, and Haibo Chen. Using restricted transactional memory to build a scalable in-memory database. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys '14)*, page 26. ACM, 2014.

[117] 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 International Conference on Management of Data*, pages 473–488. ACM, 2018.

[118] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, et al. TMO: transparent memory offloading in datacenters. In *Proceedings of the 2022 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 609–621, 2022.

[119] Wikipedia. Persistent memory. `https://en.wikipedia.org/wiki/Persistent_memory`, [n.d.].

[120] Chin-Hsien Wu, Tei-Wei Kuo, and Li Ping Chang. An efficient b-tree layer implementation for flash-memory storage systems. *ACM Trans. Embed. Comput. Syst.*, 6(3), July 2007.

[121] Kai Wu, Yingchao Huang, and Dong Li. Unimem: Runtime data managementon non-volatile memory-based heterogeneous main memory. In *Proceedings of the 2015 International Conference for High Performance Computing, Networking, Storage and Analysis (SC17)*, pages 1–14, 2017.

[122] Xiaojian Wu, Sheng Qiu, and A. L. Narasimha Reddy. SCMFS: A file system for storage class memory and its extensions. *ACM Transactions on Storage*, 9(3), August 2013.

[123] 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 '16)*, February 2016.

[124] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble page management for tiered memory systems. In *Proceedings of the 2019 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 331–345, 2019.

[125] 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 16th USENIX Conference on File and Storage Technologies (FAST '20)*, pages 169–182, 2020.

[126] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. NV-Tree: Reducing consistency cost for NVM-based single level systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*, pages 167–181, February 2015.

[127] Jifei Yi, Benchao Dong, Mingkai Dong, and Haibo Chen. On the precision of precise event based sampling. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 98–105, 2020.

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

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

[130] Yanqi Zhou, Henry Hoffmann, and David Wentzlaff. Cash: Supporting iaas customers with a sub-core configurable architecture. In *Proceedings of the 43th Int'l Symposium on Computer Architecture*, pages 682–694, 2016.

[131] Yanqi Zhou and David Wentzlaff. The sharing architecture: sub-core configurability for iaas clouds. In *Proceedings of the 2014 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*, pages 559–574, 2014.