UNIVERSITY OF CALIFORNIA, MERCED

Runtime Data Management on Non-Volatile Memory-Based High Performance Systems

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

in

Electrical Engineering and Computer Science

by

Kai Wu

Committee in charge:

Dong Li, Chair
Maya B Gokhale
Mukesh Singhal
Florin Rusu

Spring 2021

The dissertation of Kai Wu is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Professor Dong Li, Chair

Dr. Maya B Gokhale

Professor Mukesh Singhal

Professor Florin Rusu

University of California, Merced

Spring 2021

DEDICATION

To my family.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I am grateful for the support from many great people during this long adventure.

First of all, I would like to sincerely thank my advisor Professor Dong Li. His continuous guidance helped me overcome difficulties and achieved my goals throughout my graduate study, and inspired me to move towards a better career path.

I want to express my gratitude to my committee, Dr. Maya Gokhale, Professor Mukesh Singhal and Professor Florin Rusu for the time and effort they spent to help me prepare my dissertation and serve as my committee members.

I want to acknowledge Dr. Ivy Peng, Professor Feng Chen and Dr. Jialin Liu for their help and guidance in the papers that we co-authored.

I appreciate my internship days at Lawrence Livermore National Laboratory, Los Alamos National Laboratory and ByteDance Infrastructure System Lab. My thanks go to Dr. Maya Gokhale, Dr. Nathan Debardeleben, Professor Qiang Guan, Dr. Jianjun Chen, Dr. Ye Liu, Mr. Marty McFadden, Dr. Keita Iwabuchi and Dr. RogerPearce for much help along the way. I could never have imagined more enjoyable internship experiences.

I also would like to express my appreciation to colleagues and friends at UC Merced. I would like to thank Ms. Ren Jie for helping me in multiple projects and offering me with valuable suggestions. I am aslo grateful to other members of Parallel Architecture, System, and Algorithm Laboratory: Dr. Luanzheng Guo, Mr. Jiawen Liu, Ms. Wenqian Dong, Mr. Jie Liu, Mr. Wei Liu, Ms. Hanlin He and Ms. Jiaolin Luo. I give my heartfelt thanks my friends and roommates, with whom I had a wonderful time: Dr. Jing Yan, Mr. Yaorong Fan and Dr. Yi Zhu.

Last but not least I want to express my gratitude to my family for their love and support during my whole life. I want to thank my parents for they giving me unconditional love and belief in me in every choice that I have made in my life.

VITA

2010-2014	B. S. in Digital Media Technology, Harbin Normal University
2014-2016	M. S. in Computer Science and Engineering, Michigan State University
2016-2021	Ph. D. in Electrical Engineering and Computer Science, University of California, Merced

PUBLICATIONS

Kai Wu, Jie Ren, Ivy Peng and Dong Li. "ArchTM: Architecture-Aware, High Performance Transaction for Persistent Memory". In 19th USENIX Conference on File and Storage Technologies, 2021.

Kai Wu and Dong Li. "Unimem: Runtime Data Management on Non-Volatile Memory-based Heterogeneous Main Memory for High Performance Computing". In the International Journal of Computer Science and Technology, Special Section on Memory-Centric System Research for HPC, 2021.

Jie Ren, Jiaolin Luo, **Kai Wu**, Minjia Zhang, Hyeran Jeon and Dong Li. "Efficient Tensor Migration and Allocation on Heterogeneous Memory Systems for Deep Learning". In The 27th IEEE International Symposium on High-Performance Computer Architecture, 2021.

Jie Ren, Jiaolin Luo, Ivy B. Peng, **Kai Wu** and Dong Li. "Optimizing Large-Scale Plasma Simulations on Persistent Memory-based Heterogeneous Memory with Effective Data Placement Across Memory Hierarchy". In 35th International Conference on Supercomputing, 2021.

Kai Wu, Ivy B. Peng, Jie Ren and Dong Li. "Ribbon: High Performance Cache Line Flushing for Persistent Memory". In 29th International Conference on Parallel Architectures and Compilation Techniques, 2020.

Ivy B. Peng, **Kai Wu**, Jie Ren, Dong Li and Maya Gokhale. "Demystifying the Performance of HPC Scientific Applications on NVM-based Memory Systems". In 34rd IEEE International Parallel and Distributed Processing Symposium, 2020.

Jie Ren, **Kai Wu** and Dong Li. "Exploring Non-Volatility of Non-Volatile Memory for High Performance Computing Under Failures". In 22th IEEE Cluster Conference, 2020.

Ivy B. Peng, Marty McFadden, Eric Green, Keita Iwabuchi, **Kai Wu**, Dong Li, Roger Pearce and Maya Gokhale. "UMap: Enabling Application-driven Optimizations for Page Management". In Workshop on Memory Centric High Performance Computing held in conjunction with SC'19.

Kai Wu, Jie Ren and Dong Li. "Runtime Data Management on Non-Volatile Memory-based Heterogeneous Memory for Task-Parallel Programs". In 30th ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, 2018.

Kai Wu, Wenqian Dong, Qiang Guan, Nathan Debardeleben and Dong Li. "Modeling Application Resilience in Large Scale Parallel Execution". In 47th International Conference on Parallel Processing, 2018.

Jie Ren, **Kai Wu** and Dong Li. "Understanding Application Recomputability without Crash Consistency in Non-Volatile Memory". In Workshop on Memory Centric High Performance Computing held in conjunction with SC'18.

Kai Wu, Yingchao Huang and Dong Li. "Unimem: Runtime Data Management in Non-Volatile Memory-based Heterogeneous Main Memory". In 29th ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, 2017.

Shuo Yang, **Kai Wu**, Yifan Qiao, Dong Li and Jidong Zhai. "Algorithm-Directed Crash Consistence in Non-Volatile Memory for HPC". In 19th IEEE Cluster Conference, 2017.

Wei Liu, **Kai Wu**, Jialin Liu, Feng Chen and Dong Li. "Performance Evaluation and Modeling of HPC I/O on Non-Volatile Memory". In 12th International Conference on Networking, Architecture, and Storage, 2017.

ABSTRACT OF THE DISSERTATION

Runtime Data Management on Non-Volatile Memory-Based High Performance Systems

by

Kai Wu

Doctor of Philosophy in Electrical Engineering and Computer Science

University of California Merced, Spring 2021

Dong Li, Chair

Byte-addressable non-volatile memories (NVM) have been envisioned as a new tier in computer systems, providing memory-like performance and storage-level capacity and persistence. Because of the relatively high latency and low bandwidth of NVM (comparing with dynamic random-access memory (DRAM)), NVM is often paired with DRAM to build a heterogeneous main memory system (HMS). As a result, application data must be carefully placed to NVM and DRAM for best performance. Moreover, in a NVM-based HMS, data on NVM is not lost when the system crashes because of the non-volatility nature of NVM. However, because of the volatile caches and the processor's reordering of instructions, data must be logged in failure-atomic transactions and explicitly flushed from caches into NVM to ensure consistency and correctness before crashes, which can cause large runtime overhead.

This dissertation focuses on building lightweight runtime systems on the NVM-based HMS to effectively manage data placement and data crash consistency. This dissertation first studies the data placement of two types of high-performance computing (HPC) applications on NVM-based HMS (i.e., message passing interface (MPI) programs and task-parallel programs). The dissertation presents the Unimem and Tahoe runtimes to implement automatic and transparent data placement on NVM-based HMS for MVM-based applications and task-parallel applications, respectively.

Failure-atomic transactions are a critical mechanism for accessing and manipulating data on NVM with crash consistency. This dissertation then investigates performance problems in common transaction implementations on real NVM hardware

and highlights the importance of considering NVM architecture characteristics for transaction performance. The dissertation presents ArchTM, an architecture-aware NVM transaction system.

Finally, this dissertation analyzes the cache line flushing (CLF) mechanism, which is a fundamental building block for programming NVM to ensure crash consistency. This dissertation designs and implements Ribbon to optimize CLF mechanisms through a decoupled concurrency control and proactive CLF to change cache line status. Ribbon also uses cache line coalescing as an application-specific solution for those with low dirtiness in flushed cache lines.

Chapter 1

Introduction

With the rapid development of information technique, modern applications are often characterized by massive data sizes and intensive data processing. For example, the Blue Brain project aims to simulate the human brain with a daunting 100PB memory that needs to be revisited by the solver at every time step; the cosmology simulation studying Q continuum works on 2PB per simulation. Those applications pose high demands on data access performance and memory capacity.

Current computer systems commonly equipped with dynamic random access memory (DRAM) cannot satisfy such an expectation. Subject to physical and manufacturing limitations (e.g., power, density, resilience, and cost [1, 2, 3]), the DRAM density cannot continue increasing while data volume quickly increases. We have to store data on bigger but slower storage media (e.g., hard drive (HDD) and solid-state drive (SSD)) and perform frequent I/O between the storage medium and DRAM. Such an architecture sacrifices application performance, because there is a huge performance gap between DRAM (10 ns average latency) and storage (10^6 ns average latency on HDD and 10^4 ns average latency on SSD). Hence, new memory technologies that can replace DRAM or narrow the performance gap between DRAM and storage increasingly attract attention.

The emerging non-violate memory (NVM) techniques, such as phase change memory (PCM), resistive random-access memory (ReRAM), and 3D XPoints [4] are considered as a promising candidate for future big memory machine. Compare to DRAM, NVM has higher density and persistence (does not require power refresh). Compare to HDD and SSD, NVM has better performance and offers a

byte-addressable memory interface (i.e., applications access NVM via load and store instructions). However, the appealing NVM solutions cannot as a drop-in replacement for DRAM in systems because of two major challenges.

First, NVM has a higher latency and lower bandwidth than DRAM. Such NVM characteristics introduce a performance gap between NVM-based and DRAM-based systems. Though the performance gap is smaller than that of DRAM and HDD/SSD, it still causes a dramatic slow down on applications [5, 6]. To address this challenge, pairing NVM with DRAM to build a heterogeneous memory system (HMS) is a common solution. The recent release of the Intel Optane™DC persistent memory module-based architecture (named Optane PM in the rest of the dissertation) equipped with DRAM and NVM is such a system. However, efficiently running applications on NVM-based HMS is not trivial. On one hand, we want to use DRAM as much as possible to achieve the best performance; on the other hand, we often have a limited amount of DRAM on HMS because DRAM is relatively expensive and consumes more energy, compared to NVM. The above conflict presents a *data placement challenge*: Which data of the application should be placed into the limited DRAM of HMS to accomplish high performance close to the performance when the application runs on the DRAM-only system.

Second, while NVM does not lose data after a system crash due to power loss or hardware failure, there is no guarantee that the application data retained on the NVM is correct and usable by the recovery process to restart applications. Existing processors may reorder memory writes to improve performance, which results in data not necessarily being written from volatile processor caches to NVM in program order. To enforce the write ordering, applications running on NVM-based systems need to explicitly flush the cache and issue a memory barrier. However, frequently executing cache flushes and memory barriers impose high overhead on the application. Moreover, applications running on NVM need to ensure that data is modified atomically when transitioning from one consistent state to another on NVM in order to provide consistency after a crash. To meet such a requirement, failure-atomic transactions are a common mechanism for accessing and manipulating data on NVM. But the transaction mechanism is known to be expensive in terms of performance. Therefore, how to effectively ensure the *data crash consistency* on NVM brings another challenge in adopting NVM to current computer systems.

1.1 Primary Contributions

This dissertation tries to resolve the above two challenges (data placement and data crash consistency) by using software methods. Traditional storage-based software solutions are not suitable for NVM-based HMS because they are built for the performance characteristics of disks and may introduce a high software overhead. Therefore, high-performance NVM systems call for new software solutions.

Data placement for Message Passing Interface (MPI) programs. This dissertation first studies data placement for MPI programs on NVM-based HMS. MPI is one of the most widely used programming models in HPC. There are several research questions. First, how to capture and characterize memory access patterns associated with data objects? Second, how to strike a balance between different requirements on the frequency of data movement (i.e., the implementation of data placement)? Third, how to minimize the impact of data movement on application performance? To answer the above questions, we design and implement a lightweight runtime system, named "*Unimem*", to automatically and transparently decide and implement data placement on NVM-based HMS for MPI programs.

To decide what data object should be placed on DRAM, Unimem employs an online sampling-based profiling method to capture memory access patterns. This method is based on performance counters. Given the collected profiling information, we further characterize memory access patterns and associate them with data objects.

To decide the frequency of data movement, Unimem uses lightweight performance models, based on which Unimem predicts performance benefit and cost if moving data objects between NVM and DRAM. Based on the performance models, Unimem avoids unnecessary data movement while maximizing the benefits of data movement.

To minimize the impact of data movement on application performance, Unimem introduces a proactive data movement mechanism to avoid the impact of data movement on application performance. Given an execution phase and a data movement plan for the phase, this mechanism uses a helper thread to trigger data movement before the phase. The helper thread runs in parallel with the application, overlapping data movement with application execution.

Data Placement for Task Parallel Programs. This dissertation next studies the data placement for task-parallel programs on NVM-based HMS. A task-based

programming model such as OpenMP tasks, decompose a program into a set of tasks and distribute them between processing elements (e.g. CPU cores). Those programming models can improve hardware utilization and have been widely explored in HPC. Deciding data placement on HMS for task-parallel programs is a challenging problem. First, given many possible data distributions for each task and many tasks in a task-parallel program, we have to explore many possible data placement for best performance, which can be time-consuming. Second, profiling the memory accesses of tasks to decide data placement is challenging. Different tasks in a task-parallel program often work on different data. No matter which task is profiled, the profiling result for one task is not usable to direct data placement for other tasks, because of the difference in memory addresses and access patterns between tasks.

We present *Tahoe*, a runtime system that enables efficient data placement on NVM-based HMS. Leveraging the semantics and execution mode of task-parallel programs, Tahoe efficiently characterizes memory access patterns, decides data placement for many tasks, makes the best use of limited DRAM space, and reduces data movement overhead.

To address the challenge of profiling memory accesses for many tasks without causing expensive overhead, Tahoe choose a few representative tasks to profile and decide the most accessed pages. Each representative task has similar memory access patterns to many other tasks. To make the memory access information of the representative task generally applicable to other tasks with different memory pages (addresses), Tahoe leverages program semantics to transform the information of the representative task from page level to data-object level, such that other tasks can decide their potentially most accessed pages.

To determine the data placement, Tahoe is featured with a hybrid performance model to predict performance for various data placement cases. The hybrid performance model uses a machine learning model to avoids most of modeling complexity, and uses an analytical model to add modeling flexibility.

Architecture-aware transactions for NVM. This dissertation then studies the transaction mechanisms for NVM. Failure-atomic transactions are a critical mechanism for accessing and manipulating data with crash consistency on NVM. Extensive studies have proposed various transaction mechanisms that generally employ logging-based (undo or redo logging) or Copy-on-Write (CoW)-based designs.

Existing works optimize NVM transactions by reducing data copying or persistence overhead. They emulate NVM based on DRAM with increased memory latency or reduced bandwidth, but miss NVM architecture details. For instance, logging-based transactions have a double write problem because of creating logs and updating data in-place. The excessive writes to NVM mismatch with poor write performance on NVM. CoW-based transactions avoid this problem, but suffers from performance overhead due to metadata updates, which causes many small writes misaligned with NVM internal block size.

We design ArchTM, a NVM transaction system based on two design principles: avoiding small writes and encouraging sequential writes. ArchTM is a variant of CoW-based system to reduce write traffic to NVM. Unlike conventional CoW schemes, ArchTM reduces metadata modifications through a scalable lookup table on DRAM. ArchTM introduces an annotation mechanism to ensure crash consistency and a locality-aware data path in memory allocation to increases coalesable writes inside NVM devices.

Optimizing cache line flushing mechanism for NVM. Finally, this dissertation studies the cache line flushing mechanism for NVM. Cache line flushing (CLF) is a fundamental building block for programming NVM. NVM-aware workloads often rely on CLF to enforce write ordering and ensure crash consistency. However, CLF creates a performance bottleneck on NVM, which may significantly reduce the performance benefits promised by NVM. Most existing works focus on optimizing persistency semantics (e.g., skipping CLF or relaxing constraints on persist barriers), other than the CLF mechanism. They use different fault models or recovery mechanisms that are designed for specific application characteristics. All these techniques use the CLF mechanism to realize their persistency semantics.

Unlike the previous works, we focus on the CLF mechanism itself, instead of persistency semantics. Therefore, our work is generally applicable to NVM-aware applications. we examine the performance of CLF mechanism based on the performance characterization of well-established workloads on real NVM hardware (i.e., Intel Optane PM). Based on the characteristics, we introduce Ribbon, a runtime system that optimizes the CLF mechanism through concurrency control that adapts the intensity of CLF and uses proactive CLF to increase the probability of flushing clean cache lines. Ribbon, automatically detects CLF bottleneck in oversupplied or insufficient

concurrency, and adapts accordingly. Ribbon, also proactively transforms dirty or non-resident cache lines into clean resident ones to reduce the latency of CLF. Furthermore, we investigate the cause for low dirtiness of flushed cache lines in seven representative NVM-aware workloads and provide an application-specific solution to coalesce cache lines.

1.2 Outline and Previously Published Work

The remainder of the dissertation is organized as follows. Chapter 2 provides a comprehensive background for this dissertation. Chapter 3 presents the design, implementation and evaluation of Unimem,a lightweight runtime that automatically and transparently manages data placement on HMS for MPI-based HPC applications. Chapter 4 introduces Tahoe, a runtime system that addresses the data placement problem for task parallel programs on NVM-based HMS. Chapter 5 identifies that small random writes in metadata modifications and locality-oblivious memory allocation in traditional NVM transaction systems mismatch NVM architecture, and presents ArchTM, a NVM transaction system based on two design principles: avoiding small writes and encouraging sequential writes. Chapter 6 examines the performance of CLF mechanism based on the performance characterization of well-established workloads on real NVM hardware, and introduces Ribbon, a runtime system that improves the performance of CLF mechanism through concurrency control and proactive CLF. Chapter 7 concludes this dissertation by summarizing the main lessons learned, the open problems, and the topics for future work.

Chapter 2 contains material from "Demystifying the Performance of HPC Scientific Applications on NVM-based Memory Systems", by Ivy Peng, Kai Wu, Jie Ren and Dong Li, which appears in the Proceedings of the 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS'20) [7]. The dissertation author is the primary investigator and second author of this paper. The material in these chapters is copyright

Chapter 3 contains material from "Unimem: Runtime Data Management on Non-Volatile Memory-based Heterogeneous Main Memory", by Kai Wu, Yingchao Huang and Dong Li, which appears in the Proceedings of the 2017 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'17) [5].

The dissertation author is the primary investigator and first author of this paper. The material in these chapters is copyright ©2020 by the Association for Computing Machinery (ACM).

Chapter 4 contains material from "Runtime data management on non-volatile memory-based heterogeneous memory for task-parallel programs", by Kai Wu, Jie Ren and Dong Li, which appears in the Proceedings of the 2018 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'18) [8]. The dissertation author is the primary investigator and first author of this paper. The material in these chapters is copyright ©2018 by the IEEE Association.

Chapter 5 contains material from "ArchTM: Architecture-Aware, High Performance Transaction for Persistent Memor", by Kai Wu, Jie Ren, Ivy Peng and Dong Li, which appears in the Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST 2021) [9]. The dissertation author is the primary investigator and first author of this paper. The material in these chapters is copyright ©2021 by the USENIX Association.

Chapter 6 contains material from "Ribbon: High Performance Cache Line Flushing for Persistent Memory", by Kai Wu, Ivy Peng, Jie Ren and Dong Li, which appears in the Proceedings of the 2020 International Conference on Parallel Architectures and Compilation Techniques (PACT'20) [10]. The dissertation author is the primary investigator and first author of this paper. The material in these chapters is copyright ©2020 by the Association for Computing Machinery (ACM).

Chapter 2

Background

This chapter presents a comprehensive introduction to the background of this dissertation.

2.1 Non-Volatile Memory

Non-violate memory (NVM) is an emerging data media narrowing the gap between traditional volatile memory (e.g., DRAM) and persistent storage (e.g., HDD and SSD). NVM can retrieve stored information even after having been power failure. Compared to DRAM, NVM has higher density so that NVM can implement a larger capacity using the same area size. Compared to HDD/SSD, NVM can provide orders of magnitude faster-accessing speed.

Over the past ten years, both industry and academic have been working hard to develop NVM techniques. Many new NVM techniques, such as phase change memory (PCM), resistive random-access memory (ReRAM) and Spin-Transfer Torque RAM (STT-RAM), have been proposed. The recent release of the Intel Optane DC persistent memory module (named Optane in the rest of the paper) marks the first mass production of byte-addressable NVM. Table 2.1 summarizes performance characteristics of recent proposed NVM techniques and DRAM [11, 12]. Intel does not publish energy and density information of Optane DC PM yet.

Table 2.1: Performance characteristics of NVM techniques, DRAM, and NAND Flash.

	DRAM	PCM	STT-RAM	ReRAM	Optane PM	NAND Flash
Byte-addressable	Yes	Yes	Yes	Yes	Yes	No
Persistence	No	Yes	Yes	Yes	Yes	Yes
Read time (ns)	10	10 - 50	10 - 35	<10	180 - 340	10^5
Write time (ns)	10	50 - 500	50 - 90	20 - 30	90 - 390	10^5
Standby power	Refresh current	None	None	None	None	None
Energy/bit (pj^2)	2 - 4	2 - 100	0.1 - 1	0.1 - 3	N/A	10 - 10^4
Density	6 - 12	4 - 16	20 - 60	<4	N/A	1 - 4

2.2 NVM-based Heterogeneous Memory

Extensive research has proposed using NVM for implementing the main memory to exploit its high density, persistence, and power efficiency [13, 14]. Still, the current NVM technologies have lower performance than DRAM, and thus, main memory designs often pair NVM with DRAM to build a heterogeneous memory system (HMS). The Intel Optane PM-based architecture is an example of NVM-based HMS. The work of [15] has provided detailed system evaluation of the Intel Optane PM-based architecture, and we briefly summarize the system architecture (Figure 2.1) in this section.

Intel Optane DC Persistent Memory. On Optane PM-based architecture, the memory subsystem consists of DRAM DIMMs and NVDIMMs that share integrated memory controllers (iMC). Each NVDIMM has a small internal controller for address translation and a data buffer. The internal data granularity in the Optane media is 256 bytes, while the data granularity between the processor and memory subsystem is 64 bytes. Data is guaranteed to become persistent only after it reaches iMC. In cases of power failure, data in write pending queue (WPQ) in iMC will be flushed to NVDIMM by hardware. When WPQ has high occupancy, write blocking effect could stall CPU if threads have to wait for the WPQ to drain [16]. There is also a small DRAM buffer within the Optane device to improve the reuse of fetched data and reduce write-amplification [12].

System evaluation has quantified that sequential and random read accesses to NVM have a latency of 174 ns and 304 ns, respectively [15]. Write latency to NVM depends on store instructions and data sizes. For instance, 64- to 256-byte non-temporal data store has 180 – 200 ns latency [17]. On one socket, the read bandwidth to NVM can reach 39 GB/s while the peak write bandwidth is only 13 GB/s [15, 17].

Thus, the NVM exhibits about three times asymmetry in read and write bandwidth.

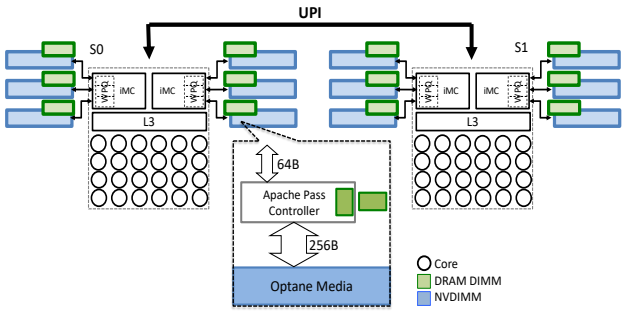


Figure 2.1: The system architecture the Intel Purley platform.

The NVDIMMs can be configured in *Memory* or *AppDirect* mode. In Memory mode, DRAM becomes a hardware-managed direct-mapped write-back cache to NVM and is transparent to applications. Note that DRAM on one socket cannot cache accesses to NVM on another socket [18]. In AppDirect mode, the NVM becomes a byte-addressable persistent memory. A *dax*-aware file system would transparently convert file read and write operations into 64-byte load and store instructions in this mode to access NVM. Also, in this mode, the NVM on each socket can be exposed as a non-uniform memory access (NUMA) node to the CPUs. Standard NUMA management routines like *numactl* can be used to control data placement in this configuration.

2.3 Crash Consistency and Cache Line Flushing

On-chip data caches are mostly implemented with volatile memory like SRAM. Because of the prevalence of volatile caches, data corruption could occur if updates to a data object stay in the cache but have not reached the persistent domain when a crash happens. A persistent domain refers to the part of the memory hierarchy that can retain data through a power failure. For instance, the system from iMC to Optane media is the persistent domain on the Optane architecture [12]. For data crash consistency (persistence), the programmer typically employs ISA-specific CLF instructions, such as `clflush`, `clflushopt`, and `clwb` on x86 machines [19], to ensure that data in a cache line is pushed to the persistent domain. The order of two CLF can be enforced by an `sfence` instruction, which ensures the second CLF does not happen before the first one reaches the persistent domain.

The standard practice to ensure persistence of a data object in NVM is to flush all cache blocks [1] of the data object [19], even though the data object may not be fully cached. Because of the complexity and overhead of tracking dirty cache lines or checking resident cache blocks for a particular data object in the existing hardware, every cache block of the data object is flushed by software, exemplified in Listing 2.1. The example is a code snippet from Intel PMDK [20].

Listing 2.1: An example of persisting a data object

```
1 /*Loop through cacheline aligned chunks*/
2 /*covering a target data object*/
3 cache_block_flush(const void *addr, size_t len)
4 {
5     unsigned __int64 ptr;
6     for (ptr = (unsigned __int64)addr & ~(FLUSH_ALIGN - 1);
7         ptr < (unsigned __int64)addr + len;
8         ptr += FLUSH_ALIGN)
9         /*clflush/clflush_opt/clwb*/
10        flush((char *)ptr);
11    /*clflush_opt and clwb needs a fence*/
12    /*to ensure its completeness*/
13    _mm_sfence();
14 }
```

Flushing cache lines from the volatile cache into the persistent domain is the building block for programming NVM. Active research in different NVM access interfaces – libraries [20, 21, 22], multi-threaded programming models [23, 24, 25], and file systems [26, 27, 28, 29] – proposes optimizations to mitigate the high overhead of CLF. We categorize existing CLF optimizations into five classes, summarized as follows.

Eager CLF triggers CLF explicitly at the application level after the data value is updated. There is no delay of CLF and no skip of CLF. This kind of CLF provides strict persistency [30], but often introduces excessive constraints on write ordering, limiting the concurrency of writes. Frequently performing eager CLF could impose high performance cost [31, 32, 33, 34, 35].

[1] We distinguish cache line and cache block in the dissertation. The cache line is a location in the cache, and the cache block refers to the data that goes into a cache line.

Asynchronous CLF removes CLF from the critical path of the application, such that CLF overhead is hidden. Asynchronous CLF can be implemented by a helper thread that performs CLF in parallel with application execution [36]. The effectiveness of asynchronous CLF depends on workload characteristics: if the time interval between CLF and the next memory fence is too short, then asynchronous CLF is not effective, and exposed to the critical path.

Deferred CLF relaxes the constraints of write ordering to improve performance. This method groups data modifications into failure-atomic intervals and delays CLF to the end of each interval. This method ensures data consistency across intervals. Once the system crashes, all or none of the data modifications in the interval become visible. The existing studies determine the interval length based on either a user-defined value [37, 38] or application semantics [23].

Passive CLF relies on natural cache eviction from the cache hierarchy to persist data. Lazy persistence [32] is one such optimization. With passive CLF, the system itself does not trigger CLF. Dirty data is written back to PM, depending on the hardware eviction. In the event of system failure, the system uses checksums to detect inconsistent data and recovers the program by recomputing inconsistent data. Lazy persistency trades CLF overhead with recovery overhead.

Bypassing CLF avoids storing modified data in the cache hierarchy and, instead, writing to PM directly [39, 40]. Specific non-temporal instructions on x86-64 architecture (e.g., `movnti` and `movntdq`) provide such support. Still, fence instructions are used to ensure the update is persisted. Bypassing CLF could avoid the overhead in cache and CLF instructions to gain performance if there is little data reuse in the cache [16].

2.4 Failure-atomic Transaction on NVM

Failure-atomic transactions are a common access interfaces to ensure crash consistency on NVM [20, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53]. Updates in a failure-atomic transaction either all succeed or fail, leaving the data on NVM in a consistent state. We refer to data objects accessed in a transaction as *persistent objects*. NVM transactions are implemented in two major paradigms – logging and copy-on-write (CoW).

2.4.1 Logging-based Transaction

Logging-based transactions can use either *undo-logging* or *redo-logging*. Both logging approaches must write twice to update a persistent object, i.e., update the log and then the data (Figure 2.2a and Figure 2.2b). This in-place update to the data could cause concurrent random writes because transactional workloads could update arbitrary persistent objects.

2.4.2 CoW-based Transaction

CoW-based transactions create a new copy of a persistent object before modifying it (Figure 2.2c). All updates are captured in the new copy, i.e., out-place updates. After persisting updates in the new copy, the system updates the pointer to the persistent object to the new copy and discards the old copy. Hence, CoW transactions write to PM only once. Even when random persistent objects are updated, persisting their new copies laid out sequentially still result in sequential writes.

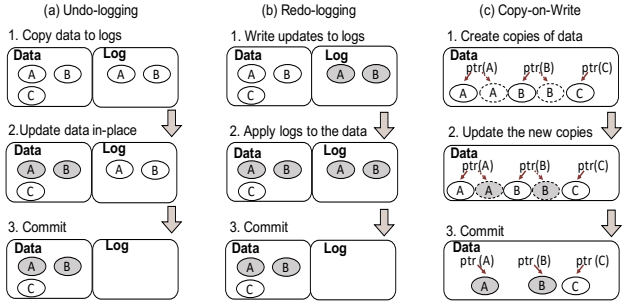


Figure 2.2: Three transaction implementations: undo-logging, redo-logging, and copy-on-write.

2.4.3 Memory Management in NVM Transactions

In logging or CoW paradigms, logs are inserted and removed, or copies of persistent objects are created and deleted in each transaction. Frequent memory allocation and deallocation in concurrent transactions require **scalable** solutions. Also, the persistence in PM imposes unique requirements of **consistency** and **low fragmentation** on memory management.

Scalable memory allocators [54, 55, 56, 57], including state-of-the-art NVM allocators [20, 58, 59], typically implement thread-local free lists and global free lists.

An allocation request is first tried on the requester thread's local free list before being forwarded to the global free list. For a deallocation request, the freed memory block is added to the requestor thread's local free list to avoid synchronization on the global free list. The existing memory allocators usually predefine a set of object size classes. For each size class, the allocator maintains a list of free memory blocks of that size. An allocation request is fulfilled by the list in the nearest size class. Memory fragmentation occurs when the selected size class is larger than the requested size. Unlike volatile memory, fragmentation on NVM has a longer-lasting impact. Volatile memory may restart the program to diminish fragmentation while fragmentation on NVM persists through restarts. Besides, a PM allocator needs to ensure its metadata in a consistent state to avoid data loss and memory leakage after crash.

2.5 MPI Programming Model

Message Passing Interface (MPI) programming model is one of the most widely used parallel programming libraries. MPI provides interfaces for programmers to define inter-processes communication methods (e.g., point-to-point, collective, or one-sided) among multiple processes from different address spaces. Each MPI-based program contains a set of autonomous processes that do not need to run the same program; Following multiple instructions multiple data (MIMD) model, processes distribute and exchange data by message passing.

2.6 Task-Based Parallel Programming Model

Taks-based programming model (e.g., OpenMP tasks, Cilk [60], and Legion [61]) is one type of shared memory parallel programming models. With task-based programming model, the programmer or compiler identifies tasks (code regions) that may run in parallel and annotates the memory footprint of task arguments (i.e., memory addresses of major data objects within tasks). The runtime system for a task-based programming model uses memory footprint information associated with tasks to identify task dependencies and build dependency graphs at runtime. Tasks without dependency can be immediately scheduled for execution on available processing elements; tasks with dependency stay in an internal data structure (e.g., a FIFO

queue) within the runtime, waiting for their dependencies to be resolved. Hence, tasks can be executed out of order by the runtime scheduler without violating program correctness.

Chapter 3

Unimem: Runtime Data Placement on NVM-based Heterogeneous Memory for MPI Programs

3.1 Overview

In this chapter, we introduce a software-based solution to decide and place data objects on NVM-based HMS for MPI programs. Using a software-based solution to manage the data placement on NVM-based HMS has multiple research challenges. First, how to capture and characterize memory access patterns associated with data objects? This question is important for making data placement decisions. As we show in Section 3.3, after we move some data object from NVM with less memory bandwidth to DRAM, there is a big performance improvement. However, we do not have such performance improvement after moving this data object from NVM with longer access latency to DRAM. We claim such data object is sensitive to memory bandwidth. Similarly, we find some data object which is only sensitive to memory latency, or sensitive to both bandwidth and latency. Characterizing data objects based on their sensitivity to bandwidth or latency is critical to model and predict performance benefit of data placement.

Second, how to strike a balance between different requirements on the frequency of data movement (i.e., the implementation of data placement)? On one hand, we want data movement to be frequent, such that data placement is adaptive to variation

of memory access patterns across execution phases. On the other hand, we want to minimize data movement to avoid performance loss.

Third, how to minimize the impact of data movement on application performance? Data movement is known to be expensive in terms of performance and energy cost. Hiding data movement cost and achieving high performance is a key to be successful in the HPC domain.

In this chapter, we introduce a runtime system (named "Unimem") that automatically and transparently decides and implements data placement. This runtime meets the above goals and addresses the above three challenges. In particular, we employ online profiling based on performance counters to capture memory access patterns for execution phases, based on which we characterize the sensitivity of data objects in each phase to memory bandwidth and latency. This addresses the first challenge. We further introduce lightweight performance models, based on which we predict performance benefit and cost if moving data objects between NVM and DRAM. Given the performance benefit and cost of data movement, we formulate the problem of deciding optimal data placement as a knapsack problem. Based on the performance models and formulation, we avoid unnecessary data movement while maximizing the benefits of data movement. This addresses the second challenge.

To avoid the impact of data movement on application performance, we introduce a proactive data movement mechanism. Given an execution phase and a data movement plan for the phase, this mechanism uses a helper thread to trigger data movement before the phase. The helper thread runs in parallel with the application, overlapping data movement with application execution. This proactive data movement mechanism takes data movement overhead off the critical path, which addresses the third challenge. To further improve performance, we introduce a series of techniques, including (1) optimizing initial data placement to reduce data movement cost at runtime, (2) exploring the tradeoff between phase local search and cross-phase global search for optimal data placement, and (3) decomposing large data objects to enable fine-grained data movement. Altogether, those techniques in combination with our performance models greatly narrow the performance gap between NVM and DRAM:

In summary, we make the following contributions.

- We study the performance of HPC workloads with large data sets on multiple nodes with various NVM bandwidth and latency, which is unprecedented. Our study reveals a big performance gap between NVM-based and DRAM-based main memories. We demonstrate the feasibility of using a runtime-based solution to narrow such gap for HPC.

- We introduce a lightweight runtime system to manage data placement without hardware modifications and disruptive changes to applications and system software.

- We evaluate Unimem with six representative HPC workloads and one production code (Nek5000). The performance difference between DRAM-only and HMS with Unimem is only 6.2% on average and 16% at most. We successfully narrow the performance gap and demonstrate better performance than a state-of-the-art software-based solution.

3.2 Definitions and Basic Assumptions

For a parallel application based on MPI, we decompose the application into phases. A phase can be a computation phase delineated by MPI operations; A phase can also be an MPI communication phase doing collective operations, point-to-point communication operations, or synchronization. For a non-blocking communication (e.g., `MPI_Isend`), the MPI communication call is not a phase. Instead, it is merged into the immediately following phase. The communication completion operation (e.g., `MPI_Wait`) is a communication phase.

Furthermore, we target on parallel applications from the HPC domain with an iterative structure. In those applications, each program phase is executed many times. Such parallel applications are very common. As an example, Figure 3.1 depicts a typical iterative structure from CG (an NAS parallel benchmark [62]), which dominates the execution time of CG.

We claim a data object is *bandwidth sensitive*, if there is a big performance difference between placing it in NVM with less memory bandwidth and DRAM. We claim a data object is *latency sensitive*, if there is a big performance difference between placing it in NVM with longer memory access latency and DRAM.

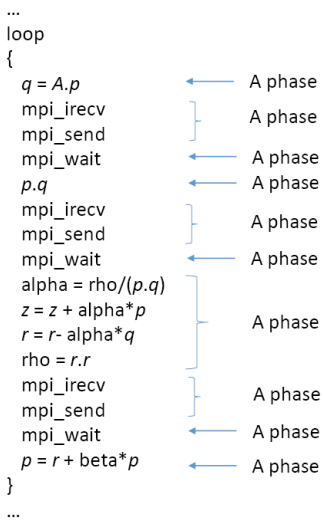


Figure 3.1: A conceptual description for an MPI-based program (CG) decomposed into phases. A is a 2D matrix, and q, p, z, and r are vectors.

3.3 Preliminary Performance Evaluation with NVM-based Main Memory

NVM has relatively long access latency and low memory bandwidth. Based on performance characteristics shown in Table 2.1, we perform preliminary performance study to quantify the impact of NVM on HPC application performance.

We use Quartz, a DRAM-based, lightweight performance emulator for NVM [63] because NVDIMM is not on the market at the time of preparing this manuscript. The existing work uses cycle-accurate simulation to study NVM performance [64, 65]. However, the long simulation time makes impossible simulate *HPC applications* with large data sets on multiple nodes. The performance of HPC workloads on NVM is always mysterious. Using Quartz, we can study performance (execution time) of HPC workloads with much shorter time. We deploy our tests on four nodes in Platform A (the configurations of those nodes and Platform A are summarized in Section 3.6.1). We change the emulated NVM bandwidth and latency, and run a set of NAS parallel benchmarks. We use Class D as input and run 16 MPI processes (4 MPI processes per node). For the benchmark FT, we use CLASS C as input because of the long execution time with Class D. Figures 3.2 and 3.3 show the emulation results.

Observation 1: We find a big performance gap between DRAM-only and NVM-only systems. This observation is contrary to an existing conclusion (i.e., no big gap)

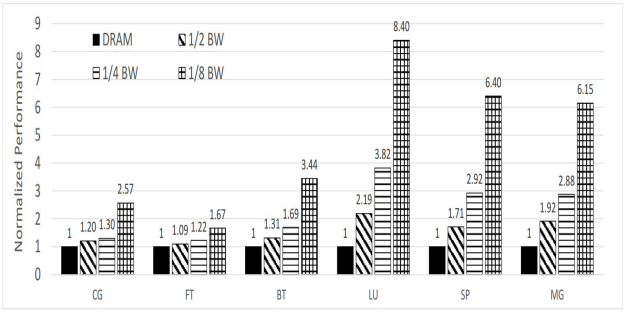


Figure 3.2: The benchmark performance (execution time) on NVM-based main memory (NVM-only) with various bandwidth. The performance is normalized to that of DRAM-only systems.

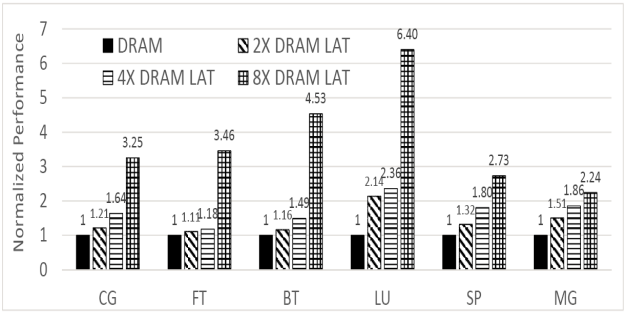


Figure 3.3: The benchmark performance (execution time) on NVM-based main memory (NVM-only) with various latency. The performance is normalized to that of DRAM-only systems.

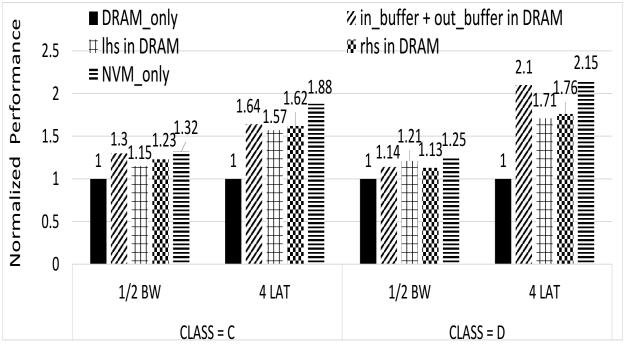


Figure 3.4: The impact of data placement on performance (execution time) of NVM-based main memory. The performance is normalized to DRAM-only systems. The legend entries "in_buffer+out_buffer", "lhs", and "rhs" are the data objects placed in DRAM in the DRAM+NVM system. The x axis shows the configuration of NVM (4x DRAM latency or 1/2 DRAM bandwidth).

for HPC workloads based on a single node simulation [64]. Furthermore, HPC application performance (execution time) is sensitive to different NVM technologies with various bandwidth and latency. With memory bandwidth reduced by only 1/2 or

latency increased by only 2x in NVM, some benchmarks already show a big slowdown. For example, LU has 2.19x and 2.14x slowdown with NVM configured with 1/2 DRAM bandwidth (Figure 3.2) and 2x DRAM latency (Figure 3.3) respectively.

We further study whether data placement in HMS can bridge the performance gap between DRAM-based and NVM-based systems. We choose SP benchmark and focus on four critical data objects of SP (the arrays lhs, rhs, in_buffer and out_buffer). We use two configurations for NVM, one with 1/2 DRAM bandwidth and the other with 4x DRAM latency. For each data object with an NVM configuration (either 1/2 DRAM bandwidth or 4x DRAM latency), we do three tests. In the first test, we use a DRAM-only system. In the second test, we use a DRAM+NVM system. For this test, a target data object is placed in DRAM (see the legend entries in Figure 3.4), while the rest of data objects are placed in NVM. In the third test, we use an NVM-only system. In each test, we use four nodes with one MPI task per node, and use CLASS C and CLASS D as input. Figure 3.4 shows the results. The results are normalized to the performance of DRAM-only.

Observation 2: A good data placement can effectively bridge the performance gap. For example, with the data object lhs placed in DRAM, we bridge the performance gap between DRAM and NVM (using the configuration of 4x DRAM latency and CLASS C) by 31% (see Figure 3.4).

Observation 3: Different data objects manifest different sensitivity to limited NVM bandwidth and latency, shown in Figure 3.4. For example, for the data objects in_buffer and out_buffer (CLASS D), there is no big performance difference (2.1 vs. 2.15) between placing them in DRAM and placing them in NVM configured with 4X DRAM latency; However, there is a big performance difference (1.14 vs. 1.25) between placing them in DRAM and placing them in NVM configured with 1/2 DRAM bandwidth (CLASS D). This indicates that the two data objects are sensitive to memory bandwidth but not memory latency. lhs (CLASS D) tells us a different story: it is sensitive to latency (1.71 vs. 2.15), but not bandwidth (1.21 vs. 1.25). Also, rhs is sensitive to both latency and bandwidth.

Different data objects have different memory access patterns which manifest different sensitivity to bandwidth and latency. A data object with a memory access pattern of bad data locality and massive, concurrent memory accesses (e.g., streaming pattern) is sensitive to memory bandwidth, while a data object with a memory access

pattern of bad data locality and dependent memory accesses (e.g., pointer-chasing) is sensitive to memory latency.

Our preliminary performance study highlights the importance of capturing memory access patterns of data objects. It also shows us that it is possible to bridge the performance gap between NVM and DRAM by appropriately directing data placement on HMS.

3.4 Unimem Design

Motivated by the preliminary performance study, we introduce a runtime system (named "Unimem") targeting on directing data placement on HMS for MPI programs.

Unimem directs data placement for data objects (e.g., multi-dimensional arrays). The data objects must be allocated using certain Unimem APIs by the programmer. We call those data objects, the *target data objects*, in the rest of the proposal. Unimem is phase based. It decides and changes data placement for target data objects for each phase based on runtime profiling and lightweight performance models.

In particular, Unimem profiles memory references to target data objects with a few invocations of each phase. Then Unimem uses performance models to predict performance benefit and cost of data placement, and formulates the problem of deciding optimal data placement as a knapsack problem. The results of the performance models and formulation direct data placement for each phase in the rest of the application execution. We describe the workflow details in this section.

3.4.1 Workflow

Unimem includes three steps in its workflow: phase profiling, performance modeling, and data placement decision and enforcement. The phase profiling happens in the first iteration of the main computation loop of the application. At the end of the first iteration, we build performance models and make data placement decision. After the first iteration, we enforce the data placement decision for each phase. We describe the three steps in details as follows.

Phase Profiling

This step collects memory access information for each phase. This information is leveraged by the second and third steps to decide data placement for each phase.

We rely on hardware performance counters widely deployed in modern processors. In particular, we collect the number of last level cache miss event, and then map the event information to data objects. Leveraging the common sampling mode in performance counters (e.g., Precise Event-Based Sampling from Intel or Instruction-based Sampling from AMD), we collect memory addresses whose associated memory references cause last level cache misses. Those memory addresses help us identify target data objects that have frequent memory accesses in main memory.

Note that the number of last level cache misses can reflect how intensive main memory accesses happen within a fixed sampling interval. It works as an indication for which target data objects potentially suffer from the performance limitation of NVM. However, there are other events that cause main memory accesses, such as cache line eviction and prefetching operations. The current performance counters either do not support counting such event (cache line eviction) or do not have the sampling mode for such event (prefetching operation). Hence, we cannot include those events when counting main memory accesses. However, the last level cache miss accounts for a large part of main memory accesses. It can work as a reliable indicator to direct data placement, as shown in the evaluation section. The last level cache miss is also one of the most common events in modern processors, which makes our runtime highly portable across HPC platforms. To compensate for the potential inaccuracy caused by the limitation of performance counters, we introduce constant factors in the performance models in Step 2.

Performance Modeling

Given the memory access information collected for each phase, we select those target data objects that have memory accesses recorded by performance counters. Those data objects are potential candidates to move from NVM to DRAM. To decide which target data objects should be moved, we introduce lightweight performance models.

General description. The performance models estimate performance benefit

(Equations 3.2 and 3.4.1) and data movement cost (Equation 3.3) between NVM and DRAM. We trigger data movement only when the benefit outweighs the cost. To calculate the performance benefit, we must decide if the data object is bandwidth sensitive or latency sensitive (Equation 3.1). This is necessary to model the performance difference between bandwidth sensitive and latency sensitive workloads.

Bandwidth sensitivity vs. latency sensitivity. To decide if a target data object in a phase is bandwidth sensitive or latency sensitive, we use Equation 3.1. This equation estimates main memory bandwidth consumption due to memory accesses to the data object (BW_{data_obj}).

$$BW_{data\_obj} = \frac{\#data\_access \times cacheline\_size}{\frac{\#samples\_with\_data\_accesses}{\#samples} \times phase\_execution\_time} \tag{3.1}$$

The numerator of Equation 3.1 is the accessed data size. $\#data\_access$ in the numerator is the number of memory accesses to the data object in main memory. $\#data\_access$ is collected in Step 1 (phase profiling) with performance counters.

For a target data object in a phase, the accessed total data size is calculated as ($\#data\_access \times cacheline\_size$).

The denominator of the equation is the fraction of the execution time that has memory accesses to the target data object in main memory. This fraction of the execution time is calculated based on $\frac{\#samples\_with\_data\_accesses}{\#samples}$, which is the ratio between the number of samples that collect non-zero accesses to the target data object and the total number of samples.

For example, suppose that the phase execution time is 10 seconds, the hardware counter sampling rate is 1000 cycles, and the CPU frequency is 1 GHz. Then we will have 10^7 samples in total during the phase execution. Assuming that 10^5 samples of all samples have memory accesses to the data object, then the fraction of the execution time that accesses the data object is $\frac{10^5}{10^7} \times 10 = 0.1s$.

Given a data object in a phase, if its BW_{data_obj} reaches $t_1\%$ of the peak NVM bandwidth BW_{peak} ($t_1 = 80$ in our evaluation), then this data object is most likely to be bandwidth sensitive. The performance benefit after moving the data object from NVM to DRAM (i.e., $BFT_{data\_obj\_bw}$) is dominated by the memory bandwidth effect, and can be calculated based on Equation 3.2, which will be discussed next. If BW_{data_obj} of the data object is less than $t_2\%$ of BW_{peak} ($t_2 = 10$ in our evaluation), then this data object is most likely to be highly latency sensitive. The performance

benefit of moving the data object from NVM to DRAM (i.e., $BFT_{data\_obj\_lat}$) is dominated by the memory latency effect, and can be calculated based on Equation 3.4.1, which will be discussed next. If BW_{data_obj} of the data object is between t_1% and t_1%, then the data object is likely to be sensitive to either bandwidth or latency. The performance benefit after data movement from NVM to DRAM is estimated by $max(BFT_{data\_obj\_bw}, BFT_{data\_obj\_lat})$. To measure BW_{peak}, we run a highly memory bandwidth intensive benchmark, the STREAM benchmark [66], with maximum memory concurrency, and use Equation 3.1 and performance counters.

Calculation of data movement benefit. Equations 3.2 and 3.4.1 calculate performance benefits (after data movement from NVM to DRAM) for bandwidth sensitive and latency sensitive data objects, respectively. The two equations are simply based on an estimation of the performance difference between running the application on NVM and on DRAM. If the data object is bandwidth-sensitive, then the application performance on a specific memory is modeled by $\frac{accssed\_data\_size}{mem\_bw}$ (mem is NVM or DRAM). $accessed\_data\_size$ is
$\#data\_access \times cacheline\_size$, the same as the one in Equation 3.1. If the data object is latency-sensitive, then the application performance on a specific memory is modeled by $\#data\_access \times mem\_lat$ (mem is NVM or DRAM).

$$BFT_{data\_obj\_bw} = (\frac{\#data\_access \times cacheline\_size}{NVM\_bw} - \frac{\#data\_access \times cacheline\_size}{DRAM\_bw}) \times CF\_bw \tag{3.2}$$

$$\text{BFT}_{data\_obj\_lat} = (\#data\_access \times NVM\_lat - \#data\_access \times DRAM\_lat) \times CF\_lat$$

In the above two equations, we have constant factors CF_bw (see Equation 3.2) and CF_lat (see Equation 3.4.1). Such constant factors are used to improve modeling accuracy. To meet high performance requirement of our runtime, the performance models are rather lightweight, and only capture the critical impacts of memory bandwidth or memory latency. However the models ignore some important performance factors (e.g., overlapping between memory accesses, and overlapping between memory accesses and computation). Also, the limitation of the sampling-based approach to count performance events can underestimate the number of memory accesses due to the inability of counting cache eviction and prefetching operations and sampling nature of the approach. The constant factors CF_bw and CF_lat work as a simple

but powerful approach to improve modeling accuracy without increasing modeling complexity and runtime overhead.

The basic idea of the two factors is to measure performance ratios between measured performance and predicted performance for representative workloads, and then use the ratios to improve online modeling accuracy for other workloads.

In particular, we run the bandwidth-sensitive benchmark STREAM to obtain CF_bw offline. We calculate the performance ratio between the predicted performance and measured performance, and such ratio is CF_bw. The predicted performance is calculated based on ($\#data\_access \times cacheline\_size/DRAM\_bw$), where $\#data\_access$ is collected with performance counters using the sampling-based approach. Hence, CF_bw accounts for the potential performance difference between our sampling-based modeling and real performance. The constant factor CF_lat is obtained in the similar way, except that we use a latency-sensitive benchmark, the pointer-chasing benchmark [67] (using a single thread and no concurrent memory accesses). Also, to calculate the predicted performance, we use ($\#dataaccess \times DRAM\_lat$). Given a hardware platform, CF_bw and CF_lat need to be calculated only once.

Calculation of data movement cost. Data placement comes with data movement cost. The data movement cost can be simply calculated based on data size and memory copy bandwidth between NVM and DRAM, which is ($\frac{data\_size}{mem\_copy\_bw}$). To reduce the data movement cost, we want to overlap data movement with application execution. This is possible with a helper thread that runs in parallel with the application to implement an asynchronous data movement. We discuss this in details in Section 3.5. In summary, the data movement cost ($COST_{data\_obj}$) is modeled in Equation 3.3 with the overlapped cost ($mem\_comp\_overlap$) included.

$$COST_{data\_obj} = max(\frac{data\_size}{mem\_copy\_bw} - mem\_comp\_overlap, 0) \tag{3.3}$$

We describe how to calculate $mem\_comp\_overlap$ as follows. To minimize the data movement cost, we want to overlap data movement with application execution as much as possible. Meanwhile, we must respect data dependency and ensure execution correctness. This means during data movement, the migrated data object must not be read or written by the application. Given the above requirement on respecting data dependency and minimizing the data movement cost, we can estimate

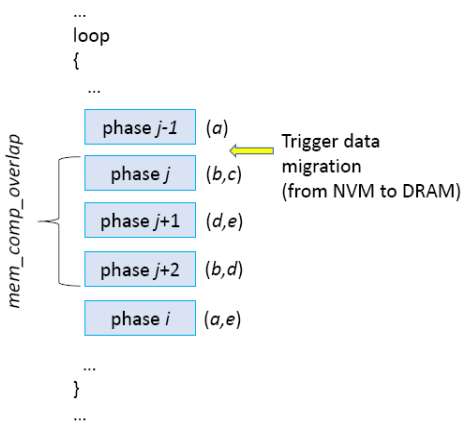


Figure 3.5: An example to show how to calculate *mem_comp_overlap* for the data object a in the phase i. The yellow arrow is the point to trigger the migration of a from NVM to DRAM for the phase i, if a is not in DRAM. The letters in brackets represent target data objects referenced in the corresponding phases.

mem_comp_overlap.

Figure 3.5 explains how to calculate *mem_comp_overlap* with an example. This example shows how to calculate *mem_comp_overlap* for a data object (a) in a specific phase (the phase i). If a is not in DRAM, we can trigger data migration of a as early as the beginning of the phase j, because a is not referenced between j and i. We cannot trigger data migration of a at the beginning of the phase $j-1$, because a is referenced there. *mem_comp_overlap* is the application execution time between the phases j and i. The data movement time, $\frac{data\_size}{mem\_copy\_bw}$, can be smaller than *mem_comp_overlap*. In this case, the data movement is completely overlapped with application execution, and the data movement cost $COST_{data\_obj}$ is 0.

Our estimation on $COST_{data\_obj}$ could be an overestimation (a conservative estimation). In particular, when a data object is to be migrated from NVM to DRAM for a phase, it is possible that the data object is already in DRAM. Use Figure 3.5 as an example again. Since the phase $j-1$ references a, it is possible that a is already in DRAM before the point to trigger the data migration. Also, $COST_{data\_obj}$ does not include the cost of moving data from DRAM to NVM when there is no enough space in DRAM and we need to switch data. Such overestimation and ignorance of data movement from DRAM to NVM are due to the fact that the data movement cost for each phase is isolatedly calculated during the modeling time. Hence, what data objects are in DRAM and whether there is enough space in DRAM is uncertain during the modeling time. We will solve the above problems in the next step (Step

3).

Data Placement Decision and Enforcement

Based on the above formulation for the benefit and cost of data movement, we determine data placement for all phases *one by one*. In particular, to determine data placement for a specific phase, we define a weight w for each target data object referenced in this phase:

$$w = BFT_{data\_obj} - COST_{data\_obj} - extra\_COST_{data\_obj} \tag{3.4}$$

$extra\_COST_{data\_obj}$ accounts for the data movement cost, when there is no enough space in DRAM to move the target data object from NVM to DRAM and we have to move data from DRAM to NVM to save space. To calculate $extra\_COST_{data\_obj}$, we must decide which data object in DRAM must be moved. We make such decision based on the sizes of data objects in DRAM. In particular, we move data objects from DRAM to NVM whose total size is just big enough to allow the target data object to move from NVM to DRAM. Note that since we determine data placements for all phases one by one, when we decide the data placement for a specific phase, we have made the data placement decisions for previous phases. Hence, we have a clear knowledge on which data objects are in DRAM and whether the target data object is already in DRAM.

Besides the weight w, each data object has a data size. Given the DRAM size limitation, our data placement problem is to maximize total weights of data objects in DRAM while satisfying the DRAM size constraint. This is a 0-1 knapsack problem [68].

The knapsack problem can typically be solved by dynamic programming in pseudo-polynomial time. If each data object has a distinct value per unit of weight ($data\_size/w$), the empirical complexity is $O((log(n))^2)$ [68], where n is the number of target data objects referenced in a phase.

The above approach can determine data placement for individual phases. We name this approach as "phase local search". Determining data placement at the granularity of individual phases can lead to the optimal data placement for each phase, but result in frequent data movements, some of which may not be able to

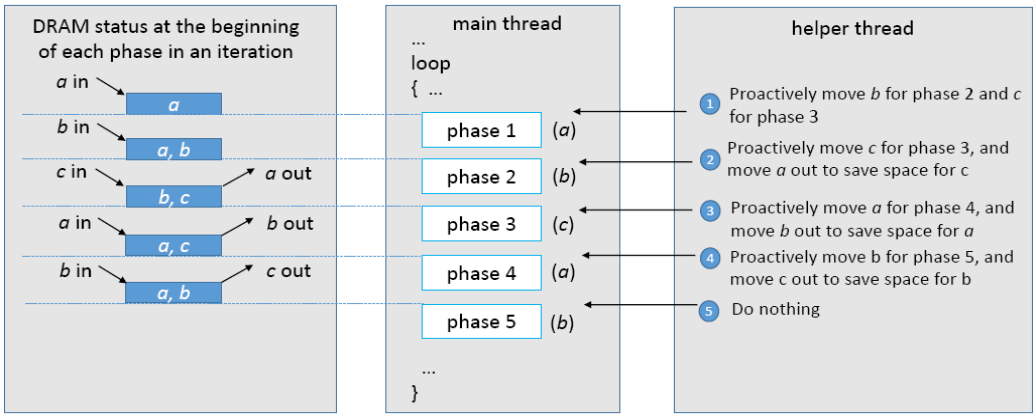


Figure 3.6: An example to show proactive data migration with a helper thread. The letters in the figure represent data objects. The letters in brackets (e.g., (a) and (b)) represent target data objects that are determined to be placed in DRAM for the corresponding phases. DRAM can hold two data objects at most.

be completely overlapped by application execution. Alternatively, determining data placement at the granularity of all phases (named "cross-phase global search") has less data movement than phase local search, because all phases are in fact treated as a combined single phase: Once the optimal data placement is determined within the combination of all phases, there is no data movement within the combination. However, the optimal data placement for the combination of all phases does not necessarily result in the best performance for each individual phase.

Based on the above discussion, we use dynamic programming to *determine the data placement using both phase local search and cross-phase global search*, and then choose the best data placement of the two searches.

After we make the data placement decision at the end of the first iteration, we enforce data placement since the second iteration. At the beginning of each phase, the runtime asks a helper thread (see Section 3.5 for implementation details) to proactively move data objects between NVM and DRAM based on the data placement decision for future phases.

Figure 3.6 gives an example for how to enforce data placement with a helper thread after determining data placement. In this example, there are three target data objects (a, b, and c) and five phases. The data placement decision for each phase is represented with letters in brackets (e.g., (a) for the phase 1). We assume DRAM can hold two data objects at most. The data movement enforced by the helper thread respects data dependence across phases and the availability of DRAM space. Such

example is a case of phase local search, where each phase makes its own decision for data placement. There are eight data movements in total. With a cross-phase global search, only two data objects will be moved to DRAM for all phases. The cross-phase global search results in only two data movements. Based on the performance modeling and dynamic programming, we can decide whether the cross-phase global search or phase local search is better.

3.4.2 Optimization

To improve runtime performance, we introduce a couple of optimization techniques as follows.

Handling workload variation across iterations. In many HPC applications, the computation and memory access patterns remain stable across iterations. This means once the data placement decision is made at the end of the first iteration, we can reuse the same decision in the rest of iterations. However, some HPC applications have workload variation across iterations. We must adjust data placement decision correspondingly.

To accommodate workload variation across iterations, Unimem monitors the performance of each phase after data movement. If there is obvious performance variation (larger than 10%), then the runtime will activate phase profiling again and adjust the data placement decision.

Initial data placement. By default, all data objects are initially placed in NVM and moved between DRAM and NVM by Unimem at runtime. However, data movement can be expensive, especially for large data objects, even though we use the proactive data movement to overlap data movement with application execution. To reduce the data movement cost, we selectively place some data objects in DRAM at the beginning of the application, instead of placing all data objects in NVM. The existing work has demonstrated performance benefit of the initial data placement on GPU with HMS [69, 70]. Our initial data placement technique on NVM-based HMS is consistent with those existing efforts.

For initial data placement, we place in DRAM those target data objects with the largest amount of memory references (subject to the DRAM space limitation). To calculate the number of memory reference for each target data object, we em-

ploy compiler analysis and represent the number of memory reference as a symbolic formula with unknown application information, similar to [71]. Such information includes the number of iterations and coefficients of array access. This information is typically available before the main computation loop and before memory allocation for target data objects. Hence it is possible to decide and implement initial data placement before main computation loop for many HPC applications. However, we cannot determine initial data placement for those data objects that do not have the information available before the main computation loop (e.g., the number of iteration is determined by a convergence test at run time).

Our method determines initial data placement simply based on the number of memory reference and ignores caching effects. The ignorance of caching effects can impact the effectiveness of initial data placement. In particular, some data objects with intensive memory references may have good reference locality and do not cause a lot of main memory accesses. However, our practice shows that in all cases of our evaluation, initial data placement based on compiler analysis makes the data placement decision consistent with the runtime data placement decision using the cross-phase global search. Using compiler analysis can work as a practical and effective solution to direct initial data placement, because the target data objects with a large amount of memory references tend to frequently access main memory.

Handling large data objects. We move data between DRAM and NVM at the granularity of data object. This means a data object larger than the DRAM space cannot be migrated. This problem is common to any software-based data management on HMS.

A method to address the above problem is to partition the large data object into multiple chunks with each chunk smaller than the DRAM size. At runtime, we can profile memory access for each chunk instead of the whole data object, and move data chunk if the benefit overweight the cost of data chunk movement. This method exposes new opportunities to manage data and improve performance.

However, this solution is not always feasible, because it can involve a lot of programming efforts to refactor the application such that memory references to the large data object are based on chunk-based partitioning. A compiler tool can be helpful to transform some regular memory references into new ones based on chunk-based partitioning (assuming the input problem size and number of loop iterations

Table 3.1: APIs for using Unimem runtime

API Name	Functionality
unimem_init	initialization for hardware counters, timers and global variables
unimem_start	identify the beginning of the main computation loop
unimem_end	identify the end of the main computation loop
unimem_malloc	identify and allocate target data objects
unimem_free	free memory allocation for target data objects

are known). However, this kind of automatic code transformation can be impotent for high-dimensional arrays with the notorious memory alias problem and irregular memory access patterns. In Unimem, we employ a conservative approach which only partitions those one-dimensional arrays with regular memory references.

In our evaluation with representative numerical kernels, we find that partitioning large data objects is often not helpful, because making the data placement decision based on chunks leads to much more frequent data movements, most of which are difficult to be overlapped with application execution and hence exposed to the critical path, but we do have a benchmark (FT) benefit from partitioning large data objects.

3.5 Implementation

We have implemented Unimem as a runtime library to perform online adaptation of data placement on HMS. To leverage the library, the programmer needs to insert a couple of APIs into the application. Such change to the application is very limited, and is used to initialize the library and identify the main computation loop and target data objects. In all applications we evaluated, the modification to the applications is less than 20 lines of code. Table 3.1 list those APIs and their functionality.

The runtime library decides data placement at the granularity of execution phase. As discussed before, a phase is delineated by MPI operations. To automatically form phases, we employ the MPI standard profiling interface (PMPI). PMPI_ function behaves in the same way as MPI_ function, but PMPI allows one to write functions that have the behavior of the standard function plus any other behavior one would like to add. Based on PMPI, we can transparently identify execution phases and control profiling without programmer intervention. Figure 3.7 depicts the general

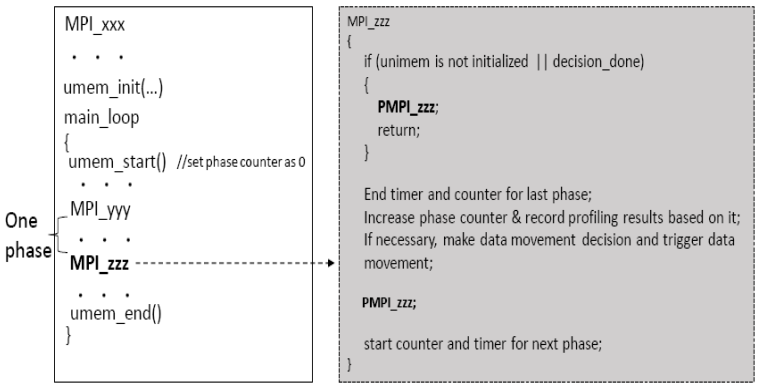


Figure 3.7: Transparently identifying phases based on PMPI.

idea. In particular, we implement an MPI wrapper based on PMPI. The wrapper encapsulates the functionality of enabling and disabling profiling and uses a global counter to identify phases.

To identify target data objects, the programmer must use `unimem_malloc` to allocate them before the main computation loop. This API allows Unimem to collect pointers pointing to target data objects. Collecting those pointers are necessary to implement data movement without asking the programmer to change the application after data movement. In particular, after data movement for a target data object, the runtime changes the data object pointer and makes it point to the new memory space of the data object without disturbing execution correctness. If there is a memory alias to the data object but such alias is created within the main computation loop, then the memory alias can still work correctly, because it is updated in each iteration and will point to the new memory space of the data object after data movement. If the memory alias to the data object is created before the main computation loop, then such memory alias information must be explicitly sent to the runtime by the programmer using `unimem_malloc`, such that the memory alias can be updated and points to the correct memory space after data movement.

The DRAM space is limited in HMS. To manage the DRAM space, we avoid making any change to the operating system (OS), and introduce a user-level service. Each node runs an instance of such service. The service coordinates the DRAM allocation from multiple MPI processes on the same node. In particular, the service responds to any DRAM allocation request from the runtime, and bounds the memory

allocation within the DRAM space allowance. Our current implementation for such service is based on a simple memory allocator without consideration of memory allocation efficiency and fragmentation, because we expect that data movement should not be frequent, and data allocation for data movement should not be frequent for performance reason. However, an advanced implementation could be based on an existing memory allocator, such as HOARD [72] and the lock-free allocator [73].

As discussed in Section 4.3 (see Step 2), we use a helper thread to proactively trigger data movement, such that data movement is overlapped with application execution. The helper thread is invoked in `unimem_init`. In the main computation loop, the helper thread and the main thread interact through a shared FIFO queue. The main thread puts data movement requests into the queue; the helper thread checks the queue, performs data movement, and removes the data movement request off the queue once the data movement is done. At the beginning of each phase, the runtime of the main thread will check the queue status to determine if all proactive data movement for the current phase is done. Hence, the queue works as a synchronization mechanism between the helper thread and the main thread. Note that checking the queue status and putting data movement requests into the queue is lightweight, because we avoid frequent data movement in our design.

As discussed in Section 4.3 (see Step 2), to ensure execution correctness, the runtime must respect data dependency across phases when moving data objects with the helper thread. The data dependency check is implemented by static analysis. We introduce an LLVM [74] pass to analyze data references to target data objects between MPI calls. To handle those unresolved control flows during the static analysis, we embed data dependency analysis result for each branch, and delay data dependency analysis until runtime. The compiler-based data dependency analysis can be conservative due to the challenge of pointer analysis [75]. There is also a large body of research related to the approximation of pointer analysis to improve compiler-based data dependency analysis. However, to simplify our implementation, we currently use a directive-based approach that allows the programmer to use directives to explicitly inform the runtime of data dependency for target data objects across phases. This approach is inspired by task dependency clauses in OpenMP, and works as a practical solution to address complicated data dependency analysis. Figure 3.8 depicts the general workflow.

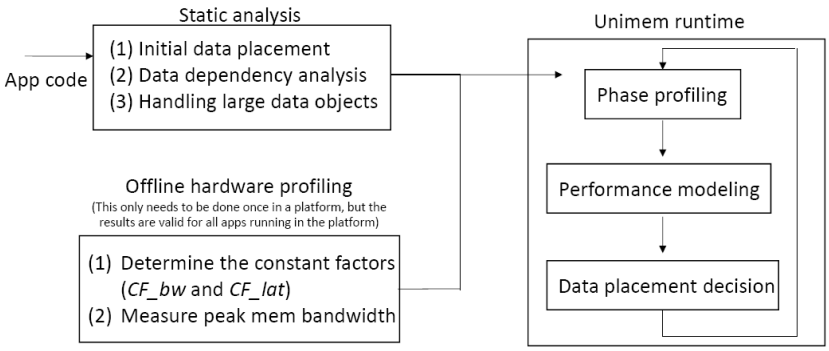


Figure 3.8: The general workflow for Unimem.

3.6 Evaluation

3.6.1 Evaluation Methodology

In our evaluation, we use Quartz emulator [63]. Quartz enables an efficient emulation of a range of NVM latency and bandwidth characteristics. Quartz has low overhead and good accuracy (with emulation errors 0.2% - 9%) [63]. We do not use cycle-accurate architecture simulators because of their slow simulation which cannot scale to large workloads. Furthermore, Quartz allows us to consider cache eviction effects, memory-level parallelism, and system-wise memory traffic, which is not available in other state-of-the-art, software-based emulation approaches [76, 77]. However, due to the limitation of Quartz, we can only emulate either bandwidth limitation or latency limitation, but cannot emulate both of them.

Using Quartz requires the user to have privilege access to the test system. We do not have such privilege access on the test platform for our strong scaling tests. Hence, instead of using Quartz, we leverage NUMA architecture to emulate NVM. In particular, we carefully manage data placement at the user level, such that, given an MPI task, a remote NUMA memory node works as NVM while the NUMA node local to the MPI task works as DRAM. The latency and bandwidth difference between the remote and local NUMA memory nodes emulates that between NVM and DRAM. On our test platform for strong scaling tests, the emulated NVM has 60% of DRAM bandwidth and 1.89x of DRAM latency.

We have two test platforms for performance evaluation. One test platform (named "Platform A") is a small cluster. Each node of it has two eight-core Intel Xeon E5-2630 processors (2.4 GHz) and 32GB DDR4. We use this platform for

Table 3.2: Target data objects in NPB benchmarks and Nek5000

Benchmark	**Target data objects**	**% of total app mem footprint**
CG	*colidx*, *a*, *w*, *z*, *p*, *q*, *r*, *rowst*, *x*	42%
FT	*u*, *u*0, *u*1, *u*2, *twiddle*	99%
BT	*rhs*, *forcing*, *u*, *us*, *vs*, *ws*, *qs*, *rho_i*, *square*, *out_buffer*, *in_buffer*, *fjac*, *njac*, *lhsa*, *lhsb*, *lhsc*	99%
LU	*u*, *rsd*, *frct*, *flux*, *a*, *b*, *c*, *d*, *buf*, *buf*1	99%
SP	*u*, *us*, *vs*, *ws*, *qs*, *rho_i*, *square*, *rhs*, *forcing*, *out_buffer*, *in_buffer*, *lhs*	98%
MG	*buff*, *u*, *v*, *r*	99%
Nek5000(eddy)	Geometry arrays and main simulation variables (48 data objects in total)	35%

tests in all figures except Figure 3.12. We deploy Quartz on such platform. The other test platform is the Edison supercomputer at Lawrence Berkeley National Lab (LBNL). We use this platform for tests in Figure 3.12. Each Edison node has two 12-core Intel Ivy Bridge processor (2.4 GHz) with 64GB DDR3. As discussed before, we perform strong scaling tests and leverage NUMA architecture to emulate NVM on this system.

We use six benchmarks from NAS parallel benchmark (NPB) suite 3.3.1, and one production scientific code Nek5000 [78]. For Nek5000, we use eddy input problem with a 256×256 mesh. The target data objects of those benchmarks are listed in Table 3.2. Those data objects are the most critical data objects accounting for more than 95% of memory footprint except CG and Nek5000. For CG, there are three large data objects (*aelt*, *acol*, and *arow*) only used for problem initialization. They are not treated as target data objects. For Nek5000, we use main simulation variables and geometry arrays in Nek5000 core. Those are the most important data objects for Nek5000 simulation. We use GNU compiler (4.4.7 on Platform A and 6.1.0 on Edison) and use default compiler options for building benchmarks. We use the sampling-based approach to collect performance events on the two platforms. The sampling interval is chosen as 1000 CPU cycles, such that the sampling overhead is ignorable while the sampling is not sparse to lose modeling accuracy.

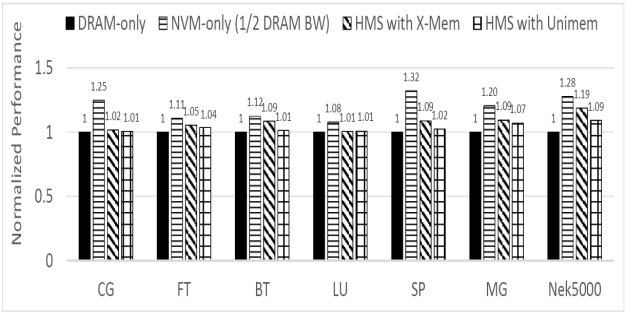


Figure 3.9: The performance (execution time) comparison between DRAM-only, NVM-only, the existing work (X-Mem), and HMS with Unimem. NVM has 1/2 DRAM bandwidth.

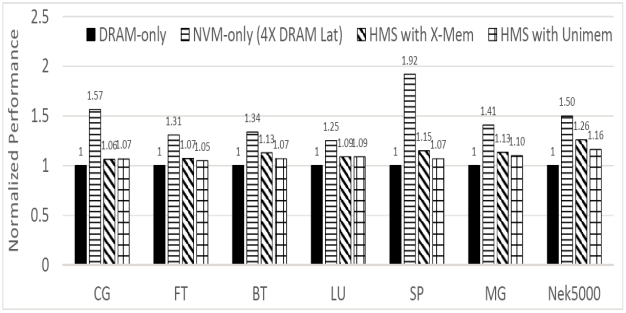


Figure 3.10: The performance (execution time) comparison between DRAM-only, NVM-only, the existing work (X-Mem), and HMS with Unimem. NVM has 4x DRAM latency.

3.6.2 Evaluation Results

The goal of our evaluation is multiple-folding. First, we want to test if our runtime can effectively direct data placement to narrow the performance gap between NVM and DRAM; Second, we want to test if our runtime is lightweight enough; Third, we want to test the performance of our runtime in various system configurations, including different DRAM sizes and different system scales. Unless indicated otherwise, performance in this section is normalized to that of the DRAM-only system.

Basic performance tests. We compare the performance (execution time) of DRAM-only, NVM-only, and HMS with Unimem. We use four nodes in Platform A with one MPI task per node. We use CLASS C as input problem for NPB benchmarks. NVM and DRAM sizes are 16GB and 256MB respectively. Figures 3.9 and 3.10 show the results. NVM is configured with 1/2 DRAM bandwidth (Figure 3.9) or 4x DRAM latency (Figure 3.10).

We first notice that there is a big performance gap between NVM-only and

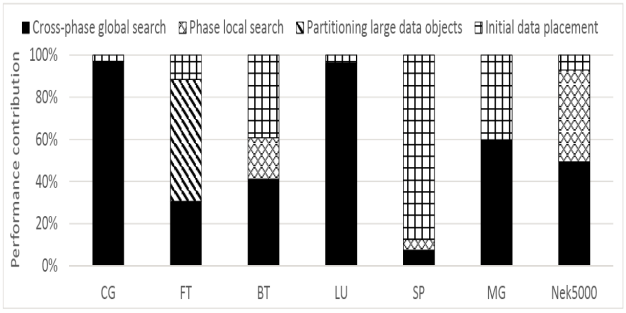


Figure 3.11: Quantifying the contributions of our four major techniques to performance improvement.

DRAM-only cases. On average, the gap is 18% for NVM with 1/2 DRAM bandwidth and 47% for NVM with 4x DRAM latency. However, Unimem greatly narrows the gap and makes performance very close to DRAM-only cases: the average performance difference between DRAM-only and HMS is only 3% for NVM with 1/2 DRAM bandwidth and 7% for NVM with 4x DRAM latency, and the performance difference is no bigger than 10% in all cases. This demonstrates that Unimem successfully directs data placement for those performance-critical data objects. This also demonstrates that Unimem is very lightweight after we optimize runtime performance and hide data movement cost.

We compare Unimem and X-Mem [79] (a recent software-based solution for data placement in HMS). The results are shown in Figures 3.9 and 3.10. X-Mem uses PIN-based *offline profiling* to characterize memory access patterns and make the decision on data placement. They do not consider data movement cost and assume a homogeneous memory access pattern within a data object. The results show that Unimem performs similarly to X-Mem, but performs 10% better than X-Mem for Nek5000. Nek5000 is a production code with various memory access patterns across phases. Unimem adapts to those variations, hence performing better. Also , Unimem does not need any offline profiling for applications.

Detailed performance analysis. Based on the results of basic performance tests, we further quantify the contributions of our runtime techniques to performance improvement on HMS. This quantification study is important to investigate how effective our techniques are and when they can be effective. We study four major techniques: (1) cross-phase global search, (2) phase local search, (3) partitioning large data objects, and (4) initial data placement.

Table 3.3: Data migration details for HMS with Unimem (NVM has 1/2 DRAM bandwidth).

Benchmark	Times of Migration	Migrated data size (MB)	Pure runtime cost	% overlap
CG	3	132	0.5%	66.7%
FT	4	201	1.5%	75%
BT	24	720	1%	87.5%
LU	3	187	1%	60%
SP	9	348	1.5%	66.7%
MG	1	17	2%	100%
Nek5000(eddy)	102	1101	3%	70.6%

We apply the four techniques one by one. In particular, we apply (1), and then apply (2) to (1), and then apply (3) to (1)+(2), and then apply (4) to (1)+(2)+(3). We measure the performance variation after applying each technique to quantify the contribution of each technique to performance. We use the same system configurations as basic performance tests with NVM configured with 1/2 DRAM bandwidth. Figure 3.11 shows the results.

We notice that cross-phase global search can be very effective. In fact, in benchmarks CG and LU, more than 90% of the contribution comes from this technique. However, cross-phase global search could lose some opportunities to improve performance on individual phases, because it uses the same data placement decision on all phases. Using phase local search can complement cross-phase global search. For BT and SP, using phase local search we improve performance by 19% and 5% respectively.

Initial data placement is very useful. In fact, it takes effect on all benchmarks. For SP, it is the most effective approach (87% contribution comes from this technique).

Partitioning large data objects does not take effect except FT, because it introduces very frequent data movement which loses performance. In FT, this technique contributes to 58% performance improvement, while the other three techniques make 42% contribution by manipulating small data objects. In general, by this study , we learn the importance of combining all techniques to maximize performance improvement for various HPC workloads.

To further study the effectiveness of Unimem, we collect some detailed data migration information for HMS with Unimem (NVM has 1/2 DRAM bandwidth). Table 3.3 shows the results. "Pure runtime cost" in the table accounts for the overhead

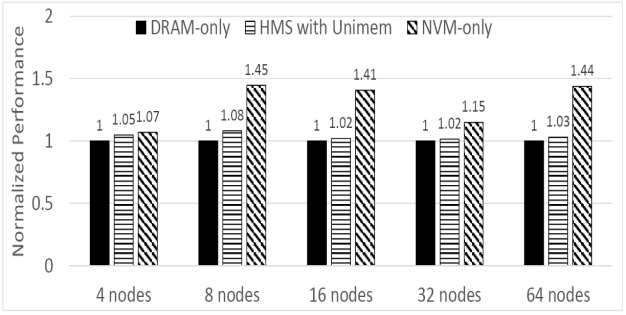


Figure 3.12: CG strong scaling tests on Edison (LBNL).

of collecting hardware counters, modeling costs, and synchronization cost between the helper thread and main thread. "Pure runtime cost" does not include data movement cost and benefit. "% overlap" in the table shows the percentage of data movement cost that is successfully overlapped with the computation.

From Table 3.3, we notice that Unimem has very small runtime overhead (less than 3% in all cases). Directed by Unimem, the data migration can happen very often (e.g., 102 times in Nek5000 and 24 times in BT), and the migrated data size can be very large (e.g., 1.1GB in Nek5000 and 720MB in BT). However, even with the frequent data migration, Unimem successfully overlaps data migration with computation (70.6% in Nek5000 and 87.5% in BT). Also, the performance benefit of data migration outweighs those non-overlapped data migration, and narrows down the performance gap between NVM and DRAM to 9% at most (see Figure 3.9).

Scalability study. To study how Unimem performs in larger system scales. We did strong scaling tests on Edison at LBNL. For each test, we use one MPI task per node and use CLASS D as input problem. We use 256MB for DRAM and 32GB for NVM. Figure 3.12 shows the results for CG. Performance (execution time) in the figures is normalized to the performance of DRAM-only.

As we change the system scale, the sizes of data objects change. The numbers of main memory accesses also change because of caching effects: Such changes in main memory accesses impact the sensitivity of data object to memory bandwidth and latency. Because of the above changes, the runtime system must be adaptive enough to make a good decision on data placement. In general, Unimem does a good job for all cases: the performance difference between DRAM-only and HMS with Uimem is no bigger than 7%.

Sensitivity study. We use various configurations of DRAM size in HMS and

test if our runtime can perform well. As DRAM size changes, we will have different opportunities to place data objects. The change of DRAM size will impact the frequency of data movement and impact whether we should decompose large data objects to improve performance. Figure 3.13 shows the results as we use 128MB, 256MB and 512MB DRAM. In all tests, we use 16GB NVM configured with 1/2 DRAM bandwidth and CLASS C as input problem. We use Platform A and four nodes (1 MPI task per node) to do the tests. In the figure, performance (execution time) is normalized to that of DRAM-only.

In general, Unimem performs well in all cases except one case: the performance difference between DRAM-only and HMS with Unimem is no bigger than 7% in all cases except MG with 128MB DRAM. For MG with 128MB DRAM, we have 13% performance difference between DRAM-only and HMS with Unimem. After careful examination, we find that DRAM is not well utilized, because large data objects cannot be placed in such small DRAM. We also cannot partition large data objects in MG by using our compiler tool because of widely employment of memory alias in the benchmark. But even so, our runtime still narrows performance gap between NVM-only and DRAM-only by 35%.

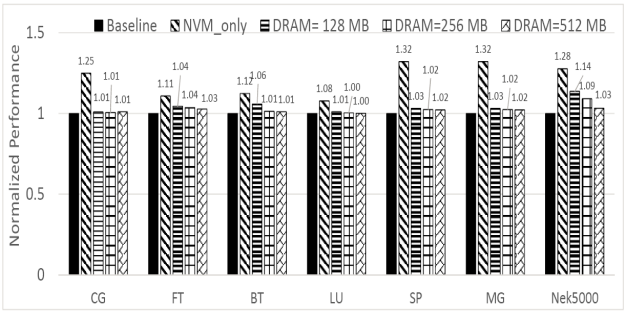


Figure 3.13: Unimem performance sensitivity to DRAM size in HMS.

3.7 Related Work

Software-managed HMS has been studied in prior work. Dulloor et. al [79] introduce a data placement runtime based on *offline profiling* of application memory access patterns. Their work targets on enterprise workloads. To decide data placement, they classify memory access patterns into streaming, pointer chasing, and random.

Giardino et. al [80] rely on OS and application co-scheduling data placement. In particular, they build APIs that allow programmers to describe their memory usage characteristics to OS, through which OS receives and implements responsive page placement and data migration. Lin et. al [81] introduce a protected OS service for asynchronous memory movement on HMS. Du et. al [82] develop a PIN-based *offline profiling* tool to collect memory traces and provide guidance for placing data on HMS.

Different from the prior efforts, our work requires neither offline profiling as in [79, 82] nor programmer involvement to identify memory access patterns as in [80]. Furthermore, our work does not require the modification of OS, which is different from [81]. Our work aims for legacy HPC applications and systems.

Some studies introduce hardware-based data placement solutions for the NVM-based HMS. Bivens et al. [83] and Qureshi et al. [84, 85] use DRAM as a set-associative cache logically placed between processor and NVM. NVM is accessed when DRAM buffer eviction or buffer miss happens. Yoon et al. [86] place data based on row buffer locality in memory devices. Wang et al. [70] rely on static analysis and advanced memory controller to monitor memory access patterns to determine data placement on GPU. Wu et al. [65] leverage the knowledge of numerical algorithms to direct data placement. They introduce hardware modifications to support massive data migration and performance optimization. Agarwal et al. [69] introduce a bandwidth-aware data placement on GPU, driven by compiler extracted insights and explicit hints from programmers.

A key limitation of the above hardware-based approaches is that they heavily rely on modified hardware to monitor memory access patterns and migrate data. Some work, such as [84, 85, 70, 86], ignores application semantics and triggers data movement based on temporal memory access patterns, which could cause unnecessary data movement. Our work avoids any hardware modification, and explores global optimization on data placement.

3.8 Summary

The limitation of NVM imposes a question on whether NVM is a feasible solution for HPC workloads. In this chapter, we quantify the performance gap between NVM-based and DRAM-based systems, and demonstrate that using a carefully designed

runtime, it is possible to significantly reduce the performance gap. We hope that our work can lay the foundation to embrace NVM for future HPC.

Chapter 4

Tahoe: Runtime Data Placement on NVM-based Heterogeneous Memory for Task-Parallel Programs

4.1 Overview

In the previous chapter, we explored the data management on MPI-based programs. With the rapid rise of massive parallelism on the many-cores platform, shared memory programming attracts more attention. In this chapter, we focus on task-parallel programs and introduce a runtime system, Tahoe, to address data placement on NVM-based HMS.

Task-based programming models for building task-parallel programs, such as OpenMP tasks, Cilk [60], and Legion [61], decompose a program into a set of tasks and distribute them between processing elements. Those programming models improve performance by exposing a higher level of concurrency than what is usually extracted by compiler and programmer. Task-based programming models and task-parallel programs have been widely explored in HPC.

Different from the existing performance optimization work for task-parallel programs, deciding on data placement on HMS for task-parallel programs is a new and challenging problem. *First*, the existing work for task-parallel programs [87, 88, 89]

studies task movement (e.g., making a task close to data on a NUMA node), while on HMS, we study data movement. Moving data to DRAM can be beneficial for performance on HMS [5, 65, 79, 80, 86, 90, 91, 92, 93], because of a relatively big performance gap between DRAM and NVM. However, data movement is expensive. As a result, we want to move data that can bring the largest performance benefit among all data and avoid less beneficial data movement. Furthermore, a task can have its data distributed on both DRAM and NVM. Given many possible data distributions for each task and many tasks in a task-parallel program, it is non-trivial to make a decision on data placement.

Second, profiling the memory accesses of tasks to decide data placement is challenging. The existing work commonly uses online performance profiling [5, 94, 95, 96, 97] for HPC applications. Leveraging iterative structures in HPC applications, profiling an execution phase can often make good performance prediction for the future execution phases. This profiling method is based on an implicit assumption that the profiled phase and future execution phases access the same data. Hence, the profiling result in one phase can be used to direct data placement for the same data for the future execution phases. However, this assumption does not hold for task-parallel programs: To enable task level and data level parallelism, different tasks in a task-parallel program often work on different data. No matter which task is profiled, the profiling result for one task is not usable to direct data placement for other tasks, because of the difference in memory addresses and access patterns between tasks. In essence, the execution model of task-parallel programs brings this unique profiling challenge.

In this chapter, we introduce a runtime system, *Tahoe*, to enable efficient data management (i.e., data placement between NVM and DRAM) on NVM-based HMS. Leveraging the *semantics and execution mode* of task-parallel programs, Tahoe efficiently characterizes memory access patterns, decides data placement for many tasks, makes the best use of limited DRAM space, and reduces data movement overhead.

To address the challenge of profiling memory accesses for many tasks without causing expensive overhead, Tahoe chooses a few representative tasks to profile and decide the most accessed pages. Each representative task has similar memory access patterns to many other tasks. To make the memory access information generally applicable to other tasks with different memory pages (addresses), Tahoe leverages

program semantics to transform the information from page level to data-object level, such that other tasks can decide their potentially most accessed pages using data object information.

To decide data placement, Tahoe is featured with a hybrid performance model to predict performance for various data placement cases. The hybrid performance model combines the power of both machine learning modeling and analytical modeling. Predicting the performance of various data placements must capture complicated (possibly non-linear) relationships between execution time and many performance events. A complicated analytical model is possible but would cause large runtime overhead and present challenges in model construction, even for simple data placement cases. We reveal that lightweight machine learning modeling is sufficient to make the prediction for simple data placement cases. However, lightweight machine learning modeling lacks flexibility, as making prediction for complicated data placement cases increases model parameters by 40%. Such a machine learning model is difficult to train and heavyweight for runtime. Analytical modeling does not have this problem because of its flexible parameter setting and formulation. Hence, to predict performance for a task with all of its data placed in one memory (simple data placement cases), we apply a machine learning model. To predict the performance for a task with its data distributed in both NVM and DRAM (complicated data placement cases), we apply a lightweight analytical model based on the machine learning modeling result. In essence, the machine learning model avoids most of modeling complexity, while the analytical model introduces modeling flexibility.

The primary contributions of this chapter are as follows:

- We introduce a runtime system for task-parallel programs to manage data placement on NVM-based HMS;
- We explore how to capture and characterize memory access information for many tasks;
- We use a hybrid performance model to make data placement decisions with high prediction accuracy (the prediction error is less than 7%);
- Evaluating with six benchmarks and one scientific application, we show that Tahoe achieves higher performance than a conventional HMS-oblivious runtime

(24% improvement on average) and two state-of-the-art HMS-aware solutions (16% and 11% improvement on average, respectively).

4.2 Definitions and Basic Assumptions

In this chapter, we use two terms, *task type* and *task size*. We define them as follows. Tasks in a typical task parallel program can run the same code region or different code regions. If some tasks run the same code region with the same input data size, we claim those tasks have the same task type. Those tasks are different *instances* of the same task type. Task size is related to task execution time. A task with a small (or big) size has a short (or long) execution time.

In this chapter, we consider the OmpSs programming model [98], which is a task-based programming model using syntax similar to the OpenMP task pragma. This programming model introduces a dependency clause that allows task arguments to be declared as *in* (for read-only arguments), out (for write-only arguments), and *inout* (for read and written arguments). Our runtime, Tahoe, is an extension of Nanos++ [99], a runtime system that supports OmpSs, OpenMP and Chapel task-based programming models.

Figure 4.1 gives an example code from a benchmark (heat) in the BSC application repository [100] to show task parallel programs. Lines 15-25 are a code region where a task construct (Lines 1-11) is enclosed in a parallel region (i.e., a three-level nested loop). All tasks running the code region have the same task type.

A task in a task-based programming model can be in different execution states. In Nanos++, a task can be in the following four states, and tasks in the state of *ready* are placed in a queue (*readyQueue*). Our runtime leverages the four states to make data migration. We review the four states as follows. (1) *Initialized*: The task is created and dependencies are computed. (2) *Ready*: All input dependencies of the task are addressed. (3) *Active*: The task has been scheduled to a processing element, and will take a finite amount of time to execute. (4) *Completed*: The task terminates, and its state transformations are guaranteed to be globally visible. The task also frees its output dependencies to other tasks.

In this chapter, we migrate data at the granularity of memory pages using *move_pages*() at the user level. In addition, we use the term "data migration" in-

```
#pragma omp task \
    in(([realN]oldPanel)[1;BS][1;BS] ...) out (...)
void jacobi(long realN, long BS, \
        double newPanel[realN][realN], \
        double oldPanel[realN][realN]) {
  for (int i=1; i <= BS; i++) {
    for (int j=1; j <= BS; j++) {
   newPanel[i][j] = 0.25 * (oldPanel[i-1][j] \
          + oldPanel[i+1][j] + oldPanel[i][j-1] \
          + oldPanel[i][j+1]);
} } }

void main(){
...
#pragma omp taskwait
for (int iters=0; iters<L; iters++) {
  int currentPanel = (iters + 1) % 2;
  int lastPanel = iters % 2;
  for (long i=BS; i <= N; i+=BS) {
    for (long j=BS; j <= N; j+=BS) {
      jacobi(realN, BS, \
            (m_t) &A[currentPanel][i-1][j-1], \
            (m_t) &A[lastPanel][i-1][j-1] );
} } }
#pragma omp taskwait
...
}
```

Figure 4.1: Code snippet from a task parallel benchmark (heat).

terchangeably with the term “page migration”; “memory page” and “memory address“ in the rest of the paper refer to “virtual memory page” and “virtual memory address”. NVM endurance is out of the scope of this work and can be handled by memory controllers [101, 102]. Many related works focus on performance, not on NVM endurance [5, 82, 79, 90].

4.3 Tahoe Design

The design goal of Tahoe is to automatically manage data placement (or migrate data) on NVM and DRAM for tasks with minimum runtime overhead. Initially, all data objects (or memory pages) in all tasks are on NVM, but Tahoe moves data objects between NVM and DRAM before task execution to improve task performance. We describe the design of Tahoe in details in this section.

4.3.1 Overview

Tahoe is built with four basic components for data management, including task metadata and profiling, performance modeling, data migration, and DRAM space management. In addition, Tahoe has three optimization techniques for performance improvement. We explain the typical workflow of Tahoe to briefly introduce the four basic components. Figure 4.2 generally depicts the workflow.

Tahoe decides if a task is scheduled to immediately run, based on *task metadata*. Before a task (named as the "target task" in the rest of the discussion) to run, Tahoe determines which memory pages of the task should be migrated from NVM to DRAM. Tahoe makes such decision based on the information provided by the components of *DRAM space management* and *task profiling*.

The task profiling component collects task execution information and memory access information by running representative tasks. A representative task has the same task type as the target task. Using the representative task, we avoid the necessity of profiling every task. The memory access information is collected by performance counters in the sampling mode, such that we can attribute memory accesses to memory pages. The memory access information is compact to enable good performance.

To handle the cases where multiple target tasks are scheduled to immediately run and the DRAM space must be partitioned between those tasks, we introduce a hybrid *performance model* to predict what is task execution time when some memory pages of a task is on DRAM, while other pages of the task are on NVM. Using the performance model, the *data migration* component makes the best use of DRAM for performance improvement.

The DRAM space management component provides information on page residency on DRAM. This component also migrates memory pages from DRAM to NVM based on the recency of task execution. The DRAM space management component ensures that DRAM does not run out of space.

We describe the above components in details as follows.

4.3.2 Task Metadata and Profiling

Our runtime leverages task metadata associated with each task to facilitate data migration. Also, data migration for each task is based on performance profiling on

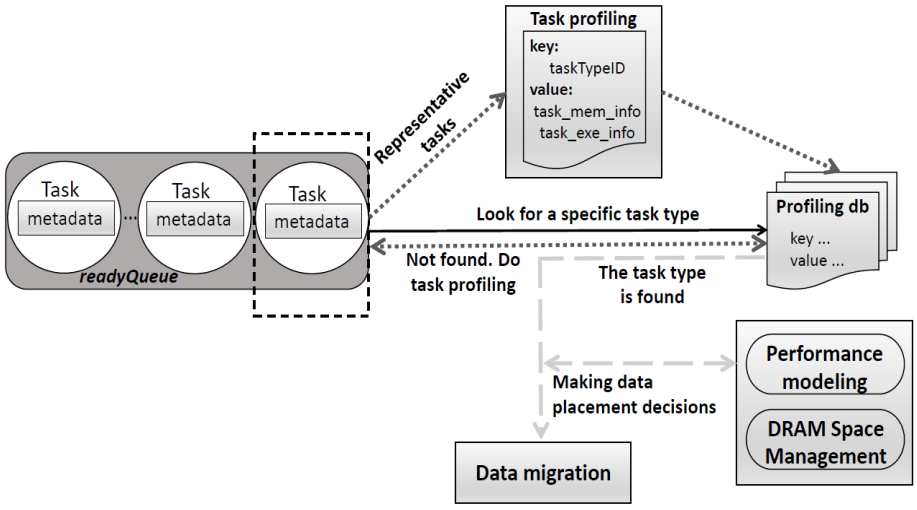


Figure 4.2: The typical workflow of Tahoe

representative tasks. We describe those details in this section.

Task metadata. Task metadata is critical for data migration. In Nanos++, each task has metadata created during task creation. The metadata includes (1) task execution state and (2) data object information for task execution. The data object information includes data addresses (starting addresses) and data sizes for data objects referenced in the task. The data addresses and data sizes information are useful for Nanos++ to identify data dependency between tasks. Nanos++ also has a FIFO queue (i.e., *readyQueue*). This queue saves those tasks that already resolve data dependency and are ready to run.

Tahoe leverages the existing task execution state in Nanos++ to decide when to trigger data migration. A task with the execution state as *Initialized* means that the task has the memory information ready, and Tahoe can use performance modeling (Section 4.3.4) to decide which data should be migrated. A task with the execution state as *ready* is ready to migrate its data, but the data migration must finish before the runtime sets the task as *active*. A task with *completed* state is ready to release its data for migration by Tahoe.

Tahoe leverages the existing data object information in Nanos++ to determine which task should wait because of data migration of other tasks. Tahoe also calculates virtual page numbers by the aligned data addresses and data sizes. The virtual page numbers are needed to profile page-level memory access information for tasks (see below).

Task profiling. To decide which memory pages should be migrated for each task, we must collect task execution information and memory access information for memory pages. The task execution information of a task includes number of instructions, last level cache miss rate, and execution time when all data of the task are on NVM. Such task execution information is necessary for using our performance model (Section 4.3.4). The memory access information includes the number of memory accesses to memory pages of the task.

To collect the above information, Tahoe profiles one instance (i.e., a representative task) of each task type, and then uses the profiling information to direct data placement for the other instances of the same task type. This profiling method is based on the observation that all instances of the same task type often perform the similar computation and have similar memory access patterns.

The task execution information can be easily measured with common performance counters in processors. To collect the memory access information, we use the common sampling mode in performance counters (e.g., Precise Event-based Sampling from Intel or Instruction-based Sampling from AMD). Such a sampling mode allows us to take a sample of a performance event (e.g., last level cache miss or first-level cache hit) every n of such events. The sampling mode allows us to correlate the sample with a memory address whose associated memory reference causes the performance event. Using the memory address and task metadata (particularly data object addresses), we can know which memory page is accessed and which data object is accessed.

The number of last level cache miss can indicate the number of main memory accesses [5, **?**]. Although other events, such as prefetching and cache coherence, can also cause main memory accesses, there is no common method to measure those events. To measure the number of main memory accesses, we use the approach in [**?**] by adding the number of hits in the first-level cache to the number of last level cache misses as main memory accesses, because the first-level cache loads include accesses to prefetched data. Using the above sampling mode, we can estimate the number of memory accesses to all memory pages of a task and decide the most accessed pages.

Profiling overhead analysis. The runtime overhead is an important concern when attributing memory samples to memory pages. In our design, such runtime overhead is small, because of the following reasons. First, the number of representative

Table 4.1: The number of task type for evaluation benchmarks.

FFT	BT	Strassen	CG	Heat	RandomAccess	SPECFEM3D
6	23	10	10	1	1	22

tasks is typically small and each task type has many instances, which means we do not have many pages for profiling. Studying all benchmarks (17 benchmarks) from the BSC application repository [100], we find that the average number of task type per benchmark is 7 (23 at most). We list the number of task types for those benchmarks used in our evaluation in Table 4.1. Each task type can have at least 30, and sometimes more than 1000 instances. Also, we observe that in many benchmarks, each representative task has a small memory footprint (less than a few megabytes) and the size of the memory footprint is independent of the input problem size of benchmarks. Having such task with a relatively small memory footprint is due to the nature of task-parallel HPC programs, which is to decompose computation into many fine-grained tasks and encourage task-parallelism.

Representation of memory access information. To reduce storage overhead of recording the number of memory accesses to each memory page of a representative task and quickly locate the most accessed pages, we use the following method: we coalesce memory pages with continuous virtual addresses and with a similar number of memory accesses (less than 10% difference) into a *memory group*. The number of memory groups in a task is much less than the number of memory pages. The number of memory accesses for each page within a memory group is the average number of memory accesses of all pages within the group. The memory access information is represented as a list of items, each of which includes the number of memory accesses and starting address for either a memory group or a memory page.

The memory access information is collected for the representative task and cannot be directly used by other tasks, because different tasks can use different virtual addresses for their data objects. To solve this problem, we map the memory access information from page level to data object level. Leveraging data semantics, the memory access information at the data object level is generally applicable to any task with the same task type as the representative task.

We use Figure 4.3 to further explain the idea. In this figure, the task i has

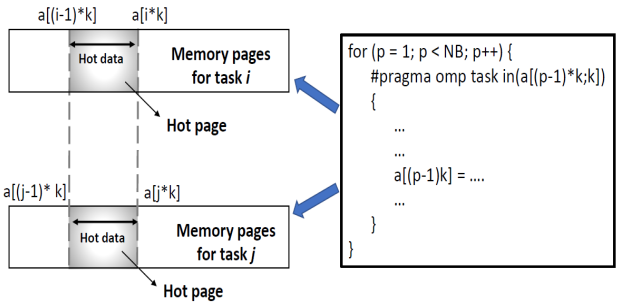


Figure 4.3: Mapping memory access information from page level to data object level.

a memory page frequently accessed. Mapping the memory access information from page level to data object level, we know that this page is filled with elements of a data object $a[]$. Hence, those elements of $a[]$ are frequently accessed. The task j has the same task type as the task i. Based on the profiling information at data object level in task i, we reason that those elements of $a[]$ in task j will be frequently accessed and the corresponding memory page will also be frequently accessed.

Putting it all together. Tahoe maintains a hashmap, named as a *profiling database*. The profiling database uses the task type as the key and the profiling information (task execution information and memory access information) as the value. A task type is represented by a concatenation of the following items: (1) the address of the first instruction in the code region of the task type; and (2) the size of each data object listed in the dependency clauses of the task.

Tahoe picks up tasks from *readyQueue* one by one to decide data placement and run tasks. For each task, Tahoe queries the profiling database to decide if a task with the same task type has been executed before. If not, Tahoe will not make any data migration for the task. Instead, the task will be scheduled to run as usual with its data on NVM. The task is a representative task for any instance of the same task type. The profiling information is collected during the execution of the representative task and saved into the profiling database. If such a type of the task has been executed before, then the profiling information is loaded from the profiling database for deciding data migration and performance modeling.

Similarity of memory access patterns between tasks. Tahoe uses a single task as a representative task for a task type, based on the assumption that all tasks with the same task type have similar memory access patterns. However, we find a couple of cases, e.g., the benchmarks *heat* and *RandomAccess* (see Table 4.4), that

violate the assumption. Nevertheless, the profiling result from the representative task still provides better guidance for data placement than an HMS-oblivious runtime (see Figure 4.4 and 4.5). Also, profiling multiple representative tasks (instead of one) for a task type can make such guidance even more useful.

4.3.3 Data Migration

Whenever there is a processing element ready to run a task, a task at the front of readyQueue will be scheduled to immediately run. Right before the task runs, Tahoe decides which memory pages of the task should be migrated from NVM to DRAM.

We must handle the following issues for data migration.

Deciding which memory pages to migrate. A task can reference many memory pages. Given the limited DRAM capacity, not all memory pages can be migrated. We must decide migrating which memory pages bring the largest performance benefit. We make such decision using two steps.

First, we decide how many memory pages can be migrated. Based on the DRAM space management (Section 4.3.5), we can know which memory pages of the task are already in DRAM. Combining such information with the availability of DRAM space, we can decide how many memory pages can be migrated from NVM to DRAM. Second, based on the profiling information (Section 4.3.2), we decide the most accessed memory pages to migrate and then update the DRAM information in the DRAM space management.

Data migration for multiple tasks. When there are multiple processing elements ready to co-run multiple tasks, we must partition the available DRAM space between the tasks to maximize performance benefit of data migration. We use a performance model to decide the partition.

Assume that we have K tasks to co-run. After deciding the DRAM space partition, a task i ($1 \leq i \leq K$) has m_i pages on DRAM and its performance is $perf_i$. To maximize the system throughput to process tasks, we have the following formulation, where $size$ is the available DRAM space and $Perf$ is the execution time to finish all K tasks:

$$\sum_{i=1}^{K} m_i = size \tag{4.1}$$

$$Perf = \max_{1 \le i \le K} perf_i \tag{4.2}$$

To know $perf_i$, we use a performance model (Equation 4.4). To solve the above equations, we use dynamic programming. To avoid the overhead of dynamic programming, when co-run tasks have the same task type, we evenly partition available DRAM space between co-run tasks without using dynamic programming. This method is based on the observation that tasks with the same task type have similar memory access patterns in most cases.

Handling conflicting decisions on page migration. A memory page can be referenced by more than one task, and multiple tasks can make conflicting decisions on the placement of a page. For such a case, we always place the page on DRAM, because those tasks that decide to place the page on NVM do not lose performance when the page is actually placed on DRAM.

4.3.4 Performance Modeling

Performance modeling is used to decide the DRAM space partition between multiple tasks, when those tasks are ready to be run by multiple processing elements. To achieve the above modeling goal, our performance model aims to predict the performance for a task when a part of its memory pages is on DRAM and the other part is on NVM (i.e., $perf_i$ for task i in Equation 4.2).

Our performance model has two parts. The first part uses a machine learning model to predict the performance of a task with all of its memory pages on DRAM (we name such a case as *complete data placement*). The second part is based on the first one and predicts the performance when some (not all) of the task's memory pages are placed on DRAM (we name such case as *partial data placement*). The second part is an analytical model.

We have the following requirements for our performance modeling. (1) Application generality: the model must work for a large variety of applications; (2) Complexity: the model must be simple enough to have low runtime overhead; (3) Usability: the model must have low programmer involvement; (4) Hardware generality: the model must be easily extensible to different hardware platforms. We describe our performance model in details as follows.

Performance Modeling for Complete Data Placement

We introduce a performance model based on machine learning. We do not use analytical modeling because when capturing the sophisticated relationship between execution time and performance events, the analytical modeling tends to be complex (e.g., [103]). It can bring large runtime overhead and construction difficulty, violating the above requirements (2)-(4).

Given a task, the machine learning model uses the following information as input: (1) last level cache miss rate, and (2) IPC (instructions per cycle) when all memory pages of the task are on NVM. The model outputs (predicts) IPC for task execution when all memory pages of the task are on DRAM. The input of the model can be obtained from the profiling database. In particular, using the task type as a key, we can get the task execution information collected from a representative task from the database. Based on this information, we calculate the model input. With the model output (i.e., predicted IPC), we calculate the task execution time of complete data placement, using the number of instructions obtained from the profiling database.

We choose last level cache miss rate and IPC as the model input, because they are highly correlated with performance variation across different cases of data placement, and hence can serve as important performance indicators. In particular, the last level cache miss rate reflects how intensively main memory is accessed. The performance of an application with a high last level cache miss rate could be sensitive to the change of main memory bandwidth and latency. IPC can reflect main memory access intensity and overlapping between computation and memory access. The performance of an application with high IPC may not be sensitive to the change of main memory bandwidth and latency.

We explore two common supervised machine learning techniques to build our models and meet the modeling requirements: multiple linear regression analysis (LR) and artificial neural network (ANN).

Multiple LR analysis. Our regression model is as follows.

$$y = \beta_1 x_1 + \beta_2 x_2 + \epsilon \tag{4.3}$$

where x_1, and x_2 are IPC and last level cache miss rate, respectively. y is the predicted IPC. β_1, β_2 and ϵ are modeling coefficients we learn through model training.

Table 4.2: Training time and prediction accuracy. NVM bandwidth is 1/x bandwidth of DRAM ($x = 4, 8$, and 16).

	Multiple LR model			**ANN model**		
NVM bandwidth	1/4	1/8	1/16	1/4	1/8	1/16
Average training time per epoch (s)	25.3	23.5	22.4	32.4	31.7	33.8
Total training time (s)	207.2	191.4	195.0	254.9	249.6	262.3
Average prediction error	10.9%	26.4%	45.9%	3.6%	4.1%	5.1%
Prediction error variance	0.2	57.2	4.7×10^3	0.007	0.016	0.017

ANN. A typical ANN has a number of neurons. Each neuron receives inputs from other neurons or ANN input, and produces an output via activation functions. Neurons, connected with weights and organized as layers, constitute the network structure of ANN.

In our model, we use a three-layer, fully-connected ANN containing one input layer with ten input neurons, one hidden layer with five neurons, and one output layer with one output neuron. We use such simple ANN to avoid large runtime overhead when making online performance prediction. We use Rectified Linear Unit (ReLu) as the activation function in our ANN.

Model training and validation. We use seven task parallel benchmarks (see Table 4.4) from the BSC application repository [100] for model training and validation. In particular, we choose every six benchmarks of the seven task-parallel benchmarks to build two models (LR and ANN), and use the one remaining benchmark for validation (training and validation use different data sets). In total, we build seven LR models and seven ANN models for cross-validation. For each model, we have at least three million tasks from six benchmarks for training, and use at least 0.7 million of tasks from one remaining benchmark for validation.

The training data is collected in a machine described in Section 4.4. On this machine, we configure our NVM emulation with three different bandwidth (1/4, 1/8, and 1/16 of DRAM bandwidth). Hence, we have three NVM cases, and for each case, we collect the training data to train the two models (LR and ANN). The average training time of those models is summarized in Table 4.2. Overall, the training time is short less than five minutes for all cases.

Performance modeling accuracy. Table 4.2 shows the prediction accuracy

and reveals that the ANN model achieves high prediction accuracy (less than 6% prediction error on average) for the three different NVM cases. LR, however, does not predict well (e.g., 45.9% prediction error on average, when the NVM bandwidth is configured as 1/16 of DRAM bandwidth). Hence we use the ANN model in Tahoe.

Modeling complexity. Our ANN model is simple. The model training happens offline, and to make a prediction at runtime, the model uses 76 floating point multiplications and 75 floating point additions. As a result, the modeling complexity is low.

In summary, our ANN model meets our modeling requirements: it has good application generality and is simple and usable. The model training time is also short.

However, the machine learning-based performance modeling is not suitable for making performance prediction for partial data placement (more complicated data placement cases), because we have to introduce at least two more input (one for DRAM and the other for NVM) to represent and distinguish cases with different numbers of memory pages on DRAM and NVM and possibly a couple more input to characterize memory access patterns to improve modeling accuracy. This increases model parameters by at least 40% (considering just two more input). Such a model is not only difficult to train but also brings large runtime overhead, which violates the model requirements on complexity, usability, and generality. This problem, in essence, comes from the lack of flexibility to build and use the machine learning model.

Performance Modeling for Partial Data Placement

We introduce an analytical model to make performance prediction for partial data placement. The model uses the prediction result of the complete data placement and uses simple formulation and parameters to capture the performance relationship between complete and partial data placement. The model avoids the problem of model training and concerns on runtime overhead in the machine learning model.

Assume that T_{c_NVM} and T_{c_DRAM} are the execution times with complete data placement on NVM and DRAM, respectively. We have performance difference ($T_{c\_NVM} - T_{c\_DRAM}$), and the performance difference between partial data placement (T_p) and complete data placement on DRAM (T_{c_DRAM}) should be less than ($T_{c\_NVM} - T_{c\_DRAM}$).

In general, more NVM accesses in partial data placement result in a larger performance difference between partial data placement and complete data placement on DRAM. Such a performance difference should be related to the ratio of NVM accesses to total memory accesses (including both DRAM and NVM accesses).

$$T_p = (T_{c\_NVM} - T_{c\_DRAM}) \times \frac{p\_nvm\_acc}{tot\_mem\_acc} + T_{c\_DRAM} \tag{4.4}$$

Equation 4.4 shows the model based on the above rationale and predicts the performance for partial data placement (T_p). T_{c_NVM} in the model is measured and obtained from the profiling database. T_{c_DRAM} is the predicted execution time with the ANN model. $(T_{c\_NVM} - T_{c\_DRAM})$ is the performance difference for complete data placement, which is the largest performance difference we can have. The performance difference for partial data placement scales the largest performance difference by (p_nvm_acc/tot_mem_acc), where p_nvm_acc is the number of NVM accesses in partial data placement and tot_mem_acc is total number of memory accesses in complete data placement. p_nvm_acc is the model input and can be leveraged to explore the performance of various data placement as in Equation 4.2. tot_mem_acc is measured and obtained from the profiling database.

To verify the modeling accuracy, we test the seven benchmarks listed in Table 4.4. We use a machine with two NUMA nodes to emulate NVM based on Quartz [63] emulator. Section 4.4 has more details on our test platform. We do not set the limitation on DRAM size. Both DRAM and NVM can hold all memory pages of the benchmarks. We collect the execution times and the number of memory accesses on NVM and DRAM under three configurations: (1) placing all memory pages on NVM, (2) memory is allocated using a round robin approach on both NVM and DRAM, and (3) placing all memory pages on DRAM. The model makes performance prediction for the second configuration, and uses the first and third configurations as model inputs. We compare the measured time and predicted time for the second configuration, and compute the prediction error shown in Table 4.3. In summary, the prediction error is less than 7%, demonstrating the effectiveness of our model.

Table 4.3: Performance prediction error for partial data placement

Benchmarks	**FFT**	**BT**	**Strassen**	**CG**	**Heat**	**RA**	**SPECFEM3D**
p_nvm_acc	5.7×10^7	1.9×10^8	7.7×10^6	4.3×10^7	5.2×10^7	1.0×10^8	7.4×10^7
tot_mem_acc	1.2×10^8	4.1×10^8	1.6×10^7	7.4×10^7	2.2×10^8	2.7×10^8	1.45×10^8
$\frac{p\_nvm\_acc}{tot\_mem\_acc}$	0.48	0.46	0.48	0.58	0.24	0.37	0.51
Prediction error	6.9%	3.6%	3.0%	1.5%	3.0%	3.0%	6.5%

4.3.5 DRAM Space Management

DRAM space management has two functionalities: (1) recording which memory pages are in DRAM; (2) migrating memory pages from DRAM to NVM when DRAM runs out of space and there is a task pending to execute.

To implement the first functionality, Tahoe represents DRAM pages as a list of memory regions. Each memory region is represented as the starting address and size of the region. Each memory region is a set of memory pages with continuous addresses. If a task wants to check which pages of the task are on DRAM, it sends the address ranges of its memory pages, and compares its address ranges with the address ranges in the list of memory regions.

All memory pages are initially allocated on NVM and no memory page is on DRAM. As memory pages are migrated from NVM to DRAM, DRAM can run out of space, and we must migrate some page from DRAM to NVM to accommodate new memory pages from the upcoming task executions.

To decide which DRAM pages should be migrated to NVM, we could use an LRU policy and migrate those pages that are the least used. However, this would require the runtime to continuously track memory references to DRAM pages, which is costly. To avoid large runtime overhead, we migrate those DRAM pages that are used by the least recently executed task. In particular, Tahoe maintains a FIFO queue with a length of ten to record DRAM memory footprints of the last ten executed tasks. If DRAM runs out of space, DRAM pages referenced by the task at the end of the queue are moved out of DRAM.

In other words, we migrate memory pages from DRAM to NVM based on the recency of task execution, not the recency of memory usage. This method has some limitation, however. A memory page used by the least recently executed task can still be referenced by recently executed tasks, and it is possible that the memory page will be accessed by the upcoming tasks too. To reduce this limitation, before

migrating pages from DRAM to NVM, we quickly examine *readyQueue* to check if the most upcoming task is going to use the pages pending to migrate from DRAM to NVM. We do not migrate those DRAM pages that are going to be used by the most upcoming task.

4.3.6 Performance Optimization

We introduce several techniques to improve performance.

Using helper thread to reduce data migration cost. After migrating data for the task at the front of *readyQueue*, it is possible that DRAM still has space. For such case, Tahoe will proactively migrate data for the task after the front task in the queue. Such proactive data migration is implemented with a helper thread running in parallel with Tahoe, overlapping with task computation and minimizing data migration cost.

Performance optimization for data migration. Calling the page migration function (i.e., *move_pages()*) involves flushing translation lookaside buffer (TLB). Migrating multiple pages of a task with one invocation of *move_pages()* often triggers TLB flush multiple times. TLB flushing is known for causing a large performance overhead [104, 105]. Hence, we combine multiple TLB flushes in one invocation of *move_pages()* into one TLB flush. Such a method reduces the number of TLB flushes, hence improve performance.

Note that an invocation of *move_pages()* only migrates pages for one task, not for multiple tasks, because a task cannot execute until the page migration function finishes. An invocation of *move_pages()* for multiple tasks delays the execution of multiple tasks and reduces system throughput.

Optimization of task scheduling. Tasks with the same parent usually perform the same computation and work on overlapped memory pages. Based on such observation, we slightly change scheduling orders of tasks in *readyQueue*, such that those tasks with the same parent are scheduled one after another. Such a task scheduling strategy maximizes DRAM page reuse before DRAM pages are evicted out of DRAM.

4.3.7 Discussions

NVM has asymmetric memory read and write latencies. However, we do not distinguish memory read and write, because using software techniques (e.g., using `mprotect` to make memory pages read-only and trigger a signal when write occurs) to collect read and write information for pages can be very costly. Most runtime designs for NVM rely on hardware mechanisms [106, 86, 107] to consider latency difference of read and write. The existing runtime solutions [79, 80, 81, 82, 5] do not consider such difference.

When profiling tasks and using performance models, we do not consider performance interferences between tasks. Those interferences can cause cache conflict misses and memory accesses. Due to the dynamic scheduling nature of task parallel programs, quantifying and predicting those performance interferences require runtime to infer possible task execution scenarios, which greatly increases runtime overhead and complicates runtime design. Hence, we do not consider performance interferences in our runtime.

4.4 Evaluation

Experiment methodology. We use a 16-core machine with two eight-core Xeon E5-2630 processors and 32GB DDR4 (two NUMA memory nodes). We use this machine for model training and validation in Section 4.3.4. We use Quartz [63] for NVM emulation. Quartz can emulate NVM with a range of latency and bandwidth, and offer high emulation accuracy. With Quartz, one NUMA node of the machine is used as NVM, while the other is as DRAM.

We use six task parallel benchmarks from the BSC application repository [100] and one production code SPECFEM3D [108]. Table 4.4 summarizes their input parameters and the ratio of the DRAM size (128 MB) to the total size of data objects of each benchmark. For performance profiling, we use the sampling-based approach with sample rate as 1000. Such sampling rate offers high modeling accuracy with tolerable runtime overhead [5].

We use six systems for evaluation: HMS with Tahoe, unmanaged HMS with default Nanos++ (i.e., the HMS-oblivious runtime), DRAM-only (no NVM) with

Table 4.4: Benchmarks for evaluation. Size ratio is the ratio of DRAM size to the total size of all data objects.

Benchmark	Input Parameter	Size ratio
FFT	4096×4096 double matrix	1:15
BT-MZ (BT)	CLASS=C NPROCS=1	1:10
Strassen	4096×4096 double matrix	1:5
CG	4096×4096 double matrix	1:6
Heat (Jacobi)	4096×4096 double matrix	1:10
RandomAccess (RA)	1024MB memory with 1000 tasks	1:9
SPECFEM3D (SF3D)	NEX_XI=128 NEX_ETA=128	1:11

Nanos++, NVM-only (no DRAM) with Nanos++, HMS with X-mem [79], and HMS with Unimem [5]. With the unmanaged HMS, Linux allocates memory with no knowledge of the underlying memory types but is restricted by a limited DRAM size. X-mem and Unimem are two recent software-based solutions for data placement on HMS. X-mem uses offline profiling to characterize memory access patterns and make the decision on data placement. Unimem makes data placement decision at the granularity of execution phases delineated by MPI operations. Because five of our benchmarks do not have MPI, we delineate execution phases by task code regions for evaluating Unimem. Unless otherwise indicated, we use eight threads for evaluation and use the performance of the unmanaged HMS for performance normalization, and NVM is configured with 1/4 DRAM bandwidth. Unless otherwise indicated, we choose 128MB as DRAM size, which is the same as recent work [109, 5, 107, 110]. Such DRAM size is smaller than the total size of all data objects of the benchmarks, such that not all memory pages of the benchmarks are on DRAM. We list the ratio of the DRAM size to the total size of data objects of each benchmark in Table 4.4.

Basic performance tests. We first compare the performance (execution time) of the six systems. NVM has 1/4 DRAM bandwidth (Figure 4.4) or 4x DRAM latency (Figure 4.5).

Using the performance of the unmanaged HMS as the baseline, X-mem, Unimem and Tahoe reduce execution time by 5%, 11% and 21% on average respectively, when NVM has 1/4 DRAM bandwidth. When NVM has 4x DRAM latency, X-mem, Unimem and Tahoe reduce execution time by 10%, 14% and 26% on average, respectively. The unmanaged HMS does not know underlying memory types in HMS. Thus, it does not make good use of DRAM. Tahoe outperforms X-mem and Unimem by

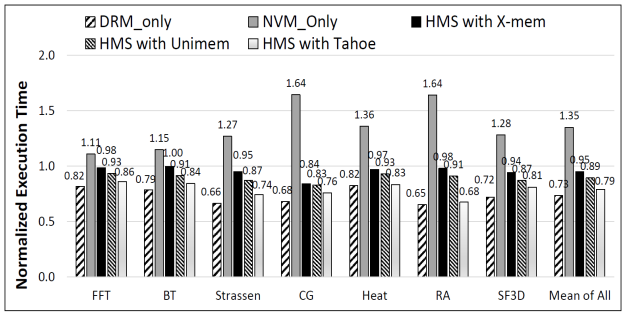


Figure 4.4: Performance (execution time) comparison between unmanaged HMS, NVM-only, X-mem, Unimem and Tahoe. The performance is normalized to that of unmanaged HMS. NVM has 1/4 DRAM bandwidth.

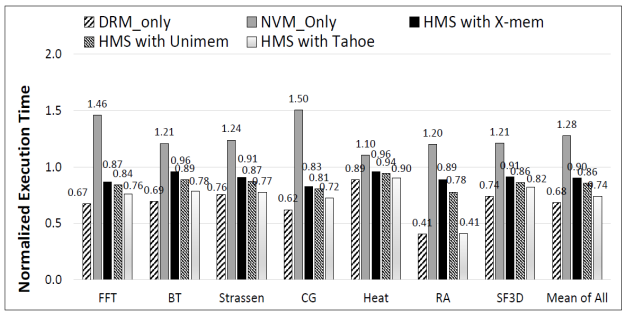


Figure 4.5: Performance (execution time) comparison between Unmanaged HMS, NVM-only, X-mem, Unimem and Tahoe. The performance is normalized to that of unmanaged HMS. NVM has 4x DRAM latency.

16% and 11% on average, respectively. Tahoe performs better than X-mem, because X-mem uses offline profiling and uses the same data placement decision for all tasks. X-mem avoids frequent data movement, but lacks the flexibility of data movement to maximize performance benefit of using DRAM. Unimem does not have the problem of X-mem, but it performs worse than Tahoe, because Unimem lacks a good capability to migrate large data objects from NVM to make best use of DRAM.

Detailed performance analysis. We quantify the contribution of our three optimization techniques to total performance improvement in Figure 4.6. The three techniques are (1) using helper thread for proactive data migration (labeled as "Using helper thread"), (2) performance optimization for data migration (labeled as "Optimized migration"), and (3) optimization of task scheduling (labeled as "Optimized scheduling").

We perform our analysis with the following method. We first remove the three techniques from Tahoe. The performance result of this case is labeled as "Preliminary

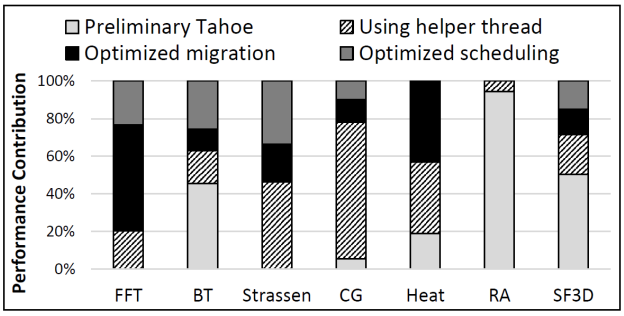


Figure 4.6: Quantifying the performance contributions of the three optimization techniques.

Tahoe". We then compute performance difference between the preliminary Tahoe and unmanaged case. Such performance difference is the performance contribution of the preliminary Tahoe. We then add the three techniques one by one. In particular, we apply (1), and then apply (2) to (1), and then apply (3) to (1)+(2). We measure performance variation for each case. Such performance variation is the performance contribution of each optimization technique. We normalize the performance contributions of all cases by the performance difference between the full-featured Tahoe and unmanaged case.

Figure 4.6 shows the results. We notice using helper thread for proactive data migration particularly works well for CG and Strassen, because the two benchmarks have many tasks with small data sizes. Those tasks cannot make best use of DRAM, hence brings opportunities for proactive data migration. The technique of optimized migration makes big contributions to FFT, because FFT has a relatively large number of page migration requests shown in Figure 4.13. The technique of optimized scheduling makes limited contributions (comparing with other techniques), except in the benchmarks FFT, Strassen, and BT. Those benchmarks often use recursive task parallelism, thus have many tasks with the same parents, which provides opportunities for applying optimized scheduling.

Performance sensitivity analysis. We change NVM bandwidth and latency, number of threads, number of nodes and DRAM size to study how Tahoe responses with the various system configurations.

Figure 4.7 shows the results when NVM has 1/4, 1/8 and 1/16 DRAM bandwidth. Tahoe brings larger performance gains (from 21% to 29%) as NVM bandwidth decreases from 1/4 to 1/16 DRAM bandwidth. This result is especially pronounced

in RandomAccess (not shown in Figure 4.7): The performance gain increases from 32% to 86% as the NVM bandwidth decreases from 1/4 to 1/8 DRAM bandwidth.

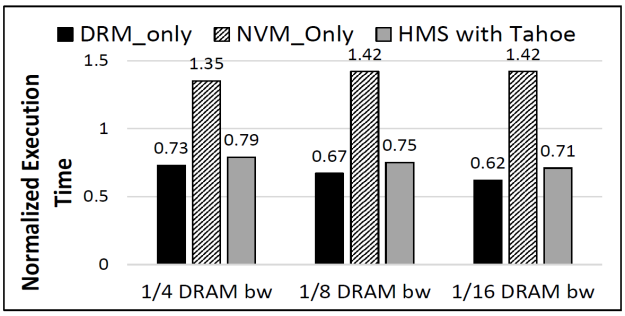


Figure 4.7: Tahoe performance (execution time) sensitivity to NVM bandwidth. The performance is average performance of all benchmarks. Performance is normalized to that of unmanaged HMS.

The performance results are slightly different when we increase NVM latency from 4x to 16x DRAM latency (Figure 4.8). Tahoe has only 4% performance variance when NVM latency increases. The biggest improvement (from 28% to 37%) happens in CG (not shown in Figure 4.8).

When changing the number of threads, our machine can only offer 8 threads at most because of Quartz emulation. To enable better performance study, we use the Edison supercomputer at Lawrence Berkeley National Lab (LBNL). Each node of Edison has two 12-core Intel Ivy Bridge processors (2.4 GHz) with 64GB DDR3 (two NUMA nodes). On this platform, we leverage its NUMA architecture to emulate NVM instead of using Quartz, because Quartz requires the user to have privilege access to the test system, and we do not have such access on Edison. On the Edison nodes, threads run on one processor using the processor's local attached NUMA node as DRAM and the remote NUMA node as NVM. The latency and bandwidth difference between the remote and local NUMA nodes emulates the difference between NVM and DRAM. The emulated NVM has 60% of DRAM bandwidth and 1.89x of DRAM latency. Because of such NVM emulation, the Edison node can offer up to 12 threads.

Figure 4.9 shows the results when we change the number of threads (from 1 to 12 threads) on an Edison node. We only report average performance of all benchmarks, because of limited paper space. Tahoe *performs well consistently* in all cases. In particularly, FFT (not shown in Figure 4.9) has only 2% performance variance when

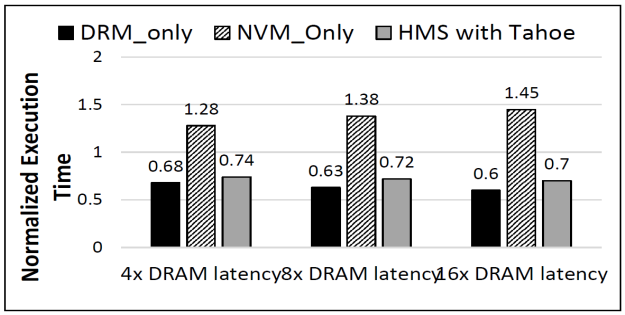


Figure 4.8: Tahoe performance (execution time) sensitivity to NVM latency. The performance is average performance of all benchmarks. Performance is normalized to that of unmanaged HMS.

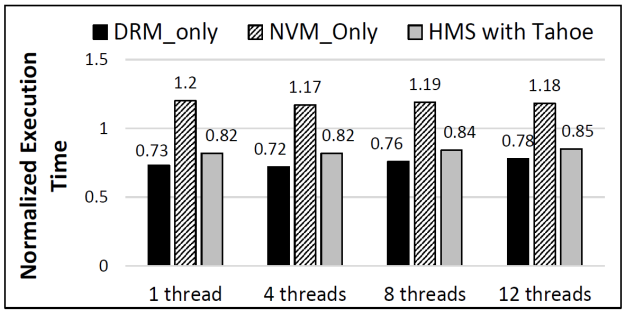


Figure 4.9: Tahoe performance (execution time) sensitivity to the number of threads on a single Edison node. Performance is average performance of all benchmarks. Performance is normalized to that of unmanaged HMS.

we change the number of threads. RandomAccess (not shown in Figure 4.9) has the largest performance difference (only 6%).

Figure 4.10 shows the results when we use different number of nodes (up to 64 nodes). We perform strong scaling tests. We only use BT and SPECFEM3D, because other benchmarks do not have MPI support. For each test, we use one MPI process per node, and each MPI process uses either 4 or 8 threads. For BT, we use CLASS D as input problem; For SPECFEM3D, we use NEX_XI = 256 and NEX_ETA = 128. As the system scale becomes larger, the performance gain of Tahoe decreases from 10% to 3% and from 16% to 4% for BT and SPECFEM3D, respectively (comparing with the unmanaged case), because the memory footprint size per node becomes smaller and more data objects can be placed into DRAM by the unmanaged case. Tahoe performs well in all cases no matter how large the memory footprint size is.

Figure 4.11 shows the results when we change the DRAM size. Overall, Tahoe brings performance benefit in all cases (comparing to the unmanaged case), but as the

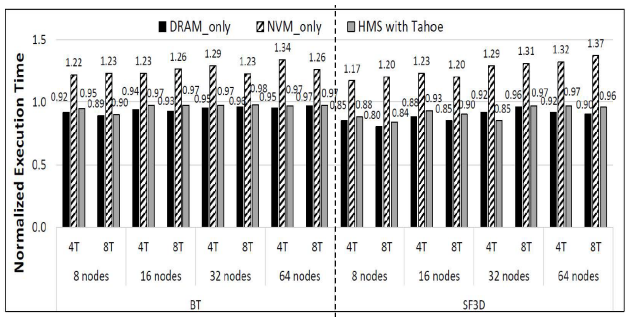


Figure 4.10: Tahoe performance (execution time) sensitivity to the number of nodes on Edison. Performance is normalized to that of unmanaged HMS.

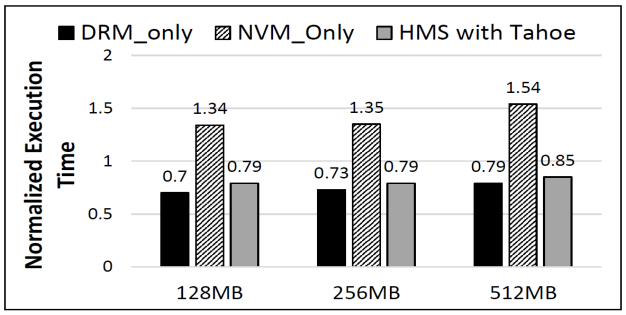


Figure 4.11: Tahoe performance (execution time) sensitivity to DRAM size. Performance is average performance of all benchmarks. Performance is normalized to that of unmanaged HMS.

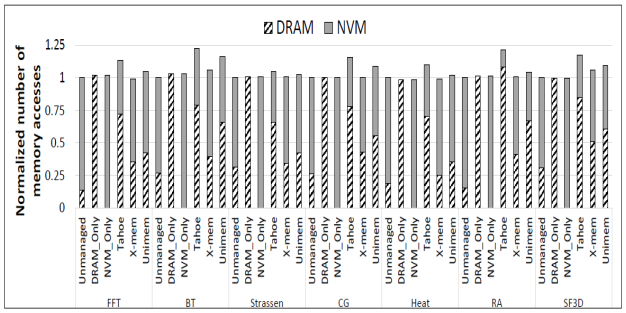


Figure 4.12: Memory access breakdowns. The number of memory accesses is normalized by that of the unmanaged cases.

DRAM size becomes bigger, the benefit decreases from 21% to 15%, because a larger DRAM provides better opportunities to place data on DRAM for the unmanaged case.

Memory utilization analysis. Figure 4.12 shows the number of main memory accesses for DRAM and NVM, normalized by the numbers with the unmanaged cases. Tahoe has larger numbers of DRAM memory accesses than other systems, and hence

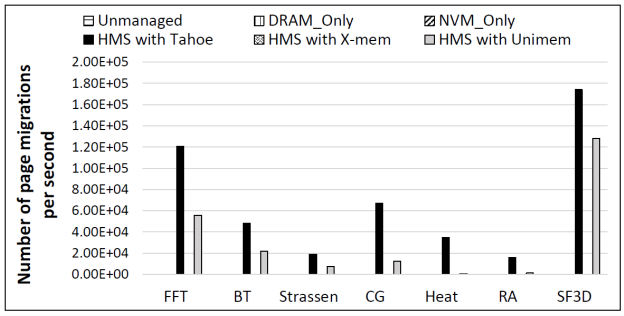


Figure 4.13: Comparing different systems in terms of number of page migrations per second.

effectively utilizes DRAM space. This result is aligned with Figures 4.4 and 4.5, where Tahoe performs consistently better than other systems.

Figure 4.13 shows number of page migrations per sec. The unmanaged and NVM-only do not have page migration. X-mem does not have either, because it is not a runtime solution. The page migration is more frequent in Tahoe than in Unimem, because Tahoe and Unimem work on different data granularities (page vs. data object). The finer-grained data migration as in Tahoe triggers more frequent data migration and makes the best use of DRAM, which transforms to better performance.

4.5 Related Work

Data management on HMS. Software-based solutions are summarized as follows. Du et. al [82] develop an offline profiling tool to analyze memory accesses to guide data placement. Lin et. al [81] introduce an OS service for asynchronous memory movement on HMS. Dulloor et. al [79] introduce a data placement runtime based on classification of memory access patterns. Giardino et. al [80] rely on OS and application co-scheduling data placement. Wu et.al [5] introduce MPI runtime for data placement. Yu et. al [90] propose three bandwidth-aware memory placement policies. Perarnau et.al [111] study data migration performance with user-space memory copy and Linux kernel-based memory migration. They demonstrate the importance of choosing a good ratio of worker threads to migration threads for performance.

Different from the prior efforts, our work does not require offline profiling as in [79, 82] nor programmer involvement to identify memory access patterns as in [80]. Our work also supports data migration for large data objects which is not fully sup-

ported in [5]. Furthermore, our work does not require the modification of OS, which is different from [81, 90]. We do not use user-space memory copy as in [111], because that may involve extensive application modification to use new data addresses after data copy.

Hardware-based solutions are summarized as follows. Yoon et al. [86] dynamically determine data placement based on row buffer locality. Wang et al. [70] use static analysis and memory controller to determine replacement on GPU. Wu et al. [65] use numerical algorithms and hardware modification to decide data replacement. Agarwal et al. [69] introduce a bandwidth-aware data placement on GPU. The major drawback of those solutions is hardware modifications. Some work, such as [84, 85, 70, 86], ignores application semantics and triggers data movement based on temporal memory access patterns, which could cause unnecessary data movement. Our work avoids hardware modification and leverage application semantics.

Performance optimization for task parallel programs. Papaefstathiou et al. [112] modify hardware to prefetch task data and guide the replacement decision in caches. Ni et al. [113] uses a runtime based on Charm++ to prefetch data into fast memory. This work, however, cannot decide optimal data placement for multiple ready tasks. Pan and Pai [114] introduce a runtime to instruct hardware to prioritize data blocks with future reuse. This work needs application and hardware modifications. Li et al. [89] adopt machine learning to estimate scheduling performance for task parallel programs. However, they cannot predict performance for various data placement cases. Our work is different from the existing efforts, and we are the first one to study performance optimization for task parallel programs on NVM-based HMS.

4.6 Summary

Using runtime of a programming model to direct data placement on HMS is promising. In this chapter, we introduce a runtime system for task parallel programs. It leverages task metadata and representative tasks to collect memory access information and make data migration decisions. It uses a hybrid performance model to decide optimal data placement for multiple tasks. Our runtime system effectively uses DRAM space for performance improvement.

Chapter 5

ArchTM: Architecture-Aware, High Performance Transaction for NVM

5.1 Overview

Crash consistency is a primary challenge in using NVM. With NVM, programs can recover their persistent data on NVM even in the event of crashes. However, such a recovery requires a guarantee that persistent data is in a consistent state, a requirement referred as the crash consistency guarantee. Failure-atomic transactions are a popular mechanism to ensure crash consistency. Extensive studies [20, 41, 42, 47, 49, 43, 44, 48, 45, 46, 50, 51, 52, 53] have proposed various transaction mechanisms that generally employ logging-based (undo or redo logging) or Copy-on-Write (CoW)-based designs.

Existing works optimize NVM transactions by reducing data copying [43, 115, 116, 117] or persistence overhead [30, 118, 119, 117, 120, 121]. They emulate NVM based on DRAM with increased memory latency or reduced bandwidth, but miss NVM architecture details. In this study, we focus on the implications of real NVM architecture (i.e., Intel Optane PM) on transaction performance. Our performance analysis on state-of-the-art NVM transaction systems identifies that the NVM micro-architecture, such as internal buffers and data block size, has significant impacts on transaction performance. The mismatch between the transaction implementation and

NVM architecture can cause 3x-58x slowdown, compared to an architecture-aware implementation.

Performance characterization of NVM architecture leads us to rethink the design of NVM transactions. Logging-based transactions have a double write problem because of creating logs and updating data in-place. The excessive writes to NVM mismatch with poor write performance on NVM. CoW-based transactions avoid this problem, but suffers from performance overhead due to metadata updates, which causes many small writes misaligned with NVM internal block size.

Therefore, high-performance NVM transactions call for new design principles tailored to the characteristics of the emerging NVM architecture, which is distinctive from conventional block devices and more than just a slower DRAM. We introduce two design principles customized to NVM architecture.

- Avoid small (less than 256 bytes) writes to NVM. Small writes in NVM suffer from write amplification because data in a small write must be aligned with the internal write block size (256 bytes) in NVM, which wastes memory bandwidth and delays transactions. Our characterization study reveals that in state-of-the-art NVM transaction systems (one in PMDK [20], Romulus [46], DUDETM [122], and an Oracle transaction system [48]), more than 78% of data objects are smaller than 64 bytes, when the transaction systems perform write operations on 512-byte persistent objects. The main source of those small data objects comes from metadata for transaction runtime state, memory allocation and object mapping.

- Encourage coalescable writes. Sequential write performs much faster than random write on NVM (e.g., for 64-byte writes, sequential write is 3.7x faster than random write). Multiple sequential writes can be coalesced in an internal buffer of Optane, enabling high performance.

We follow the above principles in ArchTM. uses a CoW-like design to avoid the double write problem in logging-based transactions. To avoid small writes, stores metadata of memory allocator and data objects on DRAM to reduce frequent small random writes to NVM. However, such a design suffers from a fundamental tradeoff between performance and crash consistency. In particular, metadata on DRAM,

although leading to high transaction performance can be lost when a crash happens, leading to a problem of identifying crash consistency of data objects.

The above problem is caused by the fact that metadata is the only connection between the transaction state and data objects for crash recovery. Such a connection is not NVM-oriented. Removing it causes isolation between transaction state and data objects. To address this challenge, ArchTM introduces a lightweight annotation mechanism. This mechanism adds data object metadata (object ID and size) and transaction ID into the data object, and adds transaction ID into the transaction metadata (i.e., the transaction state variable). The transaction ID is persistent and sets up an alternative connection between data objects and the transaction state. Using the transaction ID, the data object ID and size, ArchTM can easily locate data objects and identify their crash consistency after a crash.

To encourage coalescable writes, ArchTM makes best efforts to allow consecutive memory allocation requests to get contiguous memory allocations. This strategy is based on the observation that in a transaction, data objects that are allocated consecutively are likely to be updated together. For example, in a key-value store system, memory allocation requests for a key data object and a value data object associated with the key often happen together. Writes to the key and value data objects happen in sequential and continuous order. Hence, allocating the key and value contiguously in the address space likely results in coalescable write.

However, to implement the above strategy, we must re-examine the traditional wisdom for memory allocation. The existing memory allocators typically use multiple free lists for each thread. Each free list supports allocation requests for specific sizes. Such size-class-based memory allocation is used to reduce memory fragmentation. However, it allocates noncontiguous memory blocks to consecutive memory allocation requests if they are fulfilled by multiple free lists. Hence, there is a fundamental tradeoff between *allocation locality* and memory fragmentation.

To break this tradeoff and encourage coalescable writes, ArchTM uses a single free list and a lightweight online defragmentation mechanism. In particular, ArchTM supports locality-aware data path using the single free list for allocation and uses a recycle list to collect and merge freed memory blocks. For defragmentation, ArchTM aggregates data objects in highly fragmented memory regions to create large and contiguous memory blocks.

By designing and implementing ArchTM, we make the following contributions:

- We reveal the performance characterization of realistic NVM hardware and pinpoint the performance problems in the representative NVM transactions. Such problems are caused by the negligence of the characteristics of NVM architecture in traditional NVM transaction designs.
- We identify two fundamental tradeoffs to enable high performance NVM transactions. We introduce a new NVM transaction design, ArchTM, customized to the NVM architecture and breaking the tradeoffs.
- ArchTM beats state-of-art NVM transaction systems PMDK, Romulus, DUDETM and the Oracle system by 58x, 5x, 3x and 7x on average, using micro-benchmarks and real-world workloads on NVM hardware.

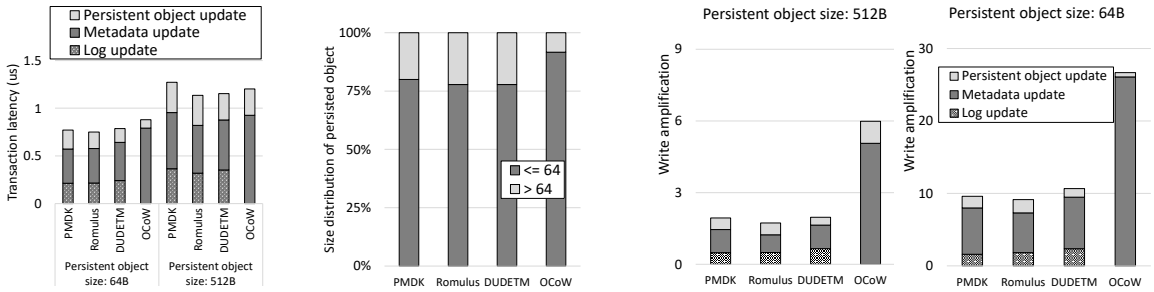
5.2 Performance Characterization

In this section, we study the performance of NVM transactions and Optane DC PM to gain insights for our design.

5.2.1 Transaction Performance Study

We study four representative NVM-based transaction systems: PMDK [20], Romulus [46], DUDETM [122], and one from Oracle [48]. PMDK uses undo-logging, Romulus and DUDETM use redo-logging, and the Oracle system (denoted as $OCoW$) uses CoW. The specification of our Optane platform is in Section 5.5. We focus on write operations because they are the most expensive transaction operation, and writes to NVM are expensive. A write operation in a transaction needs to update persistent object, log (if logging-based), and metadata. Figure 5.1a shows the latency breakdown of a write operation in NVM transactions. We report the performance on small (64-byte) and large (512-byte) persistent objects. The figure shows that most time is spent on log updates or metadata updates.

We instrument the APIs used to persist data objects (e.g., *pmemobj_persist*() in PMDK) to study the performance of write operations. The APIs use the starting address and size of the data objects as input. Figure 5.1b reports the distribution

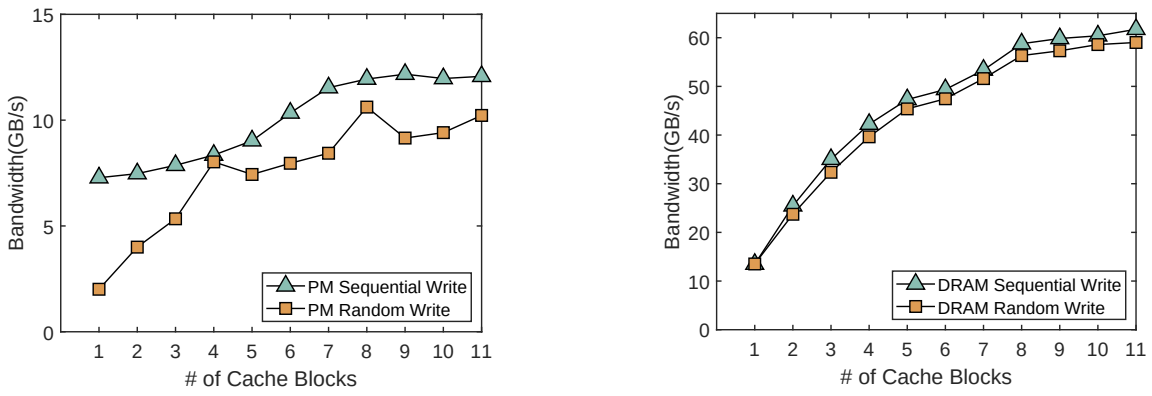


(a) Latency breakdown (b) Size distribution (c) Write amplification

Figure 5.1: Performance characterization of write operations in NVM transactions.

of the *persisted* data size in transactions that perform write operations on 512-byte persistent objects. The figure reveals that more than 78% of persisted objects are smaller than 64 bytes, i.e., a lot of small writes on NVM. Furthermore, we study write amplification, quantified as the ratio between write traffic in NVM measured by performance counters and the number of bytes modified by transactions. Figure 5.1c reports the write amplification in transactions that perform write operations on 64- and 512-byte persistent objects. All systems exhibit write amplification, inflating NVM write traffic by 1.8x - 27x.

Performance analysis. *We find that the metadata updates are the primary source of small writes.* In general, transaction systems have four types of metadata: metadata for transaction runtime, metadata for memory allocation, log metadata, and metadata for persistent objects. Metadata for transaction runtime records transaction status, e.g., COMMIT or ABORT, and transaction IDs. Metadata for memory allocation has information about memory consumption. Log metadata has information on logs (e.g., the indexing of log records), and is unique in logging-based transactions. Metadata for persistent objects store pointers to the new or old copy of persistent objects, and is unique in CoW-based transactions. By design, CoW-based systems have more metadata updates than logging-based ones. For instance, OCoW has about 270% more metadata updates than the other three logging-based systems. For each update, a CoW-based transaction must allocate a new data copy, remap pointers to the data, and deallocate the old data copy. This process generates frequent small writes to metadata for memory allocation and persistent objects.



(a) Optane PM (b) DRAM

Figure 5.2: Sequential and random write bandwidth at different write sizes on Optane PM and DRAM.

5.2.2 Performance Study of NVM Writes

We study the write performance on Optane DC PM using a microbenchmark that performs random and sequential writes. Each write is followed by cache line flushes to persist to Optane PM. Various write sizes, ranging from one to 11 cache lines, are tested. Figure 5.2 reports the bandwidth of performing 100M writes using 24 threads on Optane PM and DRAM. We have the following observations and insights for high-performance NVM transactions.

Figures 5.2a and 5.2b show that write bandwidth of Optane PM is significantly lower than that of DRAM. On our system, write bandwidth to DRAM reaches 60 GB/s but only 13 GB/s to Optane PM. Furthermore, on Optane PM, the peak write bandwidth is 13 GB/s, 3x lower than the peak read bandwidth. These results are consistent with the existing work [12]. Hence, **reducing write traffic on NVM is critical** for high-performance transactions. The logging-based transaction systems need to write data twice to update a persistent object, which causes excessive write traffic.

Figure 5.2a shows that small random writes on Optane PM perform worse than sequential writes. When writing only 64 bytes (Figure 5.2a), random write merely achieves 25% of the bandwidth of sequential write. This performance gap is caused by the 256-byte Optane internal granularity and write amplification, and the gap reduces when the write size increases. The logging-based transactions update persistent objects in-place. This could result in random writes, because persistent objects in a

transaction can be randomly distributed on Optane PM. Using out-place updates, as in CoW-based transactions, can enable sequential writes because the new copies of persistent objects are manageable and can be laid out contiguously in Optane PM.

Figure 5.2a shows that the random writes on Optane PM have performance spikes at write sizes that are a multiple of 256 bytes, e.g., four and eight cache lines. In contrast, random writes on DRAM (Figure 5.2b) exhibits no such pattern. Such performance on Optane PM is due to the effect of the write combining buffer. It buffers and combines 64-bytes stores into a 256-byte internal store. Small simultaneous writes to contiguous address space are more likely to be combined into one internal store than small writes to arbitrary addresses. Therefore, increasing the probability of concurrent writes to contiguous address space can increase the opportunity to **leverage the combining buffer** hardware to coalesce writes inside the Optane PM.

5.3 Design Principles and Major Techniques

Driven by the performance characterization and analysis of existing NVM transactions and the real NVM hardware (i.e., Optane PM), we introduce two design principles and five techniques in ArchTM for high-performance architecture-aware transactions.

- **Avoid small writes on NVM.**

  (1) Logless. ArchTM favors the CoW mechanism to reduce write traffic to NVM.

  (2) Minimize metadata modifications on NVM with guaranteed crash consistency. ArchTM keeps transient metadata on DRAM to avoid frequent metadata modifications on NVM. Also, ArchTM introduces an annotation mechanism to connect the persistent transaction state with data objects. From the transaction state of data objects, ArchTM can detect the consistency of data on NVM and recover from a crash.

  (3) Scalable persistent object referencing. ArchTM uses a scalable object lookup table on DRAM to quickly locate the latest copies of persistent objects in concurrent transactions.

- **Encourage coalescable writes**.

  (4) Consecutive allocation requests get contiguous memory blocks. ArchTM supports a locality-aware data path for small memory allocations to encourage sequential writes in transactions.

  (5) Avoid memory fragmentation. ArchTM employs a lightweight online memory defragmentation technique that examines memory usage by regions and reduces fragmentation on NVM.

5.3.1 Logless

ArchTM employs a CoW-like mechanism to reduce write traffic to NVM. Upon an update request, ArchTM creates a new copy of the persistent object and applies updates to the new copy. The out-of-place update in CoW reduces the number of NVM writes. When committing the new copy to NVM, consecutive writes into contiguous memory addresses increase the possibility of writes coalesced at the combining buffer. However, naively adopting CoW incurs excessive metadata updates on NVM due to object remapping and allocation management (Section 5.2.1). We address this challenge by maintaining metadata on DRAM.

5.3.2 Minimize Metadata Modification on NVM

ArchTM places the memory allocation metadata on DRAM. It does not record memory allocation and reclamation into logs on NVM as in previous NVM transaction systems [20, 47, 46, 21, 58]. Also, ArchTM avoids modifying the persistent object metadata on NVM by using an *object lookup table* on DRAM. This lookup table is used to locate the latest copy of a persistent object quickly. Existing CoW-based implementations [48] must modify the persistent object metadata on NVM to update the pointer to the object to the new copy (Figure 2.2c). With these metadata in DRAM, ArchTM reduces small NVM writes and accelerates the lookup, but cannot ensure crash consistency. ArchTM introduces an annotation mechanism to guarantee crash consistency.

Annotation. ArchTM annotates a transaction by adding a transaction ID into the transaction metadata (the transaction state variable). The embedded transaction

ID is persisted immediately when the transaction state changes to *start*. ArchTM also annotates a persistent object by adding the object information, i.e., object ID, object size, and transaction ID, into the object header on NVM when the object is created. During the recovery from a crash, ArchTM uses the object ID and size to identify each persistent object on NVM. Then, ArchTM uses the annotated transaction ID to identify the most recent copy of a persistent object, recycle the stale copies, and discard uncommitted modifications.

5.3.3 Scalable Object Referencing

ArchTM uses an object lookup table to find the critical information, such as the location of the latest copy of a persistent object. The table is indexed by persistent object IDs. When a persistent object is allocated, the allocator thread gets an object ID and populates the corresponding entry in the lookup table. Multiple threads can reference persistent objects from the table concurrently and efficiently because DRAM supports higher bandwidth than NVM.

The object lookup table is essential for high-performance transactions. Compared to decentralized object referencing [52, 48], the object lookup table in ArchTM resides on a contiguous DRAM space, which brings convenience for management (e.g., checkpointing) and migration. If the DRAM space is insufficient to store the whole lookup table, the spilling part of the table is placed on NVM. Compared with general concurrent index data structures, such as hash tables, our object lookup table is easy to implement and has no synchronization overhead. The competition between threads to get an entry from the lookup table cannot happen, because threads are assigned with disjoint sets of object IDs and hence update disjoint sets of table entries. The object lookup table can find the object metadata in one step because it uses the object ID as the index of the table, which differs from other indexes (e.g., hash table and B-trees) that require additional calculations or queries to find object metadata.

5.3.4 Contiguous Memory Allocations

ArchTM customizes memory allocation and reclamation for transactional workloads on NVM to maximize the possibility of sequential writes. Small allocations are the main optimization focus because sequential writes benefit small objects more than

large objects (See Figure 5.2a). In ArchTM, there are two data paths for persistent object allocation and reclamation: (1) a regular data path for large allocations and reclamations, similar to existing allocators like JEMalloc [54]; and (2) a locality-aware data path for small allocations. The latter optimizes through a single free list and global recycling procedure.

A single free list is used in ArchTM for allocating objects of various sizes. Existing approaches [20, 54, 55, 56, 57, 58, 59] use multiple free lists, each for a different allocation size. Multiple free lists could cause consecutive allocation requests of different sizes to go to different free lists. Consequently, those requests get noncontiguous memory allocations, and writing to them leads to nonsequential writes to NVM. Instead, using a single list of freed segments in sorted order would encourage consecutive requests to get sequential allocations. To maximize concurrency, ArchTM assigns each thread with a dedicated portion from the global free list (Section 5.4.1).

Recycle and merge memory blocks globally. Current approaches [20, 54, 56, 57, 58, 59] return freed memory blocks to thread-local free lists directly. This procedure avoids synchronization on managing a global free list but may harm the locality of freed memory blocks. Free memory blocks in a free list may be noncontiguous so that consecutive allocation requests get noncontiguous allocations. ArchTM runs a helper thread to collect and merge freed blocks from threads. These freed blocks are sorted and merged into a global recycle list before returning them to the global free list. The global recycling procedure does not happen in the critical path and does not affect the efficiency of memory deallocation.

5.3.5 Reduce Memory Fragmentation

Using a single free list for various allocation sizes could result in memory fragmentation. ArchTM uses a 64-byte size class in the memory allocator. An allocation smaller than the size class gets rounded up. We choose this size class to avoid false sharing in cache lines.

ArchTM introduces an online defragmentation mechanism to reduce memory fragmentation. The mechanism monitors the memory usage of the persistent object pool in the background to identify underutilized memory regions. During the memory allocation, this mechanism dynamically aggregates persistent objects distributed in

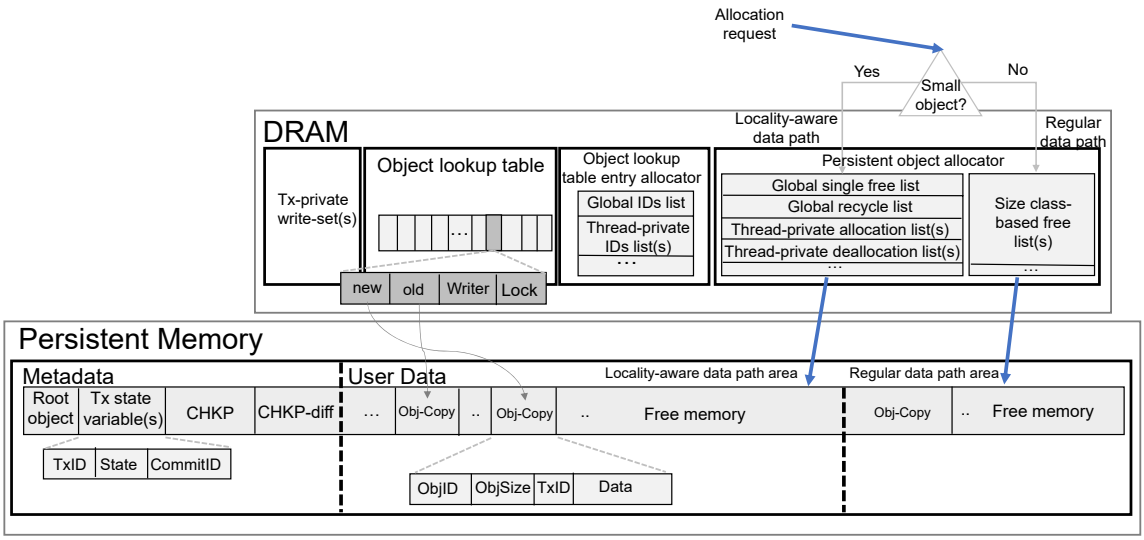


Figure 5.3: Major data structures in ArchTM

the underutilized memory regions to improve memory usage. The online defragmentation mechanism is a user-space solution that can be enabled or disabled. It requires no modifications to operating systems as required by existing solutions [123]. Also, the user-space solution is more flexible than offline static solutions [124] and can react to changes in the application during execution.

5.4 ArchTM Implementation

We describe our implementation based on Section 5.3.

5.4.1 Data Structures

Persistent Data Structures on NVM. ArchTM maintains a persistent memory pool partitioned into metadata and user data areas. As depicted in Figure 5.3, the metadata area stores a root object, a list of transaction state variables, a checkpoint field (*CHKP*), and a checkpoint-diff field (*CHKP-diff*).

The list of transaction state variables records the state of each ongoing transaction. Each variable encodes transaction state and ID, and commit ID. We use the transaction start timestamp as transaction ID, and the transaction commit timestamp as commit ID. They are global timestamps captured at the beginning and end of a transaction. ArchTM uses hardware clock (*rdtscp* in x86 architectures [125, 126])

and prevents the constant skew of the hardware clock among processors by the ORDO primitive [126] to ensure correct ordering of transactions. The transaction state indicates the progress of a transaction, e.g., BEGIN, COMMITTED, END or ABORT.

CHKP stores a persistent checkpoint of the object lookup table to speedup recovery (Section 5.4.6). *CHKP-diff* records the list of memory blocks (named *memory segments*) pre-allocated to each thread (Section 5.4.4-Allocation). *CHKP-diff* is useful to track working objects before the next checkpoint. It is implemented as an array of elements containing three fields: ID of ongoing transactions for which the segment is fetched, the segment start address and size.

The user data area stores persistent objects. Each object has an object header and data. The header contains object ID and size, and transaction ID. The user data area is divided into a regular data path area for large object allocations and a locality-aware data path area for small object allocations.

Transient Data Structures on DRAM ArchTM maintains an object lookup table and a hash set per transaction. The object lookup table is a one-dimensional array mapping a persistent object ID to a persistent object on NVM. Each NVM object has an entry in the table. An entry has four fields, i.e., a pointer to the latest copy (*new*), a pointer to the old copy (*old*), a variable (named *writer*) storing the pointer of the transaction state variable of the ongoing transaction that modifies the latest copy, and a write lock associated with the *writer* to coordinate parallel transactions. The hash set (named *write-set*) is used to collect the IDs of all persistent objects modified by a thread in an active transaction. Before committing a transaction, all objects in the hash set must be persisted.

ArchTM manages metadata for two allocators on DRAM. The first allocator allocates an entry in the object lookup table when a persistent object is created. This allocator maintains a list of free IDs for persistent objects (named *ID list*) per core. A persistent object ID is the index of an entry in the object lookup table. When the allocator allocates an entry, it gets an object ID from the ID list. When a persistent object is freed, its object ID is returned to the ID list. We reuse IDs for persistent objects to avoid the explosion of IDs. New IDs are created only when the ID list is empty.

The second allocator allocates persistent objects. It reuses the metadata structures in JEMalloc [54] for the regular data path but adds significant extensions to op-

Algorithm 1 Start, read, and write operations.

```
1:  function APT_TX_BEGIN
2:      volatile  TxID = GLOBALTIMESTAMP()
3:      TxState.ATOMIC_STORE(TxID, BEGIN)
4:      Fence()
5:  end function
6:
7:  function APT_TX_READ(TxState, objID)
8:      obj = objLookupTable[objID]
9:      if obj.new == NULL then    return obj.old
10:     end if
11:     if obj.writer → TxID == TxState.TxID then    return obj.new
12:     end if
13:     if obj.writer → State == COMMITTEDobj.writer → CommitID <= TxState.TxID then    return
    obj.new
14:     end if
15:     return obj.old
16: end function
17:
18: function APT_TX_WRITE(TxState, objID)
19:     obj ← objLookupTable[objID]
20:     if obj.new! = NULLobj.writer → TxID == TxState.TxID then
21:         return obj.new
22:     end if
23:     if LOCK(obj.writer) then
24:         obj.writer = &TxState
25:         obj.new = ALLOC(obj.old)
26:     else    ABORT_AND_RETRY( )
27:     end if
28:     obj.new = DUPLICATE(obj.old)
29:     obj.new.header.txID = TxState.TxID
30:     # append the object to write-set
31:     write_set.insert(objID)
32:     return obj.new
33: end function
```

timize small writes to NVM (Section 5.3). For the locality-aware data path, ArchTM maintains a global free list and a global recycle list. The global free list contains memory blocks available for allocations. To ensure sequentially when multiple threads access the global free list, ArchTM uses a write lock on the global free list. To mitigate contention on the global free list, each thread maintains a thread-private allocation list, which is a portion from the global free list. Only when a thread exhausts its allocation list will the thread access the global free list to get a new portion. Therefore, synchronization on the global free list is infrequent. The global recycle list collects memory blocks freed by all threads. The allocator manages a deallocation list per thread to collect deallocated memory blocks. Blocks from these thread-local deallocation lists are gathered, sorted, and merged into the global recycle list. Memory management is described in detail in Section 5.4.4.

Algorithm 2 Commit and post-commit operations.

```
 1: function APT_TX_ON_COMMIT(TxState)
 2:     if EMPTY(write_set) then    return
 3:     end if   # read-only tx
 4:     for each obj ∈ write_set do    FLUSH(obj.new)
 5:     end for   Fence()    # persist all modified objects
 6:     volatile   CommitID = GLOBALTIMESTAMP()
 7:     TxState.ATOMIC_STORE(COMMITTED, CommitID)
 8:     Fence()
 9:     APT_TX_POST_COMMIT(TxState)
10: end function
11:
12: function APT_TX_POST_COMMIT(TxState)
13:     for each tx ∈ Ongoing_TXs do
14:         if tx.TxID < TxState.CommitID then    WAIT_FOR(tx)
15:         end if
16:     end for
17:     while obj ← write_set.pop() do
18:         FREE(obj.old)    # append to the reclaim list
19:         obj.old = obj.new
20:         obj.new = NULL
21:         obj.writer = NULL
22:         UNLOCK(obj.writer)
23:     end while
24:     TxState.ATOMIC_STORE(END, INF)
25:     Fence()
26: end function
```

5.4.2 Background Threads

Background threads are helper threads transparent to the application. ArchTM uses two background threads to manage the NVM pool at runtime – the *garbage collection (GC) manager* and the *fragmentation manager*. The GC manager recycles freed persistent objects. The fragmentation manager examines memory usage by regions and aggregates memory blocks for defragmentation (see Section 5.4.4).

5.4.3 Transaction Operations

ArchTM supports five core operations to begin, read, write, commit, and post-commit in a transaction. ArchTM provides snapshot isolation [127, 128] similar to existing work [53, 129, 130, 131, 132, 133] and industrial production database systems [134, 135, 136, 137, 138, 139]. We illustrate the operations in Algorithms 1 and 2.

APT_TX_BEGIN starts a transaction and assigns a unique ID ($TxID$) based on the global timestamp (Alg. 1 Line 2) to the transaction. A transaction state variable ($TxState$) is created and stored in the metadata area on NVM. $TxState$ is a combination of the $TxID$, state and transaction commit ID ($CommitID$). At the

transaction beginning, ArchTM adds $TxID$ and the state BEGIN into $TxState$ by an atomic write.

APT_TX_READ returns a pointer to a copy of the persistent object with ($objID$). If the object is not being updated by any transactions (Alg. 1 Line 9), the pointer to the old copy is returned. If the object is being updated by the current transaction (Alg. 1 Line 11) or a transaction committed before the current transaction starts (Alg. 1 Line 13), the pointer to the new copy is returned. Otherwise, ArchTM returns the pointer of the old copy. The whole process is lock-free.

APT_TX_WRITE returns a pointer to the persistent object $objID$ ready for update. If the persistent object already has a new copy and the most recent update to the copy is performed by the current transaction, the pointer to the new copy is returned (Alg. 1 Lines 20-22). If the persistent object does not have a new copy, the application thread allocates a one, acquires the write lock of the *writer* of the object (Alg. 1 Line 23), duplicates the old copy to the new one, and then updates the new copy. The application thread also inserts the object ID into the *write-set*. If the application thread fails to obtain the write lock of the object, *APT_TX_WRITE* aborts and retries in a new transaction.

APT_TX_ON_COMMIT commits a transaction. If the transaction is read-only, no persistent operations are performed. Otherwise, ArchTM persists the modified objects recorded in the *write-set* to NVM (Alg. 2 Lines 4-5). After that, ArchTM gets a global timestamp as CommitID and updates the state to COMMITTED with CommitID in the transaction state variable by an atomic write.

APT_TX_POST_COMMIT cleans up a committed transaction. First, it checks whether there is any ongoing transaction that starts before the current transaction is fully committed (Alg. 2 Lines 13-16). It reclaims the old copy (i.e., putting the old copy in the thread-private deallocation list) after the earlier transactions are fully committed. This ensures that the old copy of the persistent object is no longer required in any ongoing transaction. The above process is similar to [140]. Afterwards, ArchTM sets the new copy as the old copy and sets the new copy as $NULL$. Finally, it resets and unlocks the writer of modified objects. ArchTM also updates and persists the transaction state to END and CommitID to INF.

5.4.4 Memory Management for Transactions

ArchTM uses a customized persistent object allocator. Depending on the size of an allocation request, ArchTM chooses the locality-aware data path for small allocations and use the regular data path for the others. We describe the locality-aware data path in this Section.

Allocation. When a thread attempts to allocate a persistent object, ArchTM searches through the thread's private allocation list to locate the first memory block larger than the requested size. If no block is found, ArchTM fetches freed memory blocks from the global free list to refill the allocation list. Each fetch takes a large and fixed-size memory segment to avoid frequent contention on the global free list. The fetching history is stored and persisted in *CHKP-diff*. Each fetching event in *CHKP-diff* contains the IDs of ongoing transactions, where the segment is fetched from, the segment start address, and the segment size. If ArchTM cannot find free memory blocks from the global free list, ArchTM replenishes memory blocks from the global recycle list to the global free list.

Deallocation (garbage collection). When a thread deallocates a persistent object, the object is ready for GC because no other transactions are accessing the object (Alg. 2 Lines 13-16). The deallocated object is added to the thread's private deallocation list. In the background, the GC manager periodically collects freed objects from threads to the global recycle list, during which freed blocks are zeroed. Synchronization between application threads and the GC manager is rare because an application thread only updates the head while the GC manager only updates the tail of a deallocation list. The global recycle list is sorted to speed up search during allocation and fragmentation ratio computation during defragmentation. Sorting is inexpensive because when freed memory blocks are added to the global recycle list, they are already mostly sorted.

Defragmentation. ArchTM implements an online defragmentation mechanism to improve the memory usage of the global recycle list. The mechanism works at the granularity of memory regions (4KB). The defragmentation manager monitors the fragmentation ratio (defined as the ratio of used memory to 4KB) of each memory region in the global recycle list. A memory region with a fragmentation ratio greater than f (f is 50% in our evaluation) is deemed underutilized. ArchTM aggregates

persistent objects in underutilized regions and migrates them to a newly allocated memory region. For migration, the defragmentation manager internally creates a "mock" write transaction to ensure the atomicity of data migration and correctness. At the end of the "mock" write transaction, the migrated objects in the original location will be reclaimed through the deallocation process.

5.4.5 Recovery Management

ArchTM follows a two-step recovery process to resume the program from a crash.

1) Detect uncommitted transactions: This is implemented by checking the state of each transaction state variable on NVM. If a state is neither COMMITTED nor END, ArchTM inserts the transaction ID of the uncommitted transaction into a temporary buffer (named $uncommittedTxIDs$).

2) Rebuild object lookup table: ArchTM creates a new object lookup table on DRAM (described in Section 5.4.1) and loads the object information to the new table. The loading process is similar to processing write operations, with the difference that the object information is retrieved from NVM instead of the user request. In particular, ArchTM scans the user data area on NVM to find persistent objects and inserts their location information (i.e., pointers to the objects on NVM) into the lookup table. ArchTM puts the location information of each persistent object in the lookup table based on the object ID which indicates where the location information is in the original lookup table. To identify an object on NVM, ArchTM relies on the object header annotated in each persistent object. The header contains the object ID and object size, which is used to isolate persistent objects from each other on NVM.

ArchTM must eliminate object copies in uncommitted transactions. If the transaction ID of an object copy is found in $uncommittedTxIDs$, the object copy is discarded, and its memory space is reclaimed.

Since ArchTM does not invalidate the memory blocks of a freed object copy until the memory manager recycles them to the global recycle list, a persistent object may have multiple copies in the NVM pool. Therefore, ArchTM must identify the latest copy and discards the others. When ArchTM reads a persistent object from NVM and finds that the object already exists in the object lookup table, ArchTM compares the transaction IDs annotated in these two copies and only keeps the latest one. The

mapping information in the object lookup table is then updated, and the old copy is reclaimed.

Crash consistency is ensured because (1) all modifications in uncommitted transactions are discarded, (2) all modifications in a committed transaction are persisted, and (3) only the latest committed copy of a persistent object is retained. All uncommitted transactions are captured in the transaction state variables stored in NVM, and all object copies with a transaction ID in these uncommitted transactions are discarded during recovery. A transaction is only marked committed after all modified persistent objects in this transaction (collected in *write-set*, (Alg. 1 Line 31)) are persisted (Alg. 2 Lines 4-5). ArchTM identifies the latest committed copy of an object by transaction IDs, which by design guarantees that a transaction ID is no earlier than the commit ID of another transaction if they update the same object (Alg. 1 Lines 23-27).

5.4.6 Reduction of Recovery Time

The recovery process may take a long time if a large number of persistent objects exist on NVM because ArchTM must scan the entire user data area to locate objects and rebuild the object lookup table. The recovery can take as long as tens of minutes on NVM with TBs of capacity.

We reduce the recovery time by incorporating an incremental checkpoint technique into ArchTM. In particular, ArchTM periodically copies the modifications of the object lookup table since the last checkpoint to NVM, such that ArchTM builds a checkpoint of the object lookup table on NVM. When restarting from a crash, ArchTM uses the checkpoint to resume the object lookup table, instead of building it from scratch.

ArchTM uses the following method to detect modifications of the lookup table since the last checkpoint. After taking an incremental checkpoint, ArchTM temporarily blocks all transactions, sets all pages of the object lookup table on DRAM as read-only by enabling write protection, and then resumes the transactions. Any following writes to those pages will trigger a write-protection page fault, indicating that the page is modified. ArchTM records the faulted pages for the next incremental checkpoint. After a page fault is triggered, the page is not write-protected, and

there will be no more page faults. At the time of incremental checkpoint, only those modified pages are copied from DRAM to NVM.

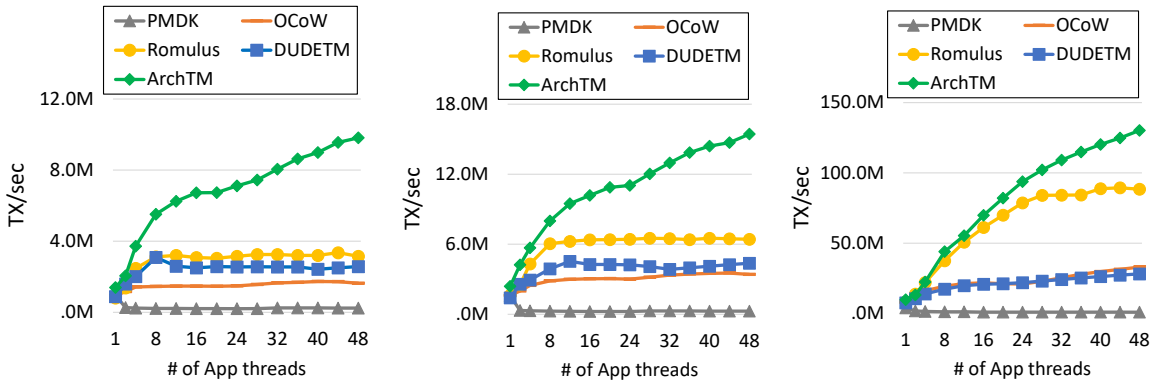
Using a persistent checkpoint of the object lookup table for recovery is not enough to reduce recovery time, because after a crash, the updates on object metadata since the last checkpoint are lost. To solve this problem, the persistent object allocator in ArchTM records the fetching history of memory segments in *CHCP-diff* (Section 5.4.4-Allocation), and those NVM segments contain the modifications of persistent objects since the last checkpoint. ArchTM scans those modified segments to find missing updates as Section 5.4.5. Note that page information collected from the above page fault mechanism cannot be used to locate missing segments, because it is on DRAM and gets lost after crash. The page information is only used to implement incremental checkpoint. Overall, ArchTM uses a combination of the checkpoint of the object lookup table and the fetching history of memory segments in *CHCP-diff* to quickly restore the object lookup table.

5.5 Evaluation

We use an Intel Purley platform that has 2nd Gen Intel® Xeon® Scalable processor, 32KB L1 caches, 1MB L2 caches, and a shared 35MB L3 cache. The memory subsystem consists of 12 DRAM DIMMs and NVDIMMs, providing a total of 192 GB DRAM and 1.5 TB NVM. We compare ArchTM with four state-of-the-art transaction systems: PMDK [20], Romulus [46], DudeTM [122], and OCoW [48]. PMDK uses libpmemobj v1.7. Libpmemobj does not support isolation, so we use a readers-writer lock to protect a transaction from concurrent accesses. Romulus uses RomulusLR for the best performance, and DudeTM uses the default persistent scheduler. We set the checkpoint frequency in ArchTM to 30 seconds, and the size of the pre-allocated NVM segment to two GB. The granularity of memory regions for defragmentation (Section 5.4.4) is 4KB.

5.5.1 Micro-benchmarks

Hash tables and red-black trees are two important concurrent data structures widely used in database workloads [141, 142, 143, 144]. We evaluate hash tables and



(a) 80% update (b) 50% update (c) 5% update

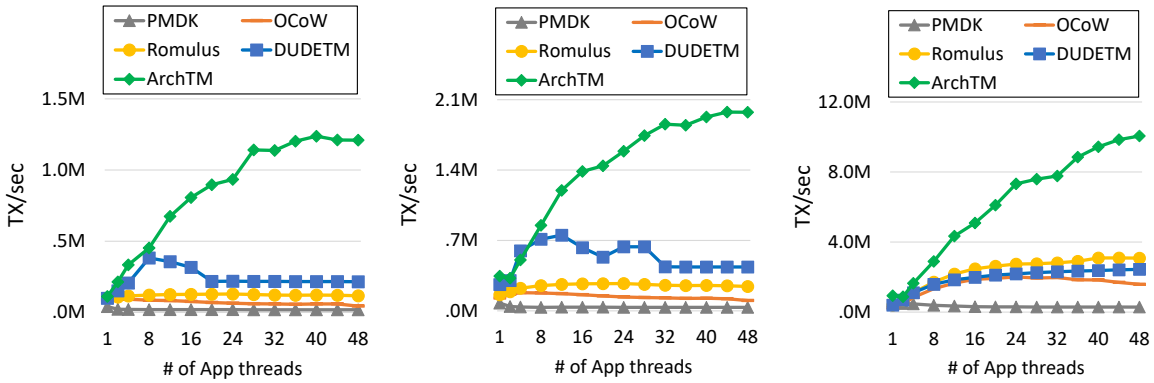
Figure 5.4: Performance and scalability of hash table.

red-black trees with three update rates (5%, 50%, and 80%) similar to [46, 52, 53, 50, 145]. Each transaction operation randomly accesses a key-value pair to read or update. Each key-value pair uses an 8-byte key and 16-byte value. Figure 5.4 and 5.5 present the performance and scalability results.

Hash table. The experiments use a hash table of 10K buckets, each as a single linked list. The hash table is initialized with 100K key-value pairs. ArchTM outperforms the other systems by 10x, 12x and 22x on average at 80%, 50%, and 5% update rates respectively (Figure 5.4). ArchTM demonstrates high scalability as the concurrency in applications increases to the maximum. In contrast, Romulus stops scaling, and DUDETM and OCoW have performance degradation when the application uses more than 16 threads.

In write-intensive workloads (Figures 5.4a and 5.4b), the sequential write technique contributes significant improvement at low application concurrency. When the number of application threads continues increasing, contention on the Optane media outweighs the write amplification. Other optimizations in ArchTM, such as the transient metadata on DRAM, start coping with this new bottleneck, and sustain performance scaling. In a read-intensive workload (Figure 5.4c), ArchTM achieves nearly linear speedup through scalable object referencing on DRAM and lock-free read operations.

Romulus scales well when the concurrency is low (i.e., 1-8 threads) for write-intensive workloads. At high concurrency, its single-threaded write operations become a performance bottleneck. DUDETM cannot consume volatile logs from DRAM to



(a) 80% update (b) 50% update (c) 5% update

Figure 5.5: Performance and scalability of red-black trees.

NVM in time, causing long delays. OCoW has frequent metadata updates on NVM for object remapping, allocation, reclamation, thereby reducing the overall throughput. PMDK shows the worst performance because it uses read-write locks extensively for logging and memory allocation.

Red-black tree. In this experiment, the red-black trees are initialized with one million key-value pairs. ArchTM outperforms the other systems by 7x-13x on average. It exhibits near-linear scalability as the number of threads increases for the read-intensive workloads (Figure 5.5c).

We notice that all three workloads have performance fluctuation at about 28 application threads, likely caused by the high contention on the Optane media. This contention point arrives later than that in the hash table, because each update in the red-black tree needs to search longer than in the hash table, reducing its write intensity.

PMDK, OCoW, Romulus, and DUDETM have lower scalability in the red-black tree than in the hash table. In write-intensive workloads (Figures 5.5a and 5.5b), the performance in these systems either fails to scale or even degrade when the concurrency increases. They suffer from the expensive synchronization [52, 53]. The lock-free operations and scalable object referencing in ArchTM avoid this contention and enables high performance at high concurrency.

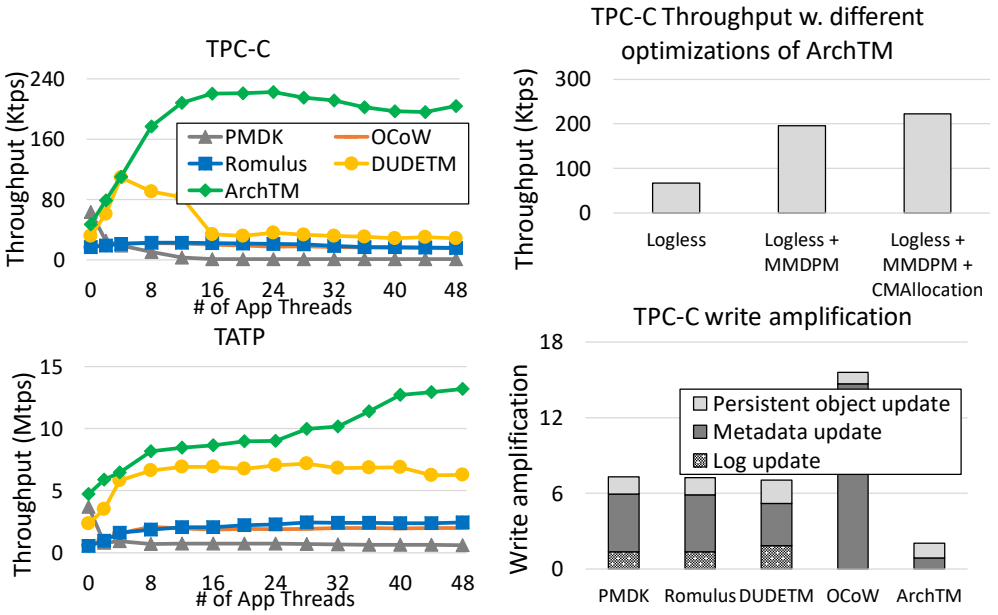
5.5.2 Real World Workloads

We run TPC-C [146] and TATP [147]) against PMEMKV [148]. PMEMKV is a in-memory key-value store developed by Intel. In this experiment, we use its *cmap* storage engine.

TPC-C. We run the *new-order* transaction test, where each application thread works on its corresponding warehouse and executes new order transactions. This workload has a 100% update rate. On average, each transaction inserts more than ten new objects into different tables and modifies more than ten existing objects. ArchTM significantly outperforms others by 10x, 9x, and 5x on average (Figure 5.6a-top). PMDK is more than 100 times slower than others when more than 12 threads are used. The performance of ArchTM scales up quickly to 24 application threads and then slightly declines due to write contention on the Optane media. DUDETM only scales up to eight threads because its performance is limited by centralized persistent logs. Once the background thread cannot flush the log buffer to NVM in time, the application threads are delayed.

TATP. TATP is widely used for online transaction processing. ArchTM outperforms DUDETM, Romulus, OCoW and PMDK by 2x, 6x, 5x, and 13x, respectively (Figure 5.6a-bottom). For evaluation, we implement three read-only and three read-write transactions similar to [53, 122]. The transactions in TATP are less write-intensive than the TPC-C test. Therefore, ArchTM achieves performance scaling up to the maximum application threads. Since TATP has less write traffic than TPC-C, DUDETM sustains performance at 16 threads and beyond.

We quantify the contribution from our design techniques to performance improvement. We separate techniques into logless, minimized metadata modification on NVM (*MMDPM*), and contiguous memory allocation (CMAlloction). Figure 5.6b-top compares the performance using different techniques when running TPC-C with 24 application threads. In this test, We use DUDETM as the baseline, and its throughput is 37 Ktps. Minimized metadata modification on NVM contributes the most (66%) performance improvement. The logless design and the contiguous memory allocation technique contribute 18% and 16% performance improvement, respectively. Using the same test configuration (Figure 5.6b-bottom), we quantify the write amplification in the five systems. The write amplification in ArchTM is only 2.03. ArchTM



(a) Throughput

(b) Detailed Performance Analysis on TPC-C

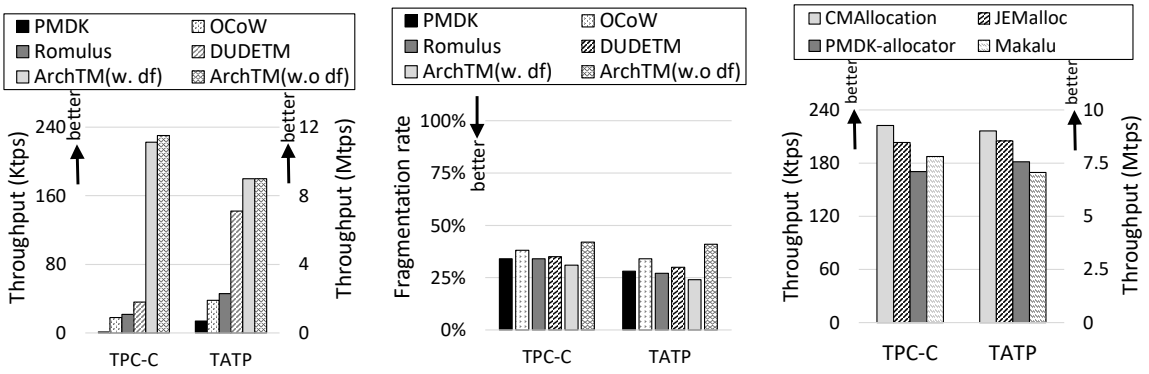
Figure 5.6: Real-world workloads with PMEMKV.

has 3x to 8x lower write amplification than the other systems.

5.5.3 Performance Analysis

Online defragmentation. We evaluate the online defragmentation technique by quantifying the system throughput and memory fragmentation rate in TPC-C and TATP against PMEMKV. Each test uses 24 application threads. We compare the performance of ArchTM with and without online defragmentation (denoted as w.df and w.o.df in Figure 5.7a), with four other NVM systems.

The two ArchTM-based systems outperform other systems by 12x and 3x on average on TPC-C and TATP, respectively. The online defragmentation in ArchTM reduces memory fragmentation from 58% to 69% with only 3% overhead on system throughput on TPC-C. TATP is less write-intensive than TPC-C, and therefore no noticeable performance loss is observed from the online defragmentation. Figure 5.7b reports the memory fragmentation rate of all systems. The memory fragmentation rate of the ArchTM with online defragmentation is 4%, 9%, 3%, and 5% lower than PMDK, OCOW, Romulus, and DUDETM respectively. *ArchTM with online de-*



(a) Impact of online defragmenation

(b) Memory fragmentation

(c) Improvement from CMAlloction.

Figure 5.7: Evaluate the effectiveness of online defragmenation and contiguous memory allocation.

fragmentation is 14% lower that without it, demonstrating the necessity of using our online defragmentation.

Contiguous memory Allocation. We evaluate the effectiveness of contiguous memory allocation (*CMAllocation*) in ArchTM. For comparison, we port ArchTM to use three state-of-the-art allocators, i.e., JEMalloc [54], PM allocator in PMDK [20], and Makalu [59]). Figure 5.7c reports the system throughput when ArchTM is equipped with the different allocators in TPC-C and TATP against PMEMKV.

The CMAllocation-based system achieves 9% and 6% higher throughput than JEMalloc-based system on TPC-C and TATP, respectively. It also offers 20% and 18% higher throughout than PMDK- and Maruku-based systems. The customized locality-aware data path enables CMAllocation to encourage sequential writes on NVM for better performance. In the PMDK and Maruku allocators, the poor scalability and frequent metadata updates become the bottleneck.

Checkpoint and Recovery Time. The checkpoint frequency trades off system throughput with recovery time. We vary the frequency from one second to 60 seconds in TPC-C against PMEMKV. We compare the system throughput with and without checkpoints, and find that checkpoints impose 11% overhead at the highest checkpoint frequency (i.e., one second). At a moderate checkpoint frequency, e.g., 30 seconds, the throughput loss diminishes to less than 1%.

We trigger a random crash after the program runs two minutes and then time the recovery. As expected, the recovery time increases linearly as the checkpoint frequency

decreases. For the 30 GB workload set of TPC-C, ArchTM recovers the system in eight seconds at a checkpoint interval of 30 seconds and the object lookup table consumes 5.6 GB DRAM. For the same experiment, the other four systems recover faster than ArchTM. The overhead in recovery in ArchTM comes from scanning the NVM data area because ArchTM needs to identify updates since the last checkpoint before the crash to rebuild the object lookup table. ArchTM trades a slightly longer recovery time for better runtime performance based on the assumption that crashes in the production environment are infrequent [32].

Transaction abort rate. Transaction aborts occur when a transaction tries to get the write lock of the writer of a persistent object but fails. We measure the abort rate. With 24 threads running highly write-intensive workloads with 80% update rate using the hash table and red-black tree, the abort rate is 1% and 2% on average, respectively. With 24 threads running the TPC-C and TATP, the abort rate is 2% and 2% on average, respectively. In general, the abort rate is very low.

5.6 Related Work

Undo-logging based NVM transactions. Intel's PMDK [20] (libpmemobj) and NV-Heap [21] use undo-logging to log persistent objects on NVM for crash recovery. Atlas [41] also uses undo-logging. It provides compiler and runtime supports to instrument writes to PM. JUSTDO logging [149] implements an Atlas-like log management system designed for machines with persistent caches. It stores the program counter and resumes the execution of critical sections from the same point where a crash happens. iDO [150] optimizes JUSTDO logging by avoiding logging each persistent store. Specifically, iDO divides the critical section into several idempotent code regions and only logs live program states at the beginning of each idempotent region within the critical section.

Redo-logging based NVM transactions. NVthreads [24] supports redo-logging for multi-threaded C/C++ programs. It logs dirty pages tracked by the OS page protection between critical sections. DUDETM [122] uses shadow DRAM to decouple transaction updates and redo-logging. It leverages a background thread to copy and persist the modifications in redo logs to hide the logging overhead. Romulus [46] and Pisces [53] use variants of redo-logging. They both keep two copies of the

data and replicate updates from one copy to the other to ensure crash consistency. Romulus uses a volatile log to record memory locations modified during a transaction to improve the performance of data copy. Pisces targets read-most workloads and explores snapshot isolation to ensure lock-free read operations.

CoW-based NVM transactions. CDDS [116], BPFS [117], and multi-version concurrency control based transactions (e.g., TimeStone [52]) create a new copy and apply updates to the new copy to avoid writing log records.

The above logging-based and CoW-based works optimize NVM transactions by reducing data replication or persistence overhead. In contrast, ArchTM introduces architecture-awareness to adapt the transaction system to leverage the micro-architecture (i.e., internal buffer and data size block) on the NVM hardware. With the architecture-awareness, ArchTM improves the efficiency of PM writes by avoiding small writes and encouraging coalescable writes.

5.7 Summary

Enabling high-performance transactions is critical for leveraging persistent memory for data-intensive applications. We reveal performance problems in common transaction implementations on real NVM hardware and highlight the importance of considering NVM architecture characteristics for transaction performance. In this chapter, we present ArchTM, an architecture-aware NVM transaction system. On average, ArchTM outperforms the state-of-the-art NVM transaction systems (PMDK, Romulus, DudeTM, and the Oracle system) by 58x, 5x, 3x, and 7x respectively.

Chapter 6

Ribbon: High Performance Cache Line Flushing for NVM

6.1 Overview

NVM technologies, such as Intel Optane DC PM [12, 16], provide large capacity, high performance, and a convenient programming interface. Data access to NVM can use `load/store` instructions as if to DRAM. However, the volatile cache hierarchy on the processor imposes challenges on data persistency and program correctness. A `store` instruction may only update data in the cache, not persisting data in NVM immediately. When data is written from the cache back to memory, the order of writes may differ from the program order due to cache replacement policies.

Data in NVM needs to be in consistency state to be able to recover the program after a system or application crash. Therefore, cache line flushing (CLF) is a fundamental building block for programming NVM. Most NVM-aware systems and applications [31, 51, 24, 151, 152, 153, 154, 155, 156, 157, 158, 26, 28, 29, 159] rely on CLF and memory fences to ensure that data is persisted in the correct order so that the state in NVM is recoverable.

CLF can be an expensive operation. CLF triggers cache-line-sized write to the memory controller, even if the cache line is only partially dirty. Also, CLF needs persist barriers, e.g., the memory fence, to ensure that flushed data has reached the persistent domain before any subsequent stores to the same cache line could happen. Our preliminary evaluation shows that CLF can reduce system throughput by 62%

for database applications like Redis. Hence, CLF creates a performance bottleneck on NVM and may significantly reduce the performance benefits promised by NVM.

Most of the existing techniques focus on optimizing persistency semantics, other than the CLF mechanism [30, 32, 149, 28, 160, 22, 161, 34]. Skipping CLF [32, 34] or relaxing constraints on persist barriers [30, 149, 160, 28, 22, 161], these techniques improve application performance by reducing CLF. Each technique may have a different fault model and recovery mechanism that is designed for specific application characteristics. Still, these techniques use CLF to implement their persistency semantics.

In this chapter, we focus on the CLF mechanism, instead of persistency semantics. Therefore, our work applies to general NVM-aware applications. We reveal the characteristics of CLF on the real NVM hardware (i.e., Intel Optane PM). Based on our performance study, we introduce a runtime system called *Ribbon* that decouples CLF from the application and applies model-guided optimizations for the best performance. Applying Ribbon on a NVM-aware application does not change its persistency semantics, i.e., fault models and recovery mechanisms, so that the program correctness is retained.

Our performance study of CLF on the real NVM hardware reveals three optimization insights. *First*, concurrent CLF can create resource contention on the hardware buffer inside NVM devices and memory controllers, which causes performance loss. We define CLF concurrency as the number of threads performing CLF simultaneously. *Second*, the status of a cache line can impact the performance of CLF considerably. For instance, flushing a clean cache line could be 3.3 times faster than flushing a dirty cache line. *Third*, many flushed cache lines have low dirtiness, wasting memory bandwidth and decreasing the efficiency of CLF. The dirtiness of a cache line is quantified as the fraction of dirty bytes in the cache line. Since a cache line is the finest granularity to enforce data persistency, the whole cache line has to be flushed, even if only one byte is dirty. Our evaluation of Redis with YCSB (Load and A-F) and TPC-C workloads shows that the average dirtiness of flushed cache lines is only 47%.

We introduce three techniques in Ribbon to improve the CLF mechanism. First, Ribbon controls the intensity of CLF by thread-level concurrency throttling. Optimal concurrency control needs to address two challenges. How to avoid the impact of

concurrency control on application computation? How to determine the appropriate CLF concurrency? Simply changing thread-level parallelism can reduce thread-level parallelism available for the application. Our solution is to decouple CLF from the application. We instrument and collect CLF in the application and manage a group of flushing threads to perform CLF. This design supports flexible concurrency control without impacting application threads. Furthermore, we introduce an adaptive algorithm to select the concurrency level of these flushing threads. The algorithm achieves a balance between mitigating contention on NVM devices and increasing CLF parallelism for utilizing memory bandwidth.

We propose a proactive CLF technique to increase the possibility of flushing clean cache lines. Flushing a clean cache line is significantly faster than flushing dirty one. Proactive CLF may change the status of a cache line from dirty to clean before the application starts flushing this cache line. Ribbon leverages hardware performance counters in the sampling mode to opportunistically detect modified cache lines with negligible performance overhead.

Ribbon coalesces cache lines of low dirtiness to reduce the number of cache lines to flush. We find that unaligned cache-line flushing and uncoordinated cache-line flushing are the main reasons for low dirtiness in flushed cache lines. These problems stem from the fact that existing memory allocation mechanisms are designed for DRAM. Ribbon introduces a customized memory allocation mechanism to coalesce cache-line flushing and improve efficiency.

We summarize our contributions as follows.

- We characterize the performance of the CLF mechanism in NVM-aware workloads on the real NVM hardware;
- We propose decoupled concurrency control, proactive CLF, and cache line coalescing to improve performance of the CLF mechanism;
- We design and implement Ribbon, a runtime to optimize NVM-aware applications automatically;
- We evaluate Ribbon on a variety of NVM-aware workloads and achieve up to 49.8% improvement (14.8% on average) in the overall application performance.

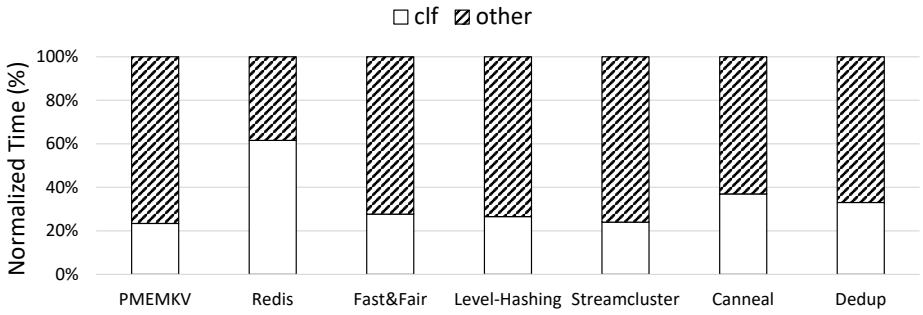


Figure 6.1: The overhead of CLF in common NVM-aware applications.

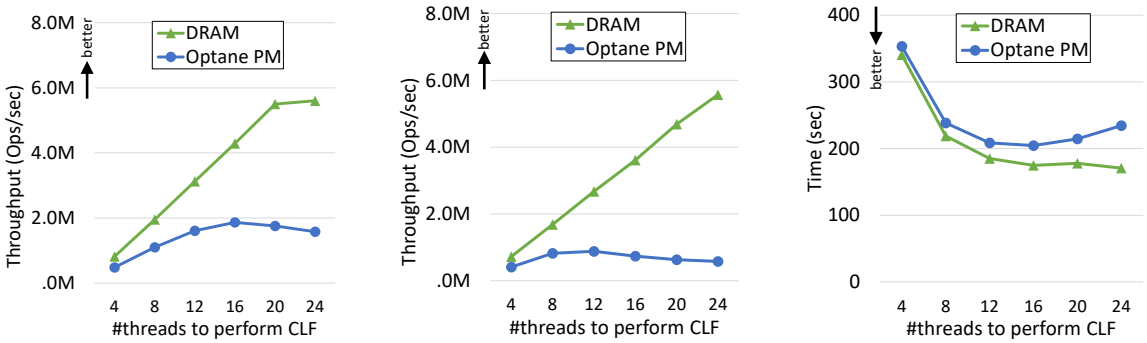
6.2 Performance Analysis of CLF

We use the Intel Optane PM hardware (specifications in Table 6.3) for the performance analysis.

Overhead of CLF in NVM-aware applications. We quantify the cost of CLF in seven representative NVM-aware applications. These applications are inmemeory databases (Intel's PMEMKV [148] and Redis [162]), NVM-optimized index data structures (Fast&Fair [154] and Level-Hashing [155]), and multi-threaded C/C++ applications (Streamcluster, Canneal and Dedup) from Parsec [163] benchmark suite. These applications rely on various persistency semantics and fault models to enable crash consistency, but all use the CLF mechanism. Table 6.4 summarizes the applications. For Parsec applications, we use the *native* input problem and report execution time. For other workloads, we run *dbench* to perform *randomfill* operations and report system throughput. Figure 6.1 shows the CLF overhead in each benchmark in the hatched bars.

The results highlight the impact of CLF on these NVM-aware workloads. For all workloads, CLF significantly affects the performance by 24%-62%. Redis shows the highest performance loss because relies on frequent CLF to persist data objects and logs to implement database transactions. The high overhead in NVM-aware workloads motivates our work to optimize the performance of the CLF mechanism.

The performance impact of CLF concurrency. We increase the number of threads to perform CLF and measure the performance of PMEMKV and Streamcluster on DRAM and Optane, respectively. Table 6.4 in Section 6.5.1 provides more details of the workloads. For PMEMKV, the key size is 20 bytes, and the value size is 256 bytes (Figure 6.2a) and 1 KB (Figure 6.2b). Figure 6.2c reports Streamcluster



(a) PMEMKV-256B. (b) PMEMKV-1KB. (c) Streamcluster.

Figure 6.2: Performance at increased numbers of threads performing CLF.

performance.

On Optane PM (Figure 6.2), all workloads reach their peak performance at a small number of threads, and then the performance starts degrading. In contrast, performance on DRAM sustains scaling as the concurrency increases. Optane shows lower scalability than DRAM because the contention at the internal buffer of Optane and the WPQ in iMC. The increasing performance gap between DRAM and Optane at a large number of threads reveals that high frequency of CLF exacerbates the scaling limitation.

We identify two optimization directions to improve CLF performance. First, the adaption in CLF concurrency should be bi-directional. At a low concurrency level, there is no sufficient writeback traffic to exploit memory bandwidth so that NVM is underutilized. In this scenario, increasing the concurrency to flush cache lines becomes essential. At a high concurrency level, NVM cannot cope with high CLF rate at the application level, and concurrency throttling becomes critical. Given the above two optimization directions, the challenges remain in how to efficiently and timely detect whether NVM is under- or over-utilized? Furthermore, what is the appropriate concurrency level?

Second, different workload characteristics, such as the value size in key-value stores and query intensity, could lead to different concurrency peak. For instance, in PMEMKV, using the 1 KB value size in Figure 6.2b reaches the peak point using 12 threads, while using the 256-byte value size in Figure 6.2a reaches the peak point using 16 threads. The different concurrency peaks necessitate a dynamic solution that enables flexible controlling of CLF concurrency.

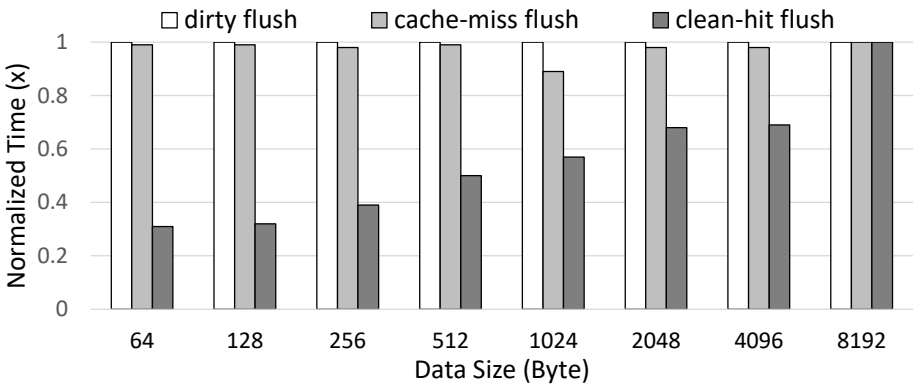


Figure 6.3: Performance of flushing cache lines in different status.

The performance impact of cache lines status. We develop micro-benchmarks to persist data objects of various sizes. Also, we control the locality and dirtiness of flushed cache blocks of those data objects, in order to measure the cost of flushing dirty (resident) cache lines, non-resident cache lines, and clean resident cache lines. Figure 6.3 presents the measured overhead of these three CLF cases.

At a small data size, e.g., 64-byte, flushing a clean cache line resident in the cache hierarchy is significantly cheaper (3.3x) than flushing a dirty cache line. Such low overhead is because of reduced overhead in cache coherence directory lookup, and also because of the elimination of writeback traffic. As a comparison, when flushing a cache line that has been evicted from the cache hierarchy, i.e., non-resident, the cost is much higher than flushing a resident cache line. The difference between a dirty flush and a cache-miss flush indicates the cost of looking up the whole cache coherence directory in our machine is high and overweights the benefit of eliminated writeback.

The low cost of flushing a clean resident cache line motivates us to design a *proactive flushing* mechanism to 'transform' dirty or non-resident flushing into clean-hit flushing ahead of time. The key idea is to complete the transformation before the latency of CLF is exposed to the critical path.

Dirtiness of flushed cache lines. We quantify the average dirtiness of flushed cache lines, denoted as R_{db}, as the ratio between the modified bytes and the cache line size. Therefore, a workload with R_{db} cache line dirtiness would waste $(1 - R_{db})$ bandwidth from the cache hierarchy to the memory subsystem. Moreover, write amplification inside the NVM hardware buffer may further increase the number of

Table 6.1: Average dirtiness of flushed cache lines.

Workloads	YCSB							TPC-C
	Load	A	B	C	D	E	F	
Dirtiness	0.43	0.55	0.56	0	0.51	0.51	0.47	0.32

clean bytes written back to NVM. For instance, if only one byte in four consecutive cache lines is updated, 256 bytes will be eventually written to Optane PM, because the internal transactions have a granularity of 256 bytes. Table 6.1 shows the results for running YCSB [164] and TPC-C [146] workloads against Redis. In general, the dirtiness is less than 0.6 in all workloads, indicating more than half memory bandwidth is wasted for writing back clean data to NVM. Thus, improving cache line dirtiness could benefit CLF performance on such NVM hardware.

6.3 Ribbon Design

We design Ribbon to accelerate the CLF mechanism in NVM–aware applications without impacting program correctness and crash recovery. Ribbon decouples the concurrency control of CLF from the application. It also proactively transforms cache lines to clean status. It uses CLF coalescing, an application-specific optimization for workloads that exhibit low dirtiness in flushed cache lines.

6.3.1 Decoupled Concurrency Control of CLF

Ribbon decouples CLF from the application and adjusts the level of CLF concurrency (the number of threads performing CLF) adaptively. Ribbon throttles CLF concurrency if contention on NVM- devices is detected. Conversely, it ramps up CLF concurrency when NVM- bandwidth is underutilized. We illustrate the workflow in Figure 6.4.

CLF Decoupling The decoupling design in Ribbon creates a thin layer (the gray box in Figure 6.4) between the application and NVM. CLF and fence instructions from the application, such as `clwb`, `clflushopt`, `clflush`, and `sfence`, are collected and queued in this layer. Ribbon uses a group of *flushing threads* to execute these intercepted instructions, respecting the order between flush and fence instructions as in the program order. Therefore, the sequence of flush and fence is unchanged,

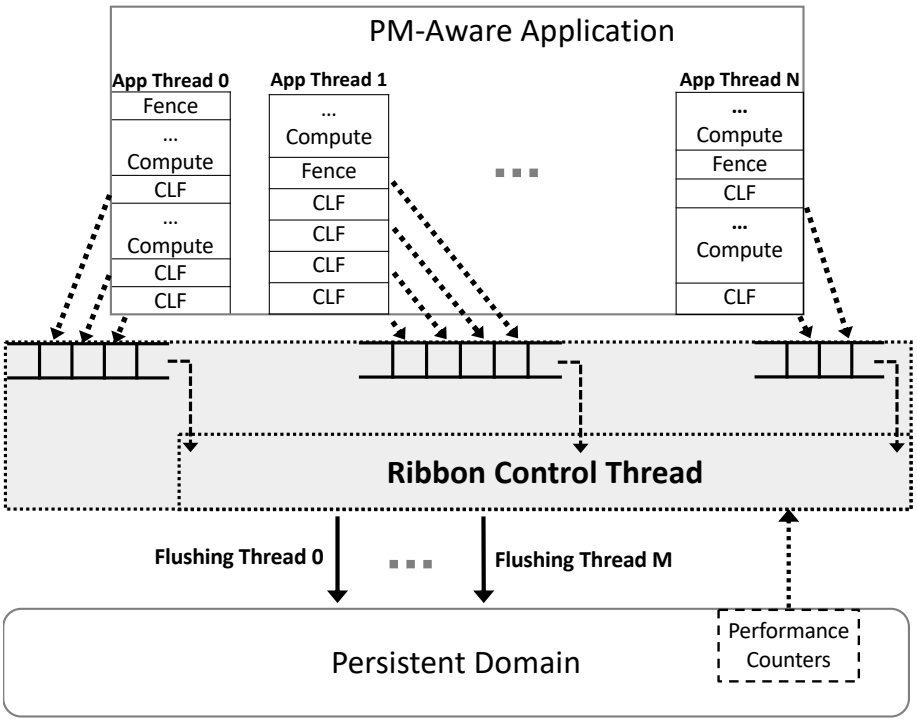


Figure 6.4: Ribbon decouples CLF from the application to its control thread. By detecting contention or underutilization on NVM, Ribbon changes the number of flushing threads to adapt the CLF concurrency.

and consistent semantics is preserved. Furthermore, Ribbon can adapt the CLF concurrency by changing the number of flushing threads.

Ribbon uses FIFO queues as a coordination mechanism between the application and flushing threads. Each application thread has a private FIFO queue, while one flushing thread may work with multiple FIFO queues. CLFs from an application thread are enqueued at the head of its queue. At the queue tail, a flushing thread dequeues and executes CLFs. Ribbon uses a circular buffer to implement the queue, and only exchanges two integers, i.e., the head and tail indexes, among threads to have a lock-less queue implementation. Synchronization between the threads is rare because, on each queue, the application thread only updates the head and the flushing threads only update the tail.

Assume there are N application threads and M flushing threads. Each flushing thread handles at most $\lfloor N/M \rfloor + 1$ application threads (queues). Ribbon throttles the CLF concurrency by reducing M to be $M < N$. Conversely, increasing M to $M > N$ would increase the CLF concurrency. Separately, a control thread detects

performance bottlenecks in NVM and adjusts the number of flushing threads.

Ribbon ensures that the flushing threads execute CLF and fence instructions in the same order as in the application thread. Each memory fence instruction in the application thread acts as the deadline for the flushing threads to finish all CLFs issued before it. Therefore, CLFs after a fence cannot be executed until CLFs before the fence are cleared from the queue. When an application thread issues a memory fence instruction, but there are pending CLF requests in the queue, Ribbon blocks the application thread. This interaction is essential for throttling the CLF concurrency and ensuring program correctness, i.e., reducing the draining rate of CLFs from the queue, without overflowing the queue.

Determining the concurrency level of CLF. A *control thread* monitors the traffic to NVM and adjusts the concurrency level of CLF (NUM_{thr}) at runtime.

The control thread monitors hardware counters in Optane PM at interval T to track the write bandwidth to NVDIMMs (BW_{NVM}). System evaluation shows that when the concurrency level increases, the bandwidth to Optane PM first increases to a peak and then starts decreasing [12, 15, 16]. BW_{NVM} reflects the speed at which the memory controller drains write requests from the WPQ. When memory contention occurs in the WPQ, reducing the concurrency level would improve BW_{NVM}. We call the concurrency levels below the one that reaches the peak performance to be *the scaling region* and above to be *the contention region*. The control thread samples BW_{NVM} at four concurrency points to estimate NUM_{thr} for achieving the peak BW_{NVM}.

The control thread first samples the bandwidth at the concurrency level P1 which is equal to the number of flushing threads that saturate bandwidth on hardware. P1 is architecture-dependent and on the Optane PM, system evaluation reveals that the peak write bandwidth is achieved at four threads [12]. Therefore, 1–$P1$ threads in NVM-aware workloads have to be in the scaling region. The control thread records the bandwidth to Optane PM at P1 to be BW_1^{NVM}. Then, it chooses a sample point at the number of cores (P4) and measures BW_4^{NVM}. On our NVM hardware, P4 is equal to 24. Next, samples are taken at $P2 = P1 + 1$ and $P3 = P4 - 1$, namely BW_2^{NVM} and BW_3^{NVM}. If BW_2^{NVM} is higher than BW_1^{NVM}, and BW_4^{NVM} is also higher than BW_3^{NVM}, it means that even the maximum parallelism has not reached the contention region. Thus, the control thread selects NUM_{thr} to be P4. If BW_2^{NVM}

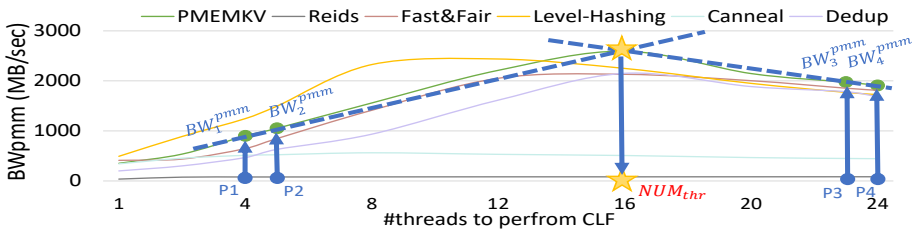
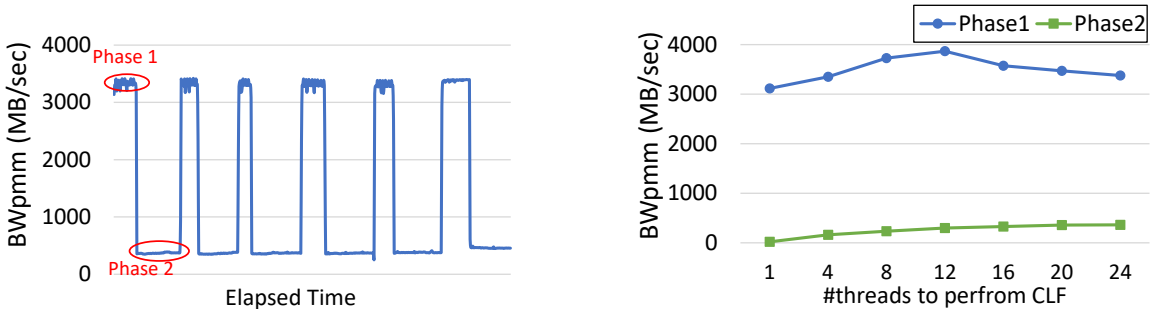


Figure 6.5: Optane PM bandwidth when running benchmarks with various numbers of flushing threads. The number of application threads is 24.

is higher than BW_1^{NVM}, but BW_4^{NVM} is lower than BW_3^{NVM}, it means that the peak is between P2 and P3. The control thread sets NUM_{thr} to be the intersection between the two lines connecting P1 to P2 and P3 to P4, respectively. Finally, if BW_2^{NVM} is lower than BW_1^{NVM}, and BW_4^{NVM} is also lower than BW_3^{NVM}, the control thread selects NUM_{thr} to be P1. In practice, the number of flushing threads is subject to the number of idle threads, and contemporary many-core platforms can provide abundant thread-level parallelism. If there are no enough idle threads to support NUM_{thr} flushing threads, Ribbon automatically disables concurrency control and regresses to use application threads to perform CLF.

We sweep all levels of CLF concurrency in all evaluated workloads and find that this algorithm can always determine the optimal concurrency level. Figure **??** reports all workloads (except one phase in Streamcluster) exhibit a similar trend, i.e., reaching a peak at a low concurrency level and then decreasing performance as concurrency increases. The dashed line and the intersection illustrate the optimal concurrency level for PMEMKV. Streamcluster contains two phases of BW_{NVM}. The first phase follows the scaling trend of other applications in Figure 6.5. In the second phase (shown in Figure 6.6), Streamcluster does not enter the contention as its bandwidth continues increasing. The control thread determines NUM_{thr} to be the maximum available concurrency.

The control thread repeats the above procedure of determining concurrency level of CLF, if the variation of BW_{NVM} is higher than a threshold, indicating there is a change in execution phases of the application and there is a need to adjust concurrency level. Based on our study, the variation threshold should be set between 20% and 30% of BW_{NVM} for best performance. If the threshold is too low (e.g., less than 20%), Ribbon triggers concurrency throttling frequently, which causes performance loss. If the threshold is too high (e.g., more than 30%), Ribbon cannot timely capture the



(a) The trace of Optane PM bandwidth.

(b) Optane PM bandwidth with various numbers of flushing threads.

Figure 6.6: Optane PM bandwidth of Streamcluster. The number of application threads is 24.

change of execution phases, which loses opportunities for performance improvement. We use 20% in Ribbon; We study the sensitivity of application performance to this parameter in Section 6.5.3.

The time interval T to track BW_{NVM} has impact on performance. On the one hand, if T is too large, infrequent monitoring may fail to capture bandwidth saturation. On the other hand, if T is too small, runtime overhead is large, thereby amortizing the performance benefit of concurrency control. We set T to one second in Ribbon to strike a balance between monitoring effectiveness and cost. We study the sensitivity of application performance to this parameter in Section 6.5.3.

6.3.2 Proactive Cache Line Flushing

Ribbon proactively flushes cache lines to transform cache lines to clean state. The proactive CLF increases the chance of flushing a clean cache line in the critical path of the application, which has lower latency than flushing a dirty cache line. We present the workflow in Figure 6.7.

Ribbon leverages the precise address sampling capability in hardware performance counters, e.g., Precise Event-Based Sampling (PEBS) from Intel processor or Instruction-based Sampling (IBS) from AMD processor, to collect the virtual memory addresses of store instructions. If a cache line is found to be updated recently, Ribbon uses a thread to proactively issue a flush (the thread is named the proactive thread). Later on, when the application thread flushes the cache line, it is likely to be in clean status. Note that the cache block may have been evicted by hardware before

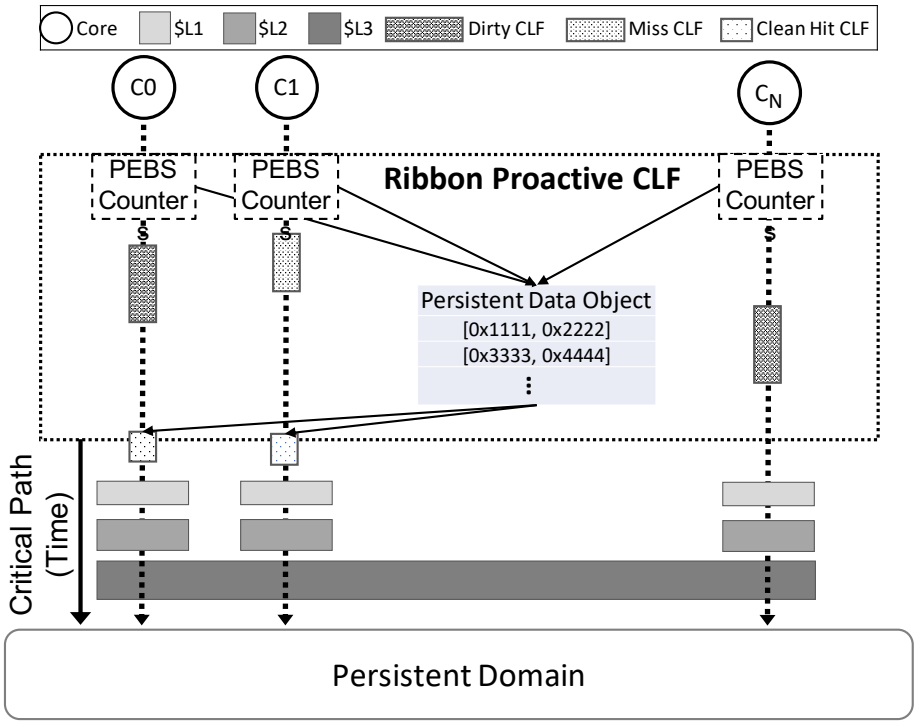


Figure 6.7: Proactive cache line flushing to improve performance.

the proactive thread flushes it. However, a redundant flush by the proactive thread has no impact on program correctness. This approach increases the probability of clean cache lines flushed by the application, which shortens the latency on the critical path.

The proactive CLF can slightly increase write traffic (see Section 6.5.3). For instance, if a cache block is written multiple times followed by one CLF in the program, using the proactive CLF may generate more than one CLF. To avoid the negative impact of extra write traffic due to the proactive CLF, Ribbon disables it once CLF concurrency is reduced because of reaching bandwidth bottleneck; The proactive CLF is re-enabled if CLF concurrency is increased.

Ribbon separates the proactive thread and flushing threads as two independent groups. The design is synchronization-free between the proactive thread and flushing threads. The design does not change which cache lines should be flushed. It also ensures that the consistency semantics in the program retains because no CLF is skipped due to the proactive CLF.

6.3.3 Coalescing Cache Line Flushing

We propose cache line coalescing as an application-specific optimization for workloads that exhibit low dirtiness in flushed cache lines. An Application is suitable for this optimization if multiple CLFs in the application meet two requirement: *First*, the multiple CLFs occur in proximity in time; *Second*, the flushed data objects are coalescable to fewer cache blocks. The first requirement ensures crash consistency after CLF coalescing. CLF coalescing delays those to-be-coalesced CLFs that happen early in the bundle of CLFs from being coalesced. However, if all CLFs in the bundle happen sequentially with no other non-coalescing CLFs occurring between these to-be-coalesced CLFs in the application, delaying the to-be-coalesced CLFs has no impact on crash consistency. The second requirement is the necessary condition to have potential performance benefits.

Listing 6.1 shows an example from Redis. Lines 8 and 12 use two CLFs for persisting *newVal* and *newKey*, respectively. Coalescing these CLFs will delay the first CLF. Between these two CLFs, there are no other CLF. Therefore, the delay of the first CLF still maintains execution correctness after a restart, i.e., the two CLFs either both succeed or fail, which is consistent with the original execution. After the coalescing, the situation that the first CLF succeeds but the second one fails is impossible, guaranteeing the consistency.

After examining NVM-aware applications in Table 6.4, we find that in-memory databases, such as PMEMKV and Redis, and customized NVM data indexes, such as Fast&Fair (B+-tree) and Level-Hashing, are prone to the low dirtiness. Parallel computing codes, such as streamcluster, caneal, and dedup from Pasec, often do not have the low dirtiness. Furthermore, we find **unaligned CLF** and **uncoordinated CLF** are the main reasons for low dirtiness in flushed cache lines.

The unaligned CLF happens when a persistent data object is unaligned with cache lines. For example, a persistent data object is 100 bytes. Ideally, the object should use two cache blocks of 64 bytes. However, the object may be unaligned at the memory allocation and ended up occupying three cache blocks. Once the object is updated, three cache blocks, i.e., 192 bytes, have to be flushed, increasing the number of CLF by 50%. Uncoordinated CLFs happen when multiple associated data objects are allocated into separate cache blocks. Here, data objects are associated if they are

Listing 6.1: An example of CLF coalescing

```
#define KEY_LEN 24
#define VALUE_LEN 100
/*The original code without coalescing*/
void setGenericCommand(client *c, char *key, char *val ...) { ...
    TX_BEGIN(server.NVM_pool) {
        char* newVal = alloc_mem(VALUE_LEN);
        dupStringObjectNVM(newVal, val);
        flush(newVal, VALUE_LEN);
        _mm_sfence();
        char* newKey = alloc_mem(KEY_LEN);
        setKeyNVM(c->db,key, newkey,newVal);
        flush(newkey, KEY_LEN);
        _mm_sfence();
    } TX_ONABORT { ... } TX_END ... }

/*The code with coalescing*/
void setGenericCommand_coalescing(client *c, char *key, char *val ...)
    { ...
    TX_BEGIN(server.NVM_pool) {
        char* mem = alloc_mem(VALUE_LEN + KEY_LEN);
        char* newVal = get_mem(0);
        dupStringObjectNVM(newVal, val);
        char* newKey = get_mem(VALUE_LEN);
        setKeyNVM(c->db,key,newKey, newVal);
        flush(mem, VALUE_LEN + KEY_LEN);
        _mm_sfence();
    } TX_ONABORT { ... } TX_END ... }
```

always updated together. Therefore, coalescing them into the same cache blocks will reduce the number of CLFs.

Implementing cache line coalescing requires replacing memory allocation and combining cache line flushes and memory fences. This transformation could be done automatically by the compiler. In practice, we find that automatic conversion is challenging because even the same application logic can have different implementations in different applications. Without application knowledge, automatic transformation is error-prone. Therefore, we provide a simple interface and leverage the programmer's application knowledge in implementation.

The remainder of the section uses Redis as an example. We use a NVM–aware version of Redis, i.e., Redis-libpmemobj [162]. As a key-value store system, Redis provides fast access to key-value pairs. Each key-value pair includes a unique key ID and their data (value). For each key-value pair, the key and value objects are allocated separately on different cache blocks. Figure 6.8 gives a case where the value object

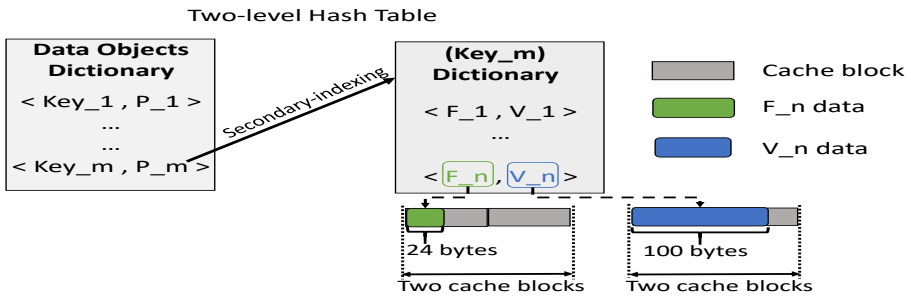


Figure 6.8: Uncoordinated cache-line flushing in the two-level hash table in Redis.

in a key-value pair is a complex data structure. This case comes from the secondary indexing in Redis. In this case, Redis updates the key (i.e., `F_n` in Figure 6.8, which is the secondary-level key) and value (i.e., `V_n` in Figure 6.8) together. Coalescing `F_n` and `V_n` objects into a fewer contiguous cache blocks reduces the number of CLFs.

To coalesce CLFs for Redis, we introduce a new memory allocation mechanism. The old implementation in Redis-libpmemobj uses the memory allocation API from PMDK's libpmemobj library, which does not consider semantics correlation between memory allocations (i.e., memory allocations for a pair of key and value). In the new implementation, we introduce a customized memory allocation API that takes an argument indicating whether the memory allocation is for a key or a value object. In the original implementation of Redis, the memory allocation for a value object happens before the memory allocation for the corresponding key object. Hence, if the memory allocation is for a value object, in our implementation of Redis, the memory allocation not only allocates memory for the value, but also for the key. The key and value objects are co-located into continuous cache blocks, which enables CLF coalescing. If the memory allocation is for a key object, no memory allocation happens, but the previously allocated memory for the key object is returned. Also, the new implementation attempts to avoid unaligned CLF.

6.3.4 Impact of Ribbon on Program Correctness

NVM-aware applications optimized with Ribbon maintain their program correctness because their fault models and recovery mechanisms remain unchanged. Ribbon does not eliminate any cache flush or fence instructions, nor changes their order in the original program. Thus, the original consistency semantics in these programs are

Table 6.2: Ribbon APIs

API name	Description
int ribbon_start(int numAppT, int eleFQueue)	Initialize the runtime system and resource (e.g., flushing threads and FIFO queues)
int ribbon_flush(void* addr, size_t len)	Put CLF requests into fluhsing queues
void* ribbon_alloc(size_t len, int type)	Memory allocation for coalescing CLF
int ribbon_stop()	Terminate runtime and release resources
int ribbon_fence()	Ensure all pending CLF requests are flushed
int ribbon_free(void* addr)	Free a memory allocation

preserved even in the presence of crashes. The advantage of Ribbon is to reduce the latency of these CLF instructions on the critical path by improving the bandwidth to NVM or increasing the probability of a clean cache line. Although the proactive CLF may introduce additional cache flushes, they do not occur on the critical path. Also, changing the state of cache lines has no impact on the fault model in these NVM-aware applications because cache line eviction and replacement is hardware-managed and outside the application control. Coalescing multiple cache lines into one does not eliminate the flush and fence instructions in the program. However, it can reduce write amplification so that these instructions could complete at reduced latency. When a crash occurs, each program will be restored to a consistent state by employing its original recovery mechanism, e.g., undo/redo logging.

6.4 Implementation

Programming APIs. Ribbon is implemented as a user-level library to provide CLF performance optimization. Ribbon provides a small set of APIs and is designed to minimize the porting efforts in existing NVM-aware applications and libraries, such as Intel PMDK [19], Mnemosyne [77], and NVthreads [24]. Table 6.2 summarizes main APIs.

ribbon_start() initializes the flushing threads, control thread and proactive CLF thread. This routine creates a pool of flushing threads and FIFO queues, and initializes performance counters. This routine is called only once before main execution phase starts. *ribbon_stop()* frees all runtime resources created in *ribbon_start()*. This routine is called only once before the end of the main program. *ribbon_flush()* and *ribbon_fence* are used to intercept cache flush and memory fence calls in the pro-

Table 6.3: Experiment Platform Specifications

Processor	2nd Gen Intel® Xeon® Scalable processor
Cores	2.4 GHz (3.9 GHz Turbo frequency) × 24 cores (48 HT)
L1-icache	private, 32 KB, 8-way set associative, write-back
L1-dcache	private, 32 KB, 8-way set associative, write-back
L2-cache	private, 1MB, 16-way set associative, write-back
L3-Cache	shared, 35.75 MB, 11-way set associative, non-inclusive write-back
DRAM	16-GB DDR4 DIMM x 6 per socket
PM	128-GB Optane DC NVDIMM x 6 per socket
Interconnect	Intel® UPI at 10.4 GT/s, 10.4GT/s, and 9.6 GT/s

gram. *ribbon_flush()* places a CLF request at the head of the private FIFO queue of the issuing thread. *ribbon_fence()* checks if all pending requests in the FIFO queue have drained. If not, Ribbon blocks the application thread. *ribbon_alloc()* and *ribbon_free()* are used to replace the memory allocation and free APIs in the *pmemobj* library in Redis. The two APIs are used to allocate and free memory from/to NVM for coalescing CLF.

Using the above APIs to replace CLF and memory fence can be done automatically by a compiler. To enable CLF coalescing in Redis, we make modifications manually. The statistics of code modification given by git diff is: 10 files changed, 293 insertions(+), 64 deletions(-).

System optimization. Ribbon includes several optimization techniques to enable high performance. We use FIFO queues to coordinate between the application thread and flushing threads. When the number of flushing threads is more than the number of application threads, multiple flushing threads fetch CLF requests from one FIFO queue, which raise contention. To avoid the contention, we dedicate one flushing thread to fetch CLF requests from the queue and then assigns them to other flushing threads. Our implementation uses the most recent `clwb` instruction to flush cache blocks.

6.5 Evaluation

In this section, we evaluate the performance of Ribbon.

Table 6.4: A summary of evaluated workloads

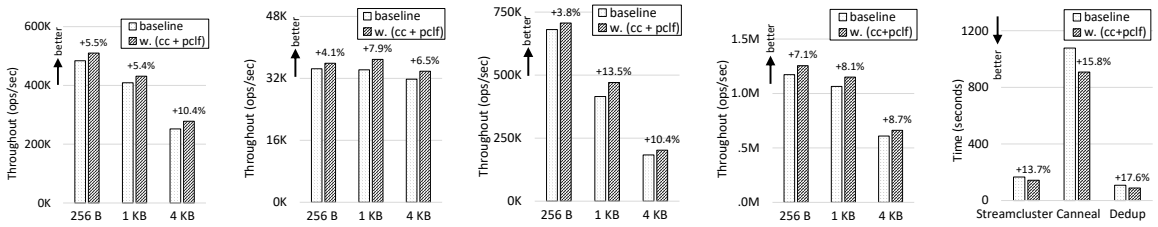
Application	Program Type	NVM Access Layer
PMEMKV	Database	Library/PMDK(undo&redo)
Redis	Database	Library/PMDK(undo&redo)
Fast&Fair (B+-tree)	NVM-aware index	Native (add custom assembly instructions)
Level-Hashing	NVM-aware index	Native (add custom assembly instructions)
Streamcluster	Lock-based parallel code	Library/NVthreads (redo)
Canneal	Lock-based parallel code	Library/NVthreads (redo)
Dedup	Lock-based parallel code	Library/NVthreads (redo)

6.5.1 Methodology

Experiment platform. We evaluate Ribbon on the Intel Optane persistent memory. Table 6.3 describes the configuration of the testbed. The system consists of two sockets, each with two integrated memory controllers (iMCs) and six memory channels. Each DRAM DIMM has 16 GB capacity while a NVDIMM has 128 GB capacity. In total, the system has 192 GB DRAM, and 1.5 TB Intel Optane DC persistent memory. We use one socket for performance study to eliminate NUMA effects. The persistent domain starts from iMC, i.e., a memory fence only returns after the flushed data has reached iMC.

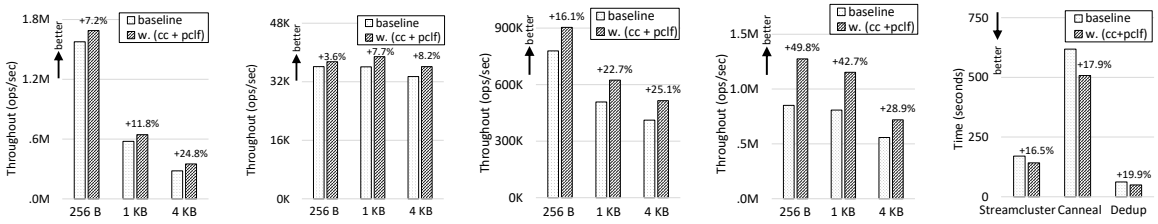
Applications with various NVM access interfaces. We select seven representative PM-aware workloads from diverse domains, including in-memory database (PMEMKV [148] and Redis [162]), NVM-aware index data structures (Fast&Fair [154] and Level-Hashing [155]) and C++ parallel computing applications (Streamcluster, Canneal, and Dedup from Parsec benchmark suite [163]). For PMEMKV, we use its *cmap* storage engine.

These applications also use different interfaces to access NVM, such as high-level NVM-aware libraries and native direct interaction. Table 6.4 summarizes the application characteristics and NVM access interfaces for each workload. PMEMKV and Redis use *libpmemobj* from Intel PMDK [19] library to access and persist data. *libpmemobj* is a logging-based transaction system, which implements undo logging to protect user data and redo logging to protect metadata. The Parsec applications guarantee data consistency by the NVthreads library [24]. NVthreads supports a redo-logging for multi-threaded *C/C++* programs. The two NVM-aware index data structures use custom assembly instructions to flush data from the cache to NVM, and add fences to ensure the order between these flushes and other application accesses



(a) PMEMKV. (b) Redis. (c) Fast&Fair. (d) Level-Hashing. (e) Parsec.

Figure 6.9: Overall performance (App threads = 4).



(a) PMEMKV. (b) Redis. (c) Fast&Fair. (d) Level-Hashing. (e) Parsec.

Figure 6.10: Overall performance (App threads = 24).

to the data.

6.5.2 Overall Performance

We evaluate each workload at a low and high thread-level parallelism (using 4 and 24 application threads respectively). For Redis, we cannot change the number of threads to run it, because it is a single-thread server; To evaluate Redis, we change the number of client threads (using 4 and 24). PMEMKV, Redis, Fast&Fair, and Level-Hashing run the *dbench* benchmark to execute one hundred million *randomfill* operations. The key size is 20 bytes, and three value sizes (256 bytes, 1 KB, and 4 KB) are tested. Streamcluster, Canneal, and Dedup use the *Native* input problem in [24].

Ribbon demonstrates its generality in these NVM-aware frameworks that employ different fault models, recovery mechanisms, and interfaces to access NVM. Ribbon achieves performance improvement in all seven workloads at different application concurrency, without changing any CLF policy. Figures 6.9 and 6.10 present the performance of Ribbon (*w. cc+pclf*) in comparison to the original implementation

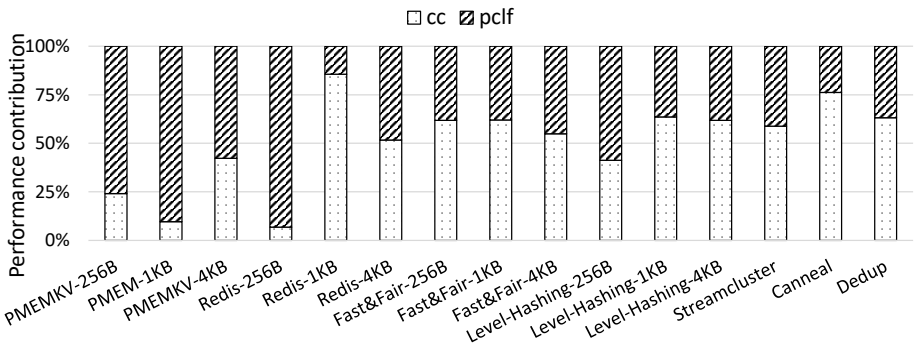


Figure 6.11: A breakdown of performance improvement from the concurrency control and proactive CLF (App threads = 4).

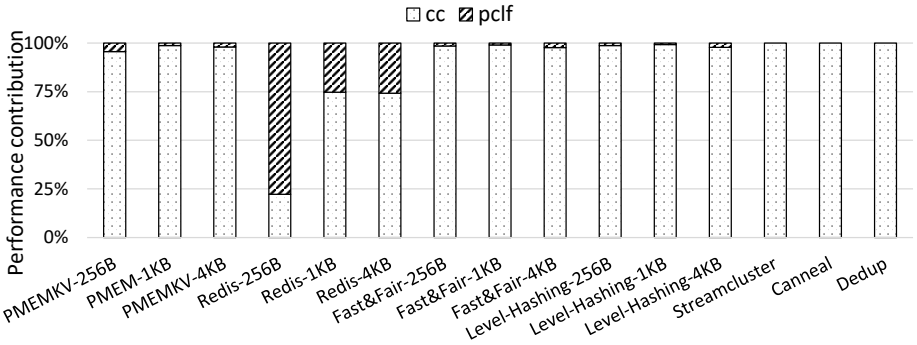


Figure 6.12: A breakdown of performance improvement from the concurrency control and proactive CLF (App threads = 24).

(*baseline*). At four application threads, Ribbon increases the concurrency of CLF and achieves up to 17.6% improvement (9.3% on average). In contrast, at 24 application threads, Ribbon detects memory contention and improves the performance by up to 49.8% (20.2% on average).

Ribbon brings performance benefits to all tested workloads. Among them, Ribbon delivers more performance benefits to those that use large value sizes (1 KB and 4 KB in our evaluation) and high application threads concurrency (24 application threads in our evaluation). These cases can result in memory contention or lack of CLF parallelism, which provides more opportunity to Ribbon.

We analyze the effectiveness of each optimization technique by breaking down their contribution to performance improvement. In particular, we apply the concurrency control first (*w. cc*) and measure performance improvement. Then, on top of it we apply the proactive CLF (*w. pclf*) and measure performance improvement. Figures 6.11 and 6.12 presents the breakdown with four and 24 application threads.

Table 6.5: Sensitivity study on bandwidth variance threshold and monitor interval (App threads = 24).

BW Variance Threshold	Improvement
10%	7.4%
20%	16.5%
30%	17.3%
50%	14.6%

Interval (sec)	Improvement
0.1	9.7%
1	16.5%
5	13.8%
10	11.3%

We find that the concurrency control and proactive CLF contribute comparably to the performance improvement at a low number of application threads (Figures 6.11). At a large number of application threads, most performance improvement attribute to the concurrency control technique (Figure 6.12). The difference is because the contention on NVM devices increases when CLFs are issued by more threads, which compete in inserting flushed data to WPQ (the start of the persistent domain). Therefore, CLF tends to create a performance bottleneck at a large number of application threads, which is addressed by the concurrency control. Note that Redis also benefits substantially from the proactive CLF even at the high number of client threads because it is a single-threaded server, and CLF contention is not its bottleneck.

6.5.3 Sensitivity Evaluation

We use Streamcluster with *Native* input problem for sensitivity study because this workload has execution phases with various bandwidth consumption, imposing challenges on concurrency control and proactive CLF.

Sensitivity on bandwidth variance threshold. We use four thresholds for study. Table 6.5-left shows the results and the tradeoff between low and high threshold values. 20%-30% leads to the largest improvement (Ribbon uses 20%).

Sensitivity on monitor interval T. We use four intervals for study. Table 6.5-right shows the improvement achieved at various interval values. The highest

Table 6.6: Sensitivity study on proactive CLF

#app threads	1	2	4	8	12	16	17-24
Improvement	5.4%	6.6%	7.5%	6.3%	4%	2.1%	0
Normalized BW cost	6.9%	6.3%	5.6%	7.2%	8.4%	8.9%	0

improvement is achieved at one second. (Ribbon uses one second for T).

Sensitivity on proactive CLF. We evaluate how the proactive CLF responses given various bandwidth consumption of the application. Ribbon should avoid the negative impact of the proactive CLF on memory bandwidth. To evaluate the proactive CLF itself, we disable the concurrency control, but integrate the algorithm of determining concurrency level into the proactive CLF to detect bandwidth contention. When the concurrency level needs to be reduced according to the algorithm, we do not change concurrency but disable the proactive CLF. We sweep the number of application threads from one to 24. We report the bandwidth consumption of the proactive CLF normalized to the total bandwidth consumption in Table 6.6. We report application performance normalized to that without the proactive CLF.

The proactive CLF improves the performance by 2.1%-7.5% when the number of applications increases from one to 16. In these cases, the proactive CLF takes a small portion (5.6%-8.9%) of the total bandwidth consumption. When the application uses more than 16 application threads, the proactive CLF is disabled because of the detection of bandwidth contention. As a result, there is no performance improvement.

6.5.4 Heavily Loaded System Evaluation

We evaluate Ribbon on a heavily loaded machine to understand the impact of Ribbon on application performance. In this evaluation, we co-run three different application combinations. For each combination, we run two applications, each using 24 application threads. PMEMKV, Level-Hashing, and Fast&Fair run *dbench* to execute one hundred million *randomfill* operations and use 256 bytes as the value size. Streamcluster and Canneal use the native input problem. We report the experimental results in Figure 6.13.

We observe that all workloads can benefit from Ribbon significantly. Compared with the system without Ribbon, Ribbon improves the performance of PMEMKV and Streamcluster by 20.4% and 27.7%, respectively. When Level-Hashing and Canneal

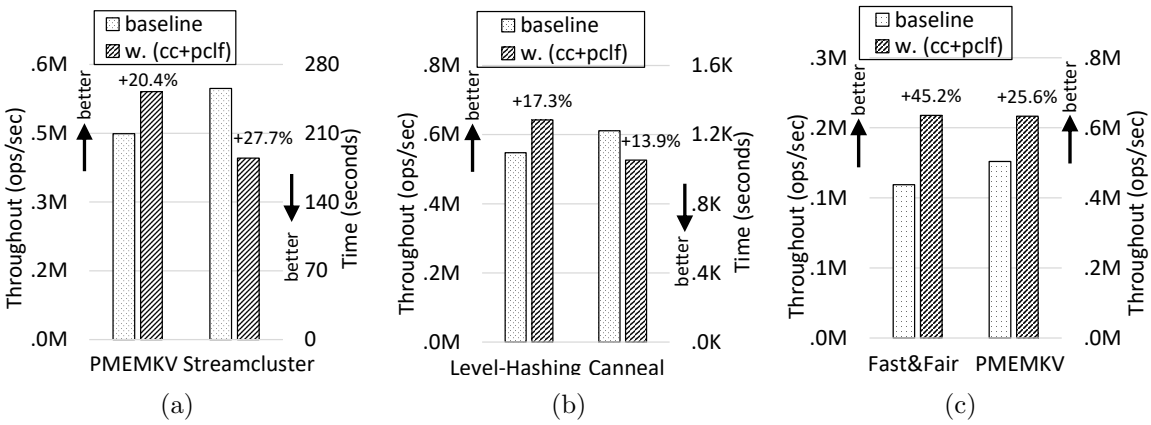


Figure 6.13: Heavily loaded system

Table 6.7: Quantify the dirtiness of flushed cache lines in Redis.

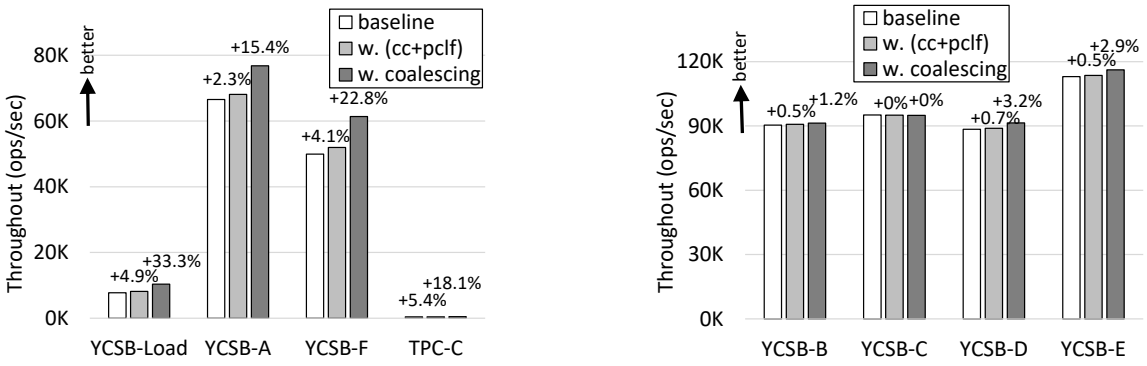
Workloads	YCSB							TPC-C
	Load	A	B	C	D	E	F	
w.o coalescing	0.43	0.55	0.56	0	0.51	0.51	0.47	0.32
w. coalescing	0.62	0.66	0.67	0	0.63	0.63	0.68	0.40

co-run on the same machine, Ribbon speeds up the two applications by 17.3% and 13.9%, respectively. Fast&Fair and PMEMKV co-run achieve the most improvement from Ribbon, reaching 45.2% and 25.6% improvement, respectively. When multiple applications share a machine, Ribbon predicts the optimal system-wide CLF concurrency according to the method described in Section 6.3.1. Ribbon decides the number of flushing threads for each application based on the CLF throughput ratio of the two applications.

6.5.5 Coalescing of Cache Line Flushing

We evaluate the effectiveness of CLF coalescing in Redis running YCSB [**?**] and TPC-C [146] benchmarks. For YCSB, we use its default configuration. The key and value sizes are 24 bytes and 100 bytes, respectively. We run 24 clients threads.

Our first evaluation compares the dirtiness of flushed cache lines with and without CLF coalescing. Table 6.7 presents the results. The baseline version results in 0.32 to 0.56 cache line dirtiness in tested workloads, except for the read-only workloads (YCSB-C). After the optimization, the cache line dirtiness is increased to 0.4-0.68. For each workload, the coalescing effectively reduces traffic and CLF by 20%-45%.



(a) Write-intensive workloads

(b) Read-mostly workloads

Figure 6.14: The performance improvement by the CLF coalescing.

We quantify the impact of the improved cache line dirtiness on the overall performance, as reported in Figure 6.14. The increased dirtiness results in 18% to 33% performance improvement (*w. coalescing*) for write-intensive workloads. For the read-mostly workloads, performance improvements are less than write-based workloads, because these read-only workloads generate far less write traffic. We also compare the CLF coalescing technology with the combination of the concurrency control and proactive CLF (*w. cc+pclf*). We observe that the CLF coalescing achieves 5.3x higher performance improvements than the combination. This performance improvement highlights the effectiveness of the CLF coalescing.

6.6 Related Work

Persistency models have been proposed to characterize and direct CLF. Pelley et al. [30] introduce strict and relaxed persistency and consider persistency models as an extension to memory consistency model. They propose strict, epoch, and strand persistency models and provide a persistent queue implementation. Other works [149, 28, 160, 22, 161] propose various optimizations to relax the constraints on persistence ordering. Ribbon is generally applicable to various persistency models.

CLF-oriented optimizations. Lazy Persistency [32] avoids eager cache flushing and relies on natural eviction from the cache hierarchy to persist data. Their solution detects persistency failures by calculating the checksum of each persistency region. This approach trades off rare persistency failure with a complex recovery

procedure. NV-Tree [152] quantifies that CLF causes over 90% persistency cost in persistent B+-tree data structure. They propose to decouple tree leaves from internal nodes and only maintain the persistency of leaf nodes. In-cacheline log [38] supports fine-grained checkpointing that writes the cache hierarchy to NVM at the beginning of each epoch. They place undo log and its logged data structure in the same cache line to reduce CLF. Link-free and soft algorithms [159] implement a durable concurrent set that only persists set members but avoids persisting pointers to eliminate unnecessary CLF. Software Cache [121] implements a resizable cache to combine writebacks and reduce CLF. Hardware modifications in the cache hierarchy and new instructions [158, 165] are also proposed to reduce the latency of CLF. Also, some cache designs use (relaxed) non-volatile memories [166, 167, 168], which naturally eliminates CLF.

Many other efforts that use CLF to enable crash consistency provide solutions in NVM-aware programming models [24, 23], language-level persistency [169, 25]. Our solution is generally applicable to most of the existing software interfaces as their building block relies on CLF. Unlike hardware-based solutions, we do not change hardware. We use commonly available hardware counters on existing architectures. We summarize software and hardware-based solutions, as well as optimizations for concurrency controls as follows.

System software, such as file systems PMFS [28] and BPFS [26], introduce a buffered epoch persistency model. Persistent operations within an epoch can be reordered to improve the persist concurrency, while orders of persists across epochs are enforced. SCMFS [27] and NOVA [29] are NVM-aware file systems with failure-atomic, scalability optimizations.

Libraries, such as Mnemosyne [22] and NV-Heaps [170], support programmer's annotation of persistent data structures. Mnemosyne [22] keeps a per-thread log for improving concurrency and uses streaming writes to NVM. NV-Heaps [170] provides type-safe pointers and garbage collection for failure atomicity on NVM. Kamino-tx [171] and Intel's PMDK [20] enable transactional updates to NVM.

Hardware-based solutions extend existing instruction sets [119, 172, 173], modify cache hierarchy or add new interfaces to memory subsystems [160, 174], to provide low-overhead support for crash consistency on NVM. Recently, works that rely on a hybrid of DRAM and NVM memory subsystem [120, 153, 175] to speedup

logging into DRAM and persist later to NVM off the critical path.

Concurrency control has been studied on GPU and CPU to improve performance. Kayiran et al. [176] propose a mechanism to balance the system-wide memory and interconnect congestion and dynamically decide the level of GPU concurrency. Li et al. [177] reduce thread-level parallelism to mitigate page thrashing, which brings significant pressure on memory management, on Unified Memory. On the Optane architecture, Yang et al. [16] identify contention on a single DIMM, when a large number of threads access it. Their work proposes using non-interleaved memory mapping onto NVM and binds each DIMM to a specific thread to avoid contention. Our approach requires no modification in virtual memory mapping and can dynamically adjust concurrency without statically binding NVDIMMs to threads. Curtis-Maury et al. [178] and Li et al. [179, 180] use performance models to select thread-level or process-level concurrency for best performance on CPU. Our design does not use performance models because and provides focused guidance on CLF.

6.7 Summary

CLF is critical for ensuring data consistency in NVM. It is a building block for many NVM-aware applications and systems. However, the high overhead of CLF creates a new "memory wall" unseen in the traditional volatile memory. We analyze the performance of CLF in diverse NVM-aware workloads on NVM hardware. We design and implement Ribbon to optimize CLF mechanisms through a decoupled concurrency control and proactive CLF to change cache line status. Ribbon also uses cache line coalescing as an application-specific solution for those with low dirtiness in flushed cache lines, achieving an average 13.9% improvement (up to 33.3%). For a variety of workloads, Ribbon achieves up to 49.8% improvement (14.8% on average) of the performance.

Chapter 7

Conclusion and Future Work

NVM technologies have many appealing features relative to traditional DRAM and disk technologies. They fundamentally change the landscape of memory and storage systems. However, adopting NVM into the current computer systems raises many challenges, such as data placement and data crash consistency. Existing software solutions can compromise the NVM performance with their high software overheads, and fail to achieve high-performance and cost-efficiency. This dissertation studies the data placement and data crash consistency problems of NVM-based HMS, and proposes a collection of new software solutions to achieve high performance.

This dissertation first presents Unimem, a runtime solution that automatically and transparently manage data placement on HMS for iterative MPI-based applications. Leveraging online profiling and performance models, Unimem characterizes memory access patterns associated with data objects, and minimizes unnecessary data movement. Experimental results show that Unimem effectively bridges the performance gap between NVM and DRAM, and demonstrate better performance than a state-of-the-art software-based solution.

Then this dissertation proposes Tahoe, a runtime system that addresses the data placement problem for task parallel programs on NVM-based HMS. Tahoe leverages task metadata and representative tasks to collect memory access information and make data migration decisions. Tahoe uses a hybrid performance model that combines the power of both machine learning modeling and analytical modeling to decide optimal data placement for multiple tasks. Experimental results demostrate Tahoe achieves higher performance than a conventional HMS-oblivious runtime and

state-of-the-art HMS-aware solutions.

This dissertation also investigates the transactions performance on NVM. We identify that small random writes in metadata modifications and locality-oblivious memory allocation in traditional NVM transaction systems mismatch NVM architecture. We present ArchTM, a NVM transaction system based on two design principles: avoiding small writes and encouraging sequential writes. ArchTM is a variant of CoW-based system to reduce write traffic to NVM. Unlike conventional CoW schemes, ArchTM reduces metadata modifications through a scalable lookup table on DRAM. ArchTM introduces an annotation mechanism to ensure crash consistency and a locality-aware data path in memory allocation to increases coalesable writes inside NVM devices.

Finally, this dissertation optimizes the cache line flushing (CLF) mechanism which is critical to ensure crash consistency on NVM. We reveal the characteristics of CLF on real NVM hardware. Based on the characterization, we introduce a runtime system, called Ribbon. Ribbon decouples CLF from the application and applies model-guided optimizations to the CLF mechanism. We also investigate the cause for low dirtiness in flushed cache lines in in-memory database workloads and provide cache line coalescing as an application specific solution to achieve performance.

Although we believe that the software solutions of this dissertation are beneficial and useful, they also have limitations. For instance, Chapter 3 and Chapter 4 propose runtime solutions to solve the data placement of two types of HPC applications on the NVM-based HMS. In these two works, we assume a single application occupies a bare-metal machine. However, in current data centers, it is also common for multiple applications to share a machine running in a virtualization environment. The data placement problem is especially challenging for multiple applications in a virtualized system. Because multiple applications can compete for memory capacity on HMS. When deciding data placement, we have to consider the competition between different applications. To solve it, we plan to use a machine learning model to capture complex memory-access patterns manifested by multiple applications, and predict future memory accesses. Based on the prediction results of the machine learning model, we can prefetch data from NVM to DRAM, or proactively evict data from DRAM to NVM, which can reduce the data movement overhead.

Bibliography

[1] S. Lee. Technology scaling challenges and opportunities of memory devices. In *2016 IEEE International Electron Devices Meeting (IEDM)*, pages 1.1.1–1.1.8, Dec 2016.

[2] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 361–372, Piscataway, NJ, USA, 2014. IEEE Press.

[3] J. A. Mandelman, R. H. Dennard, G. B. Bronner, J. K. DeBrosse, R. Divakaruni, Y. Li, and C. J. Radens. Challenges and future directions for the scaling of dynamic random-access memory (dram). *IBM Journal of Research and Development*, 46(2.3):187–212, March 2002.

[4] F. T. Hady, A. Foong, B. Veal, and D. Williams. Platform storage performance with 3d xpoint technology. *Proceedings of the IEEE*, 105(9):1822–1833, Sep. 2017.

[5] Kai Wu, Yingchao Huang, and Dong Li. Unimem: Runtime Data Management on Non-volatile Memory-based Heterogeneous Main Memory. In *SC*, 2017.

[6] A. K. Jain, S. Lloyd, and M. Gokhale. Performance assessment of emerging memories through fpga emulation. *IEEE Micro*, 39(1):8–16, Jan 2019.

[7] Ivy Peng, Kai Wu, Jie Ren, Dong Li, and Maya Gokhale. Demystifying the performance of hpc scientific applications on nvm-based memory systems. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020.

[8] Kai Wu, Jie Ren, and Dong Li. Runtime data management on non-volatile memory-based heterogeneous memory for task-parallel programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18. IEEE Press, 2018.

[9] Kai Wu, Jie Ren, Ivy Peng, and Dong Li. Archtm: Architecture-aware, high performance transaction for persistent memory. In *19th USENIX Conference on*

File and Storage Technologies (FAST 21), pages 141–153. USENIX Association, February 2021.

[10] Kai Wu, Ivy Peng, Jie Ren, and Dong Li. Ribbon: High performance cache line flushing for persistent memory. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, PACT '20. Association for Computing Machinery, 2020.

[11] Kosuke Suzuki and Steven Swanson. The Non-Volatile Memory Technology Database (NVMDB). Technical Report CS2015-1011, Department of Computer Science & Engineering, University of California, San Diego, 2015. http://nvmdb.ucsd.edu.

[12] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the intel optane DC persistent memory module. *CoRR*, abs/1903.05714, 2019.

[13] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 24–33, New York, NY, USA, 2009. ACM.

[14] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. *ACM SIGARCH Computer Architecture News*, 37(3):2–13, 2009.

[15] Ivy B. Peng, Maya B. Gokhale, and Eric W. Green. System evaluation of the intel optane byte-addressable NVM. In *Proceedings of the International Symposium on Memory Systems*. ACM, 2019.

[16] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, 2020.

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

[18] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali. Single machine graph analytics on massive datasets using intel optane dc persistent memory, 2019.

[19] Intel. Persistent memory development kit, 2014.

[20] Intel. Persistent Memory Development Kit. `https://pmem.io/`.

[21] 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 Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[22] Haris Volos, Andres Jaan Tack, and Michael M Swift. Mnemosyne: Lightweight persistent memory. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 91–104. ACM, 2011.

[23] Dhruva R Chakrabarti, Hans-J Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *ACM SIGPLAN Notices*, volume 49, pages 433–452. ACM, 2014.

[24] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. Nvthreads: Practical persistence for multi-threaded applications. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 468–482. ACM, 2017.

[25] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M Chen, and Thomas F Wenisch. Persistency for synchronization-free regions. In *ACM SIGPLAN Notices*, volume 53, pages 46–61. ACM, 2018.

[26] 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 ACM SIGOPS 22nd symposium on Operating systems principles*, pages 133–146. ACM, 2009.

[27] Xiaojian Wu and AL Reddy. Scmfs: a file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 39. ACM, 2011.

[28] 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 Ninth European Conference on Computer Systems*, page 15. ACM, 2014.

[29] 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, 2016.

[30] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory Persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, 2014.

[31] S. Yang, K. Wu, Y. Qiao, D. Li, and J. Zhai. Algorithm-Directed Crash Consistence in Non-volatile Memory for HPC. In *2017 IEEE International Conference on Cluster Computing*, 2017.

[32] M. Alshboul, J. Tuck, and Y. Solihin. Lazy Persistency: A High-Performing and Write-Efficient Software Persistency Technique. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture*, June 2018.

[33] Yiying Zhang and Steven Swanson. A study of application performance with non-volatile main memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10. IEEE, 2015.

[34] Jie Ren, Kai Wu, and Dong Li. Exploring Non-Volatility of Non-Volatile Memory for High Performance Computing Under Failures. In *2017 IEEE International Conference on Cluster Computing*, 2020.

[35] Jie Ren, Kai Wu, and Dong Li. Understanding application recomputability without crash consistency in non-volatile memory. In *Proceedings of the Workshop on Memory Centric High Performance Computing*, MCHPC'18, 2018.

[36] E. R. Giles, K. Doshi, and P. Varman. SoftWrAP: A Lightweight Framework for Transactional Support of Storage Class Memory. In *2015 31st Symposium on Mass Storage Systems and Technologies*, May 2015.

[37] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey III, Dhruva R. Chakrabarti, and Michael L. Scott. Dalí: A Periodically Persistent Hash Map. In *31st International Symposium on Distributed Computing (DISC 2017)*, Leibniz International Proceedings in Informatics (LIPIcs), 2017.

[38] Nachshon Cohen, David T Aksun, Hillel Avni, and James R Larus. Fine-grain checkpointing with in-cache-line logging. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 441–454. ACM, 2019.

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

[40] Pradeep Fernando, Ada Gavrilovska, Sudarsun Kannan, and Greg Eisenhauer. Nvstream: Accelerating hpc workflows with nvram-based transport for streaming objects. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '18, 2018.

[41] 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*, 2014.

[42] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. High-Performance Transactions for Persistent Memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[43] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. Atomic In-place Updates for Non-volatile Main Memories with Kamino-Tx. In *Proceedings of the Twelfth European Conference on Computer Systems*, 2017.

[44] H. Shu, H. Chen, H. Liu, Y. Lu, Q. Hu, and J. Shu. Empirical Study of Transactional Management for Persistent Memory. In *2018 IEEE 7th Non-Volatile Memory Systems and Applications Symposium*, 2018.

[45] H. Wan, Y. Lu, Y. Xu, and J. Shu. Empirical Study of Redo and Undo Logging in Persistent Memory. In *5th Non-Volatile Memory Systems and Applications Symposium*, 2016.

[46] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, page 271–282, New York, NY, USA, 2018. Association for Computing Machinery.

[47] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2011.

[48] Virendra J. Marathe, Achin Mishra, Amee Trivedi, Yihe Huang, Faisal Zaghloul, Sanidhya Kashyap, Margo Seltzer, Tim Harris, Steve Byan, Bill Bridge, and Dave Dice. Persistent memory transactions. *CoRR*, abs/1804.00701, 2018.

[49] Y. Lu, J. Shu, L. Sun, and O. Mutlu. Loose-Ordering Consistency for Persistent Memory. In *IEEE 32nd International Conference on Computer Design*, 2014.

[50] Pedro Ramalhete, Andreia Correia, Pascal Felber, and Nachshon Cohen. Onefile: A wait-free persistent transactional memory. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019, Portland, OR, USA, June 24-27, 2019*, 2019.

[51] P. Zardoshti, T. Zhou, Y. Liu, and M. Spear. Optimizing persistent memory transactions. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2019.

[52] R. Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. Durable transactional memory can scale with timestone. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, 2020.

[53] Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. Pisces: A scalable and efficient persistent transactional memory. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[54] Jason Evans. A scalable concurrent malloc (3) implementation for freebsd. Technical report, 2006. http://jemalloc.net/.

[55] Wolfram Gloger. Wolfram Gloger's malloc. `http://www.malloc.de/en/`.

[56] Paul Menage Sanjay Ghemawat. TCMalloc: Thread-Caching Malloc. `http://goog-perftools.sourceforge.net/doc/`.

[57] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, 2000.

[58] David Schwalb, Tim Berning, Martin Faust, Markus Dreseler, and Hasso Plattner. nvm malloc: Memory allocation for nvram. In *ADMS@VLDB*, 2015.

[59] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. Makalu: Fast recoverable allocation of non-volatile memory. *SIGPLAN Not.*, 51(10):677–694, October 2016.

[60] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. In *PPOPP*, 1995.

[61] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing Locality and Independence with Logical Regions. In *SC*, 2012.

[62] D. H. Bailey, L. Dagum, E. Barszcz, and H. D. Simon. Nas parallel benchmark results. In *Supercomputing '92: Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 386–393, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[63] Haris Volos, Guilherme Magalhaes, Ludmila Cherkasova, and Jun Li. Quartz: A Lightweight Performance Emulator for Persistent Memory Software. In *Annual Middleware Conference (Middleware)*, 2015.

[64] Dong Li, Jeffrey Vetter, Gabriel Marin, Collin McCurdy, Cristi Cira, Zhuo Liu, and Weikuan Yu. Identifying Opportunities for Byte-Addressable Non-Volatile Memory in Extreme-Scale Scientific Applications. In *International Parallel and Distributed Processing Symposium*, 2012.

[65] Panruo Wu, Dong Li, Zizhong Chen, Jeffrey Vetter, and Sparsh Mittal. Algorithm-Directed Data Placement in Explicitly Managed No-Volatile Memory. In *ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2016.

[66] John D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. `https://www.cs.virginia.edu/stream`.

[67] Tim Besard. Pointer-chasing memory benchmark. `https://github.com/maleadt/pChase`.

[68] M. Silvano and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, 1990.

[69] Neha Agarwal, David Nellans, Mark Stephenson, Mike O'Connor, and Stephen W. Keckler. Page Placement Strategies for GPUs within Heterogeneous Memory Systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[70] Bin Wang, Bo Wu, Dong Li, Xipeng Shen, Weikuan Yu, Yizheng Jiao, and Jeffrey S. Vetter. Exploring Hybrid Memory for GPU Energy Efficiency through Software-Hardware Co-Design. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013.

[71] Chen Ding and Ken Kennedy. Bandwidth-based performance tuning and prediction. In *International Conference on Parallel Computing and Distributed Systems*, 1999.

[72] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[73] Maged M. Michael. Scalable Lock-free Dynamic Memory Allocation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2004.

[74] C. Lattner. *LLVM: An Infrastructure for Multi-Stage Optimization*. PhD thesis, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, 2002.

[75] Venkatesan T. Chakaravarthy. New results on the computability and complexity of points–to analysis. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2003.

[76] Peter Macko. PCMSIM: A Simple PCM Block Device Simulator for Linux. `https://code.google.com/p/pcmsim`.

[77] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight Persistent Memory. In *Architectural Support for Programming Languages and Operating Systems*, 2011.

[78] P. Fischer and J. Lottes. *nek5000 Web page*. http://nek5000.mcs.anl.gov, Web page, 2008.

[79] 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 Eleventh European Conference on Computer Systems (EuroSys)*, 2016.

[80] Michael Giardino, Kshitij Doshi, and Bonnie Ferri. Soft2LM: Application Guided Heterogeneous Memory Management. In *International Conference on Networking, Architecture, and Storage (NAS)*, 2016.

[81] Felix Xiaozhu Lin and Xu Liu. memif: Towards Programming Heterogeneous Memory Asynchronously. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[82] Du Shen, Xu Liu, and Felix Xiaozhu Lin. Characterizing Emerging Heterogeneous Memory. In *ACM SIGPLAN International Symposium on Memory Management (ISMM)*, 2016.

[83] Alan Bivens, Parijat Dube, Michele Franceschini, John Karidis, Luis Lastras, and Mickey Tsao. Architectural Design for Next Generation Heterogeneous Memory Systems. In *International Memory Workshop*, 2010.

[84] Moinuddin K. Qureshi, Michele Franchescini, Vijayalakshmi Srinivasan, Luis Lastras, Bulent Abali, and John Karidis. Enhancing Lifetime and Security of PCM-Based Main Memory with Start-Gap Wear Leveling. In *MICRO*, 2009.

[85] Moinuddin K. Qureshi, Viji Srinivasan, and Jude A. Rivers. Scalable High-Performance Main Memory System Using Phase-Change Memory Technology. In *ISCA*, 2009.

[86] HanBin Yoon, Justin Meza, Rachata Ausavarungnirun, Rachael Harding, and Onur Mutlu. Row Buffer Locality Aware Caching Policies for Hybrid Memories. In *International Conference on Computer Design (ICCD)*, 2012.

[87] François Broquedis, Nathalie Furmento, Brice Goglin, Raymond Namyst, and Pierre-André Wacrenier. Dynamic Task and Data Placement over NUMA Architectures: An OpenMP Runtime Perspective. In *Proceedings of the 5th International Workshop on OpenMP: Evolving OpenMP in an Age of Extreme Parallelism*, IWOMP, 2009.

[88] Stephen L Olivier, Allan K Porterfield, Kyle B Wheeler, Michael Spiegel, and Jan F Prins. OpenMP Task Scheduling Strategies for Multicore NUMA Systems. *IJHPCA*, 26(2), 2012.

[89] J. Li, X. Ma, K. Singh, M. Schulz, B. R. de Supinski, and S. A. McKee. Machine Learning based Online Performance Prediction for Runtime Parallelization and Task Scheduling. In *ISPASS*, 2009.

[90] Seongdae Yu, Seongbeom Park, and Woongki Baek. Design and Implementation of Bandwidth-aware Memory Placement and Migration Policies for Heterogeneous Memory Systems. In *Proceedings of the International Conference on Supercomputing (ICS)*, 2017.

[91] Wei Liu, Kai Wu, Jialin Liu, Feng Chen, and Dong Li. Performance evaluation and modeling of hpc i/o on non-volatile memory. In *2017 International Conference on Networking, Architecture, and Storage (NAS)*, 2017.

[92] Ivy Peng, Marty McFadden, Eric Green, Keita Iwabuchi, Kai Wu, Dong Li, Roger Pearce, and Maya Gokhale. Umap: Enabling application-driven optimizations for page management. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, 2019.

[93] Jie Ren, Jiaolin Luo, Kai Wu, Minjia Zhang, Hyeran Jeon, and Dong Li. Sentinel: Efficient tensor migration and allocation on heterogeneous memory systems for deep learning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.

[94] D. Li, B. R. de Supinski, M. Schulz, K. Cameron, and D. S. Nikolopoulos. Hybrid MPI/OpenMP Power-aware Computing. In *International Symposium on Parallel Distributed Processing (IPDPS)*, 2010.

[95] Matthew Curtis-Maury, Ankur Shah, Filip Blagojevic, Dimitrios S. Nikolopoulos, Bronis R. de Supinski, and Martin Schulz. Prediction Models for Multi-dimensional Power-performance Optimization on Many Cores. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[96] N. Kappiah, V. W. Freeh, and D. K. Lowenthal. Just In Time Dynamic Voltage Scaling: Exploiting Inter-Node Slack to Save Energy in MPI Programs. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, 2005.

[97] C. Liu, A. Sivasubramaniam, M. Kandemir, and M. J. Irwin. Exploiting Barriers to Optimize Power Consumption of CMPs. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.

[98] ALEJANDRO DURAN, EDUARD AYGUADÉ, ROSA M. BADIA, JESÚS LABARTA, LUIS MARTINELL, XAVIER MARTORELL, and JUDIT PLANAS. OmpSs: A Proposal for Programming Heterogeneous Multi-core Architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[99] Barcelona Supercomputing Center. Nanos ++. `https://pm.bsc.es/nanox`.

[100] Barcelona Supercomputing Center. BSC application repository. `https://pm.bsc.es/projects/bar/wiki/Applications`.

[101] Matt Poremba and Yuan Xie. Nvmain: An architectural-level main memory simulator for emerging non-volatile memories. In *ISVLSI*, 2012.

[102] Jeffrey C. Mogul, Eduardo Argollo, Mehul Shah, and Paolo Faraboschi. Operating system support for nvm+dram hybrid main memory. In *HotOS*, 2009.

[103] Y. Huang and D. Li. Performance Modeling for Optimal Data Placement on GPU with Heterogeneous Memory Systems. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, 2017.

[104] Sparsh Mittal. A Survey of Techniques for Architecting TLBs. *Concurrency and Computation: Practice and Experience*, 29(10):e4061–n/a, 2017.

[105] Nadav Amit. Optimizing the TLB Shootdown Algorithm with Page Access Tracking. In *USENIX ATC*, 2017.

[106] Wei Wei, Dejun Jiang, Sally A. McKee, Jin Xiong, and Mingyu Chen. Exploiting Program Semantics to Place Data in Hybrid Memory. In *PACT*, 2015.

[107] Luiz Ramos, Eugene Gorbatov, and Ricardo Bianchini. Page Placement in Hybrid Memory Systems. In *International Conference on Supercomputing (ICS)*, 2011.

[108] D. Komatitsch and J. Tromp. Introduction to the Spectral Element Method for Three-dimensional Seismic Wave Propagation. *Geophysical Journal International*, 139(3):806–822, Dec 1999.

[109] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian. CHOP: Integrating DRAM Caches for CMP Server Platforms. *IEEE Micro*, 31(1):99–108, 2011.

[110] S. Bock, B. R. Childers, R. Melhem, and D. Mossé. Concurrent Migration of Multiple Pages in Software-managed Hybrid Main Memory. In *ICCD*, 2016.

[111] S. Perarnau, J. A. Zounmevo, B. Gerofi, K. Iskra, and P. Beckman. Exploring Data Migration for Future Deep-Memory Many-Core Systems. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, 2016.

[112] Vassilis Papaefstathiou, Manolis G.H. Katevenis, Dimitrios S. Nikolopoulos, and Dionisios Pnevmatikatos. Prefetching and Cache Management Using Task Lifetimes. In *ICS*, 2013.

[113] X. Ni, N. Jain, K. Chandrasekar, and L. V. Kale. Runtime Techniques for Programming with Fast and Slow Memory. In *CLUSTER*, 2017.

[114] Abhisek Pan and Vijay S. Pai. Runtime-driven Shared Last-level Cache Management for Task-parallel Programs. In *SC*, 2015.

[115] Joy Arulraj, Matthew Perron, and Andrew Pavlo. Write-behind logging. *Proc. VLDB Endow.*, 10(4):337–348, November 2016.

[116] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, page 5, USA, 2011. USENIX Association.

[117] 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 ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, 2009.

[118] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. Efficient Persist Barriers for Multicores. In *International Symposium on Microarchitecture*, 2015.

[119] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch. Delegated Persist Ordering. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture*, 2016.

[120] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutiu. ThyNVM: Enabling Software-transparent Crash Consistency in Persistent Memory Systems. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture*, 2015.

[121] P. Li, D. R. Chakrabarti, C. Ding, and L. Yuan. Adaptive software caching for efficient nvram data persistence. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017.

[122] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. Dudetm: Building durable transactions with decoupling for persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, 2017.

[123] Theodore Michailidis, Alex Delis, and Mema Roussopoulos. Mega: Overcoming traditional problems with os huge page management. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, SYSTOR '19, 2019.

[124] Bobby Powers, David Tench, Emery D. Berger, and Andrew McGregor. Mesh: Compacting memory management for c/c++ applications. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, 2019.

[125] Rdtscp — read time-stamp counter and processor id. `www.felixcloutier.com/x86/rdtscp`.

[126] Sanidhya Kashyap, Changwoo Min, Kangnyeon Kim, and Taesoo Kim. A scalable ordering primitive for multicore machines. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, 2018.

[127] A. Adya. Weak consistency: A generalized theory and optimistic implementations for distributed transactions. Technical report, USA, 1999.

[128] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. In *Proceedings of the 1995*

ACM SIGMOD International Conference on Management of Data, SIGMOD '95, 1995.

[129] Heiner Litz, David Cheriton, Amin Firoozshahian, Omid Azizi, and John P. Stevenson. Si-tm: Reducing transactional memory abort rates through snapshot isolation. *SIGPLAN Not.*, 49(4):383–398, February 2014.

[130] Lois Orosa and Rodolfo Azevedo. Logsi-htm: Log based snapshot isolation in hardware transactional memory. 07 2015.

[131] Torvald Riegel, Christof Fetzer, , and Pascal Felber. Snapshot isolation for software transactional memory. In *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, TRANSACT'06, 2006.

[132] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Feral concurrency control: An empirical investigation of modern application integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, 2015.

[133] S. Lu, A. Bernstein, and P. Lewis. Correct execution of transactions at different isolation levels. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1070–1081, 2004.

[134] Microsoft. Snapshot Isolation in SQL Server. `https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/sql/snapshot-isolation-in-sql-server`.

[135] MySQL. `https://www.mysql.com/`.

[136] Oracle. `www.oracle.com`.

[137] Redis. `https://redis.io/`.

[138] PostgreSQL. `https://www.postgresql.org/`.

[139] mongoDB. `https://www.mongodb.com/`.

[140] Alexander Matveev, Nir Shavit, Pascal Felber, and Patrick Marlier. Read-log-update: A lightweight synchronization mechanism for concurrent programming. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15. Association for Computing Machinery, 2015.

[141] H. Garcia-Molina and K. Salem. Main memory database systems: an overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, 1992.

[142] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. Meet the walkers: Accelerating index traversals for in-memory databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, 2013.

[143] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, 2017.

[144] L. Qiaoyu, L. Jianwei, and X. Yubin. Performance analysis of data organization of the real-time memory database based on red-black tree. In *2010 International Conference on Computing, Control and Industrial Engineering*, 2010.

[145] Pengfei Zuo, Yu Hua, and Jie Wu. Level hashing: A high-performance and flexible-resizing persistent hashing index structure. *ACM Trans. Storage*, 2019.

[146] Scott T. Leutenegger and Daniel Dias. A Modeling Study of the TPC-C Benchmark. In *SIGMOD Record*, 1993.

[147] Simo Neuvonen, Antoni Wolski, Markku manner, and Vilho Raatikka. Telecom Application Transaction Processing Benchmark . `http://tatpbenchmark.sourceforge.net/`.

[148] Key/value datastore for persistent memory. `https://github.com/pmem/pmemkv`.

[149] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via justdo logging. *ACM SIGARCH Computer Architecture News*, 44(2):427–442, 2016.

[150] Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung. ido: Compiler-directed failure atomicity for nonvolatile memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[151] Shimin Chen and Qin Jin. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment*, 8(7):786–797, 2015.

[152] 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 *13th {USENIX} Conference on File and Storage Technologies ({FAST} 15)*, pages 167–181, 2015.

[153] 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 International Conference on Management of Data*, pages 371–386. ACM, 2016.

[154] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable transient inconsistency in byte-addressable persistent b+-tree. In *16th {USENIX} Conference on File and Storage Technologies ({FAST} 18)*, pages 187–200, 2018.

[155] Pengfei Zuo, Yu Hua, and Jie Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 461–476, 2018.

[156] Katelin A Bailey, Peter Hornyack, Luis Ceze, Steven D Gribble, and Henry M Levy. Exploring storage class memory with key value stores. In *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, page 4. ACM, 2013.

[157] Joy Arulraj, Andrew Pavlo, and Subramanya R Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 707–722. ACM, 2015.

[158] Sanketh Nalli, Swapnil Haria, Mark D Hill, Michael M Swift, Haris Volos, and Kimberly Keeton. An analysis of persistent memory use with whisper. In *ACM SIGARCH Computer Architecture News*, volume 45, pages 135–148. ACM, 2017.

[159] Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. Efficient lock-free durable sets. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):128, 2019.

[160] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. Kiln: Closing the Performance Gap between Systems with and without Persistent Support. In *MICRO*, 2013.

[161] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M Chen, and Thomas F Wenisch. High-performance transactions for persistent memories. *ACM SIGPLAN Notices*, 51(4):399–411, 2016.

[162] Intel. Redis, enhanced to use persistent memory - limited prototype. `https://github.com/pmem/redis/tree/3.2-nvml`.

[163] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81, 2008.

[164] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, 2010.

[165] Seunghee Shin, James Tuck, and Yan Solihin. Hiding the long latency of persist barriers using speculative execution. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 175–186. IEEE, 2017.

[166] Mitchelle Rasquinha, Dhruv Choudhary, Subho Chatterjee, Saibal Mukhopadhyay, and Sudhakar Yalamanchili. An energy efficient cache design using spin torque transfer (stt) ram. In *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design*, pages 389–394. ACM, 2010.

[167] Clinton W Smullen, Vidyabhushan Mohan, Anurag Nigam, Sudhanva Gurumurthi, and Mircea R Stan. Relaxing non-volatility for fast and energy-efficient stt-ram caches. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 50–61. IEEE, 2011.

[168] Zhenyu Sun, Xiuyuan Bi, Hai Helen Li, Weng-Fai Wong, Zhong-Liang Ong, Xiaochun Zhu, and Wenqing Wu. Multi retention level stt-ram cache designs with a dynamic refresh scheme. In *proceedings of the 44th annual IEEE/ACM international symposium on microarchitecture*, pages 329–338. ACM, 2011.

[169] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M Chen, Satish Narayanasamy, and Thomas F Wenisch. Language-level persistency. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 481–493. IEEE, 2017.

[170] Joel Coburn, Adrian Caulfield, Ameen Akel, Laura Grupp, Rajesh Gupta, Ranjit Jhala, and Steve Swanson. NV-heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[171] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. Atomic in-place updates for non-volatile main memories with kamino-tx. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 499–512. ACM, 2017.

[172] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. Efficient Persist Barriers for Multicores. In *Proceedings of the 48th International Symposium on Microarchitecture*.

[173] Seunghee Shin, Satish Kumar Tirukkovalluri, James Tuck, and Yan Solihin. Proteus: A flexible and fast software supported hardware logging approach for nvm. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 178–190. IEEE, 2017.

[174] Arpit Joshi, Vijay Nagarajan, Stratis Viglas, and Marcelo Cintra. Atom: Atomic durability in non-volatile memory through hardware logging. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 361–372. IEEE, 2017.

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

decoupling for persistent memory. In *ACM SIGARCH Computer Architecture News*, volume 45, pages 329–343. ACM, 2017.

[176] Onur Kayiran, Nachiappan Chidambaram Nachiappan, Adwait Jog, Rachata Ausavarungnirun, Mahmut T Kandemir, Gabriel H Loh, Onur Mutlu, and Chita R Das. Managing gpu concurrency in heterogeneous architectures. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 114–126. IEEE, 2014.

[177] Chen Li, Rachata Ausavarungnirun, Christopher J Rossbach, Youtao Zhang, Onur Mutlu, Yang Guo, and Jun Yang. A framework for memory oversubscription management in graphics processing units. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 49–63. ACM, 2019.

[178] Matthew Curtis-Maury, Ankur Shah, Filip Blagojevic, Dimitrios S. Nikolopoulos, Bronis R. de Supinski, and Martin Schulz. Prediction models for multi-dimensional power-performance optimization on many cores. In *International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[179] Dong Li, Bronis de Supinski, Martin Schulz, Dimitrios S. Nikolopoulos, and Kirk W. Cameron. Hybrid MPI/OpenMP Power-Aware Computing. In *International Parallel and Distributed Processing Symposium*, 2010.

[180] Dong Li, Dimitrios S. Nikolopoulos, Kirk W. Cameron, Bronis de Supinski, and Martin Schulz. Power-Aware MPI Task Aggregation Prediction for High-End Computing Systems. In *International Parallel and Distributed Processing Symposium*, 2010.