

## ABSTRACT

WANG, TIANCONG. Architectural and Language Support for Efficient Programming on Non-Volatile Memory. (Under the direction of James Tuck).

Emerging Non-Volatile Main Memory (NVMM) technologies create a new opportunity for writing programs with a large, byte-addressable persistent storage that can be accessed through regular memory instructions. These new memory-as-storage technologies impose a significant long-term challenges to current programming models. All of our existing programming models assumes data in memory to be volatile and when data need to be persistent, we use file operations to serialize them into files. In this dissertation, we propose several architectural and language supports to improve the efficiency of programming with NVMM, in the aspect of both performance and programmability.

The first part of this dissertation deals with the problem of addressing persistent data in NVMM. In particular, some emerging persistent programming frameworks, like the Persistent Memory Development Kit (PMDK), implement relocatable persistent objects that can be mapped anywhere in the virtual address space. To make this work, persistent objects are referenced using object identifiers (ObjectID), rather than pointers, that need to be translated to an address before the object can be read or written. Frequent translation from ObjectID to address incurs significant overhead. We propose treating ObjectIDs as a new persistent memory address space and provide hardware support for efficiently translating ObjectIDs to virtual addresses. With our design, a program can use load and store instructions to directly access persistent data using ObjectIDs, and these new instructions can reduce the programming complexity of this system. We also describe several possible microarchitectural designs and evaluate them. We demonstrate for the *Pipelined* implementation that our design has an average speedup of 1.96 $\times$  and 1.58 $\times$  on an in-order and out-of-order processor, respectively, over the baseline software translation system on six microbenchmarks of RANDOM pattern (place persistent data randomly into 32 persistent pools). For the same in-order and out-of-order microarchitectures, we measure a speedup of 1.17 $\times$  and 1.12 $\times$ , respectively, on the TPC-C application, and 1.19  $\times$  and 1.18  $\times$  speedup, respectively on Vacation application, on of SEPARATE pattern (when B+Trees or linked-list and red-black trees are put in different pools and rewritten to use our new hardware).

The second part of this dissertation deals with validating permissions to access ObjectIDs for programs. The ObjectIDs consist of an identifier to a relocatable region, or a *pool*, and a byte offset within the pool to locate persistent data. This provides direct access to persistent memory to programmers but also incurs possible security issue when a program tries to access a pool without having permissions. It incurs large overhead and programming burden if a programmer has to check permission on every ObjectID before accessing it. We identify permission checking in hardware as a critical mechanism that must be included when translating ObjectIDs to addresses in order to

simplify programming and fully benefit from hardware translation. To support it, we add a System Persistent Object Table (SPOT) to support translation and permissions checks on ObjectIDs. The SPOT holds all known pools, their physical address, and their permissions information in memory. When a program attempts to access a persistent object, the SPOT is consulted and its permissions are verified without trapping to the operating system. We have implemented our new design in a cycle accurate simulator and compared it with software only approaches and prior work. We find that our design offers a compelling 3.3x speedup on average for the microbenchmarks we studied for RANDOM pattern and 1.4x and 1.7x speedup on TPC-C and vacation respectively for SEPARATE pattern.

The last part of this dissertation deals with a new dimension introduced by persistent programs: how to migrate persistent data in the presence of code changes. In this work, we propose *the persistent data retention model*, and we argue it should be specified as part of persistent programming models to describe what happens to persistent data when code that uses it is modified. We identify two models present in prior work but not described as such, the Reset Model and Manual Model, and we propose a new one called the Automatic Model. The Reset Model discards all persistent data when a program changes leading to performance overheads and write amplification. In contrast, if data is to be retained, the Manual Model relies on the programmer to implement code that upgrades data from one version of the program to the next. This reduces overheads but places a larger burden on the programmer. We propose the Automatic Model to assist a programmer by automating some or all of the conversion. We describe one such automatic approach, Lazily Extendable Data Structures, that uses language extensions and compiler support to reduce the effort and complexity associated with updating persistent data. We evaluate our PDRMs in the context of the Persistent Memory Development Kit (PMDK) using kernels and the TPC-C application. Manual Model shows an overhead of 2.90% to 4.10% on average, and LEDS shows overhead of 0.45% to 10.27% on average, depending on the workload. LEDS reduces the number of writes by 26.36% compared to Manual Model. Furthermore, LEDS significantly reduces the programming complexity by relying on the compiler to migrate persistent data.

© Copyright 2018 by Tiancong Wang

All Rights Reserved

Architectural and Language Support for Efficient Programming on Non-Volatile Memory

by  
Tiancong Wang

A dissertation submitted to the Graduate Faculty of  
North Carolina State University  
in partial fulfillment of the  
requirements for the Degree of  
Doctor of Philosophy

Computer Engineering

Raleigh, North Carolina

2018

APPROVED BY:

---

Yan Solihin

---

Greg Byrd

---

Xipeng Shen

---

James Tuck  
Chair of Advisory Committee

## **DEDICATION**

To my father and mother, for bringing me to this wonderful world, for loving me to your hearts, and  
for making me the luckiest son in the world.

And, to my lovely wife, for being a friend, a soul-mate and a supporter in both of my work and life.

## **BIOGRAPHY**

The author was born in a small town named Anshan in northeastern of China. I am the only child in my family. I was first introduced to computer programming when I was ten years old, and my first programming language is Logo. I taught myself Pascal and algorithms during my high school and got first prize in Olympics competition in Information in 2008. The prize got me admission to enter Peking University in 2009. In order to explore more on the computer system, I selected Microelectronics as my major. I was exposed to a lot of low-level silicon device designs and theories, as well as circuit designs. In 2013, I got an offer from North Carolina State University in the PhD program in Computer Engineering. I started my first year of PhD exploring many topics in the computer area and computer architecture and compilers became my interest so I joined Dr. James Tuck's group in 2014. Since then, I have learned a lot on hardware/software co-designs and started my projects on solving issues related to programming on Non-Volatile Memory. I have interned on the summers of 2016 and 2017, at Synopsys and Google respectively. I will join Google after I graduate. I have defended my thesis on August 17th, 2018.

## ACKNOWLEDGEMENTS

This dissertation would not exist without the guidance and support from many people - my advisor, many professors, my labmates, my friends and family.

I would like to thank my advisor, Dr. James Tuck, to my heart. Thank you for being a role model, inspiring me to love what I do and help others with my best effort. Thank you for being a great mentor, guiding me through my confusions on the topics and helping me figure out technical problems. Also thank you for being a good friend, always offering help when I need it the most.

I also would like to thank Dr. Yan Solihin for cooperating on the MICRO-50 paper and offering help on my researches. I also want to thank other members on my committee, Dr. Greg Byrd and Dr. Xipeng Shen, for offering your valuable opinions to improve my researches.

I want to thank the senior lab-mates, Bagus Wibowo, Joonmoo Huh, Abhinav Agrawal, Seunghee Shin and George Patsilaras for offering your valuable experiences in both academics and also personal lives. I also want to thank Sakthikumaran Sambasivam for helping me on the MICRO-50 and ISCA-18 papers. I also want to thank other lab-mates who offer suggestions on my researches and presentations.

Lastly, I want to thank my parents for your sacrifices to always give me the best in my whole life and thank my wife for accompanying me through many obstacles in the PhD journey.

To all the people who have offered your help to me in my life, thank you and I believe I couldn't achieve anything without your help.

# TABLE OF CONTENTS

<b>LIST OF TABLES</b> .....	<b>viii</b>
<b>LIST OF FIGURES</b> .....	<b>x</b>
<b>Chapter 1 Introduction</b> .....	<b>1</b>
1.1 Address Translation .....	2
1.2 Permission Checks .....	3
1.3 Persistent Data Retention Models .....	4
1.4 Contributions .....	5
1.5 Organization of the Dissertation .....	5
<b>Chapter 2 Background</b> .....	<b>7</b>
2.1 Non-Volatile Main Memory (NVMM) Technologies .....	7
2.2 Persistent Programming .....	8
2.2.1 Persistent Memory Development Kit (PMDK) .....	10
2.3 Our vision and objectives: making persistent programming easier .....	10
<b>Chapter 3 Hardware Supported Persistent Object Address Translation</b> .....	<b>12</b>
3.1 Motivation: Baseline Persistent Programming Model and its limitations .....	13
3.1.1 The Persistent Programming Interface .....	13
3.1.2 Persistent Linked List Example .....	17
3.2 Design: Hardware-supported Address Translation .....	20
3.2.1 New Instructions .....	20
3.2.2 Overall Architecture .....	20
3.2.3 Supporting System and Library Modifications .....	22
3.3 Microarchitecture .....	22
3.3.1 Persistent Object Look-aside Buffer .....	22
3.3.2 Persistent Object Table .....	24
3.3.3 Memory Disambiguation .....	26
3.3.4 Out-of-Order Pipeline .....	27
3.3.5 In-order Pipeline .....	28
3.4 Methodology .....	28
3.4.1 Simulation .....	28
3.5 Evaluation .....	32
3.5.1 Overall Performance .....	32
3.5.2 Reduce Instructions .....	34
3.5.3 Overhead of Durability and Atomicity .....	34
3.5.4 Sensitivity Analysis: POLB size .....	35
3.5.5 Impact of Hardware POT Walk .....	37
3.6 Summary .....	37
<b>Chapter 4 Hardware Supported Permission Checks On Persistent Objects For Performance and Programmability</b> .....	<b>39</b>

4.1	Motivation . . . . .	40
4.1.1	Example . . . . .	40
4.1.2	Permissions Checking Support . . . . .	41
4.1.3	Optimizing Checks in Software . . . . .	42
4.1.4	Permissions Checks with Hardware Supported Translation . . . . .	44
4.2	Design: System Persistent Object Table (SPOT) . . . . .	45
4.2.1	System Persistent Object Table . . . . .	45
4.2.2	Pool Open Using SPOT . . . . .	45
4.2.3	Permission Checking Using SPOT . . . . .	46
4.2.4	Organization of the SPOT . . . . .	46
4.3	Implementation . . . . .	47
4.3.1	System Persistent Object Table . . . . .	47
4.3.2	Permission Checking Logic . . . . .	49
4.4	Evaluation . . . . .	51
4.4.1	Methodology . . . . .	51
4.4.2	Overall Performance . . . . .	52
4.4.3	In-order vs Out-of-order . . . . .	54
4.4.4	Impact of Page Size . . . . .	54
4.4.5	Comparison of HwT+SPOT and native hardware support plus SPOT . . . . .	55
4.4.6	Storage Overhead of SPOT . . . . .	57
4.5	Summary . . . . .	57
<b>Chapter 5 Persistent Data Retention Models . . . . .</b>		<b>58</b>
5.1	Motivation . . . . .	59
5.1.1	Comparison between persistent programming and file-based programming . . . . .	59
5.1.2	PDRMs in existing persistent programming models . . . . .	60
5.1.3	PMDK Library . . . . .	61
5.2	Persistent Data Retention Models . . . . .	63
5.2.1	Reset Model . . . . .	63
5.2.2	Manual Model . . . . .	63
5.2.3	Automatic Model . . . . .	64
5.2.4	Lazily Extendable Data Structures . . . . .	64
5.3	Implementations on PMDK . . . . .	65
5.3.1	Reset Model . . . . .	65
5.3.2	Implementation of Manual Model . . . . .	65
5.3.3	Implementation of LEDS . . . . .	69
5.3.4	Discussion . . . . .	71
5.4	Methodology . . . . .	72
5.4.1	Coverage Test . . . . .	72
5.4.2	Workloads . . . . .	74
5.5	Evaluation . . . . .	76
5.5.1	Coverage Test . . . . .	76
5.5.2	Overhead in Manual Model and Automatic Model . . . . .	77
5.5.3	Breakdown of overhead in Automatic Model . . . . .	78
5.5.4	Sensitivity Analysis: Size of data set . . . . .	79

5.5.5	Sensitivity Analysis: Ratio of working data set over total data set . . . . .	80
5.5.6	Write Amplification . . . . .	80
5.6	Summary . . . . .	82
<b>Chapter 6</b>	<b>Related Work . . . . .</b>	<b>83</b>
6.1	Persistent Programming Interfaces . . . . .	83
6.1.1	Address Translation . . . . .	84
6.1.2	Permission Checking . . . . .	84
6.1.3	Persistent Data Retention Model . . . . .	85
6.2	NVMM File Systems and Persistent Stores . . . . .	85
6.3	Durability and Failure Safety . . . . .	86
6.3.1	Dynamic Software Update (DSU) . . . . .	86
6.3.2	Struct Splitting . . . . .	86
<b>Chapter 7</b>	<b>Conclusion . . . . .</b>	<b>87</b>
7.1	Future Directions . . . . .	88
<b>BIBLIOGRAPHY</b>	<b>. . . . .</b>	<b>89</b>

## LIST OF TABLES

Table 2.1	Characteristics of PCM compared with DRAM and flash. [Vol11] . . . . .	8
Table 3.1	Summary of library calls required to support persistent memory programming.	14
Table 3.2	Average instructions executed in the <code>oid_direct</code> function with different number of pools used (on ALL and EACH benchmarks), as described in Section 3.4.1.1. Last column shows the miss rate for the most recent translation predictor on the EACH pattern. . . . .	19
Table 3.3	New instructions to support translation directly in hardware. . . . .	20
Table 3.4	Architecture configuration used for simulation. . . . .	28
Table 3.5	Summary of workloads used in the simulation. . . . .	30
Table 3.6	Pool usage access patterns. . . . .	31
Table 3.7	Combined architecture and benchmark configurations. . . . .	31
Table 3.8	POLB miss rate of OPT benchmark pattern on <i>Pipelined</i> and <i>Parallel</i> design respectively. <i>Pipelined</i> only shows the results of EACH here, because ALL and RANDOM with <i>Pipelined</i> only misses 1 and 32 times (0.000%) during POLB warm-up respectively. . . . .	33
Table 3.9	Percentage of instructions saved with OPT benchmarks on micro-benchmarks and TPC-C . . . . .	34
Table 3.10	POLB miss rates on OPT_NTX using the RANDOM usage pattern on <i>Pipelined</i> and <i>Parallel</i> designs while increasing the size of the POLB. . . . .	36
Table 4.1	Pool and ObjectID Extended API to support permissions checking. . . . .	42
Table 4.2	Summary of the designs used in the evaluations. . . . .	52
Table 4.3	POLB miss rate for different page size with RANDOM/SEPARATE on HwT+SPOT on out-of-order processor . . . . .	55
Table 4.4	Comparison of SPOT walk performed in HwT+SPOT with system that has a direct-mapped POT. for different page sizes on RANDOM/SEPARATE on out-of-order processor. . . . .	56
Table 4.5	The size of tables on all levels of SPOT used in our workloads, for 4KB, 2MB and 1GB page size. . . . .	57
Table 5.1	PDRMs in existing persistent programming models that supports user-defined data types. . . . .	62
Table 5.2	PMDK interfaces that are referred to in this Chapter. . . . .	62
Table 5.3	Summary of the environment for experiments . . . . .	73
Table 5.4	Micro-benchmarks for coverage test . . . . .	74
Table 5.5	Description of the workloads from PMDK . . . . .	75
Table 5.6	Description of different configurations of inputs to the benchmarks in PMDK.	76
Table 5.7	Terms used in the experiments. . . . .	76
Table 5.8	The lines of code added in order to retaining persistent data for the four categories in Table 5.4, with native PMDK APIs, Manual Model and Automatic Model respectively. . . . .	76

Table 5.9	Number of writes for different Persistent Data Retention Models after 100000 insertions in P-ORIGINAL. . . . .	82
-----------	--	----

## LIST OF FIGURES

Figure 2.1	An overview of NVM in current and future computer system. (a) Traditional view of system architecture; (b) Heterogeneous memory where some non-volatile memory extends or replaces DRAM to provide persistent access; (c) NVMM replaces DRAM and disk. . . . .	9
Figure 2.2	PM File Mode interface, depicted by the red dashed line, providing the standard file operations but with memory mapped files going directly to NVM, as described in [Rud13]. . . . .	9
Figure 3.1	Components of an ObjectID. . . . .	15
Figure 3.2	Relationship of pool, object, virtual, and physical address space. . . . .	16
Figure 3.3	Pseudo-code of <code>oid_direct</code> . <code>most_recent_pool_id</code> and <code>most_recent_base_address</code> holds the most recent translated pool identifier and base address. <code>OIDTranslationMap</code> is a hash map that holds all the translations from pool id to base address. . . . .	17
Figure 3.4	Example of a persistent linked list insertion and search using our programming interface. . . . .	18
Figure 3.5	Overall architecture depicting Persistent Object Look-aside Buffer and Persistent Object Table. . . . .	21
Figure 3.6	Two designs for the Persistent Object Look-aside Buffer. . . . .	23
Figure 3.7	Hardware-based POT walk. . . . .	25
Figure 3.8	An example of hardware supported translation on an out-of-order processor with <i>Pipelined</i> design. The code snippet disassembles a simple C expression, <code>a++</code> , but the variable <code>a</code> is held in a persistent pool. The data field of the LSQ is not shown. . . . .	27
Figure 3.9	Speedup of OPT over BASE for each usage pattern on in-order (a) and out-of-order (b) designs. The red dots show the speedup on an ideal architecture where there's no penalty to perform a hardware translation. . . . .	32
Figure 3.10	Performance of all OPT_NTX benchmarks without persistence and transaction supports on both Pipelined and Parallel designs. The results are normalized to the BASE_NTX benchmark running on baseline system. . . . .	35
Figure 3.11	Sensitivity analysis to POLB size. Each bar shows the normalized speedup of OPT over BASE for the EACH pattern on Pipelined and Parallel design on in-order processor, with no POLB and POLB size=1, 4, 32, and 128, respectively. . . . .	36
Figure 3.12	Sensitivity analysis to POT walk penalty. Each bar shows the normalized speedup of OPT over BASE for the EACH pattern on the in-order <i>Pipelined</i> design. . . . .	37
Figure 4.1	Persistent data structure spread across pools. Some pools are mapped and others are not. . . . .	40
Figure 4.2	Example of a persistent linked list traversal with permission check. . . . .	42
Figure 4.3	Pseudo-code of <code>oid_check_direct</code> . The if-else part is added to handle the situation where a translation is missing (i.e. pool not mapped). . . . .	43

Figure 4.4	Overview of the design of automatic permission checks with System Persistent Object Table (SPOT). . . . .	46
Figure 4.5	SPOT entry details on systems with 4KB or 2MB page size. . . . .	48
Figure 4.6	Hardware SPOT walk on 4KB page size system. 2MB and 1GB have similar procedure with different number of SPOT levels. . . . .	49
Figure 4.7	Flowchart of the added hardware logic for permission check used during SPOT walk. . . . .	50
Figure 4.8	Speedup of SwT-Opt, HwT and HwT+SPOT over SwT-Base for each usage pattern on an out-of-order processor. (a) Shows the results of microbenchmarks with ALL and applications with COMBINED. (b) Shows only microbenchmarks with EACH. (c) Shows microbenchmarks with RANDOM and applications with SEPARATE. . . . .	53
Figure 4.9	Performance of different designs on the in-order processor. . . . .	54
Figure 4.10	Performance of HwT+SPOT normalized to SwT-Base on architectures with different page sizes for the RANDOM/SEPARATE pattern. . . . .	55
Figure 4.11	Comparison of proposed system with a system with a direct-mapped POT and adds SPOT, for 4KB page size and RANDOM/SEPARATE workloads. . . . .	56
Figure 5.1	A comparison of having a persistent data structure using (a) traditional file-based programming models and (b) using persistent programming models. . . . .	61
Figure 5.2	An example of retention with Automatic Model: struct A adds a floating-point type field to the struct A in the second version, and copy the integer value. . . . .	65
Figure 5.3	A description of the differences in Manual Model and Automatic Model using LEDS when retaining persistent data in a changed program. The example shows a linked list with three nodes and searching in the updated linked list hits the first node. In Manual Model, a new linked list needs to be created. And in Automatic Model using LEDS, only the node accessed in the new program (first node) is being updated. . . . .	66
Figure 5.4	An example manual retention of a linked-list when adding a floating point field and retaining the value from the integer field, with native PMDK libraries. . . . .	67
Figure 5.4	An example manual retention of a linked-list when adding a floating point field and retaining the value from the integer field, with native PMDK libraries. . . . .	68
Figure 5.5	An example of retention with Automatic Model using LEDS: adding a floating point value and initializing its value from the integer field of last run. . . . .	70
Figure 5.6	The original root object for coverage test. . . . .	73
Figure 5.7	Overall performance of the overhead of Manual Model and Automatic Model on different workloads. (a) LAYOUT-CHANGE: key changes from 32-bit to 64-bit. (b) LAYOUT-ADD: add a new field to each node. The result of PMDK kernels is measured with 100000 operations in both P-ORIGINAL and updated program, and the result of TPC-C is measures with 10 warehouses and 200000 random client operations. . . . .	77
Figure 5.8	A breakdown of the overheads in Automatic Model. The experiment is done with PMDK-DEL workloads of LAYOUT-CHANGE, with 100000 insertions in P-ORIGINAL and 100000 deletions in P-AUTO. . . . .	79

Figure 5.9 Sensitivity analysis of the impact of different data set on both (a) Manual Model and (b) Automatic Model. The experiment is done with PMDK-DEL of LAYOUT-CHANGE, with different number of operations of 100, 1000, 10000, 100000, resulting in different size of data set for retention. . . . . 80

Figure 5.10 Sensitivity analysis of different ratio of working data set in updated program on both (a) Manual Model and (b) Automatic Model. The experiment is done with PMDK-DEL of LAYOUT-CHANGE, where P-ORIGINAL performs 100000 insertions and update program deletes 0.1%, 1%, 10% and 100% of the keys respectively. The y-axis is on logarithmic scale. . . . . 81

## CHAPTER

# 1

# INTRODUCTION

Non-volatile memory technologies are advancing rapidly and may supplant DRAM as main memory given their higher capacities, lower stand-by power, and reasonably fast access latencies [IM15; Raj14; Lee10; Kaw07; Kul13; AS10; Wan15; Mor13; KK09; Zha13]. Products based on Intel’s and Micron’s 3D Xpoint memory technology are already in the market [IM15]. These new *non-volatile main memory* (NVMM) technologies are byte-addressable and have reasonably fast access latencies [IM15; Raj14; Lee10; Kaw07; Kul13; AS10].

As a short term solution, these NVMM technologies can replace either DRAM technologies to provide denser main memory or replace disk technologies to provide faster storage. NVMM has the potential to bring revolutionary changes to the community when considering it provides a new opportunities for programmers to operate on persistent data directly in memory.

Prior works on NVMM file systems[Con09; Dul14; WR11; Vol14; XS16; Xu17] gave programmers direct access to NVMM using regular loads and stores. This enabled programmers to store important data in *persistent* data structures in memory instead of serializing it to the file system. This creates many challenges to existing programming models because they all assume data in memory to be volatile, and when data needs to be persistent, programmers must use file operations to serialize them into files.

There are many prior works on new programming models on NVMM, or *persistent programming*, but many aspects are yet to be explored. In this dissertation, we will explore architectural and language support for multiple aspects of persistent programming. We will answer three questions:

*how programmers can efficiently manipulate persistent data, how they can ensure correctness when accessing data across different persistent regions and what happens to persistent data in memory when the source code changes.*

## 1.1 Address Translation

One area that needs more study is the mechanism that programmers will use to manipulate persistent objects. Ideally, programmers would access persistent objects the same way they access volatile memory, using pointers and references. In fact, many prior works assume precisely this with no modification. For example, Mnemosyne [Vol11] proposes adding a new keyword to the C language, namely `persistent`, that allows some data to be placed in non-volatile memory regions. The assumption is that a segment of the virtual address space is reserved for persistent objects. However, assuming a fixed location within the address space prevents or restricts the use of Address Space Layout Randomization (ASLR) [Bha03], an affordable and prevalent security mechanism in most current systems.

Alternatively, NVHeaps [Cob11] and NVM Library(NVML) [Pme], two recently proposed libraries for persistent memory, allow objects to be relocated within the virtual address space. In these environments, programmers will open persistent regions or *pools*, much the way that files are opened on systems today, and they will be mapped into the virtual address space arbitrarily. Objects within the pools will not be referenced using addresses. Instead, object identifiers (ObjectIDs), consisting of a pool identifier and a byte offset within the pool, are used to identify and locate objects. ObjectIDs are general enough to access objects within the same pools or different pools, and can serve as the basis for building linked structures within and across pools. However, ObjectIDs require *translation to a virtual address* before they can be used to access a persistent object.

ObjectIDs enable persistent memory programming on current systems with ASLR. However, they pose significant challenges too. Translation of ObjectIDs is an onerous burden to place on programmers [Mar17]. Anywhere an object is dereferenced, a translation is needed. Given the already challenging problem of ensuring failure-safety [Pel14; Con09; Jos15a; Pme; Lu14a; Koll16b], this translation requirement adds a significant additional complexity to the code. Furthermore, translation can be a significant performance overhead. At a minimum, we need to perform a translation of ObjectID to base address by looking up the ObjectID in a table. Even if this translation is cached in a fast data structure, like in NVML, the overhead for frequently accessing this structure could be significant because it would require many instructions per translation.

Simultaneous work by Chen *et al.* evaluated various low-level software mechanisms to reduce overhead of manipulating ObjectIDs. However, we observe that the translation of ObjectIDs to virtual addresses is a problem that bears similarity to the problem of translating virtual addresses to physical addresses. We propose adding hardware and software support to enable ObjectID

translation directly in the microarchitecture. In our design, programmers directly reference memory using ObjectIDs. We interpret ObjectIDs as a new address space layer that sits on top of virtual memory, and we provide new load and store instructions that directly access non-volatile memory using ObjectIDs.

Chapter 3 will cover more details in hardware supported address translation. The work shows that on microbenchmarks that place persistent data randomly into persistent pools, *Pipelined* implementation has an average speedup of  $1.96\times$  and  $1.58\times$  on an in-order and out-of-order processor, respectively, over the baseline system. This work was published in The 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50) [Wan17].

## 1.2 Permission Checks

Even though hardware-supported translation of ObjectIDs make it easier to program and use persistent objects, they are not as simple to use as normal pointers. Consider the following scenario: after opening one pool and finding an ObjectID within it, we may want to dereference it. However, unless we have additional information, it is unknown if the corresponding object is in an already-opened pool or not. The programmer must reason about whether the persistent pool has already been opened and mapped into the process' address space. Similarly, permissions (i.e. like file permissions in a conventional file system) also must be checked to ensure the pool can be mapped to the current process. Without additional checks to ensure the pool is opened, the programmer may incur memory protection errors.

Interestingly, this is quite different from programming with conventional pointers or references. Typically, if a pointer address is known and non-NULL, it can be assumed legal and dereferenced, assuming a correctly implemented program. However, ObjectIDs do not come with this guarantee because they persist across runs of the program, creating an additional step in reasoning about their correctness. In current designs, the burden to perform these correctness checks rests fully on the programmer to implement them in software.

Instead, we propose hardware support for validating ObjectIDs and checking their permissions on demand. We borrow from well known architecture techniques used to support Virtual Memory, and we extend them to support common file system operations to make using persistent objects simpler and higher performing. In particular, we extend the work of hardware address translation in Chapter 3. We replace the per-process Persistent Object Table (POT) with a System Persistent Object Table (SPOT) that holds all known pools in the system, their physical address, and their permissions information. When a program suffers a POLB miss, a privileged hardware SPOT walk traverses the SPOT to find the relevant entry, check permissions, and copy it back into the POLB. If the persistent ObjectID is known and the permissions check out, the requesting process obtains the mapping and continues execution without trapping to the operating system.

With this design, we can improve program’s performance significantly. Part of the performance improvement comes from our new hardware. Permission checking is performed by hardware along with fast hardware-supported address translation. Also, the hardware SPOT walk allows us to avoid system calls for mapping pools to the program. Another part of the performance comes from simplifying the software. Programmers can omit frequent and costly library calls that check the permission of pools they wish to reference. This streamlines the code and reduces overhead, pushing some of the complexity into low latency hardware and system code.

Chapter 4 will cover more details in the hardware permission checks. This work We find that our design offers a compelling  $2.9\times$  speedup on average for microbenchmarks that access pools with the RANDOM pattern and  $1.4\times$  and  $1.8\times$  speedup on TPC-C and vacation, respectively. This work was published in The 45th International Symposium on Computer Architecture (ISCA 2018) [Wan18].

### 1.3 Persistent Data Retention Models

Programming persistent data structures poses many new challenges. Many prior works have pointed out the problem of *failure-safety* [Pel14; Con09; Jos15a; Pme]. Other works [Cob11; Vol11; Che17; Wan17] strive to make it easier to put data into persistent memory and use it. Chapter3 and Chapter 4 will discuss how to make it easier to write persistent data structures. We identify another new dimension introduced by persistent programming: *the problem of managing changes to the declarations and meanings of persistent data across the lifespan of software*.

Since persistent data is retained across runs of a program, does a declaration of a type refer to the persistent data created in a previous run of the program or to the organization of the data on the next run of the program? In systems with a conventional memory and storage hierarchy, programmers trust that data types describe what will happen when the program runs next, because there is no data already in memory – memory is repopulated from scratch each run. However, in the context of persistent memory the declarations describe both – what’s already in persistent memory and what will happen on the next run.

To begin addressing this problem, we propose that programming environments define a *Persistent Data Retention Model* (PDRM) that explains how persistent data is *retained* as programs are modified and change over time. Programmers need, at a minimum, a clearly specified model of behavior. The simplest one is that discard all persistent data anytime a program changes and reset persistent memory. We call it the *Reset Model*. Alternatively, we can make the programmer fully responsible for distinguishing such data and managing it across changes to the code. This is the conventional way for how files are handled, and it requires substantial programmer effort, so we call it the *Manual Model*. Last, we define the *Automatic Model*. The Automatic Model assists programmers in the conversion of data and ensures the data continues to be used appropriately after code changes. This requires a mechanism that can distinguish which data can be retained,

along with possible limitations or restrictions, and which cannot after a program is modified.

Chapter 5 will discuss the three Persistent Data Retention Models with more details.

## 1.4 Contributions

This dissertation makes contributions in each of the three parts.

For the hardware translation part, we describe an efficient way to manipulate persistent data. (1) We describe novel architectural and system-level extensions that enable translation of ObjectIDs to addresses, eliminating the need for programmers to directly translate ObjectIDs and the performance overhead of translation. (2) We design and evaluate critical microarchitectural choices as it relates to translation and disambiguation of addresses for persistent objects. We consider two different designs for Persistent Object Look-aside Buffer (POLB), one that translates to the virtual address and relies on a conventional Translation Look-aside Buffer (TLB) for conversion to the physical address, and another that works in parallel with the TLB and directly translates to the physical address. We also consider the design implications of these POLBs on an out-of-order superscalar processor's memory disambiguation logic. (3) We implement this system in a simulator based on Sniper [Car14]. (4) We evaluate the system based on a suite of microbenchmarks and applications.

For the permission check part, we propose the problem of permission checks in persistent programming with multiple persistent regions, or pools. 1) We raise the problem to reconsider providing permission checks of persistent pointers instead of leaving the burden to programmers. 2) We explore different design choices to provide permission checks on demand with native software translations and hardware supports. 3) We propose a design with hardware and system support with small modifications to prior work and provide automatic permission checks and then evaluate the design.

For the PDRM part, we are the first to explore the problem of defining what happens to the persistent data when a persistent program changes. We propose three PDRMs and give definitions to the three models. We also give implementations of the three model, in Persistent Memory Development Kit (PMDK, formerly known as NVML), especially a technique of Automatic Model, called Lazily Extendable Data Structures (LEDS). Our experiments on various benchmarks show that Automatic Model can save a lot of programmers' effort with the expense of comparable overhead with Manual Model.

## 1.5 Organization of the Dissertation

The rest of the dissertation is organized as follows. Chapter 2 covers background on NVMM technologies. Chapter 3 discuss the work of hardware support ObjectID translation. Section 3.1 presents the baseline programming model and overheads with software translations. Section 3.2 describe

the overall design to use Persistent Object Lookaside Buffer (POLB) and Persistent Object Table (POT) to support direct load and store with new instructions. Section 3.3 discusses microarchitectural implementation details, and Section 3.5 presents an evaluation on the benchmark suite. Chapter 4 discuss the work of permission check. Section 4.1 brings up motivations of permission check and discuss different designs to provide permission security on persistent pools. Section 4.2 brings a hardware optimized design, including the System Persistent Object Table (SPOT), that can reduce the overhead and also reduce the programming burden. Section 4.3 presents details of architectural and system implementations. Section 4.4 evaluates the implementation on the same benchmark suite from Chapter 3 with permission checks added. Chapter 5 discuss the work of Persistent Data Migration Models (PDMM). Section 5.1 gives a comparison of the traditional file-based persistent programming with persistent memory programming models for NVMM and also explores the PDRMs in existing persistent programming extensions. Section 5.2 introduces the definition of Reset Model, Manual Model and Automatic Model in general persistent programming languages and points out key differences between Manual Model and Automatic Model. Section 5.3 demonstrates implementations of each model built in PMDK libraries, and Section 5.5 demonstrates the overhead of Manual Model and Automatic Model and analyzes the components of the overhead and also characterizes the two models. Chapter 6 discusses related works and Chapter 7 concludes the dissertation and also proposes some of the possible future works.

## CHAPTER

# 2

# BACKGROUND

## 2.1 Non-Volatile Main Memory (NVMM) Technologies

In a conventional computer system, disk is much slower than memory, but it is necessary because it provides long-term persistent storage. Data that needs to be saved must be written to disk. Disks are connected to the computer system through slow I/O buses, so programmers take pains to hide the longer latencies required to read and write from disk.

Recent breakthroughs in Non-Volatile Memory (NVM) technologies are poised to disrupt the conventional computer system design<sup>1</sup>. NVM provides near-DRAM speed and byte-addressable persistent storage. These technologies allow persistent storage to be attached to the memory bus and accessed through load and store instructions [FW08]. Since it can be deployed as main memory, we refer to this usage as *Non-Volatile Main Memory (NVMM)*.

There are many different NVMM technologies, like Phase-Change Memory (PCM) [Lee10], Spin-Torque-Transfer RAM (STT-RAM) [Kul13] and memristors [Str08]. They all have slightly different attributes. we will choose PCM as the main NVMM technology to model in this dissertation because Intel and Micron has already shipped 3D Xpoint [IM15] based on PCM technology.

In this dissertation, our evaluation is based on battery backed DRAM, i.e. same read/write latency as DRAM but with persistence. The comparison of DRAM, NAND flash and PCM is summarized in

---

<sup>1</sup>These NVM technologies are also known as Storage-Class Memory (SCM) in other literature [Vol11; FW08; Lam10; WR11].

Table 2.1. While NVM technologies are under development, we make several assumptions on the NVMM behaviors based on many prior works [Vol11]. Although current PCM technologies are slower than DRAM, especially when writing, some research prototypes [Bed04; Vil10] predict it can achieve similar read and write speed as modern-day DRAM. (1) We assume NVMM can support 64-bit atomic read/write. [Con09]. (2) We assume data is considered to be persistent until it's flushed to the NVM by some hardware instructions (e.g. `clwb` by Intel), similar to the `fsync` call for file systems.

**Table 2.1** Characteristics of PCM compared with DRAM and flash. [Vol11]

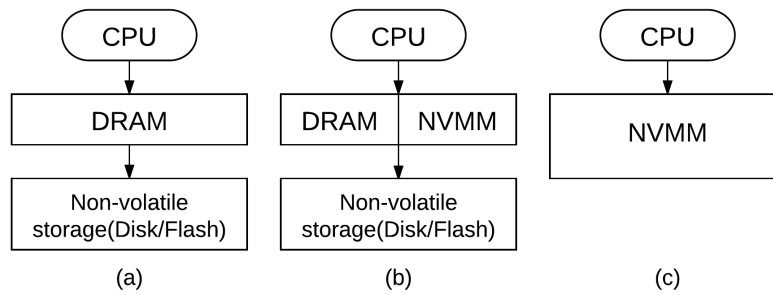
Technology	Read	Write	Endurance
DRAM	60 ns	60 ns	$>10^{16}$
NAND flash	25 $\mu$ s	200-500 $\mu$ s	$10^4$ - $10^5$
PCM (current)	115 ns	120 $\mu$ s	$10^6$
PCM (predicted)	50-85 ns	150-1000 ns	$10^8$ - $10^{12}$

## 2.2 Persistent Programming

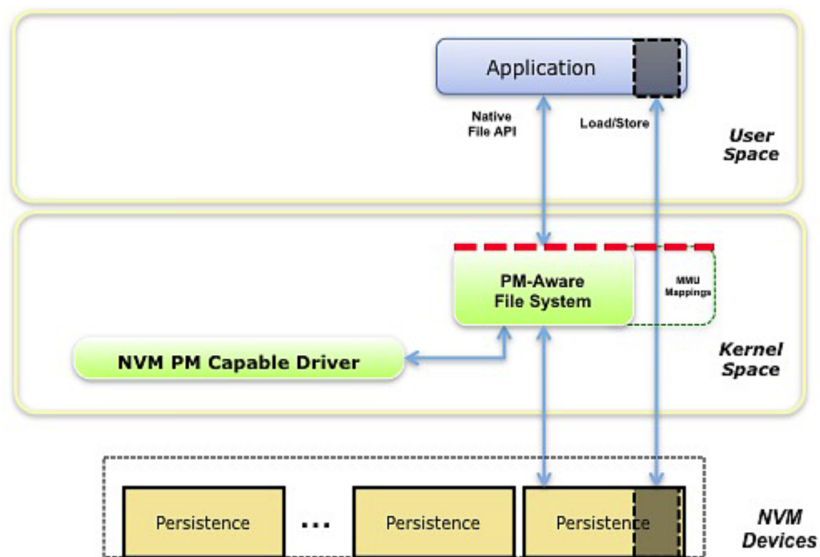
Emerging NVMM technologies will encourage significant changes to the computer system. The boundary of memory and disk will eventually blur. In this dissertation, we envision the future computer system will lean towards the one depicted in Figure 2.1 (b) and eventually Figure 2.1 (c). In this dissertation, we discuss programming problems existing in a system like Figure 2.1 (b) or (c). These systems are going to significantly change our current programming models because programmers can store persistent data in memory, instead of being forced to serialize them to disk using file operations. We refer to this as *programming with persistent memory*, or sometimes simply *persistent programming*, in the following sections.

There are many ways to provide persistent programming interfaces to programmers. Rudoff [Rud13] summarized four ways to provide NVM programming models, where the PM Volume Mode and PM File Mode are directly dealing with emerging NVMM technologies. The major differences of the two models are whether user programs have direct access to NVMM. The PM Volume Mode provides direct access to NVMM only from kernel code, so user programs need to use system calls to access persistent data. Meanwhile, in PM File Mode, the operating system maps a region of NVMM directly to a program's address space and allows user programs to use normal load and store instructions to access persistent data. The following sections in this dissertation focus on this model and its interface is depicted in Figure 2.2.

Many prior works [Vol11; Cob11; Pme; Ora15; Zho16] belong to this memory-mapped file model. The rest of the dissertation will focus on two major persistent programming models, Mnemosyne [Vol11] and Persistent Memory Development Kit (PMDK, previously known as NVM Library) [Pme].



**Figure 2.1** An overview of NVM in current and future computer system. (a) Traditional view of system architecture; (b) Heterogeneous memory where some non-volatile memory extends or replaces DRAM to provide persistent access; (c) NVMM replaces DRAM and disk.



**Figure 2.2** PM File Mode interface, depicted by the red dashed line, providing the standard file operations but with memory mapped files going directly to NVM, as described in [Rud13].

Mnemosyne [Vol11] proposes adding a new keyword to the C language, namely `persistent`, that allows static or global data to be placed in non-volatile memory regions. It also provides dynamic allocation interfaces similar to `malloc()` and `free()`, namely `pmalloc` and `pfree`. The assumption is that a segment of the virtual address space is reserved for persistent objects. This makes minimal changes to the current programming model. Persistent Memory Development Kit, on the other hand, points out assuming a fixed location within the address space prevents or restricts the use of Address Space Layout Randomization (ASLR) [Bha03], an affordable and prevalent security mechanism in most current systems. Thus they introduce the concept of relocatable persistent region, or *pool*, that is similar to the concept of a file, and programmers need to specify which pool is used to allocated persistent data. More details about these works will be covered in Section ??.

### 2.2.1 Persistent Memory Development Kit (PMDK)

In this dissertation, we will mainly focus on Persistent Memory Development Kit (PMDK), it provides a collection of libraries for various use cases, which is tuned and validated to production quality and thoroughly documented [Pme]. It targets server-class applications that require skilled programmers to write programs with low-level interfaces. There are 10 libraries in the PMDK and we will use the interfaces exclusively from `libpmemobj` as described in the man page [Lib].

The main idea of PMDK is to store persistent regions in file abstractions, or *pools*. Pools are directly mapped to a program's address space and programmers can refer to persistent data with persistent pointers called ObjectIDs, or `PMEMoid`, which are the equivalent of addresses to persistent memory.

In PMDK, all persistent data is held within pools. There are no persistent global variables, as in Mnemosyne [Vol11], that are outside of a pool. Each pool has a root object, which is defined as a struct, to store variables associated with the pool, some of which may be pointers to data structures in the pool. To modify persistent data, a coder would modify one or more struct definitions contained within the persistent pool.

In Chapter 3 and Chapter 4, we will model a reduced collection of the interfaces of PMDK, we will cover more details on the interfaces we modeled in Table 3.1 and Table 4.1. In Chapter 5, we will directly use the `libpmemobj` in PMDK and all the functions we will cover are summarized in Table 5.2.

## 2.3 Our vision and objectives: making persistent programming easier

The adoption of NVMM in future computer systems will not happen all at once and is widely believed to be a multi-staged road [Swa17]. According to Prof. Swanson [Swa17], in Stage 1 (Legacy Support and Giant Memory), DRAM-starved applications can leverage the denser NVMM to increase

memory capacity, or IO-starved applications can use the persistency of NVMM while ignoring its byte-addressability. Either case is an extension in Figure 2.1 (a). Neither of the usage will be satisfying since they never fully take advantage of NVMM. Thus, in Stage 2 (DAX and Expert Programmers), we can use the NVMM as a part of the memory to create persistent data structures. As a step towards the goal, we can provide direct access (DAX) to programmers so that they can map a file to memory and operate on it directly. Many libraries mentioned in Section 2.2 ([Vol11; Cob11; Ora15; Pme]) provide interfaces for programmers to program with persistent memory, but it requires skilled programmers to write a robust persistent program. Our goal in this dissertation is to provide some solutions to eventually help develop a computer system towards Stage 3 (Democratizing Persistence), where the notion of persistence is integrated into the programming language, so the compiler and runtime can work together to make NVMM easy-to-use for typical programmers.

## CHAPTER

# 3

# HARDWARE SUPPORTED PERSISTENT OBJECT ADDRESS TRANSLATION

In this Chapter, we will first answer the question: *how programmers can efficiently manipulate persistent data?* With prior works like PMDK [Pme], programmers can use ObjectIDs to create permanent pointers using persistent pools that survive multiple program runs. However, the usage of ObjectID remains a burden to programmers and also incur significant overhead. We will first visit this programming scheme in Section 3.1. Then we propose treating ObjectIDs as a new persistent memory address space and provide hardware support for efficiently translating ObjectIDs to virtual addresses. With our design, a program can use load and store instructions to directly access persistent data using ObjectIDs, and these new instructions can reduce the programming complexity of this system. The details of the design will be discussed in Section 3.2. It's also interesting to consider whether the ObjectID can be translated to virtual address or physical address since we are moving the translation into hardware side. We will discuss it and other microarchitectural considerations in Section 3.3. Section 3.4 will discuss the setup of our simulation environment, which will also be used in the next Chapter.

In Section 3.5, we evaluate our design on Sniper modeling both in-order and out-of-order processors with 6 micro-benchmarks and the TPC-C application. The results show our design can give significant speedup over the baseline system using software translation. We demonstrate for the *Pipelined* implementation that our design has an average speedup of  $1.96\times$  and  $1.58\times$  on an

in-order and out-of-order processor, respectively, over the baseline system on microbenchmarks that place persistent data randomly into persistent pools. For the same in-order and out-of-order microarchitectures, we measure a speedup of  $1.17\times$  and  $1.12\times$ , and  $1.19\times$  and  $1.18\times$  respectively, on the TPC-C application and Vacation application.

This Chapter is written based on the published work

### 3.1 Motivation: Baseline Persistent Programming Model and its limitations

In this section, we demonstrate the programming overheads and challenges of *persistent programming* using the programming interface shown in Table 3.1. We first describe the interface and how to use it. Then, we present a code example and point out key overheads. We also discuss other overheads that are commonly presented in the context of logging for failure-safety.

#### 3.1.1 The Persistent Programming Interface

Many designs for persistent programming interface are possible [Vol11; Cob11; Pme; Zho16] and the community has not settled on any design yet. In this chapter, we base our interface on the Persistent Memory Development Kit (PMDK, formerly known as NVM Library) [Pme], an open-source library developed by the PMEM team at Intel that was inspired by prior works [Cob11; Vol11]. We select it because it has garnered significant support from industry. However, because of its size and complexity, we have reduced it to an essential set of functions in order to give it C language semantics and to narrow the focus to specific aspects of the interface that are important to our work in this Chapter. We expect that the findings of our work extend to PMDK and other systems with a design based on ObjectIDs.

Table 3.1 contains a number of functions that support a few keys tasks: pool management, object management, translation, support for durability, and support for failure-safety.

##### 3.1.1.1 Pool Management

As in most prior works [Pme; Cob11; Vol11], we assume that persistent objects are grouped into file-like entities, called *pools*, that are mapped in their entirety into a process' address space. This is similar in concept to *mmap* on Linux. Pools are then created from scratch (`pool_create`), opened (`pool_open`), or closed (`pool_close`) just like files. Like files, opening or creating a pool would require an OS-level system call that verifies permissions and maps the pool into the address space. We do not further consider these mechanisms since they are similar, in principle, to ones that already exist for managing files.

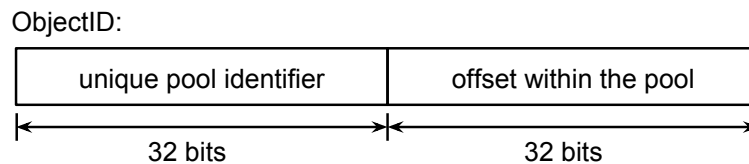
**Table 3.1** Summary of library calls required to support persistent memory programming.

Function Calls	Descriptions
<b>Pool Management</b>	
pool* pool_create(name, size, mode)	Create a pool with the specified <i>size</i> and associate it with a <i>name</i> .
pool* pool_open(name)	Reopen a pool that was previously created. Permissions will be checked.
pool_close()	Close a pool.
OID pool_root(pool* p, size)	Return the root object of the pool <i>p</i> with specific <i>size</i> . The root object is intended for programmers to design as a directory of the contents in the pool.
<b>Object Management</b>	
OID pmalloc (pool* p, size)	Allocate a chunk of persistent data with the given <i>size</i> on pool <i>p</i> and return the ObjectID of the first byte.
pfree(oid)	Free persistent data pointed to by the ObjectID.
<b>Translation</b>	
void* oid_direct(oid)	Translate an ObjectID to a virtual address. Used when there's no hardware translation.
<b>Durability</b>	
persist(oid, size)	Make the region durable that begins at the ObjectID and ends <i>size</i> bytes later.
<b>Failure Safety</b>	
tx_begin(pool* p)	Start a new transaction related to the pool <i>p</i> .
tx_add_range(oid, size)	Take a "snapshot" of the current persistent data region referenced by the ObjectID and save it into the undo log.
OID tx_pmalloc(size)	Atomically allocate persistent data. Perform the functionality of pmalloc() but record it in the undo log in case the transaction fails.
tx_pfree(oid)	Free persistent data referenced by ObjectID within a transaction.
tx_end()	End the current transaction. If the transaction successfully ends, all the persistent data logged by tx_add_range() and tx_pmalloc() will be made durable.

However, this is where the analogy to files ends, because pools are not byte serialized and they contain objects. We assume, like prior work, that all pools must contain a root object, which serves as an entry point for the pool and aggregates key information for the pool. The last function shown under Pool Management is `pool_root`, and it provides access to the root object of a given pool.

### 3.1.1.2 Object Management

All objects within a pool are persistent, and we can allocate and deallocate them within the pool the same way that the heap works. We provide `pmalloc` and `pfree` functions. Unlike `malloc`, `pmalloc` needs to know the pool the object is being created within, and then it returns an `ObjectID` to the newly allocated object.



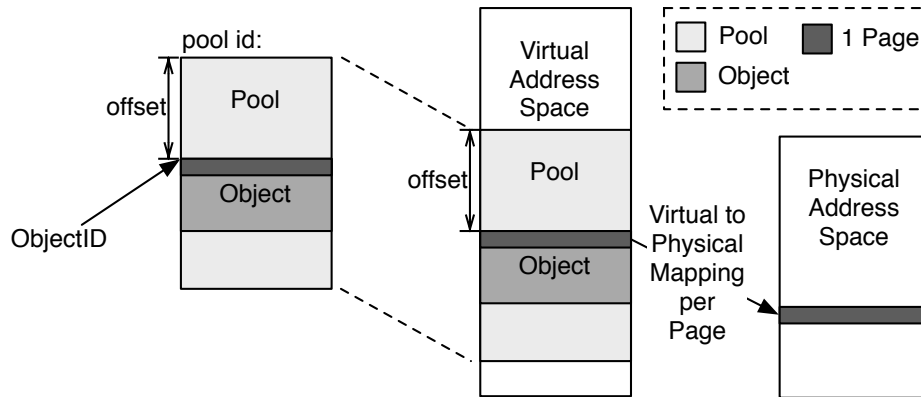
**Figure 3.1** Components of an ObjectID.

As depicted by Figure 3.1, the ObjectID is the concatenation of a unique identifier given to the pool and an offset within the pool where the object is located. We assume that the ObjectID is 64 bits to make it possible to hold it in one register. We devote the upper 32 bits to the pool identifier, a unique system-wide number assigned to a pool when it is created, and the lower 32 bits for the offset. This can be interpreted as a segmented address space, where each segment is 4GB. However, it can also be interpreted as a flat address space since an object in one pool can reference any other pool using a legitimate ObjectID.

### 3.1.1.3 Translation

As shown in Figure 3.2, each pool is mapped into the virtual address space, and each page of the pool is individually mapped to a physical address using conventional approaches for virtual memory. It's worth noting that the same pool might map to different virtual addresses in different processes. Furthermore, the mapping to physical address is handled by the virtual memory manager for the system in the conventional way.

An ObjectID can be converted to a virtual address as long as we know the location of the pool. In our API, such a translation is provided by the `oid_direct` function. Once a pool is mapped



**Figure 3.2** Relationship of pool, object, virtual, and physical address space.

into an address space, it has a base address, and the relation of pool id to base address is tracked in an efficient data structure, like a hash table. For any given ObjectID, the translated address is obtained by searching the data structure for the unique pool id, returning the address where the pool is mapped, and then adding the offset.

PMDK uses a combination of a hash table and a last value predictor to translate ObjectIDs. The most recent pool id look-up is held in a variable. If the next translation request is for the same pool, the base address is quickly determined without searching the hash table. Otherwise, if there is no match, the hash table is searched. Such an approach is efficient when translations tend to occur for the same pool, but it loses efficiency when ObjectIDs are spread across many pools.

We adopt this same strategy in our `oid_direct` implementation, illustrated as pseudo-code in Figure 3.3. We use a global variable pair, `most_recent_pool_id` and `most_recent_base_addr`, to hold the most recent translations. If the `pool_id` part of the ObjectID matches, the translation can be calculated with the most recently translated pool id. Otherwise, a search is performed on the hash table (`OIDTranslationMap` in the pseudo-code) to find the translation, and it also updates the most recently searched pair for use by subsequent translations.

If a translation is requested for a pool that has not been opened, it is considered an error, as it's the case in PMDK. Ideally, the API should be extended with error codes to allow recovery or handling by the programmer, but we omit this support and further discussion for the sake of keeping the API small. In Chapter 4, we will revisit it, where it might not be an error if the programmer "forgets" to open the the pool but has permission to access it.

```

1 void* oid_direct(OID oid)
2 {
3     if (most_recent_info_valid && oid.pool_id == most_recent_pool_id)
4         return most_recent_base_addr + oid.offset;
5     most_recent_info_valid = 1;
6     most_recent_pool_id = oid.pool_id;
7     most_recent_base_addr = OIDTranslationMap->find(oid.pool_id);
8     return most_recent_base_addr + oid.offset;
9 }

```

**Figure 3.3** Pseudo-code of `oid_direct`. `most_recent_pool_id` and `most_recent_base_address` holds the most recent translated pool identifier and base address. `OIDTranslationMap` is a hash map that holds all the translations from pool id to base address.

### 3.1.1.4 Durability and Failure-Safety

Durability and failure-safety are two major aspects of persistent programming, and many prior works have dealt with these topics [Pel14; Con09; Jos15a; Pme; Lu14a; Kol16b].

Durability refers to the process of ensuring that modified data has been written back to non-volatile storage. Intel’s NVML extensions describe new instructions, like `CLWB` and `CLFLUSHOPT` [Pme; Int16], that force data to write back to memory. These instructions, in conjunction with `SFENCE`, allow the programmer to guarantee that data is written back to persistent memory. The `persist` function guarantees that all data starting at a given `ObjectID` and extending size additional bytes are written back to persistent storage.

Failure-safety deals with the larger problem of how to update a persistent data structure so that it is always in a consistent state or recoverable to one. Prior works have described how to use write-ahead undo or redo logging to provide transactional semantics when persisting data. We add support for write-ahead undo logging.

Transactions begin and end using the `tx_begin` and `tx_end` functions. Any persist object can be logged using the `tx_add_range` function which copies the specified range of addresses (`ObjectID` to `ObjectID+size`) to the log and makes them persistent. `tx_pmalloc` and `tx_free` are special versions of `pmalloc` and `pfree` that can be undone.

### 3.1.2 Persistent Linked List Example

Figure 3.4 shows an example of inserting a node and searching for a node in a linked list. First, consider the struct definition. Each node of the list contains a value and an `ObjectID` to the next node in the list. This design allows the list to be fully contained in a single pool or span multiple pools.

The `insert` routine takes a pool, head, and value as arguments. Next, it creates a new node in the specified pool. `pmalloc` returns an `ObjectID` so we have to convert it to an address before we

```

1 typedef struct {
2     int value;
3     OID next;
4 } node;
5 // insert a node that contains value
6 OID insert (pool* p, OID head, int value){
7     OID new_oid;
8     node* temp;
9     assert(new_oid = pmalloc(p, sizeof(node)));
10    temp = (node*)oid_direct(new_oid);
11    temp->value = value;
12    // link to OID of old head
13    temp->next = head;
14    head = new_oid;
15    // return new OID
16    return head;
17 }
18 // find the first node that matches data
19 OID find (OID head, int data){
20     OID tmp = head;
21     while (tmp != OID_NULL) {
22         node *x = (node*)oid_direct(tmp);
23         if (x->value == data)
24             return tmp;
25         tmp = x->next;
26     }
27     return OID_NULL;
28 }

```

**Figure 3.4** Example of a persistent linked list insertion and search using our programming interface.

can initialize the new node. Next, we set the next pointer to the head and return the new node as the head of the list. We have to be careful to set the next field using an ObjectID and to return the ObjectID rather than the translated address.

The find routine traverses a list looking for a node with a matching value. During each iteration a translation is required in order to access the elements of the node. Note that since the list could span multiple pools, we are assuming that all such pools have already been mapped using `pool_open` or `pool_create`.

### 3.1.2.1 Overheads

The persistent linked list incurs more overhead compared with a normal linked list. Translations from ObjectID to address are needed as compared to a conventional design. In the event that a list is fully contained within the same pool, some translation overhead can be eliminated within the `oid_direct` function because it remembers the last one and can avoid a costlier search operation. Nonetheless, up to 100s of instructions could be needed to perform a translation depending on

implementation and number of pools used by a program at any given time. Table 3.2 shows that the `oid_direct` function requires approximately 17 instructions when the last translation is reused (second column), but it increases to approximately 97 instructions, on average, when most translations require the cost of a full look-up (third column). Also, as translation must precede execution, these overheads fall on the critical path and have a major impact on performance.

**Table 3.2** Average instructions executed in the `oid_direct` function with different number of pools used (on ALL and EACH benchmarks), as described in Section 3.4.1.1. Last column shows the miss rate for the most recent translation predictor on the EACH pattern.

Bench.	Insns with one pool	Insn with multiple pools	Miss on recent
Linked-list	17.0	99.3	99.7%
B-Tree	16.8	77.8	62.2%
Red-black Tree	17.0	104.7	91.0%
B+ Tree	17.0	95.0	80.3%
Binary Search Tree	16.8	107.3	96.7%
String Position Swap	17.0	102.7	99.9%
Geometric Mean	17.0	97.3	87.2%

These translation overheads also bring many indirect inefficiencies. For example, they increase the working set of the program, adding to increased pressure on the cache, and the compiler may be less efficient since it cannot interpret ObjectIDs as addresses, limiting the success of many common optimizations.

### 3.1.2.2 Programmability Concerns

PMDK [Pme] targets server class applications that primarily require system programming experts to explicitly manage persistent data. Hence, it makes little effort to simplify persistent programming. However, we argue that NVMM has the potential to be powerful on a wide range of systems, from client applications to embedded domains. Prior work has pointed out many of the other challenges of persistent programming [Cob11; Pme; Vol11; Mar17], like providing failure-safety and ensuring persistent objects are not leaked. Hence, given the already complex nature of programming for persistence, it is important to ease the burden on the programmer when possible.

Unfortunately, in the current API, implementing a simple data structure is somewhat more complex [Mar17]. In the case of the linked list, rather than keeping track of the address for a node, it is imperative to track both the ObjectID and the translated address. Depending on the operation, one or the other may be required. Programming with pointers is already challenging and error prone, and ObjectIDs add another kind of reference that must be properly used by the programmer. Hence, rather than reasoning about one pointer, the programmer must reason about two, the normal

address and the ObjectID.

Our hardware support can reduce this complexity by allowing a single reference, the ObjectID, to serve both purposes. To fully exploit the hardware, languages and compilers may need to be modified in order to expose this simplicity to the programmer. We do not consider these additional efforts in this paper, but we believe that such extensions are feasible.

## 3.2 Design: Hardware-supported Address Translation

We identify that translation from ObjectIDs to virtual address places a burden on the programmer and incurs significant performance overheads. We propose new instructions and architectural support to accelerate ObjectID translation.

### 3.2.1 New Instructions

In our design, we interpret the space of all ObjectIDs as a new address space and provide hardware support for loading and storing directly to this address space. Rather than requiring translation in software from ObjectID to virtual address, two new instructions will support translation in hardware, as shown in Table 3.3. `nvld` allows a load directly from an ObjectID. New hardware will look-up the ObjectID, convert it to a virtual address, and perform the load all in one instruction. Likewise, `nvst` will write to a location specified by an ObjectID.

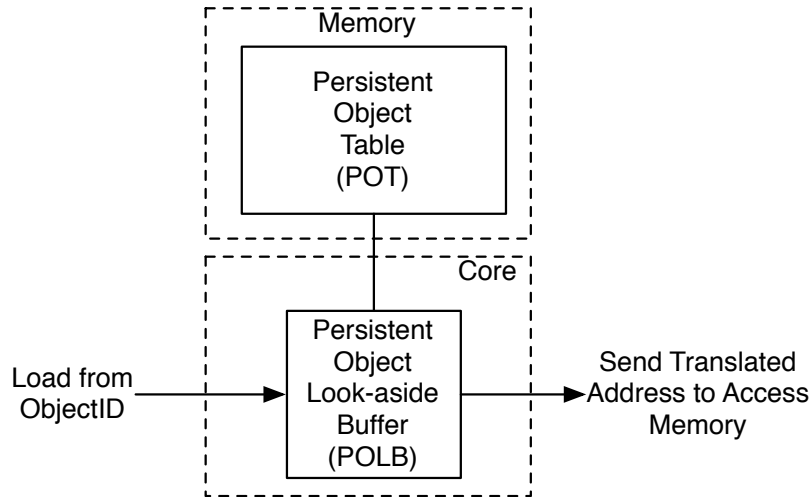
While we do not show them explicitly, we assume that each instruction comes with variants for accessing data of varying widths (e.g. bytes, words, double words, and so on). Ultimately, the language and compiler would need to facilitate selection of the appropriate width operation, but we do not consider this issue more.

**Table 3.3** New instructions to support translation directly in hardware.

Instruction	Description
<code>nvld rd,rs1,imm</code>	$rd = \text{MEM}[\text{Lookup}(rs1) + \text{imm}]$
<code>nvst rs1,rs2,imm</code>	$\text{MEM}[\text{Lookup}(rs2) + \text{imm}] = rs1$

### 3.2.2 Overall Architecture

Figure 3.5 provides a depiction of our architecture. To support translation from ObjectID to virtual address when an `nvld` or `nvst` executes, we need two structures, the Persistent Object Look-aside Buffer (POLB) and Persistent Object Table (POT).



**Figure 3.5** Overall architecture depicting Persistent Object Look-aside Buffer and Persistent Object Table.

The POLB is similar to the TLB. It contains entries for the pools that have been mapped into a process’s address space. The entry allows for translation from an ObjectID to an actual address, possibly virtual or physical (Section 3.3 will cover more details). The POLB is located inside the core and is meant to be accessed with low latency so as to incur very little overhead on `nvld` and `nvst` operations. If an entry is present, hardware is allowed to perform the translation.

The POT serves a similar role as a page table. The POT tracks the current pool mappings for a process. In our design, the POT converts a pool id to a virtual address.

The POLB and POT work together to provide high performance translation. If a translation is requested for an entry that is not present, the POT is checked to determine if the pool is mapped into the address space already. If a matching entry is present in the POT, it is simply moved to the POLB by hardware. Note that the POT serves as a backing store for all such information, whereas the POLB acts as a cache, holding only the most recent translations used by the core. However, if a requested translation is unknown to both the POLB and POT, it is treated like a page fault and a trap to the operating system is required to handle it, which may abort the program or invoke some form of signal handling to allow the application to recover.

### 3.2.2.1 Microarchitectural Considerations

While the design of the POLB and POT bear similarity to the TLB and Page Table, they are different. There are a variety of microarchitectural design choices. For example, should the POLB translate ObjectIDs to virtual addresses or physical addresses. If the POLB produces virtual addresses, that implies an additional *pipelined* access to the TLB. On the other hand, if the POLB produces physical

addresses, a cache access can proceed in *parallel* with the POLB look-up. However, this leads to other complexities.

Another aspect is the problem of aliasing among normal loads and stores and `nvld` and `nvst`. It is possible for code to simultaneously access a persistent object through both kinds of instructions. This implies that the architecture must disambiguate virtual addresses and references based on ObjectIDs at the same time. For high performance, memory disambiguation and load-store forwarding should operate correctly in the presence of aliased operations. For example, a store using an ObjectID should be able to forward data directly to a regular load.

Furthermore, these design choices are inter-dependent. We consider two overall design strategies for the POLB and POT, one that pipelines POLB accesses with the TLB and another that operates in parallel with the cache. Because these choices interact with memory disambiguation, we also consider these designs in the context of an out-of-order and in-order processor pipeline.

Their complete implementations are described in Section 3.3.

### **3.2.3 Supporting System and Library Modifications**

We modify our programming interface from Section 3.1 with support for our new instructions and hardware.

In `pool_create()` or `pool_open()`, after the pool is created or verified, the persistent data region will be mapped into the process' address space. Also, the POT must be updated with a new entry for the pool.

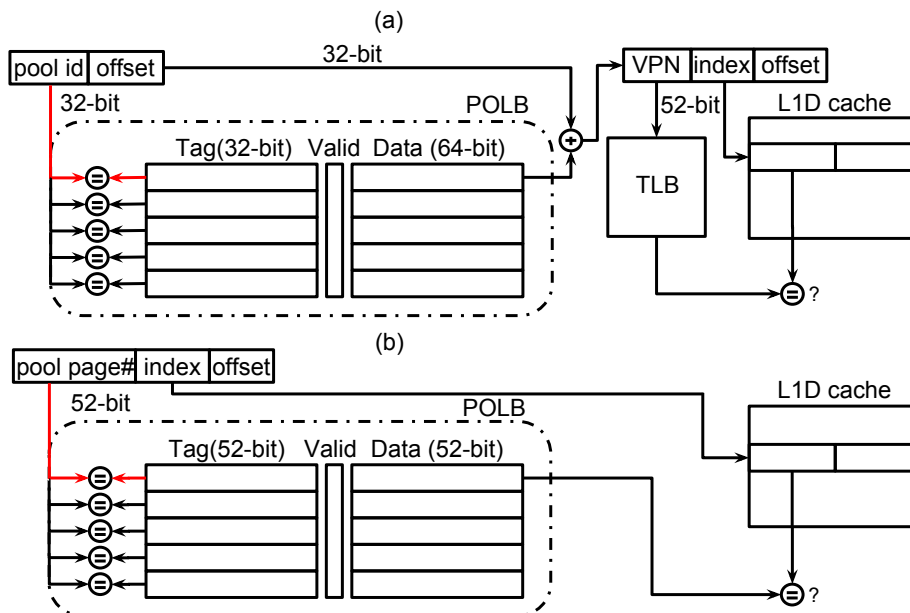
Now, `oid_direct()` is no longer required. A programmer can dereference an ObjectID to read or write persistent data directly. Several of the library functions can also benefit from hardware acceleration. For example, `tx_add_range()` can use these more efficient instructions to when copying data into undo logs. We modify all of the library calls that access persistent data to use faster `nvld` and `nvst` instructions.

## **3.3 Microarchitecture**

We present the microarchitectural details of our new structures and how they integrate into the pipeline.

### **3.3.1 Persistent Object Look-aside Buffer**

The POLB is implemented as a small cache-like structure. Each entry in the POLB contains a tag, a valid bit, and a data value. The tag array is implemented as a Content Addressable Memory (CAM) for efficient searching.



**Figure 3.6** Two designs for the Persistent Object Look-aside Buffer.

### 3.3.1.1 Pipelined

In the case of the *Pipelined* design, the POLB stores the translation from the pool identifier to the base address. In this case, the tag will hold a pool identifier, and the data field holds a 64-bit virtual address. On a lookup, the pool identifier from an ObjectID is checked against the tag array. If a valid entry is found, the corresponding base address is obtained and added to the offset field of the ObjectID. The resulting virtual address is sent to the TLB and L1 data cache to access the memory hierarchy. Figure 3.6(a) shows the process of translation for *Pipelined*.

### 3.3.1.2 Parallel

For *Parallel* the fields of the POLB have a different purpose because the goal is direct conversion to a physical address. In this case, we need to know both the pool identifier and the page within the pool that is being referenced. This requires taking some of the bits from both the pool identifier and the offset of the ObjectID. We assume 4 KB pages and a virtually-indexed physically-tagged cache, so the lower 12 bits can be forwarded directly to the L1 data cache. The upper 52 bits of the ObjectID are used to search the POLB for a match. If a valid entry is found, a 52 bit physical address is obtained and used for a tag comparison with the L1 data cache. Figure 3.6(b) shows the process of translation for *Parallel*.

### 3.3.1.3 Comparison

*Pipelined* imposes a delay for translation from ObjectID to virtual address before accessing the cache. It's unlikely the POLB delay can be added with no impact on load latency. *Parallel* can avoid this delay by operating in parallel. This appears to lean in favor of *Parallel* in the absence of other microarchitectural considerations.

On the other hand, the size of the POLB in *Pipelined* only needs to be big enough to track all pools currently mapped in the process's address space, whereas *Parallel* needs to track all active pages in the pool. Some of these pages may be redundantly tracked in the TLB, since we allow regular loads and stores to the same virtual addresses that *nvlds* and *nvsts* are accessing through ObjectIDs.

If the added cost of the *Pipelined* design can be hidden or otherwise avoided, it may be preferable given its smaller size and that it avoids duplicate mappings with the TLB.

### 3.3.2 Persistent Object Table

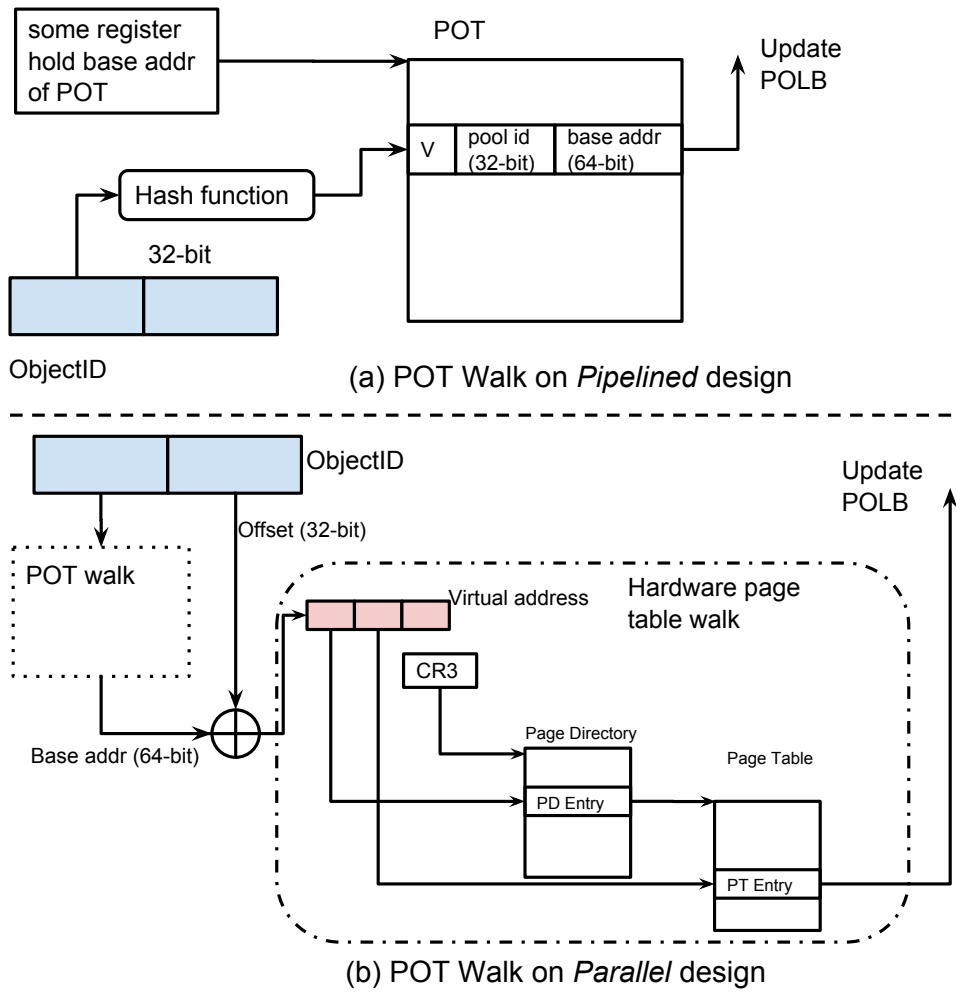
We design the POT under a different set of assumptions than a typical page table. If we assume that pools are like files, then it is reasonable that hundreds to thousands of files may be the limit that is needed. Many Linux systems limit the number of open files by a single process to 1024 or less, which implies that a relatively small table would suffice.

The POT is accessed on a POLB miss, as described in Section 3.2, to look-up entries that need to be placed in the POLB. The POT must contain the information needed to determine if the access is legal and to fill in an entry in the POLB. Each entry in the POT contains a pool identifier and a corresponding virtual base address for the pool. We reserve a pool id of 0 to indicate a NULL pool which cannot exist. This allows us to initialize all entries as invalid.

For both *Pipelined* and *Parallel*, we design the look-up of the POT based on the X86 architecture's hardware page table walk. Figure 3.7 depicts the POT walk. We presume a new architectural register can be used to hold the base virtual address of the POT in memory. The pool identifier from the referenced ObjectID will be taken and calculated through a hash function to get the index within the POT. A legal translation occurs when the POT entry is valid and the pool id in the entry matches the one in the ObjectID.

If a legal entry is found that does not match, linear probing is used to find a match. If an invalid entry is encountered, it means the translation is missing in the POT, and an exception is raised. The OS may abort the program due to an illegal access or allow the program to recover in some way. (For example, a signal handler may establish a legal translation by proper use of the pool creation/opening interface.)

For *Pipelined*, the process is complete after finding a match in the POT. The pool identifier and virtual base address are copied to the POLB and the valid bit in the POLB is set.



**Figure 3.7** Hardware-based POT walk.

In the case of *Parallel*, the process is not yet complete because a physical address has not yet been determined. We could redesign the POT for the sake of *Parallel* so that it could provide the physical page. However, given the assumption of few pools, we choose not to replicate the page table. We opt instead to use the same POT design for both *Parallel* and *Pipelined*. Hence, after the POT walk, the base address found in POT and the offset in ObjectID will be used to perform a page table walk to find the physical frame number (PFN). Then, a new entry can be created in the POLB with the pool identifier and the PFN. Given the need to walk both the POT and the page table, we expect the POLB miss for *Parallel* to take longer to resolve than one in *Pipelined*. The lower part of Figure 3.7 shows the additional page table walk required for *Parallel*.

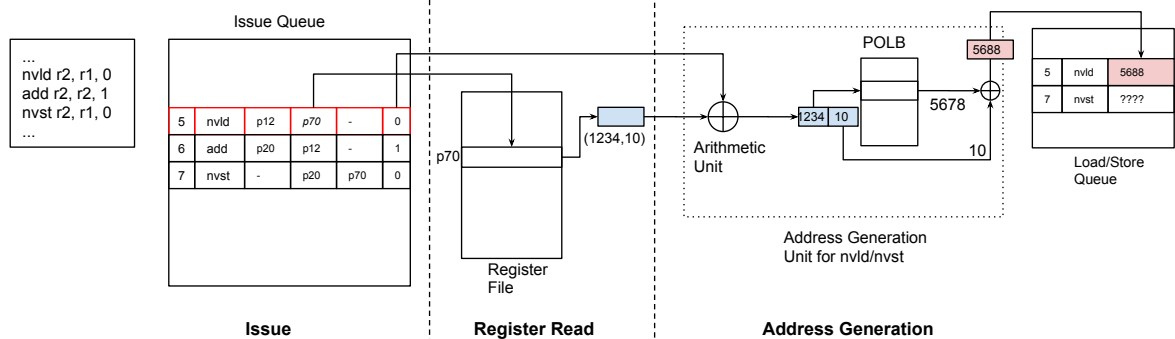
### 3.3.3 Memory Disambiguation

*nvld* and *nvst* introduce new challenges for enforcing memory dependencies and for load-store forwarding. These instructions reference memory using an ObjectID but regular loads and stores use virtual addresses. Conventional memory disambiguation logic in a processor core would not be able to determine whether an ObjectID and regular address alias. Only after converting the ObjectID to a virtual address would such a dependence enforcement be possible.

For *Pipelined*, it is reasonable to convert an ObjectID using the POLB before filling the *nvld*'s or *nvst*'s address in the Load Store Queue (LSQ). This would require placing the POLB in the address generation pipeline stage. This stage occurs between wake-up and before execution so as to give load instructions a cycle to compute their effective address. It would also ensure that the LSQ can perform memory disambiguation and forwarding correctly because it would only ever see virtual addresses.

For *Parallel*, the POLB must be placed in parallel with the cache access since it produces a physical address. However, we cannot allow the LSQ to hold ObjectIDs for pending *nvld* and *nvst* operations, without some modifications. An ObjectID could happen to have the same value as an address currently operated on by the program, leading to a false dependence edge. In the case of a store address that happens to match an *nvld*'s ObjectID (or vice versa), we might perform erroneous store-load forwarding. To prevent this, at a minimum, we would need to mark each entry in the LSQ with a bit indicating whether the entry's address is an ObjectID or not, to properly enforce dependencies and forward data. However, this still does not cover the case of loads bypassing aliased stores. To detect when ObjectID's alias with other normal loads or stores, we would need additional hardware.

Given the design complexity of the *Parallel* design, we choose not to consider it further for out-of-order execution. We do compare *Parallel* and *Pipelined* in the context of an in-order core (Section 3.3.5). However, it's worth pointing out that techniques for aggressive load speculation may be helpful for implementing *Parallel* on an out-of-order processor.



**Figure 3.8** An example of hardware supported translation on an out-of-order processor with *Pipelined* design. The code snippet disassembles a simple C expression, `a++`, but the variable `a` is held in a persistent pool. The data field of the LSQ is not shown.

### 3.3.4 Out-of-Order Pipeline

We integrated the *Pipelined* design for the POLB and POT into an aggressive out-of-order superscalar pipeline. `nvld` and `nvst` flow through the pipeline like regular load and store instructions. When they are decoded, they are reserved an entry in the LSQ in program order. However, at this time, their source operands are not yet known. For a load, that is only the ObjectID, but for a store it includes both the ObjectID and the data operand.

The `nvld` and `nvst` instructions wait in the Issue Queue (IQ) until their source operands are ready. When woken up, they read their source registers and then subsequently their address is calculated through an address generation (AGEN) stage. In this stage, the POLB is accessed, the ObjectID is translated to a virtual address, and the virtual address is placed in the LSQ entry for the instruction. The LSQ enforces memory dependencies and ordering with these virtual addresses in the usual way.

If the lookup on the POLB misses, the Address Generation Unit stalls for the POT walk to complete. If it finds a matching entry, the entry is added to the POLB and the ObjectID translation is forwarded to the LSQ. In the event of a POT miss, the Re-order Buffer entry for the instruction is marked as raising an exception. Once it reaches the head of the re-order buffer, the exception is raised, flushing the pipeline, and the control is handed over to the OS for handling.

Figure 3.8 illustrates an example of how `nvld` and `nvst` instructions are executed on out-of-order pipeline. Assume the instructions are already decoded, renamed and dispatched to the IQ. In the example, the `nvld` instruction is ready to execute with its operand `p70` ready. It flows into Register Read (RR) stage to read register file which stores an ObjectID of `(1234, 10)`. Then in Address Generation (AGEN) stage, the ObjectID is calculated first and it's still `(1234, 10)` because of the immediate being 0. It then goes through POLB to obtain base address of a pool (`5678`) and then calculates the actual virtual address of the access (`5688`). Then the virtual address is placed in LSQ for memory disambiguation.

**Table 3.4** Architecture configuration used for simulation.

Component	Configuration
In-order core	4 cores, X86-64, in-order, 2.66GHz, Branch predictor: Pentium M, Branch misp.: 8 cycles, page size: 4KB, cache block size: 64 Bytes
Out-of-order core	4 cores, X86-64, out-of-order, 2.66GHz, Issue width: 4, LQ: 48, SQ: 32, ROB: 128, Branch predictor: Pentium M, Branch misp.: 8 cycles, page size: 4KB, cache block size: 64 Bytes
Cache	D-TLB: 64, I-TLB: 128, L1D: 8-way 32KB, 3 cycles (1ns), L1I: 4-way 32KB, 3 cycles (1ns), L2: 8-way 256KB, 8 cycles (3ns), L3: 16-way 8MB, 27 cycles (10ns)
POLB	POLB access: 3 cycles (1ns), POT walk: 30 cycles (11ns)
clwb latency	100 cycles (38ns)
DRAM	1GB, 120 cycles (45 ns)
NVMM	Battery-backed DRAM, 120 cycles

### 3.3.5 In-order Pipeline

We also implement on a traditional five-stage pipelined processor: instruction fetch (IF), instruction decode (ID), execution (EX), memory accesses (MEM) and write back (WB).

We assume the processor is updated to support our new instructions. In the EX stage, the ObjectID may go through additional arithmetic logic to perform offset related calculations to determine the final ObjectID, like addresses in memory access instructions. In the MEM stage, they access memory. As mentioned in Section 3.2, for *Pipelined*, the ObjectID first goes through the POLB to calculate the virtual address and then accesses the TLB and data cache in parallel. For *Parallel*, the POLB is accessed in parallel with the cache.

In the case of a POLB miss, the in-order pipeline stalls until the POT walk is completed. If an POT miss occurs, the instruction raises an exception, flushes the pipeline, and hands control to the OS for handling.

## 3.4 Methodology

### 3.4.1 Simulation

We extend Sniper 6.1 [Car14], a cycle-accurate X86 simulator, to model our microarchitecture. We adopt Sniper’s most recent timing model for the Instruction Window-Centric core model (ROB core model) to perform both in-order and out-of-order simulations. The simulator models a QuadCore Intel Xeon X5550 Gainestown (Nehalem-EP) processor shown in Table 3.4. We model the in-order processor using the same frequency and architecture.

We use Pin as the front-end for Sniper. We did not modify the x86 ISA or compilers to compile

new instructions. Instead, to emulate `nvld` and `nvst`, we use normal loads and stores but assign ObjectIDs in such a way that they are clearly distinguishable from the rest of the program's address space. When we identify loads and stores to persistent memory, we mark them and execute them according to our microarchitectural description. This strategy is beneficial because it allows the compiler to heavily optimize the code, demonstrating the full benefit of programming directly with ObjectIDs rather than translating them in software.

The POLB size we choose, by default, is a 32-entry CAM according to our sensitivity analysis in Section 3.5.4. It results in a  $32 \times 32 / 8 = 128$  bytes tag array and  $32 \times 64 / 8 = 256$  bytes data array in *Pipelined* design, and *Parallel* processors, and  $32 \times 52 / 8 = 208$  bytes tag and data array respectively in *Parallel* design. We model the tag look-up and virtual address translation to be 1 ns, i.e. 3 cycles. Sniper does not contain a model for a detailed page table walk, instead it charges 30 cycles for the TLB miss penalty. We also model a fixed 30-cycle POLB miss penalty on *Pipelined*, since it would require, on average, one access to memory. We charge a 60-cycle miss penalty on *Parallel* to estimate the combined POT walk and page table walk. For our design, we believe this is fairly pessimistic since the POT entries would likely hit in the cache and be serviced in a short period of time. However, we perform a sensitivity study to determine the impact of much longer POT walk latencies in Section 3.5.4.

We set the POT size per process to 16384 entries, because we do not expect programs in the near future to leverage a large number of pools. Such a POT requires 256 KB of memory.

We pessimistically model the CLWB instruction for making data durable with a fixed 100 cycle latency, estimated by the CLFLUSH latency on a similar X86 architecture [Ins17].

As for the library extension, we implement all the APIs summarized in Table 3.1. We implement two versions of the library, one that exploits our new instructions and one that does not. Whenever we run a workload on our proposed hardware, we use the optimized version of the library. Otherwise, we use the software-only implementation.

#### **3.4.1.1 Workloads**

We evaluate our design with six micro-benchmarks and the TPC-C [Tpc] application and Vacation from STAMP benchmark suite [Min08], which is also evaluated in [Nal17], listed in Table 3.5. Four of the micro-benchmarks (LL, SPS, BT and RBT) are similar to prior works [Cob11], but they use our library calls to allocate persistent data structures and maintain durability and failure-safety. The B+ Tree benchmark is derived from the core structure of the TPC-C application. In TPC-C, we move the data structures in the form of a B+ Tree to persistent pools. Also we retain TPC-C's own failure-safe logging implementation without modification for persistence. We rewrite core data structures (linked list and red black tree) to persistent pools in Vacation and use interfaces in Table 3.1, similar to WHISPER [Nal17].

**Table 3.5** Summary of workloads used in the simulation.

<b>Name</b>	<b>Abbr.</b>	<b>Description</b>
Linked-list	LL	Search 700 random integers in the linked-list. If the number is found, remove it. Otherwise, insert a new node with the number as key
Binary Search Tree	BST	Search 5000 random integers in the binary tree. If the number is found, remove it and replace the node with maximum key on its left sub-tree. Otherwise, insert a new node with the number as key.
String Position Swap	SPS	Randomly swap a pair of strings in a 32KB string array and repeat 10000 times.
Red-black Tree	RBT	Search 3000 random integers in the RB-Tree. If the number is found, remove it and re-balance the tree according to the red-black rule. Otherwise, insert a new node with the number as key and also re-balance the tree.
B-Tree (order=7)	BT	Search 5000 random integers in the B-Tree. If the number is missing, insert a new node with the number as key and the tree will be re-balanced.
B+ Tree (order=7)	B+T	Search 5000 random integers in the B+ Tree. If the number is found, remove it. Otherwise, insert a new node with the number as key. Both insertion and deletion need to re-balance the tree.
TPC-C	TPCC	Generate 1 warehouse according the parameters in TPC-C spec [Tpc] and perform 1000 transactions.
Vacation	VAC	Travel reservation system with 4096 tasks and 4 queries per each task. Use the suggested high contention configuration (-n4 -q60 -u90 -r16384 -t4096) [Min08].

**Table 3.6** Pool usage access patterns.

Patterns	Description
ALL	All persistent data are in one pool.
EACH	Each structure created by a program is put in separate pools.
RANDOM	The number of pools fixed to be 32 and newly created structure will be allocated in a pool indexed by the key of structure modulo by 32.
COMBINED	All B+-Tree-based structures in TPC-C benchmark are allocated in one pool.
SEPARATE	Each B+-Tree-based structure is put in separate pools.

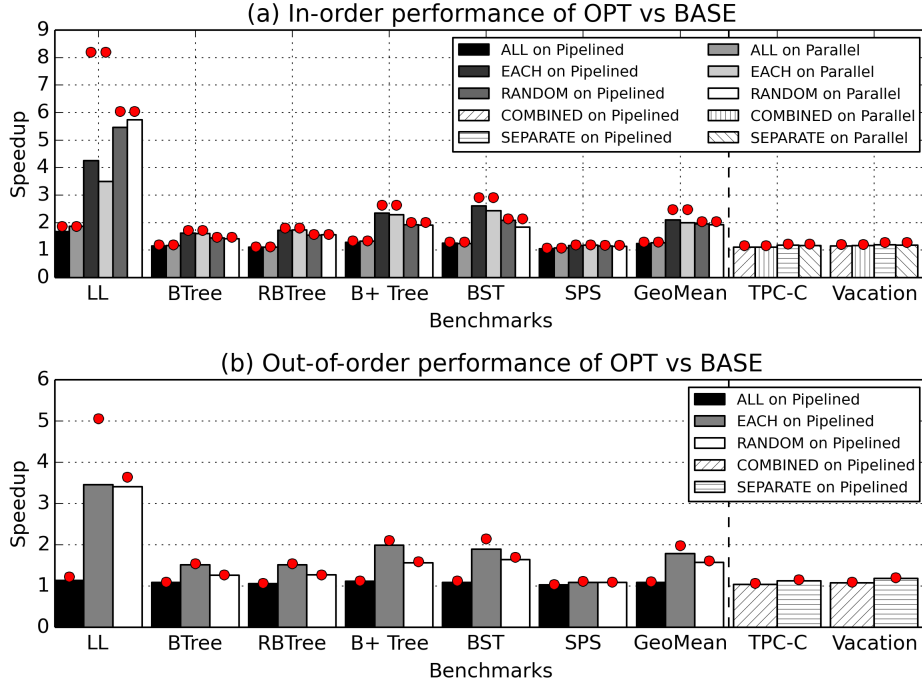
**Table 3.7** Combined architecture and benchmark configurations.

Configuration	Description
BASE	No hardware support for translation. Includes support for failure-safety and durability.
OPT	BASE optimized to use our hardware support.
BASE_NTX	BASE without support for failure-safety and durability.
OPT_NTX	OPT without support for failure-safety and durability.

We develop three different pool usage patterns for each microbenchmark, namely ALL, EACH and RANDOM, and two different usage pattern COMBINED and SEPARATE for TPC-C and vacation, summarized in Table 3.6. This approach allows us to evaluate a variety of access patterns on our microbenchmarks and understand their implications on our proposed hardware.

We run each benchmark on four different configurations: BASE, OPT, BASE\_NTX, OPT\_NTX, summarized in Table 3.7. BASE is fully software without our hardware support. It allocates the persistent data structures using the APIs, and it provides failure-safety through the logging and `persist()` API. OPT also allocates persistent data but instead of using `oid_direct()` to translate from ObjectIDs, all reads and writes will be directly performed with ObjectIDs. It also guarantees atomicity and durability of updates. We also compare to configurations without failure-safety and durability support. These are named BASE\_NTX and OPT\_NTX. They do not provide failure-safety but help determine the performance impact from atomicity and failure-safety. This is important to consider because logging code also uses ObjectID translation and extracts some benefit from our new hardware, but it also introduces significant overheads in the form of logging code and persists.

Each benchmark used in Chapter 3 may have any combination of pool usage pattern and architectural configuration, e.g. executing the Linked-list benchmark with the EACH pattern and OPT configuration means that all nodes to be inserted in the Linked-list will be allocated a new pool and hardware translations are used to access persistent objects.



**Figure 3.9** Speedup of OPT over BASE for each usage pattern on in-order (a) and out-of-order (b) designs. The red dots show the speedup on an ideal architecture where there’s no penalty to perform a hardware translation.

## 3.5 Evaluation

### 3.5.1 Overall Performance

First, we look at the overall performance improvement provided by our proposed architectures. We start with the results for the in-order processor shown in Figure 3.9(a). Here, we show the speedup of OPT over BASE for each benchmark. This comparison showcases the advantages of hardware translation. For each usage pattern, we evaluate both *Pipelined* and *Parallel* to see the impact of each architecture. We also place a red dot over each bar to show best possible performance gain if using an architecture with no hardware translation overhead.

First, we consider the performance of the different usage patterns. For the RANDOM pattern, the BASE design will incur many slow translations using the look-up table since pool accesses occur randomly and are unpredictable, and this means our architectures can offer significant speedups. *Pipelined* attains a  $1.96 \times$  average speedup, and *Parallel* attains a  $1.92 \times$  average speedup. ALL shows the minimal speedup among the three patterns because it uses only one pool. The BASE works well because it is able to leverage its last translation to avoid a look-up, and that reduces the benefit

**Table 3.8** POLB miss rate of OPT benchmark pattern on *Pipelined* and *Parallel* design respectively. *Pipelined* only shows the results of EACH here, because ALL and RANDOM with *Pipelined* only misses 1 and 32 times (0.000%) during POLB warm-up respectively.

Bench.	Parallel			Pipelined
	ALL/COMBINED	RANDOM	EACH/SEPARATE	EACH/SEPARATE
LL	0.0%	0.5%	32.5%	32.4%
BST	0.6%	2.8%	8.1%	7.3%
RBT	0.6%	2.5%	4.1%	3.1%
BT	0.3%	1.4%	2.5%	1.7%
B+T	0.0%	1.0%	2.5%	1.5%
SPS	0.0%	1.2%	2.4%	1.2%
TPCC	2.3%	-	2.5%	0.0%

of hardware translation. ALL results in the minimum number of dynamic instructions among the three.

In the EACH pattern, all objects are placed in different pools. One might conclude that this pattern would attain the best speedup on our architecture. However, that is not always the case. There is a performance trade-off between the number of translations saved and the number of misses in the POLB. EACH and RANDOM benchmarks require significantly more translations, which gives more speedup than ALL. Comparing EACH and RANDOM, the one that achieves better performance depends on the predictor miss-rate in the BASE version versus POLB contention caused by an increasing number of pools. According to Table 3.8, if a benchmark doesn't create large contention on a 32-entry sized POLB in EACH, it should have a bigger speedup for EACH than RANDOM. On the other hand, LL has a large miss rate on the POLB compared to RANDOM, so the miss penalty of the POLB negates some of the benefits of eliminating software look-ups. Thus RANDOM is better than EACH for LL.

Now consider the differences between *Pipelined* and *Parallel*. *Pipelined* performs better than *Parallel* in all benchmarks. This suggests that the extra cycles spent on POLB accesses are a smaller overhead than the additional POLB miss penalty for *Parallel*. Table 3.8 shows the POLB miss rate for both *Pipelined* and *Parallel*. *Pipelined* has much lower miss rate than *Parallel*, and only has 1 and 32 misses (0.00%) on ALL and RANDOM benchmarks respectively. From this analysis, the *Pipelined* appears to be the better choice given its simplicity and performance advantage. Overall, both are reasonable approaches.

The TPC-C application gives additional perspective on the importance of our hardware, because the translation overheads are incurred within a larger application, not a kernel with frequent translations. We measure a  $1.10 \times$  and  $1.17 \times$  speedup on COMBINED and SEPARATE on TPCC, and  $1.14 \times$  and  $1.19 \times$  speedup on COMBINED and SEPARATE on vacation respectively. Even in applications, the speedups are significant.

**Table 3.9** Percentage of instructions saved with OPT benchmarks on micro-benchmarks and TPC-C

<b>Bench.</b>	<b>ALL</b>	<b>EACH</b>	<b>RANDOM</b>
LL	65.6%	82.0%	75.9%
BST	58.0%	79.6%	69.7%
RBT	52.5%	75.1%	65.6%
BT	52.0%	74.0%	63.5%
B+T	47.1%	69.5%	58.8%
SPS	35.9%	53.3%	46.9%
TPCC	47.9%	63.0%	-

We also evaluate the ideal performance attained where there’s no penalty to access the POLB and no miss penalty on the hardware POT walk, thereby measuring a theoretical upper-bound for our hardware. These results are shown as the dot above each bar. The results show that our *Pipelined* and *Parallel* design, although they introduce overhead, can compete with the ideal for most usage patterns and workloads. The notable exception is LL on the EACH pattern. In this case, even though the total number of pools is not the largest among our workloads, each operation on the linked list involves a traversal starting from the head of list, which leads to poor temporal locality and high POLB contention.

Figure 3.9(b) shows the speedup of the *Pipelined* design on out-of-order processor. Overall, it shows a similar trend across the usage patterns as the in-order architecture. A key difference is that it shows less speedup than in-order because out-of-order execution hides some of the overhead of slow software translation because it extracts more instruction level parallelism.

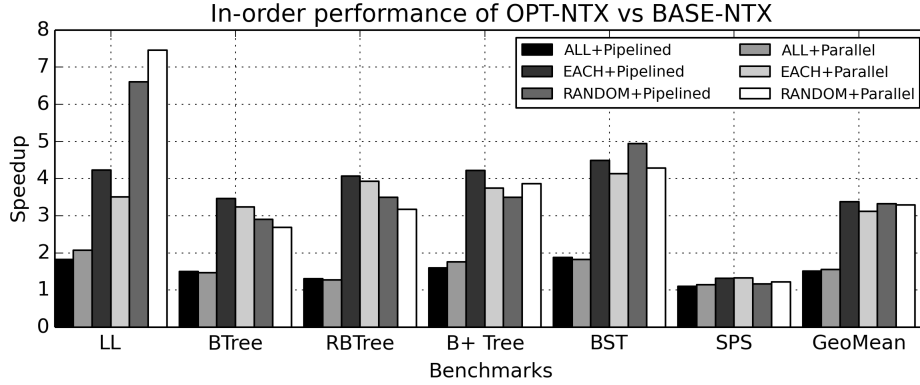
There are a couple of workload specific behaviors worth pointing out. LL has more speedup than the tree structures because of more translations that are on the critical path, and LL has longer chains of accesses than tree structures. SPS, on the other hand, is array-based and does not need a translation per element in the BASE case. Thus it has the least speed up on both in-order and out-of-order implementations.

### 3.5.2 Reduce Instructions

As evaluated in Section 3.1.2.1, software translations can incur overheads in terms of number of instructions. Table 3.9 shows that with hardware translation involved, the number of instructions is significantly reduced in the microbenchmarks and TPC-C.

### 3.5.3 Overhead of Durability and Atomicity

We also evaluate our workloads without persistence and atomicity support. The OPT\_NTX and BASE\_NTX are evaluated on both the *Pipelined* and *Parallel* designs. The results on the in-order processor are shown in Figure 3.10 normalized to BASE\_NTX. The speedup on both designs are



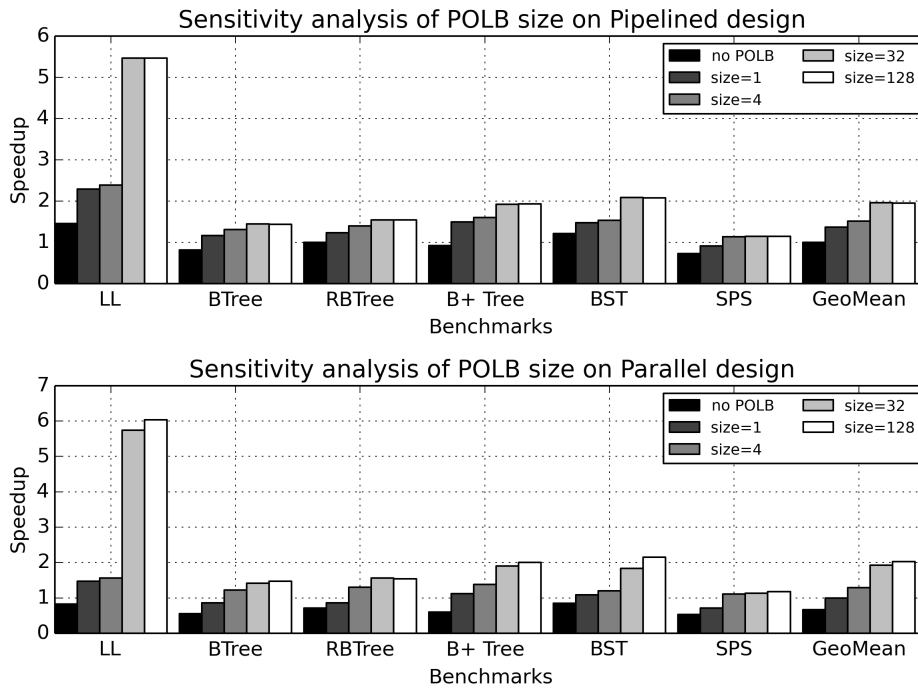
**Figure 3.10** Performance of all OPT\_NT\_X benchmarks without persistence and transaction supports on both Pipelined and Parallel designs. The results are normalized to the BASE\_NT\_X benchmark running on baseline system.

higher than the prior case with persistence and atomicity support. One reason for this difference is that without the added overhead of logging which requires additional translations the working set of pools is small enough that each pool can fit within just one page. Hence, the *Parallel* design only needs one entry per pool in the POLB, which reduces the contention and makes *Parallel* faster than *Pipelined* on EACH. However on the RANDOM pattern, *Pipelined* still has more speedup than *Parallel*. *Pipelined* does not miss on the POLB (except during warm-up) when running a RANDOM pattern, but *Parallel* has more pages of data within one pool and that results in more POLB contention especially in comparison to the ALL pattern.

### 3.5.4 Sensitivity Analysis: POLB size

We select a fixed POLB size of 32 entries on the previous studies. In this section, we perform a sensitivity analysis on the POLB size for workloads with RANDOM patterns. By design, these workloads use 32 pools. We sweep the number of POLB entries from 1 to 128. We also include the results when there is no POLB in the system forcing all hardware translations to perform a POT walk.

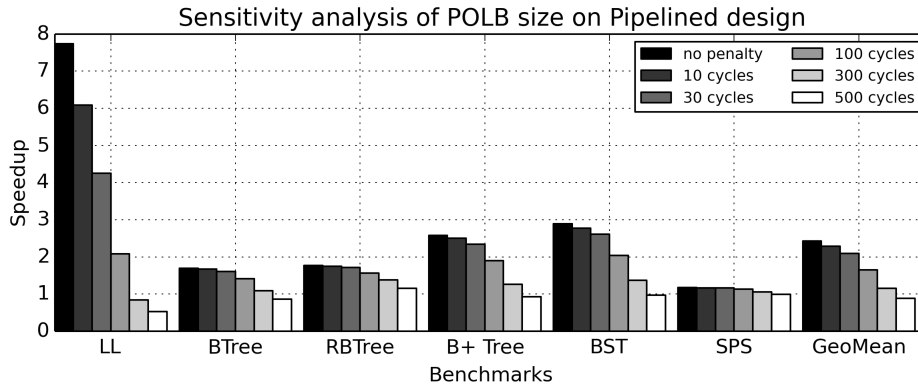
The results are shown in Figure 3.11. We see that without a POLB most workloads slow down. Also, the speedup saturates at a specific POLB size once the POLB is large enough to eliminate most misses. The *Parallel* design requires a larger POLB than *Pipelined* because it incurs more conflict misses, as shown in Table 3.10. We can surmise that a 32-entry POLB will be sufficient for many workloads, under the expectation that programmers will not operate on hundreds of pools in near future applications.



**Figure 3.11** Sensitivity analysis to POLB size. Each bar shows the normalized speedup of OPT over BASE for the EACH pattern on Pipelined and Parallel design on in-order processor, with no POLB and POLB size=1, 4, 32, and 128, respectively.

**Table 3.10** POLB miss rates on OPT\_NTX using the RANDOM usage pattern on *Pipelined* and *Parallel* designs while increasing the size of the POLB.

Bench.	Pipelined				Parallel			
	1	4	32	128	1	4	32	128
LL	31.7%	28.3%	0.0%	0.0%	32.1%	28.4%	0.5%	0.0%
BST	19.7%	16.4%	0.0%	0.0%	25.9%	17.6%	2.8%	0.0%
RBT	36.9%	8.1%	0.0%	0.0%	58.7%	11.5%	2.5%	0.0%
BT	17.4%	4.7%	0.0%	0.0%	29.4%	6.4%	1.4%	0.0%
B+T	8.7%	3.8%	0.0%	0.0%	18.7%	5.0%	1.0%	0.0%
SPS	40.8%	1.1%	0.0%	0.0%	48.1%	2.3%	1.2%	0.0%



**Figure 3.12** Sensitivity analysis to POT walk penalty. Each bar shows the normalized speedup of OPT over BASE for the EACH pattern on the in-order *Pipelined* design.

### 3.5.5 Impact of Hardware POT Walk

In our model, we assume that the hardware POT walk has a fixed 30-cycle latency. In this section, we estimate the impact of the POT walk by varying the fixed latency from 10 cycles, 30 cycles, 100 cycles, 300 cycles, and 500 cycles. We choose 10 cycles and 30 cycles because we expect caching to reduce the penalty of POT accesses. We sweep higher POT latencies to explore their impact on performance. We also evaluate the results with an ideal system that charges no POT walk penalty. The performance speedups are shown in Figure 3.12.

POT walk latency has a higher impact on workloads with a high POLB miss rate, namely LL. We can conclude that an actual hardware POT walk averaging around 30 cycles would have a small impact on the performance of the system, but an even smaller POT walk penalty will benefit benchmarks with higher POLB miss rates, namely LL.

## 3.6 Summary

In this work, we propose treating ObjectIDs as a new persistent memory address space and provide hardware support for efficiently translating ObjectIDs to virtual addresses. With our design, a program can use load and store instructions to directly access persistent data using ObjectIDs, and these new instructions can reduce the programming complexity of this system. We also describe several possible microarchitectural designs and evaluate them. The *Pipelined* approach is preferred for out-of-order processors because the address translation operation needs to occur before memory disambiguation. Fortunately, this design choice did not significantly hurt the performance potential of our approach, given the large savings of removing frequent software translation from the program. The *Pipelined* design offers a small advantage over *Parallel* on in-order cores due to its lower miss rate

and lower miss penalty. On microbenchmarks that place persistent data randomly into persistent pools, the *Pipelined* implementation has an average speedup of 1.96× and 1.58× on an in-order and out-of-order processor, respectively, over the baseline system.

## CHAPTER

# 4

# HARDWARE SUPPORTED PERMISSION CHECKS ON PERSISTENT OBJECTS FOR PERFORMANCE AND PROGRAMMABILITY

In last Chapter, we have shown that hardware-supported address translation for ObjectIDs provides significant performance improvement and simplifies programming, which makes it easier for programmers to manipulate persistent data. However, in order to access persistent pools, programs need to have permissions to do so. All prior works did not consider the large overheads incurred to check permissions before accessing persistent objects and leave the burden to programmers. In this Chapter, we will answer the question: *how to ensure correctness when accessing data across different persistent regions?* Prior works, including Chapter 3, assume it's the programmers responsibility to check permission and map all the pools at the beginning of the program so later accesses to the pools will succeed. This is an optimization but requires significant effort from programmers. The goal of this Chapter is that *we want to make permission check transparent to programmers and the ObjectID is automatically validated when being accessed.*

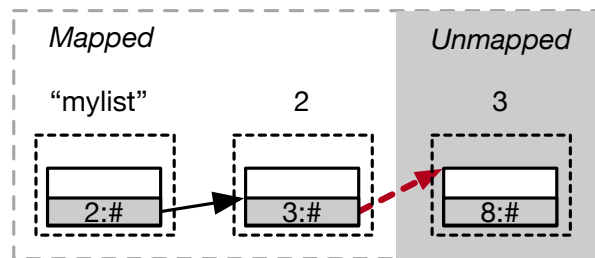
In Section 4.1, we identify permission checking is a problem that can't be solely rely on program-

mers effort. In Section 4.2, we identify that hardware-supported translation provides an opportunity and hardware support as a critical mechanism that must be included when translating ObjectIDs to addresses in order to simplify programming and fully benefit from hardware translation. To support it, in Section 4.3, we add a System Persistent Object Table (SPOT) to support translation and permissions checks on ObjectIDs. The SPOT holds all known pools, their physical address, and their permissions information in memory. When a program attempts to access a persistent object, the SPOT is consulted and permissions are verified without trapping to the operating system. We have implemented the design in a similar simulation environment and benchmarks as in previous Chapter (Section 3.4) and compared it with software only approaches and prior work. In Section 4.4, we find that our design offers a compelling 3.3x speedup on average for microbenchmarks that access pools with the RANDOM pattern and 1.4x and 1.7x speedup on TPC-C and vacation, respectively, for the SEPARATE pattern.

## 4.1 Motivation

### 4.1.1 Example

We consider an example to motivate the additional problem of permissions checking on persistent objects. Figure 4.1 shows a persistent list. Each pool contains one object. After an initial partial traversal of the list, some of the pools have been opened and some have not. The red dotted-line indicates an ObjectID pointing to an object in an unopened pool. Before a dereference to the object in the unopened pool occurs, the programmer must open the pool to map it into memory and check that the access is permissible.



**Figure 4.1** Persistent data structure spread across pools. Some pools are mapped and others are not.

This leads to new challenges that are different from typical linked data structures. Normally, if an address is not NULL and a program is free of memory related bugs, it is legal to dereference that address. In this case, persistent objects were created in a previous run of the program and persist

until the next run. The ObjectID remains valid across program runs, however, the ObjectID would no longer be legal to access in the second run unless it were first opened. This leads to the challenge that the programmer must somehow know which parts of a linked data structure are mapped and which are not. If it is not easy to know, it is better to be conservative and check to make sure the pool is opened before accessing it.

Moreover, between two runs of the program, other programs may access this data structure and update it. New objects may be added. Hence, the programmer may not assume that the same ObjectIDs are present and remain valid from one run to the next. Hence, the legitimacy of the ObjectID must be verified again.

#### 4.1.2 Permissions Checking Support

A correct persistent programming model should perform permission checking on all ObjectID before dereferencing it. This should be the baseline of all system but it's very slow considering the permission check takes long time. With a library like Table 3.1, programmers should check permissions of all ObjectID before dereferencing it. It obviously will have big overhead to use persistent pools, so we need optimizations.

Prior works like PMDK[Pme], including Chapter 3, assume that the permissions are checked when programmers call `pool_open` and it's programmers responsibility to call `pool_open` to open and map the pools that will be used in the program. It can be seen as an optimization of reducing overhead of permission check but increases programming burden. Sometimes, it's hard to open pools with names when considering the pools can be shared between programs.

In this Chapter, we will go another route. We want to remove the burden from programmers to manually open all the pools upfront. We will propose several ways to check permissions on ObjectID while accessing it. Inspired by the linked list example, we add a few new functions to the API shown in Table 3.1 to support permissions checking, as shown in Table 4.1.

First, consider `oid_check`, which checks the mapping and permissions of a pool using a known ObjectID. This function can be implemented fully in library code by remembering which pools have already been opened. If the pool has been opened once before, we can assume that the system previously granted access. This is the approach used by Persistent Memory Development Kit (PMDK, formerly known as NVML) [Pme]. This approach can be efficient because it can leverage a fast data structure to perform the check, like a hash table or binary tree, and avoid any overhead of trapping to the operating system. It operates under the implicit assumption that once access to a persistent region is granted it is never revoked.

This function is helpful if an ObjectID is obtained in a linked persistent data structure, but whether or not it refers to a pool that has already been opened is unknown.

Figure 4.2 shows a linked list traversal modified to properly check permissions. Before attempting

**Table 4.1** Pool and ObjectID Extended API to support permissions checking.

Function Calls	Descriptions
<b>Pool Management</b>	
pool* oid_open(oid)	Reopen a pool using an ObjectID rather than a name. Permissions will be checked.
<b>Permissions Management</b>	
int oid_check(oid,mode)	Check if corresponding pool is open. Return an error code if not.
void* oid_check_direct(oid, mode)	Translate but perform permissions checks first.

to convert the ObjectID to virtual address, the status of the ObjectID is determined by calling `oid_check`. If the pool it is contained within is not already open, `oid_open` is called to force the mapping of the pool and the permission check. The repeated calls to `oid_check` incur a significant and necessary overhead.

```
1 typedef struct {
2     int value;
3     OID next;
4 } node;
5 // find the first node that matches data
6 OID find (OID head, int data){
7     OID tmp = head;
8     while (tmp != OID_NULL) {
9         if (!oid_check(tmp,"rw"))
10            oid_open(tmp,"rw");
11        node *x = (node*)oid_direct(tmp);
12        if (x->value == data)
13            return tmp;
14        tmp = x->next;
15    }
16    return OID_NULL;
17 }
```

**Figure 4.2** Example of a persistent linked list traversal with permission check.

### 4.1.3 Optimizing Checks in Software

We can further optimize the linked list traversal specifically, and permissions checking in general, by embedding the check into the translation function. We provide a function, similar to similar to `oid_direct`, called `oid_check_direct` to perform software translation and permissions checking at

the same time. By merging these two operations, we can leverage the fact that translations have significant locality. If the last translation was for the same pool, there is no need for any additional checks.

The pseudo-code in Figure 4.3 describes the procedures in `oid_check_direct` and differentiates the additions to the baseline `oid_direct` function. The new function will try to translate the ObjectID using the most recent translation, but if that fails, it will trap to the OS to check for permission to access the pool and will map the pool if permission is granted.

```
1 void* oid_check_direct(pool_id, offset):
2     if (most_recent_info_valid && oid.pool_id == most_recent_pool_id)
3         return most_recent_base_addr + oid.offset;
4     most_recent_info_valid = 1;
5     most_recent_pool_id = oid.pool_id;
6     most_recent_base_addr = OIDTranslationMap->find(oid.pool_id);
7     if most_recent_base_addr != NULL
8         return most_recent_base_addr + oid.offset;
9     // Different from oid_direct
10    else
11        //Trap to kernel-level
12        valid = check_user_permission(user ID, pool_id);
13        if valid
14            base_addr = map_pool(pool_id);
15            OIDTranslationMap->insert(pool_id, base_addr);
16        else
17            abort program
18        //Return to user-level
19        most_recent_base_addr = OIDTranslationMap->find(oid.pool_id);
20        return most_recent_base_addr + oid.offset;
```

**Figure 4.3** Pseudo-code of `oid_check_direct`. The if-else part is added to handle the situation where a translation is missing (i.e. pool not mapped).

During the translation, the pool ID will be searched in the hash table for translation as normal. But when the pool ID is not found in the table, instead of aborting the program, we provide a series of kernel-level actions that are equivalent to `pool_open`. We will perform a system check to see if the programmer has permission<sup>1</sup> to access the pool. If the permission is granted, the pool will be mapped into the program's address space and a translation entry will be added to the table. Thus the translation can proceed without problem and further translations will also proceed without operating system involvement. If the permission is denied, the program will be aborted.

The action taken when the pool is not already open is analogous to opening a file and the

---

<sup>1</sup>We will grant maximum permission level to the programmer. For example, if the programmer has read and write permission to the pool, we will grant both permissions.

associated permissions checks therein. It is also similar to the method for handling a page fault, at least in principle. The physical page of the persistent object was once legal to access, but the mapping must be renewed on subsequent runs of the program. Interestingly, the data may still reside in the same location in NVMM. Unlike a page fault that moves the data into memory from disk, in our case, we simply need to restore the mapping. However, the OS must take action to create the mapping.

We can optimize the code in Figure 4.2 to use this new function. We remove lines 9 and 10 and replace the call to `oid_direct` with `oid_check_direct`.

However, it is worth noting that `oid_check_direct` cannot be replaced with a single `nvld` or `nvst` instruction because they do not include permissions checks.

#### 4.1.4 Permissions Checks with Hardware Supported Translation

Software translation imposes significant overheads [Wan17; Che17] and permission checks in software add even more overheads, so it is advantageous to integrate these permissions checks with hardware that performs translation.

Without adding any additional hardware support, we can implement an approach that relies on hardware translation with software level checking of permissions. For example, this amounts to the same code as in Figure 4.2 with `oid_direct` replaced with `nvld`. However, the added software permission checks bring a significant overhead (shown in Section 4.4). Instead, these checks would ideally occur as part of the `nvld` not as a separate functionality.

With modest changes, the *Pipelined* design (Section 3.3.1.1) can support streamlined permissions checks. Suppose an `nvld` or `nvst` accesses an ObjectID for a pool that has not yet been opened. In this case, the POT will miss. We modify the design to raise an exception on a POT miss, and the OS runs an exception handler, like a page fault handler, that will find the corresponding pool for the ObjectID. If such a pool exists and if the running process has permission, it can be immediately mapped into the POT. This mechanism allows the programmer to avoid checks on ObjectIDs that are known to be legitimate. This extension adds no additional cost to the *Pipelined* design proposed in previous Chapter. However, it does incur a significant performance penalty on a POT miss. An OS exception must be raised and a handler invoked. This may add significant overhead when traversing data structures that spread across many different pools.

We argue for an even more streamlined design. Persistent objects are expected to be permanently resident in memory. Unlike pages that have been swapped to disk or files that need to be copied into memory from disk, few actions are needed to make a persistent object accessible, since it still resides in memory. In reality, all that is needed is knowledge of its location in memory. A system-level table that works like a page table could provide this information at relatively low hardware complexity. With such a design, we do not need system calls (in software) or a trap to an OS exception handler

(with hardware support) to map pools into the program.

Such integration is non-trivial. The hardware would need access to the privileged system-level data structure that would identify the physical location and permissions information of persistent pools. It's worth noting that this is substantially different from the information provided by the page table. In the next section, we build on this idea, and we describe our hardware support for performing permissions checks directly in hardware.

## **4.2 Design: System Persistent Object Table (SPOT)**

### **4.2.1 System Persistent Object Table**

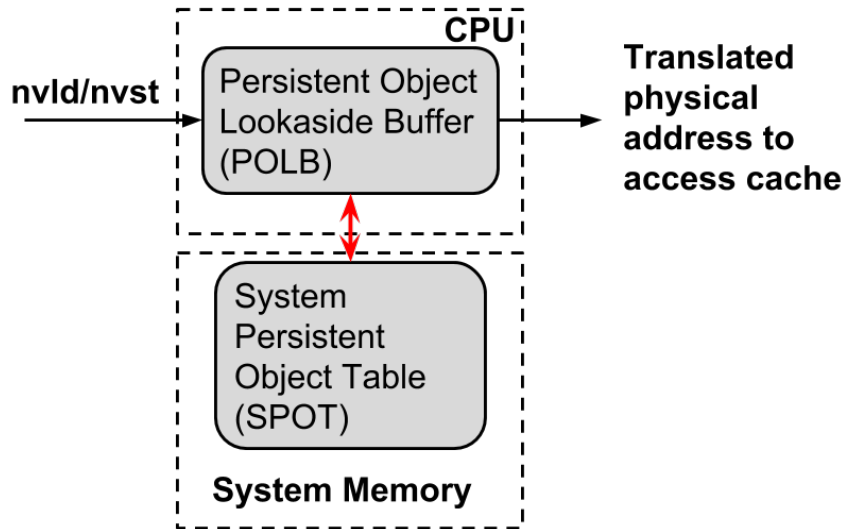
Once created, persistent objects are always in memory. Rather than relying on the operating system to re-map them into a process' address space, we provide hardware support to carry-out this action and to check permissions. We add a system data structure called the *System Persistent Object Table* (SPOT) to keep a record of all the pools created in the system. When a pool is first created, an entry is added to the SPOT that stores the physical address of the pool and important information, like the owner, group, and permission information.

The SPOT serves as a backing store for the POLB and replaces the POT in Chapter 3.2, as shown in Figure 4.4. In the event of a miss in the POLB, a hardware walk over the SPOT, similar to a page table walk, finds the corresponding pool's entry in the table and adds it to the POLB if allowed. As long as the desired object is found in the SPOT and permissions check out, this entire process occurs without trapping to the operating system. However, if the SPOT indicates that the object is not valid or there are insufficient privileges to access the pool, an exception is raised for OS handling. If the OS cannot resolve the request, a memory protection violation is signaled.

Note, the SPOT walk is different from a typical page table walk that occurs on a TLB miss. A page table only contains entries for pages that have been mapped into the virtual address space. It does not contain pages for files or other objects that are not yet known to the program. Furthermore, for entries in the page table to be updated, an OS-level routine must take an action. On the other hand, the SPOT does contain persistent objects that may not be legal for a program to access, and hardware helps decide whether or not it is legal for the program to access them.

### **4.2.2 Pool Open Using SPOT**

When an entry from the SPOT is moved to the POLB by hardware, this is equivalent to the `pool_open` function in the API. It legalizes the `ObjectID` by creating a translation. In prior work, it was assumed that a persistent pool was first mapped into the address space of the process before accessing it [Pme; Wan17]. However, we are skipping that step by establishing a direct translation from `ObjectID` to



**Figure 4.4** Overview of the design of automatic permission checks with System Persistent Object Table (SPOT).

physical address. We design our POLB to translate directly from ObjectID to physical address. This implies building on top of the *Parallel* design (Section 3.3.1.2).

### 4.2.3 Permission Checking Using SPOT

We also add hardware that is equivalent to the `pool_check` function. Each entry in the SPOT stores the physical address of a pool and important information, like the owner, group, and permission list. We borrow our design from standard Unix file system implementations that specify owner, group, and access settings (e.g. read, write, and execute privilege) for each file.

In order to support this check in hardware, we add protected registers to the core that specify the current user and the groups to which the user belongs. These registers can only be set during privileged execution.

Before an entry is copied from the SPOT, the user and group are validated against the SPOT entry. For example, if the current user matches the owner of the persistent pool, then access is permitted. Or, if the user is different from the owner but the user is a member of the same group, some level of shared access may be permitted.

We discuss the permission checking logic more in Section 4.3.2.

### 4.2.4 Organization of the SPOT

The SPOT will hold a record of the persistent pools in the system. The ObjectID format allows up to  $2^{32}$  unique pool identifiers, so we design the SPOT for scalability up to a large number of persistent

pools. X86 page tables use a multi-level design. Such designs provide low memory overhead in the common case, allow for a full address space, and work well even with small page sizes. For these reasons, we also propose a multi-level design for the SPOT.

The organization of the table is dependent on the layout and management of persistent pools. If pools are always laid out contiguously in the physical address space, then only the top 32-bits of the ObjectID matter for translation. We can design fewer levels of table by translating at large granularities, and this results in less memory overhead and fewer accesses to memory to look up a SPOT entry.

On the other hand, it is more likely that persistent pools are not contiguous, in order to allow the system to flexibly manage the NVMM device with pools of varying sizes. This implies translation at the granularity of the page size. In addition to incurring more overhead from additional tables, there is also potentially more redundant information. For example, the permissions information is only needed for each pool, not for each page. To optimize for this case, we design a multi-level SPOT that translates at page granularity but only stores permission information at pool granularity. These designs are presented in Section 4.3.

## 4.3 Implementation

We now describe the hardware implementation for the SPOT and translation and permission checks.

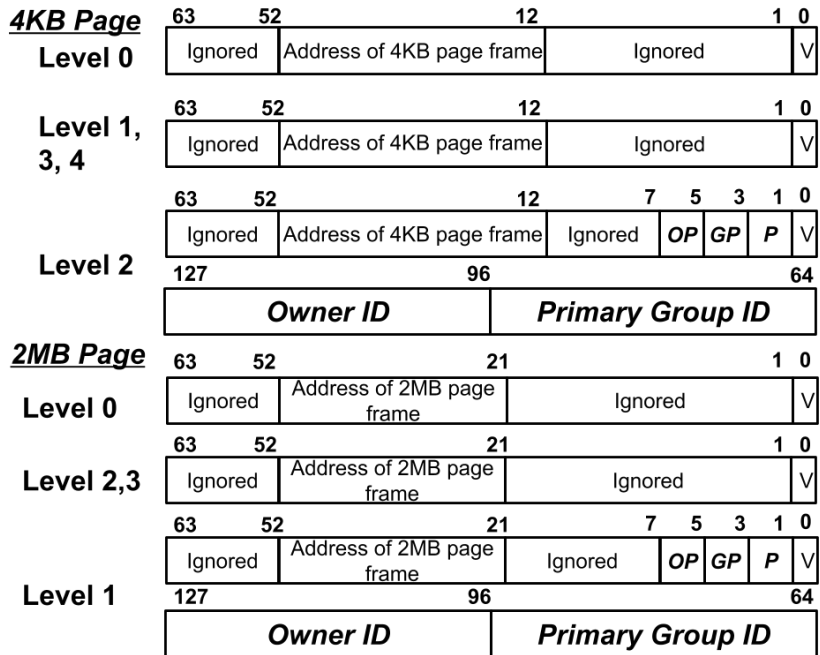
### 4.3.1 System Persistent Object Table

When a translation is not found in the POLB, a privileged hardware SPOT walk traverses the SPOT looking for the corresponding pool and object information. The starting address of the SPOT is stored in a control register so that hardware can perform an SPOT walk, similar to a page table walk.

We adopt a multi-level design for the SPOT, and we consider multiple designs to account for different page sizes. Page size is a key architectural parameter for the design of the SPOT. Larger pages mean a smaller table and fewer look-up steps. However, larger pages have the downside of greater internal fragmentation.

Figure 4.5 shows the design of the SPOT entry depending on the page size and for each level of the table. We assume the same page sizes as the Intel 64 architecture, where page size can vary among 4KB, 2MB or 1GB [Int]. Each entry either stores the address of the next level of the table, or entries in the last level table store the physical page frame number for the page.

Another functionality of the SPOT is to perform the permission check. One challenge is that we only need the owner, group, and permissions bits per pool, but we need translations per page. If we repeat the owner, group, and permissions bits in every entry of the last level of the SPOT, it would add a significant amount of redundant information for all pages in a pool. Instead, we place the

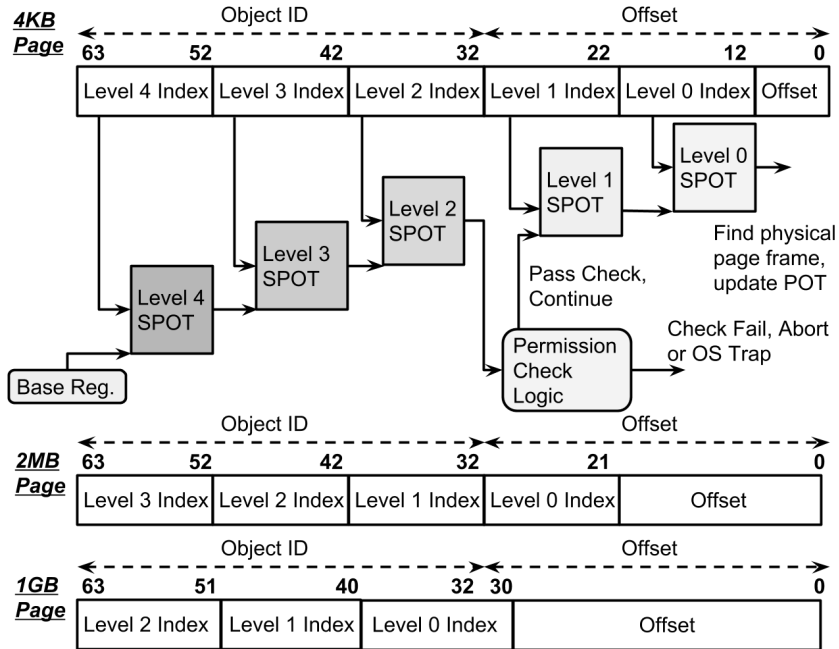


V: Valid bit. 1: Present. 0: Not Present  
 OP (2 bits): Owner Permission bits. 1x: Write. 10:Read-only. 0: No permission.  
 GP (2 bits): Group Permission bits. Permission of owner's primary group. Same bit representation.  
 P (2 bits): Other Permission bits, for all the other users. Same bit representation.  
 Owner ID (32 bits): Set by O/S. Represent the owner of this page.  
 Primary Group ID (32 bits): Set by O/S. Represent the primary group of this page.

**Figure 4.5** SPOT entry details on systems with 4KB or 2MB page size.

permissions information at the granularity of a pool, and we only store translations at the granularity per page. To save the owner ID and group ID, the mid-level tables need an extra 64 bits compared to the other table entries. The permission check is depicted in Figure 4.6 with the Permission Check Logic reading the entry from Level 2 of the SPOT.

The overall procedure for the SPOT walk is demonstrated in Figure 4.6 with a 4KB page example. The starting address is stored in a special control register and preset by privileged operations. The 64-bit ObjectID is used to walk the SPOT. Each level takes part of the ObjectID to calculate the index for its table. A look-up at that index yields the starting physical address for the next level of the table. We use the level 2 table to perform the permission check logic (described in Section 4.3.2) to verify if the user has permission or not. Lower levels will continue to be accessed only after the permissions are validated. After the lowest level entry is read out, the physical page frame is obtained and used to provide an entry to the POLB along with the correct read and write permissions from level 2 (more



**Figure 4.6** Hardware SPOT walk on 4KB page size system. 2MB and 1GB have similar procedure with different number of SPOT levels.

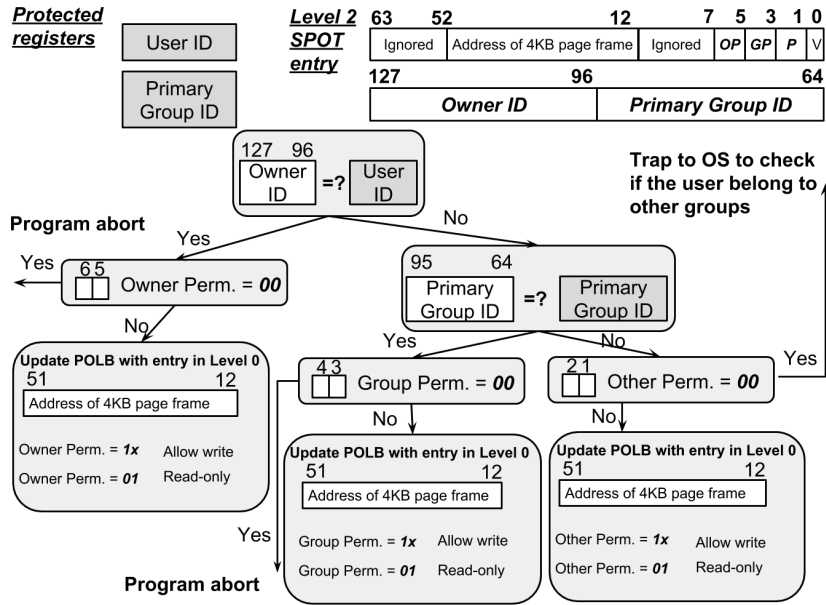
details in Section 4.3.2).

### 4.3.2 Permission Checking Logic

To support permission checking logic, we add permissions information to the mid-level SPOT tables, and we add protected control registers to the core to indicate the user ID and primary group ID. In reality, a user may be part of more than one group. The number of registers devoted to group IDs could be adjusted to match the common case, provided it is not too large<sup>2</sup>. Each user is assigned to a primary group and we store the primary group ID to check permissions. These registers would be set as part of the context switch code in the OS. We assume 32-bits is sufficient for the user ID and primary group ID. So, a 64-bit register can hold both of the IDs.

Figure 4.7 illustrates the logic added to perform permission checking. First, bits [127:96] will be read out to compare the current user ID against the owner ID. If the IDs match, meaning the current user is the owner, the POLB will be updated with the physical page frame number in bits [51:12] from the last level entry of the SPOT, provided the read/write permission checks out. The operation of `nvst` will fail if the Owner Permission bits are `01` because that means the entry is read-only. Both `nvld` and `nvst` will succeed if the Owner Permission bits are `11` or `10`.

<sup>2</sup>An informal inspection of Linux workstations suggests that most users are only a member of a handful of groups at most.



**Figure 4.7** Flowchart of the added hardware logic for permission check used during SPOT walk.

If the IDs don't match, primary group IDs will be checked to see if the current user is in the primary group of the owner. If so, and the group is granted access (Group Permission is not 00), the POT will still be updated with physical page frame number and set to the corresponding read and write permission. If the group matches but has no access, an exception is raised for handling by the OS or user program. If the current user is not in the group, the Other Permission bits will be used to check permission. If the Other Permission bits are 00, an OS exception will be raised and checks if the user belongs to other groups of the owner. Otherwise, the permission can be granted and the POT updated with the physical page frame number and correct read/write permission according to the Other Permission bits.

#### 4.3.2.1 Load-Store Disambiguation in *Parallel*

As reported in previous Chapter, the *Parallel* design adds complexity to load-store disambiguation in out-of-order processors. We assume oracle support for load-store speculation, so that we can evaluate the *Parallel* design on out-of-order processors. Many prior works have considered how to support aggressive load and store speculation [Rot05; Gan05], and these designs need to be re-considered in the context of ObjectIDs. We leave such considerations to future work.

### 4.3.2.2 Getting Unique Pool Identifiers

Pools need unique identifiers. One way to obtain them is using a utility like the the Universally Unique Identifier [PLS05] in Linux. Although the probability of a duplicate in UUID is not zero, it's very close to zero in practice. In our implementation, the Pool ID is a 32-bit number, so we need some function to map the 128-bit number into a 32-bit pool ID which increases the likelihood of collision. However, if a collision happens, we can linearly search for the next available pool ID by linear probing on the SPOT.

### 4.3.2.3 Concurrency and Updates

Since the SPOT is shared, concurrent updates will occur. These updates are made by the operating system and must be synchronized.

## 4.4 Evaluation

### 4.4.1 Methodology

#### 4.4.1.1 Simulation Environment

We evaluate the system by extending the simulation environment described in Section 3.4. We extend Sniper to simulate system behaviors like the page table walk.<sup>3</sup> We fully emulate the page table walk. We read the base register of the highest level of the page table and access each entry in all levels until we find the page table entry. The latencies are added up depending on which level of the cache hierarchy each entry is found within. We use a similar mechanism to simulate the SPOT walk.

#### 4.4.1.2 Different Designs

We implement permission checks in software as described in Section 4.1.2 and Section 4.1.3, referred to as SwT-Base and SwT-Opt respectively. SwT-Base is a naive implementation, and SwT-Opt is similar to NVML [Pme]. We use C++ Standard Template Library (STL) map (tree-based implementation) to implement the software translation table and the system-level data structure that holds all of the persistent pools (i.e. equivalent to the SPOT but in software).

We also implement two designs that use hardware-supported translation. One has hardware support for translation that doesn't integrate the permission check (i.e. no SPOT). In this design, the programmer must call `oid_check` manually, as described in Section 4.1.4. This design will be referred to as HwT.

---

<sup>3</sup>Sniper doesn't have a model for the page table walk, by default. Instead, it charges a fixed 30 ns (80 cycles) for a TLB miss.

**Table 4.2** Summary of the designs used in the evaluations.

Design	Description
SwT-Base	Naive implementation. It requires programmers to call <code>oid_check</code> to determine status of a pool and manually open a pool. Described in Section 4.1.2.
SwT-Opt	Programmers can use <code>oid_check_direct</code> to translate ObjectID, to perform software translation while checking permissions. Described in Section 4.1.3.
HwT	Baseline hardware translation support, where programmers have to call software <code>oid_check</code> function to check permissions. Described in Section 4.1.4.
HwT+SPOT	Hardware translation support with automatic permission check with SPOT.

Finally, we implement our proposed design with the SPOT. It is called HwT+SPOT, and we compare its performance with the other three designs. The designs are summarized in Table 4.2.

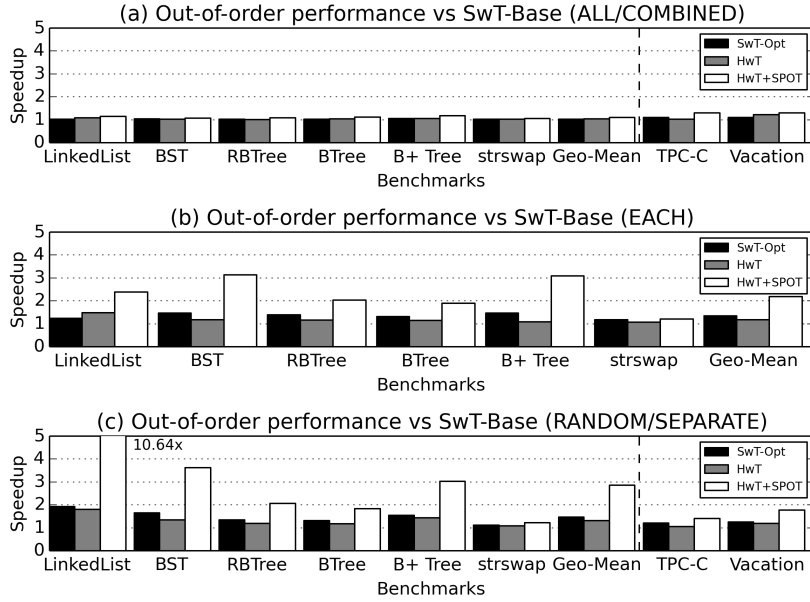
#### 4.4.1.3 Workloads

We evaluate our designs with the same six micro-benchmarks and TPC-C and Vacation application described in Section 3.4.1.1. All the benchmarks allocate their core data structure (B+ Tree in TPC-C and linked list and red black tree in Vacation) in persistent storage with the APIs in Table 3.1 and Table 4.1. For microbenchmarks, we use the transaction support in the APIs, and for the applications we retain their existing failure-safety support. We also retain the pool usage patterns described in Table 3.6. In this work, we assume programmers don't have knowledge of all the relevant pools to open at the beginning of a program. Instead, permission checking is embedded in the traversal code of each data structure, through either software interfaces (SwT-Base and HwT) or translation procedures (SwT-Opt and HwT+SPOT).

#### 4.4.2 Overall Performance

First, we look at the overall performance improvement provided by our proposed architectures. Figure 4.8 shows the results on the out-of-order processor normalized to SwT-Base. We compare the performance of all four designs. We show the results for ALL, EACH and RANDOM patterns on microbenchmarks and COMBINED and SEPARATE patterns on applications.

For the case that only one pool is used in the system (ALL/COMBINED), Figure 4.8(a) shows a speedup for HwT+SPOT over SwT-Base of 1.1x on average for the microbenchmarks, 1.3x on TPC-C, and 1.3x on vacation. Overall, no design is significantly better or worse because only a single pool needs to be translated and have its permission checked.



**Figure 4.8** Speedup of SwT-Opt, HwT and HwT+SPOT over SwT-Base for each usage pattern on an out-of-order processor. (a) Shows the results of microbenchmarks with ALL and applications with COMBINED. (b) Shows only microbenchmarks with EACH. (c) Shows microbenchmarks with RANDOM and applications with SEPARATE.

EACH, shown in Figure 4.8(b), puts every node in the data structure in separate pools, which results in hundreds to thousands of pools used in each program. This case shows a clear advantage for HwT+SPOT, since it is able to accelerate mapping and permissions checking operations. HwT+SPOT shows a speedup of 2.2x on average for the microbenchmarks and 1.4x and 1.8x on TPC-C and vacation, respectively. Interestingly, SwT-Opt is slightly better than HwT (1.3x vs 1.1x speedup on average of microbenchmarks). SwT-Opt has significant overhead when performing translations, and that is not present for HwT. However, HwT has overhead performing permissions checks that negate much of the benefits of hardware translation, since a permissions check is triggered for each node visited. These checks are costly, involving a trap to the OS.

The RANDOM/SEPARATE pattern is shown in Figure 4.8(c). For HwT+SPOT, RANDOM/SEPARATE has the biggest performance speedup across all the workloads and designs. Since fewer pools need to be mapped<sup>4</sup>, there are more POLB hits, and less time is spent walking the SPOT. HwT+SPOT has 2.9x speedup on average for the microbenchmarks and 1.4x and 1.8x speedup on TPC-C and vacation, respectively.

<sup>4</sup>Most of the pools created in vacation are outside of simulation region, only 42 linked lists are created during the simulation.

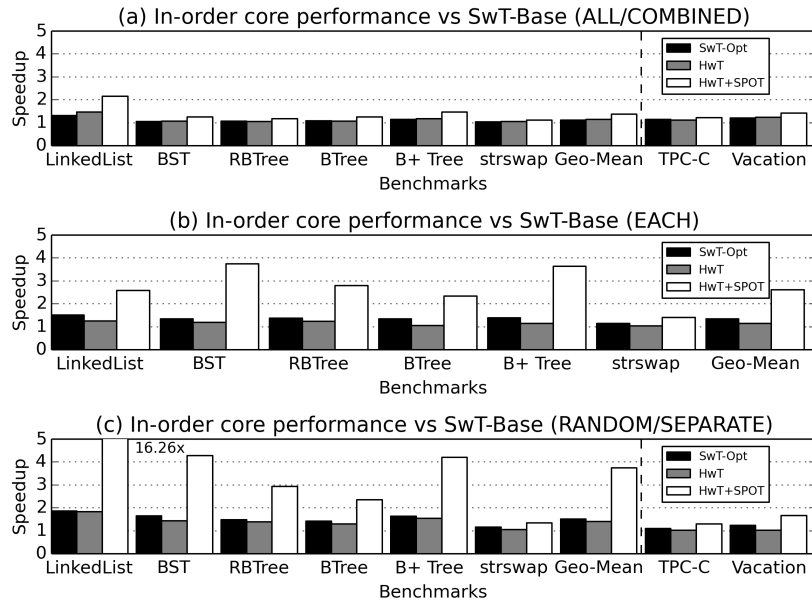


Figure 4.9 Performance of different designs on the in-order processor.

#### 4.4.3 In-order vs Out-of-order

We also show the performance of all designs on an in-order processor in Figure 4.9. The speedup is normalized to SwT-Base on the in-order processor. From the results we see that ALL/COMBINED has similar speedup as an out-of-order processor. EACH and RANDOM/SEPARATE both show larger speedup on the in-order processor than out-of-order processor, especially HwT+SPOT. The permission check performance is more critical to overall performance because the slow software checks can't be hidden by out-of-order execution.

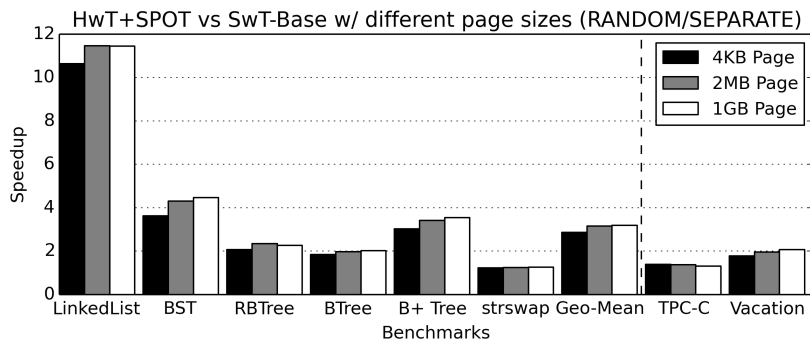
#### 4.4.4 Impact of Page Size

The page size has a significant impact on the performance of HwT+SPOT because the POLB and SPOT store pool information at the page granularity. To understand this effect, we vary the page size among 4KB, 2MB and 1GB for SwT-Base and HwT+SPOT and show the performance in Figure 4.10.

Since larger page size can also benefit the other data in the program, the speedup shown in Figure 4.10 is normalized to SwT-Base running with the same page size. Overall, a 2MB page size has a large improvement on the performance over the 4KB page size because it reduces the POLB miss rate and number of SPOT walks, as shown in Table 4.3. Larger page size can reduce the POLB miss rate and SPOT walk latency because most of the workloads can fit all of their persistent data in one

**Table 4.3** POLB miss rate for different page size with RANDOM/SEPARATE on HwT+SPOT on out-of-order processor

Bench.	4KB Page	2MB Page	1GB Page
LinkedList	0.45%	0.00%	0.00%
BST	3.80%	0.01%	0.01%
RBTree	2.56%	0.00%	0.00%
B-Tree	1.61%	0.00%	0.00%
B+Tree	1.36%	0.00%	0.00%
SPS	1.22%	0.00%	0.00%
TPC-C	3.35%	0.00%	0.00%
Vacation	6.08%	0.17%	0.17%



**Figure 4.10** Performance of HwT+SPOT normalized to SwT-Base on architectures with different page sizes for the RANDOM/SEPARATE pattern.

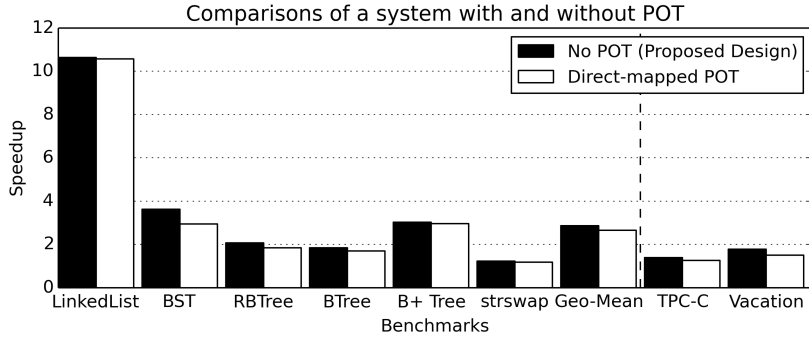
large page leading to fewer misses on the POLB.<sup>5</sup> On the other hand, the performance improvement from 2MB to 1GB is smaller because 2 MBs is enough to hold the data in each pool, and in most cases, the POLB miss rate and number of SPOT walks does not drop any further. Thus we conclude that a 2MB page size is large enough to get most of the benefit from HwT+SPOT for our workloads.

#### 4.4.5 Comparison of HwT+SPOT and native hardware support plus SPOT

In this Chapter, in HwT+SPOT design, we replaced the POT in prior work [Wan17] with the SPOT to hold both the translation and permission. Thus when POLB misses, it performs SPOT walk to validate the permission and also establish translation. There are other design choices, like adding the SPOT to the design in Section 3.2 as a backing store for POT. So that when POLB misses it walks POT first to see if the entry was evicted from POLB before. If not, it then performs SPOT walk to validate permission and establish translations.

A key issue is the size and organization of the POT. We could organize it as multi-level table,

<sup>5</sup>Microbenchmarks use 32 pools. TPC-C use 7 pools. Vacation access 3230 pools in the simulation region.



**Figure 4.11** Comparison of proposed system with a system with a direct-mapped POT and adds SPOT, for 4KB page size and RANDOM/SEPARATE workloads.

like the SPOT. However, we choose instead to use a relatively small, simple linear table design. This will result in collisions, but we can access the SPOT on a miss. As long as the POT is large enough to make misses infrequent, we can avoid replicating information across many POTs, one for each process. Hence, we use a contiguous 2MB memory space (512 4KB pages or one 2MB page) and then the POT can hold  $2MB / 16B = 128K$  entries. We compare such a system adding SPOT to check permissions with HwT+SPOT and show the comparison in Figure 4.11.

It shows that HwT+SPOT out-performs the design with POT in all the workloads. It's because that HwT+SPOT reduces the latency to walk POT when the page needs permission check for the first time. And as shown in Table 4.4, also although SPOT is a multi-level design, it has similar walk latency with POT because of caching. Thus we advocate the design without POT because it reduces the complexity and has better performance.

**Table 4.4** Comparison of SPOT walk performed in HwT+SPOT with system that has a direct-mapped POT. for different page sizes on RANDOM/SEPARATE on out-of-order processor.

Bench.	No POT		Direct-mapped POT			
	SPOT Walks (POLB misses)	Latency (ns)	POT Walks (POLB misses)	Latency (ns)	SPOT Walks (POT misses)	Latency (ns)
LinkedList	3153	19	3153	14	64	203
BST	21701	20	21701	35	127	127
RBTree	19658	20	19658	34	137	126
B-Tree	12626	22	12626	29	113	137
B+Tree	12740	22	12740	22	66	194
SPS	20042	21	20042	44	65	194
TPC-C	109208	9	109208	9	1793	24
Vacation	136038	16	136038	18	4471	211

#### 4.4.6 Storage Overhead of SPOT

The memory size of SPOT is dominated by the last level tables. For the workloads used in our simulation with EACH/SEPARATE usage pattern, the total size of the SPOT used for each benchmark is summarized in Table 4.5. The number is calculated by summing the size of all levels of the SPOT created by a workload. Since the Pool IDs are randomly generated and a workload tends to use a few pages in a pool, the lower-level tables have a similar number as other levels. With larger page size, the number of levels of table in the SPOT are reduced so the storage overhead decreases.

**Table 4.5** The size of tables on all levels of SPOT used in our workloads, for 4KB, 2MB and 1GB page size.

Bench.	4KB Page	2MB Page	1GB Page
Linkedlist	24 MB	24 MB	19 MB
BST	161 MB	161 MB	117 MB
RBTree	100 MB	100 MB	76 MB
btree	52 MB	52 MB	41 MB
bplustree	41 MB	41 MB	33 MB
strswap	37 MB	37 MB	29 MB
TPC-C	312 KB	312 KB	288 KB
Vacation	113 MB	113 MB	85 MB

## 4.5 Summary

ObjectIDs may point to unmapped persistent regions, thereby requiring the programmer to reason about the permissions of objects and whether or not they are mapped. In this Chapter, we remove this burden from the programmer by performing translation and permissions checks in hardware and by removing the requirement that objects be manually opened and mapped before being accessed.

To support it, we add a System Persistent Object Table (SPOT) that holds all known pools in the system, their physical address, and their permissions information. When a program attempts to access an unmapped or unchecked object, a privileged hardware SPOT walk finds the relevant entry, checks its permissions, and copies its entry into the POLB to allow translation and access all without trapping to the operating system. We have implemented our new design in a cycle accurate simulator and compared it with software only approaches and prior work [Wan17]. We find that our design offers a compelling 2.9x speedup on average for microbenchmarks that access pools with the RANDOM pattern and 1.4x and 1.8x speedup on TPC-C and vacation, respectively, using the SEPARATE pattern.

## CHAPTER

# 5

# PERSISTENT DATA RETENTION MODELS

In previous Chapters, we have focused on providing architectural support for programmers to write faster persistent programs with less efforts. In this Chapter, we will shift the steer to focus on development of persistent programs. We will answer the question: *what happens to persistent data in memory when the source code changes?*

Persistent programming with NVMM provides an exciting change to current computer system: programmers can create persistent data structures with a program. So data can be placed in persistent memory on one run of a program and then accessed again on a subsequent run. However, this creates a semantic challenge for programmers with regard to how persistent data is described and manipulated over time. Since persistent data is retained across runs of a program, does a declaration of a type refer to the persistent data created in a previous run of the program or to the organization of the data on the next run of the program? In systems with a conventional memory and storage hierarchy, programmers trust that data types describe what will happen when the program runs next, because there is no data already in memory – memory is repopulated from scratch each run. However, in the context of persistent memory the declarations describe both – what’s already in persistent memory and what will happen on the next run. Section 5.1 will cover more details in comparing the persistent programming with traditional file-based programming.

To begin addressing this problem, we propose that programming environments define a *Persistent Data Retention Model* (PDRM) that explains how persistent data is *retained* as programs are modified and change over time. Programmers need, at a minimum, a clearly specified model of

behavior.

A Persistent Data Retention Model describes what happens to persistent data when code that uses it is modified. In Section 5.2, we identify two models present in prior work but not described as such, the Reset Model and Manual Model, and we propose a new one called the Automatic Model. The Reset Model discards all persistent data when a program changes leading to performance overheads and write amplification. In contrast, if data is to be retained, the Manual Model relies on the programmer to implement code that upgrades data from one version of the program to the next. This reduces overheads but places a larger burden on the programmer.

We propose the Automatic Model to assist a programmer by automating some or all of the conversion. We describe one such automatic approach, Lazily Extendable Data Structures (LEDS), that uses language extensions and compiler support to reduce the effort and complexity associated with updating persistent data. In Section 5.3, we implement all the three models in Persistent Memory Development Kit (PMDK) and in Section 5.5, we evaluate our PDRMs in PMDK using kernels and the TPC-C application. Manual Model shows an overhead of 2.90% to 4.10% on average, and LEDS shows overhead of 0.45% to 10.27% on average, depending on the workload. LEDS reduces the number of writes by 26.36% compared to Manual Model. Furthermore, LEDS significantly reduces the programming complexity by relying on the compiler to migrate persistent data.

## 5.1 Motivation

### 5.1.1 Comparison between persistent programming and file-based programming

Persistent programming not only brings challenges to hardware and architectural designs, but also has significant changes to programming schemes. We identify a new aspect that was not addressed in any prior works: we all know programs can evolve to different versions in the lifetime of development, what happens to the persistent data created by each version? Does a declaration of a type refer to the persistent data created in a previous run of the program or to the organization of the data on the next run of the program? This is a new problem only arise in persistent programming.

Consider an analogy with files on conventional systems without NVMM. If a file format changes, the software is modified to support reading the old file type and converting it as needed into a new format. For data stored serially on secondary storage, this is reasonable since it must be moved from disk to memory on each invocation of the program. But, persistent memory is different in that persistent structures never need to be serialized. However, if the type declarations used in the previous execution change, then the address calculations for the persistent data accessed in the next execution will be invalid or erroneous. Current languages and compilers have no automatic way of detecting the discrepancy or helping the programmer deal with it.<sup>1</sup>

---

<sup>1</sup>There are lot's of software-based solutions here. The programmer could add fields to the data structures, like a magic

Let's compare traditional file-based support for persistent data and new persistent memory programming models with an example. As shown in Figure 5.1(a), with traditional file-based programming models, there are two copies of the data existing in the program: one in volatile memory for use in computation and one in a file. The interpretation of the two copies is written by the programmer. Here, the struct clearly only describes the copy in memory, not storage.

For persistent memory, as shown in Figure 5.1(b), the programmer only needs to define a data structure and denote that it resides in NVMM. The struct itself now describes storage<sup>2</sup>. Since we are considering persistent programming models that provide direct access to programmers, there's only one copy of the data structure and it's persistent.

However, it also introduces a problem when a program changes the struct definition. In the programs in Figure 5.1, suppose the programmer wants to add a new field *int z* to the *struct example*. It's easy for traditional models in (a). They can modify the volatile data structure freely because the struct does not describe the file contents. However, with persistent programming model in (b), the persistent data structure is laid out according to the original struct (two integers), but after the change the program believes the persistent struct has three integers. Thus it will read the wrong data if it continues using the old data still held in persistent memory.

There are many reasonable software approaches to fix this problem. We may want to reset the persistent memory (Reset Model). However, the two integer fields *x* and *y* of the struct might still be useful and should not be discarded. For example, a programmer might only add a field in a well-established data structure. Thus all the other fields in the data structure should still be useful. Furthermore, if discarded, they may simply be recomputed leading to detrimental write amplification and lowering the lifetime of the device. Instead, the Manual Model and the Automatic Model allow the programmer to retain the persistent data but with different complexity and overhead.

Hence, the key issue is this: declarations now may describe persistent data. As a result, modifications to declarations for persistent data need additional attention by the programmer to ensure that the persistent data can evolve as expected when program version evolves. We want to minimize programmer effort while retaining high performance and write endurance.

### 5.1.2 PDRMs in existing persistent programming models

We found no literature that discusses a similar concept as Persistent Data Retention Models. However, there is recognition that such support is needed. Table 5.1 summarizes the support in existing persistent programming models. All the libraries we examined could support Reset Model because they use a file abstraction for persistent data.

---

word, to track versions and manually convert between types on any change to the code. However, this is an onerous and error-prone requirement to put on all persistent data.

<sup>2</sup>Here we assume some persistent programming model like Mnemosyne. Other persistent programming models like NV-Heaps and PMDK can also achieve this with more complicated interfaces.

```

struct example{
    int x;
    int y;
};
struct example array[100];
int main(){
    // Interpreter
    FILE* fp = open("somefile.txt", "r");
    while (fscanf(fp, "%d %d",
                 &array[i].x, &array[i].y) != EOF){
        i++;
    }
    fclose(fp);
    // Some operations
    ...
    // Another interpreter to save data to files
}

```

(a) Traditional file-based programming

```

struct example{
    int x;
    int y;
};
persistent struct example array[100];
int main(){
    //Some operations on array
    ...
}

```

(b) Persistent programming

**Figure 5.1** A comparison of having a persistent data structure using (a) traditional file-based programming models and (b) using persistent programming models.

For NV-Heaps and PMDK, Manual Model can be implemented by manually duplicating the data structure and copying from an old struct layout to a new one. NVM-direct [Ora15] identified the problem of needing to upgrade data structures and even suggests allocating the struct with more memory than needed to support in-place updates. They also define an `upgrade()` function that can perform an in-place upgrade on the struct. However, the function may fail if there's not enough space for the upgrade and will force a reset.

### 5.1.3 PMDK Library

PMDK library is already introduced in Section 2.2.1. The functions we will refer to in this Chapter is summarized in Table 5.2. Also, a coder would modify one or more struct definitions contained within the persistent pool. Thus in our Automatic Model approach (Section 5.3), we will focus only

**Table 5.1** PDRMs in existing persistent programming models that supports user-defined data types.

Programming Models	Reset Model	Manual Model	Automatic Model
Mnemosyne [Vol11]	Yes	Partial; Only for dynamic allocations	No
NV-Heaps [Cob11]	Yes	Yes	No
NVM-direct [Ora15]	Yes	Yes; upgrade() function	No
PMDK [Pme]	Yes; Also, can detect a change of struct layout and resets	Yes	No

on dealing with struct layout changes.

**Table 5.2** PMDK interfaces that are referred to in this Chapter.

Interface	Description
PMEMobjpool* <b>pmemobj_open</b> (const char* path, const char* layout)	Open a pool that associates with a name of path and with the layout of layout.
<b>POBJ_ROOT</b> (PMEMobjpool* pop, type)	Creates or resizes the root object for pool pop and return a typed object of type
void* <b>pmemobj_direct</b> (PMEMoid oid)	Translate oid into the virtual address in the current program of
<b>TOID</b> (type)	A typed PMEMoid that points to an object of type.
<b>D_RO</b> (TOID oid)	A macro defines a pmemobj_direct call for the oid and cast the address into a typed pointer for read
<b>D_RW</b> (TOID oid)	A macro defines a pmemobj_direct call for the oid and cast the address into a typed pointer for write
<b>TX_BEGIN</b> {...} <b>TX_END</b>	A transaction that guarantees a series of operations either all succeed, or non succeed.
<b>TX_ZNEW</b> (type)	Allocates a persistent object of type and zeros the whole object inside a transaction.
<b>TX_FREE</b> (TOID oid)	Frees a typed persistent object oid inside a transaction.

PMDK offers some features to help detect if a pool will be misinterpreted. Each pool can be tagged with a layout name when it's created. If the pool is opened with a layout name that's different than the one used in creation, the library will generate an error and abort the program.

## 5.2 Persistent Data Retention Models

As discussed in Section 5.1, persistent data is tied to their program in persistent programming. In this section, we describe three different models to retain persistent data when a program evolves from one version to another. We only briefly describe Reset Model and Manual Model because they reflect the state-of-the-art.

### 5.2.1 Reset Model

*Reset Model* mandates that the persistent data region is cleared/reset for a modified program in order to prevent data misinterpretation. The Reset Model is compelling for its simplicity, effectiveness, and soundness. It's detrimental due to write amplification from recomputing data and rebuilding data structures. Write amplification is most severe when the working set of a program is much larger than the amount of data affected by the data structure change.

It's easy for programming models to support Reset Model as long as they can easily clear the persistent regions. For example, for all the persistent programming languages in Table 5.1, they use files to hold persistent data. So when a program is modified, the file should be deleted so the persistent data of the program is cleared. Note, this does not happen automatically, even though it is a necessity for some of the systems to function properly.

### 5.2.2 Manual Model

The *Manual Model* requires programmers to transform all the related persistent data to reflect the changes in a newer version of program.

A sufficient requirement to provide the Manual Model is that persistent data is dynamically allocated in a persistent heap. Then programmers can allocate new persistent objects, copy the old ones, and fix-up all of the relevant pointers between persistent objects. The Manual Model may not work for persistent data that is statically mapped by the compiler, like in Mnemosyne, unless an interface is provided for re-linking the data to a new location, which has been proposed in other contexts [HN05] and could be applied here.

The Manual Model can be implemented as a separate conversion program to be executed during re-compilation or as part of the larger program, running only when data layout changes occur. The potential advantage of Manual Model is that programmers have the freedom to either retain or reset data precisely as needed. However, this approach places a large burden on the programmer. The burden of Manual Model makes sense if persistent data is either rarely used or predominantly read-only.

### 5.2.3 Automatic Model

We wish to make retention of data simpler than in the Manual Model, thus we propose *Automatic Model*. Ideally, Automatic Model would fully or partially automate the update process from one version of code to the next, requiring no special intervention. We believe a wide variety of techniques may fall into this model. On a path toward that goal, we propose Lazily Extendable Data Structures (LEDS).

### 5.2.4 Lazily Extendable Data Structures

Using new language support, we only require that programmers provide a high level description of how each structure is extended from its previous definition. The extensions allow the structure to grow to encompass new fields or pointers. The compiler analyzes the description and inserts code that will upgrade each object from the old definition to the new one when it is encountered during a subsequent program execution.

An example of possible C/C++ language extensions are shown in Figure 5.2. The *extendible* keyword identifies the struct as one that may be extended in the future. The *EXTENSION* block can be added once to indicate new fields that are desired. Within the block, any number of new fields may be added, including a nested *EXTENSION* block that holds another set of extensions. The *INIT* block defines how to initialize the new fields from existing fields.

Each *EXTENSION* block represents one change or *upgrade* to the data structure. In our current design, extension blocks can only be added and should never be removed or changed, as this would again lead to a mismatch between the struct and the persistent data in memory.

Based on the extension blocks, the compiler inserts checks into the code to test if an upgrade is needed or not at each access to an object of the same type. How this check is done will be clear after we explain the upgrade process.

If an upgrade is needed, we do not actually allocate a new object and copy over the old one, as in the Manual Model. Instead, for all structs marked with the *extendible* keyword, we embed an extra unused pointer field at the end of the struct. If a programmer adds an extension, we allocate it as a new object in the same persistent pool and store its address in the pointer. This means we do not make a copy, and hopefully do not significantly increase the number of writes. It also means we do not need to fix-up references to the upgraded object held elsewhere in memory. However, accesses to the fields in the extension have to use an additional load through the pointer to access the desired field, which adds overhead.

The presence of this pointer field makes it simple to test if an object has already been upgraded. If the pointer is NULL, it has not yet been updated; otherwise, it has been.

Our approach supports arbitrarily deep nesting. Any new extension field is always given an extra pointer field to point to the next extension.

```

extendible struct A{
    int val;
};

```

First version of struct A.

```

extendible struct A{
    int val;
    EXTENSION{
        double val_dbl;
        INIT(A* obj){
            // initialize new field using old val
            val_dbl = obj->val;
        }
    }
};

```

Second version of struct A.

**Figure 5.2** An example of retention with Automatic Model: struct A adds a floating-point type field to the struct A in the second version, and copy the integer value.

A comparison between Manual Model and Automatic Model is shown in Figure 5.3. Note that Manual Model, in general, would make a copy of the existing data structure. However, in Automatic Model, we extend the existing data by allocating a linked node.

## 5.3 Implementations on PMDK

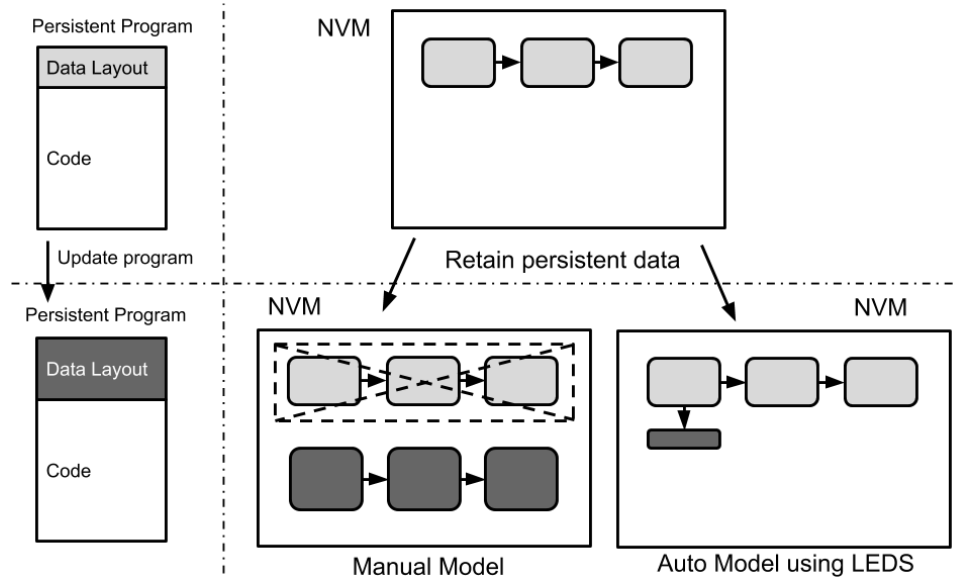
In this section, we discuss the implementations of three PDRMs in libpmemobj in PMDK [Pme].

### 5.3.1 Reset Model

To support the Reset Model, we manually delete all pools used by a program before the next execution. One can always choose to reset a pool used in a program when it's recompiled, just to avoid misinterpretation. An optimization is to detect if the pool will be misinterpreted by the program. The PMDK library provides a way to detect it: each pool is associated with a layout name when it's created. If the pool is opened with a layout name that's different than the one used in creation, the library will generate an error and abort the program. Therefore, it forces the programmer to reset the pool instead of reusing the data.

### 5.3.2 Implementation of Manual Model

PMDK supports the Manual Model. To convert persistent data, the coder must read out the persistent data using the current layout from the pool, transform it into a new layout by copying it into a new



**Figure 5.3** A description of the differences in Manual Model and Automatic Model using LEDS when retaining persistent data in a changed program. The example shows a linked list with three nodes and searching in the updated linked list hits the first node. In Manual Model, a new linked list needs to be created. And in Automatic Model using LEDS, only the node accessed in the new program (first node) is being updated.

pool or a different object in the same pool, just like what we would expect from the file-based approach. Figure 5.4 gives an example of adding field to a persistent linked list using the interfaces from Table 5.2 in PMDK.

This example reveals that retaining only one field in the root object, which is a linked data structure, takes a non-trivial amount of effort from a programmer, and it has proven to be error prone in the context of PMDK. We make some observations from this example. First, some of this code is general enough that it could be put into libraries. For example, the function `root_retain` is for creating a new root object and retaining fields from an old one. Most of the code applies generally to other programs: it needs to allocate a new root object, read out all the fields from the old root and copy to it. Since there's only one root object in a pool, we should use a temporary holder to hold the contents of a new root before actually making it the official root object. The only part that might be different is how to retain each field (e.g. `linkedlist_retain`). Thus if a function is created for each field that needs retention, `root_retain` can automatically call those functions on each field. The other observation is that the compiler/libraries can't fully automate this whole process. It's because `linkedlist_retain` needs to iterate through every `old_LL` object in the old linked list in order to create a new linked list. The iteration might be different depending on the data structure (e.g. graph versus linked list) and hard to automate. Furthermore, it's also complicated to retain data

```

struct old_LL {
    int val;
    TOID(old_LL) next; // TOID: PMEMoid with a specified type, used with
    D_RO and D_RW
};
struct LL {
    int val;
    float val_fl; // Added field in the new version.
    TOID(LL) next;
};
struct old_root_t{
    TOID(old_LL) ll_head;
};
struct new_root_t{
    TOID(LL) ll_head;
};

```

(a) Struct definitions

```

TOID(LL) linkedlist_retain(PMEMobjpool* pop, const TOID(old_LL)& old_head)
{
    TOID(LL) head = TX_ZNEW(LL); // TX_ZNEW: Allocate a new object inside
    a transaction
    D_RW(head)->val = D_RO(old_head)->val; // D_RO and D_RW: translates a
    PMEMoid type into an address, and reads/write from it
    D_RW(head)->val_fl = D_RO(old_head)->val;
    TOID(old_LL) old_p = D_RO(old_head)->next;
    TOID(LL) p = head; // Retain the next node of p
    TX_FREE(old_head); // TX_FREE: Free an object inside a transaction
    while(!TOID_IS_NULL(old_p)){
        D_RW(p)->next = TX_ZNEW(LL);
        // Retain the content from old_p to D_RW(p)->next
        D_RW(D_RW(p)->next)->val = D_RO(old_p)->val;
        D_RW(D_RW(p)->next)->val_fl = D_RO(old_p)->val;
        // Move on to the next node in the linked list and free the old
    node
        TOID(old_LL) to_free = old_p;
        old_p = D_RO(old_p)->next;
        p = D_RO(p)->next;
        TX_FREE(to_free);
    }
    return head;
}

```

(b) Retain the linked list

**Figure 5.4** An example manual retention of a linked-list when adding a floating point field and retaining the value from the integer field, with native PMDK libraries.

```

TOID(new_root_t) root_retain(PMEMobjpool* pop)
{
    TOID(old_root_t) old_root = POBJ_ROOT(pop, old_root_t); // POBJ_ROOT:
    Obtain the root object of a pool
    // Create a temporary holder of the new root
    TOID(new_root_t) new_root_temp = TX_ZNEW(new_root_t);
    // Copy the whole linkedlist pointed by the head field
    D_RW(new_root_temp)->head = linkedlist_retain(pop, D_RO(old_root)->
    head);
    // Allocate a new root and copy from the temp
    TX_FREE(old_root);
    TOID(new_root_t) new_root = POBJ_ROOT(pop, new_root_t);
    TX_MEMCPY(new_root, new_root_temp); // TX_MEMCPY: Perform memcpy()-
    like operations from one PMEMoid to another inside a transaction
    return new_root;
}

```

(c) Retain the root object

```

int main(){
    // The whole program performs retention
    PMEMobjpool* pop = pmemobj_open("poolname", "Layout"); // pmemobj_open
: Reopen a pool with a name that was created before in another program
    TX_BEGIN(pop) { // TX_BEGIN and TX_END: Denotes the beginning and
    ending of a transaction.
        TOID(new_root_t) new_root = root_retain(pop);
    } TX_END;
    return 0;
}

```

(d) Main function

**Figure 5.4** An example manual retention of a linked-list when adding a floating point field and retaining the value from the integer field, with native PMDK libraries.

when there are nested structs. Even if only one struct changes its layout, all the objects related to it should change (e.g. in the example, only *struct LL* changes but it also leads to convert the root object, which contains a field related to *struct LL*). Thus in Manual Model, we need the programmer to manually iterate through the old data structure but there could be some library and compiler support to generate some common code to reduce the amount of programming. We do not explore this possibility further.

### 5.3.3 Implementation of LEDS

We also implement LEDS for the PMDK libraries. We have not yet implemented a front-end for the language extensions, because we wanted to justify that such an effort would be worthwhile. Hence, we focus our implementation on accurately measuring the performance and write amplification of the proposed language extensions. For our implementation, we manually implemented the code using the extensions we described earlier and manually lower them into C/C++ language implementations.

#### 5.3.3.1 Lowering the *extendible* Struct.

An *extendible* struct is supported by splitting its definition into multiple structs, one for the original struct and one for each *EXTENSION* block. A pointer is placed at the end of each struct marked with the *extendible* keyword to point to the next extension. By convention, we refer to this pointer as *to\_extend*, and it will be used to access the extension indirectly. We are essentially building a linked list from the extension fields.

Extensions are enforced by the compiler to be strictly nested, to reflect the sequential order of updates to a program. As shown in Figure 5.5, the original *struct LL* is split into two structs, *LL* and *LL\_EXT*.

#### 5.3.3.2 Lazy updates.

The compiler must generate code that upgrades an object on the first access to one of its extended fields. Note, we could attempt to upgrade on any access, but we make this as lazy as possible to reduce the overhead of benign updates to code, for example, adding a field that is not used.

For each struct marked *extendible*, we traverse the full code and find all references to its fields that are in an *EXTENSION* block. Just before the access (either a load or store), we insert code to check if the corresponding *to\_extend* field used to access it is non-NULL. If non-NULL, the object has already been extended and the code falls-through to the access. For the other case, we add code to allocate the extension and insert the initialization code specified in the *INIT* initializer. Last, the appropriate indirect access using the *to\_extend* pointer is generated.

```

extendible struct LL{
    int val;
    TOID(struct LL) next;
    EXTENSION{
        float val_fl;
        INIT(LL* obj){
            val_fl = obj->val;
        }
    }
};
...
bool search(float v, TOID(LL) head){
    TOID(LL) p = head;
    while (!TOID_IS_NULL(p)){
        if (v == D_RO(p)->val_fl)
            return true;
        p = D_RO(p)->next;
    }
}

```

(a) Written by programmers.

```

struct LL{
    int val;
    PMEMoid to_extend;
};
struct LL_EXT{
    double val;
    PMEMoid to_extend;
    LL_EXT(const LL* obj) {
        val_fl = obj->val;
        to_extend = OID_NULL;
    }
};
...
bool search(float v, TOID(LL) head){
    TOID(LL) p = head;
    while (!TOID_IS_NULL(p)){
        CHECK_EXT(p);
        struct LL_EXT* ext = pmemobj_direct(D_RO(p)->to_extend);
        if (v == ext->val)
            return true;
        p = D_RO(p)->next;
    }
}

```

(b) Generated by the compiler.

**Figure 5.5** An example of retention with Automatic Model using LEDS: adding a floating point value and initializing its value from the integer field of last run.

Since LEDS is a lazy approach, a persistent object might not be accessed after an update is made to its structure definition. In this case, when the programmer accesses fields in the newest extension, the routine also needs to check all the intermediate *to\_extend* fields that lead to the newest extension in sequence, allocating extensions that turn out to be NULL. For any given extension, the extension only needs to be allocated once, whether in the current run or a later run of the program.

The NULL pointer check on *to\_extend* is only needed on the first access of one of the fields it directly points to. However, in general, deciding if it is the first access is difficult. Therefore, by default, we always insert a check before any dereference of an extension field.

Figure 5.5(a) shows the same data structure from Figure 5.4 rewritten for PMDK using LEDS. Now, *linkedlist\_retain* and *root\_retain* are no longer needed.

## 5.3.4 Discussion

### 5.3.4.1 Overheads and Optimizations

LEDS has the potential to incur higher overheads than a manual approach. However, based on our observations, many of these overheads can be reduced through optimization.

#### 5.3.4.1.1 Extra Checks.

Because LEDS lazily updates structures, every run of the program will need to check if extendible objects have been updated before accessing any of their extended fields. Since the compiler may not be able to determine the first access of each object, many redundant checks may be inserted. Furthermore, for nested *EXTENSION* blocks, there will be multiple levels of check if the field is placed in an innermost *EXTENSION* block.

The compiler can reduce redundant checks on extended objects. For example, a redundant check B can be removed if a check A of the same object is placed in a basic block that dominates B's basic block. We do not evaluate this optimization further in this paper.

#### 5.3.4.1.2 ObjectID Translation.

In PMDK, persistent objects are accessed using ObjectIDs. ObjectIDs must be translated to an address before they can be used to access memory. This overhead can be reduced through compiler or hardware support [Che17; Wan17].

The compiler can also reduce the cost of translations by caching them and re-using them. Thus it saves some redundant translations. We evaluate this optimization in Section 5.5.3.

### 5.3.4.1.3 Unnecessary Copies.

Lastly, the compiler can avoid allocating extensions for objects that only exist in volatile memory for a short period of time. The programmer might create a temporary local variable to hold a copy of persistent data. When copying a field to such a local, we may not need to perform a deep copy if the fields are never modified. We can perform a shallow copy instead. Two of our workloads, namely `btree` and `ctree`, benefit from this optimization. We evaluate this optimization in Section 5.5.3.

### 5.3.4.2 Liabilities.

LEDS keeps the original data layout by adding new fields as extensions. This breaks an assumption that programmers and compilers often make: all the data of an object is stored contiguously in the address space. For example, copying structs is a simple `memcpy` operation. However, that will not work for our extendible structs because they will need a deep copy instead. When a compiler detects a copy, it can insert code as necessary to make a deep copy. However, if a programmer manually copies a struct, it could lead to an error, perhaps adding some difficulty for using LEDS.

The extension field also adds some complexity in terms of freeing the extendible object. When `free()` is called on the extendible object, it needs an explicit call to free the extensions as well. The compiler can insert a routine to make sure the extensions of the object are all freed, including the extensions of the extensions. However, again, there may be cases this goes undetected due to pointer casting, allowing some extended objects to leak. However, these dangers are not significantly different than those already present in the C language.

## 5.4 Methodology

The evaluation is performed on a workstation summarized in Table 5.3. We implement the Manual Model and using interfaces in Persistent Memory Development Kit (PMDK) [Pme], formerly known as NVM Library (NVML), developed by Intel. We also implement Lazily Extendable Data Structures (LEDS) as an Automatic Model using PMDK libraries. From now on, we will refer to LEDS as Automatic Model. We run the benchmarks on a hard disk instead of real NVM hardware. PMDK supports hard disk with NVM interfaces and simulates the `clflush` instructions (used for making sure the data is persistent) with an `msync()` call. In our experiment, we use the configuration of `PMEM_IS_PMEM_FORCE=1` provided by PMDK to avoid issuing `msync()` and unnecessarily slowing down the program.

### 5.4.1 Coverage Test

We created several micro-benchmarks to cover situations where data in a pool needs retention. To test the coverage of both Manual Model and LEDS, we create a pool with a root object that contains

**Table 5.3** Summary of the environment for experiments

Component	Configuration
Processor	Intel Core 2 Duo CPU E8400 @ 3.00GHz
CPU Cache	L1D: 32KB, L1I: 32KB, L2: 6MB
Memory	4GB DRAM
Operating System	Linux version 2.6.32 (Red Hat Enterprise Linux Workstation release 6.9)
Hard Disk	110G
Library	Persistent Memory Development Kit (PMDK) Version 1.3.1

different types of fields, as described in Figure 5.6, and 11 different retention situations as shown in Table 5.4. The coverage test is grouped according to four different categories. Both use the interfaces in Manual Model and Automatic Model to evaluate how well the models perform retention. In our experiments, we manually modify all of the programs following the same algorithm we describe for the compiler in Section 5.3.

```
1 struct A {
2     int val;
3     struct X{
4         int val;
5     } x;
6 };
7 struct B {
8     unsigned val;
9 };
10 struct LL {
11     int val;
12     TOID(LL) next;
13 };
14 struct my_root{
15     int counter;
16     struct A a;
17     int data[100];
18     TOID(int) int_ptr;
19     TOID(struct B) b_ptr;
20     TOID(struct LL) head;
21 };
```

**Figure 5.6** The original root object for coverage test.

**Table 5.4** Micro-benchmarks for coverage test

Category	Situation	Actual Changes
Basic field changes to root object	Add a new field to the root object	Add char name[100]
	Change a type of the root object	int data[100] to double data[100]
	Remove a field from root object	Remove int counter
	Change an ObjectID field of base types	TOID(int) int_ptr to TOID(unsigned)
Struct field changes in root object	Add a new field to struct A	Add float fl
	Change a type of a field in struct A	int val to unsigned
	Add a new field to struct X	Add float fl
ObjectID of struct changes in root object	Add a new field to struct B	Add char name
	Change a type of a field in struct B	int val to float
Linked data structure changes in root object	Add a new field to struct LL	Add char name
	Change a type of a field in struct LL	int val to float

### 5.4.2 Workloads

The workloads are shown in Table 5.5. We select multiple implementations of maps (list map, hash map and tree map) from examples/libpmemobj in PMDK as our workloads, and we implement a TPC-C application with PMDK interfaces.

The original programs are modified to use a 32-bit key<sup>3</sup> and are referred as P-ORIGINAL. In order to measure the overhead of different retention situations, we make some changes to the layout (LAYOUT-X). In order to perform retention and preserve the data in the P-ORIGINAL, we write retentions and updated programs for both the Manual Model and the Automatic Model. For the Manual Model, we write a separate program to perform the retention (P-RETAIN), so the updated program P-MANUAL only needs to change the definition in the P-ORIGINAL. With Automatic Model, we integrate the retention in the updated program P-AUTO using our proposed language features and their required transformation.

In our experiments, we make two different data layout changes: LAYOUT-CHANGE is that we change the 32-bit key to 64-bit key and LAYOUT-ADD is that we add a new field to the node, a char array as the name. This measures the impact of different usage scenarios when changing a field: the key field will be used heavily, whereas the newly added field might be used very lightly. We expect the real world usage will be a combination of the two situations.

The experiments measure the overhead of retention in both models: in Manual Model, the overhead of retention is the time spent on the retention program P-RETAIN. So we calculate the

<sup>3</sup>PMDK kernels implements 64-bit key so we change it to 32-bit, in order to retain the keys in the updated program.

**Table 5.5** Description of the workloads from PMDK

Workloads	Descriptions
skiplist	Skip list data structure with 4 levels.
ctree	Crit-bit Tree
btree	B-Tree of Order of 8
rbtree	Red Black Tree
hashmap_tx	Hash map implementations using transaction APIs, with 10 buckets initially
TPC-C	The implementation is based from an implementations that uses volatile memory. We move the B+ Trees (Order of 8) on to the NVM with PMDK interfaces.

overhead as:

$$Overhead_{Manual} = \frac{T_{P-RETAIN}}{T_{P-MANUAL}} \times 100\% \quad (5.1)$$

In Automatic Model, the overhead of retention is the additional runtime needed by Automatic Model over Manual Model. This measure shows what would have been possible if the new struct were contiguous and had no overhead to create. We calculate the overhead as:

$$Overhead_{Auto} = \frac{T_{P-AUTO} - T_{P-MANUAL}}{T_{P-MANUAL}} \times 100\% \quad (5.2)$$

In our experiments, the overhead is calculated by executing the P-RETAIN, P-MANUAL and P-AUTO for five times and calculate the average execution time before feeding into the two equations to calculate overheads.

For the PMDK map kernels, we write a main program that performs all three operations (insertion, deletion and search) on the map randomly: we randomly generate a number of integers and search them in the map. If it's found, we remove the key and value pair from the map. Otherwise, we insert a key-value pair for the integer. The sequence of random numbers is provided as an input to ensure the same behavior across all executions. In our experiments, we use three different types of inputs to evaluate different behaviors, PMDK-INS, PMDK-DEL and PMDK-RAND, which is named after the behavior in the update program with explanations in Table 5.6. These configurations are used to measure the impact of different sequences of operations in the update program, which might result in different overheads for the retention models.

**Table 5.6** Description of different configurations of inputs to the benchmarks in PMDK.

Name	Abbr.	Descriptions
Deletions Only	PMDK-DEL	P-ORIGINAL inserts N unique keys and P-MANUAL/P-AUTO deletes the same N unique keys.
Insertions Only	PMDK-INS	P-ORIGINAL inserts N unique keys and P-MANUAL/P-AUTO inserts another N unique keys.
Combination of Insertions and Deletions	PMDK-RAND	P-ORIGINAL inserts N random keys that may be repeated and P-MANUAL/P-AUTO uses the same input. The update program will have a combination of insertions and deletions.

**Table 5.7** Terms used in the experiments.

Category	Term	Explanation
Different layout changes	LAYOUT-CHANGE	Change the field <i>key</i> from 32-bit to 64-bit.
	LAYOUT-ADD	Add a new field of a char array.
Different programs in the experiments	P-ORIGINAL	The original version of program with 32-bit keys.
	P-RETAIN	The program used to perform retention in Manual Model and executed before the updated program.
	P-MANUAL	The program with updated layout definition in Manual Model.
	P-AUTO	The program with updated layout definition and primitives for retention in LEDS.

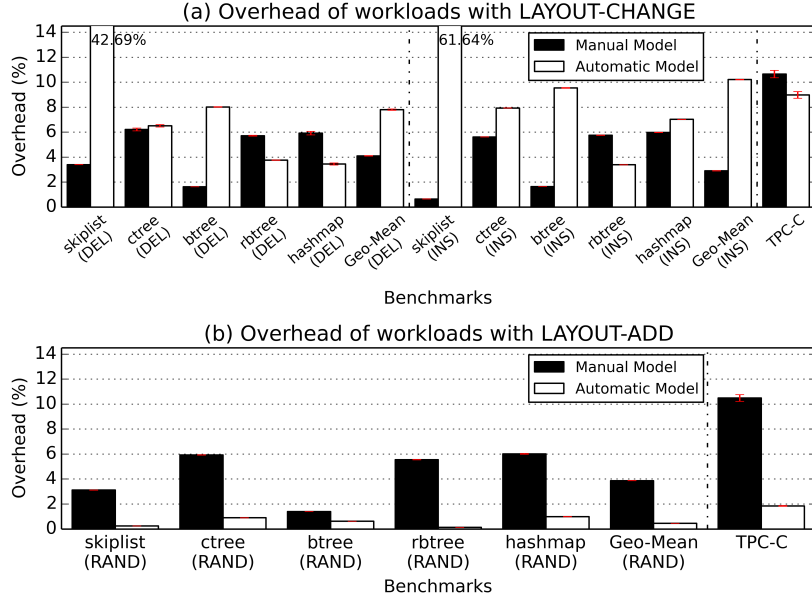
## 5.5 Evaluation

### 5.5.1 Coverage Test

We use Manual Model and Automatic Model to retain data for all the 11 situations with four different programs. It shows that both of the models can retain data properly in these situations. Table 5.8 shows the percentage in the number of code that Manual Model and Automatic Model can save.

**Table 5.8** The lines of code added in order to retaining persistent data for the four categories in Table 5.4, with native PMDK APIs, Manual Model and Automatic Model respectively.

Category	Manual with native APIs	Manual Model (Pct% saved)	Auto Model (Pct% saved)
Root changes	84	70 (16.7%)	15 (82.1%)
Struct base	102	90 (11.8%)	11 (89.2%)
Struct ptr	92	77 (16.3%)	6 (93.5%)
Linked list	106	83 (21.7%)	8 (92.5%)



**Figure 5.7** Overall performance of the overhead of Manual Model and Automatic Model on different workloads. (a) LAYOUT-CHANGE: key changes from 32-bit to 64-bit. (b) LAYOUT-ADD: add a new field to each node. The result of PMDK kernels is measured with 100000 operations in both P-ORIGINAL and updated program, and the result of TPC-C is measures with 10 warehouses and 200000 random client operations.

## 5.5.2 Overhead in Manual Model and Automatic Model

In this section, we present the overhead of the Manual Model and the Automatic Model with two different LAYOUT changes on the workloads mentioned in Table 5.5. The overheads calculated by equations Eq. 5.1 and Eq.5.2 are shown in Figure 5.7.

We analyze the PMDK kernels first. For the Manual Model, both LAYOUT-CHANGE and LAYOUT-ADD need to duplicate the old data structure because the size of the node changes. The execution time of P-RETAIN is similar in these benchmarks but the overhead is slightly different due to different execution times for the P-MANUAL, as a result of a different number of insertions or deletions. For most workloads, an insertion operation and a deletion operation take a similar amount of time, except for skiplist where skiplist(INS) runs much longer than skiplist(DEL) due to larger data size. Hence, the overhead of retention is amortized in that case. For LAYOUT-CHANGE, Manual Model has 4.10% and 2.90% overhead in PMDK-DEL and PMDK-INS kernels respectively, while LAYOUT-ADD has 3.87% overhead in PMDK-RAND kernels. Note, these overheads are similar.

For Automatic Model, the overhead can vary significantly across workloads. First, LAYOUT-CHANGE has a bigger impact on the overhead (7.81%, 10.21% for PMDK-DEL and PMDK-INS) than LAYOUT-ADD (0.45%) because the key is constantly accessed. Hence, it needs checking, redirection, and translation on the field quite often. However, with LAYOUT-ADD, it's a newly added field that is not

otherwise used, so it adds little overhead. In reality, we expect applications will have a combination of changes akin to LAYOUT-CHANGE and LAYOUT-ADD so the performance will be somewhere in-between.

When comparing the PMDK-DEL and PMDK-INS, Automatic Model can have larger overhead in the latter because there are more keys in PMDK-INS and every access to the key can add overhead. PMDK-INS can also have larger overhead because newly allocated nodes carry extra work to build the extensions. We will break down the overhead of Automatic Model model in Section 5.5.3.

When comparing the Manual Model and the Automatic Model across the workloads, some general trends can be observed. The Manual Model has similar overhead to retain data whether it's adding new fields or changing existing fields (as long as the size changes) because it needs to allocate a new data structure and copy from the old one. But, the overhead paid in the Automatic Model is relative to the number of accesses of the changed fields. If the accesses are rare, the Automatic Model can have much smaller overhead than Manual Model.

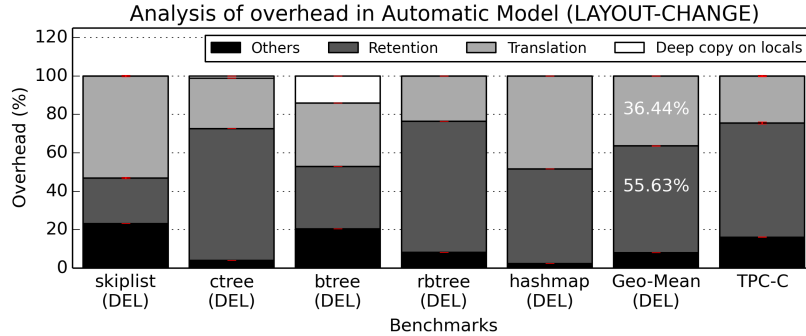
As for the TPC-C application, Manual Model has similar overheads with LAYOUT-CHANGE and LAYOUT-ADD, with 10.65% and 10.49% overheads respectively. The Automatic Model attains a better performance in these two scenarios, with 8.98% in LAYOUT-CHANGE and 1.85% in LAYOUT-ADD.

Furthermore, in TPC-C, the Automatic Model has smaller overhead than the Manual Model, which shows another advantage of the Automatic Model. The Automatic Model only transforms what is needed instead of transforming all the data. Our experiment shows that the second run of the TPC-C after the update only accesses about 67.4% of the nodes from the previous run, so Automatic Model only performs 67.4% of the work that Manual Model does. We further analyze the sensitivity on the ratio of working data set over total data set in Section 5.5.4.

### 5.5.3 Breakdown of overhead in Automatic Model

There are multiple components of the overheads in Automatic Model, as analyzed in Section 5.3.4. We will evaluate three major sources of overhead: the time for allocating extension object and copying fields, the extra redirection and translation cost to access extensions, and the overhead when copying objects with extensions. To measure the time that P-AUTO programs spent on each overhead, we implement multiple programs that remove each component from the P-AUTO program individually and see how much overhead it can save. Figure 5.8 shows the fraction of each component for each workload.

The most significant component is the cost of allocating the extension and initializing it. We see that PMDK-DEL spends on average 55.63% of the overhead for allocation. The second most significant component is the overhead of translation. On average PMDK-DEL spends 36.44% of the overhead for such translations. We are using native PMDK translation functions and we think with faster translation techniques like [Wan17; Che17], this overhead can be reduced. It's also worth



**Figure 5.8** A breakdown of the overheads in Automatic Model. The experiment is done with PMDK-DEL workloads of LAYOUT-CHANGE, with 100000 insertions in P-ORIGINAL and 100000 deletions in P-AUTO.

noting that even with virtual addresses stored in the *to\_extend* field, accessing a field in extension still has an extra level of redirection because it needs extra loads to locate the extension object first. The portion of such overhead is illustrated in the Others part in the Figure 5.8 and is non-trivial.

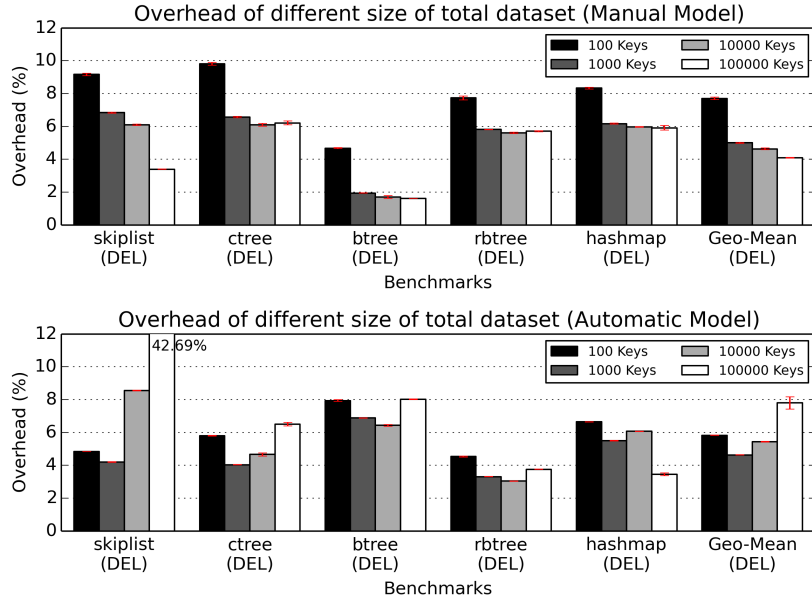
The last component of overhead we analyzed is the cost of making deep copies of objects with extensions. Only two workloads ctree and btree have this situation. In btree, 14.10% of the overhead falls into this category, and we can save the unnecessary allocations in about 51.61% of the copies.

#### 5.5.4 Sensitivity Analysis: Size of data set

Now we consider the impact that the size of the data set has on the overhead. In this experiment, we vary the total amount of data and the total number of operations performed on the data in the same proportion. Figure 5.9 shows the overhead of Manual Model and Automatic Model with multiple data set sizes to retain.

In the case of Manual Model, for sizes above 100 keys, the behavior is fairly uniform. skiplist is an exception because with increasing  $N$  each deletion might take longer to perform due to the larger data structure. This makes the cost of P-RETAIN relatively less than P-ORIGINAL.

For the Automatic Model, the results are also fairly uniform, with a slight worsening for large data sizes. The overhead comes from the time spent on allocation and copying new extension objects and also the accesses to the fields placed in extensions. Thus for larger  $N$ , each deletion will access a larger number of keys that will increase the overhead. The trend is the most significant for skiplist that needs to access at most  $N$  keys for each deletion. Evenso, for some workloads, like rbtree, ctree, and hashmap, Automatic Model performs better than Manual Model.



**Figure 5.9** Sensitivity analysis of the impact of different data set on both (a) Manual Model and (b) Automatic Model. The experiment is done with PMDK-DEL of LAYOUT-CHANGE, with different number of operations of 100, 1000, 10000, 100000, resulting in different size of data set for retention.

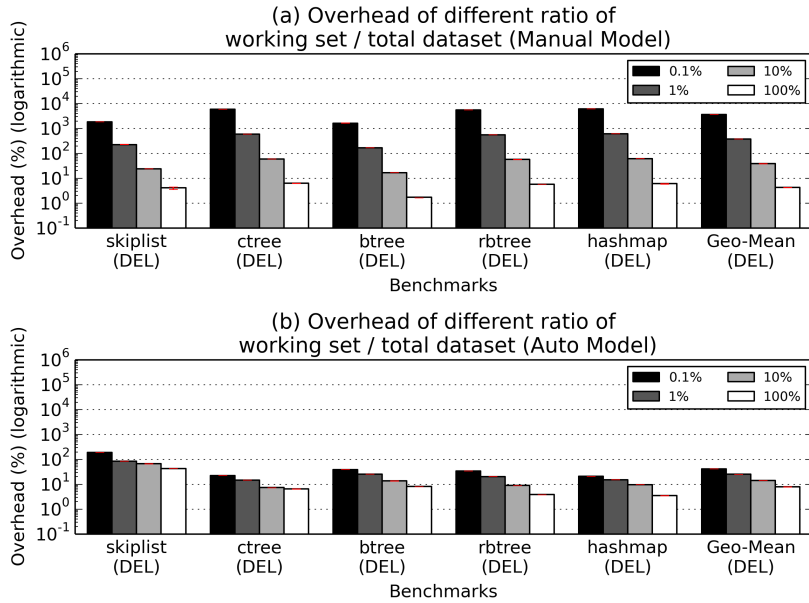
### 5.5.5 Sensitivity Analysis: Ratio of working data set over total data set

The Automatic Model only transforms data that is accessed. For programs with a small working set compared to their total data set, Automatic Model has an advantage. We vary the number of deletions performed in the update program P-MANUAL and P-AUTO and present the overhead of both Manual Model and Automatic Model in Figure 5.10. For small working sets, Automatic Model significantly outperforms Manual Model, even for the 10% case.

### 5.5.6 Write Amplification

NVMM technologies have limited write endurance, hence techniques that increase writes are undesirable. Reducing the number of writes is always prudent. In this section, we estimate the number of bytes written under the Reset Model, Manual Model and Automatic Model when converting old data structure. Table 5.9 shows the number of additional writes incurred by the PDRMs for PMDK-DEL and TPC-C on the LAYOUT-CHANGE workload.

For the Reset Model, the data of original program is discarded after 100000 runs. There's retention of the data. When the program changes, we need to repeat 100000 runs and during each run, we need to repeat the writes to NVM as the original program. Table 5.9 shows an estimation of the number of bytes written when the program is re-executed.



**Figure 5.10** Sensitivity analysis of different ratio of working data set in updated program on both (a) Manual Model and (b) Automatic Model. The experiment is done with PMDK-DEL of LAYOUT-CHANGE, where P-ORIGINAL performs 100000 insertions and update program deletes 0.1%, 1%, 10% and 100% of the keys respectively. The y-axis is on logarithmic scale.

In Manual Model, when we modify the key field from 32-bit to 64-bit, we need to create new objects. Also, this adds extra writes for the fields that do not change since they are copied. Different workloads have different structures to hold the key-value pair.

Table 5.9 shows the definition of each struct that needs retention in the second column. The {uint64\_t and PMEMoid} pair is the key-value pair, and note that we need to retain all the structs that have the key-value struct as its field. The size of each struct is calculated in the second column<sup>4</sup>. All the bytes that are written during the P-RETAIN are calculated in the third column.

Meanwhile, with the Automatic Model, we only need to allocate data for the fields placed in the extension, which in this case is the 64-bit key. We also need the to\_extend field in the extension in order for further extension, resulting in 24 bytes total. We also need to write the extension field of the object (with the allocated ObjectID), which is 16 Bytes. Thus in Automatic Model, each retention needs to write 40 Bytes of data, and the total number of bytes to write is calculated in the last column. From the results in column 3 and 5, we can conclude that Automatic Model writes fewer bytes to NVM than Manual Model in these scenarios. For TPC-C, LEDS reduces the number of writes by 7.2× over Manual Model.

Automatic Model reduces amplification and increases the write endurance of NVM by only

<sup>4</sup>The size of PMEMoid and TOID is 128-bit (16 Bytes), and TOID is a typed PMEMoid type in PMDK

**Table 5.9** Number of writes for different Persistent Data Retention Models after 100000 insertions in P-ORIGINAL.

Workloads	Reset	Manual			Auto	
	Total Bytes (MB)	Struct in retention program	Each mig. (B)	Total Bytes (MB)	Each mig. (B)	Total Bytes (MB)
skiplist	9.9	struct { <b>uint64_t</b> , PMEMoid}, TOID[4]	88	8.4	40	3.8
ctree	6.4	int, struct { <b>uint64_t</b> , PMEMoid} [2]	52	4.9	40	4.3
btree	7.4	int, struct { <b>uint64_t</b> , PMEMoid} [8], TOID[8]	324	6.7	40	9.5
rbtree	8.8	<b>uint64_t</b> , PMEMoid, enum, TOID, TOID [2]	76	7.2	40	3.8
hashmap	4.6	<b>uint64_t</b> , PMEMoid, TOID	40	3.8	40	3.8
TPC-C	94.2	unsigned, <b>uint64_t</b> [8], PMEMoid [8]	196	81.0	40	11.1

updating the data that is accessed instead of copying the whole data structure.

## 5.6 Summary

We identify an important dimension introduced by persistent programs: how to retain persistent data in the presence of data layout changes. A key challenge is giving the programmer a way to reason about such changes. We refer to this as the Persistent Data Retention Model, and we propose LEDS, an example of the Automatic Model. We show that LEDS offers competitive performance and significantly fewer writes than the state-of-the-art for the workloads studied.

## CHAPTER

# 6

## RELATED WORK

### 6.1 Persistent Programming Interfaces

Many works have focused on how to create persistent regions of memory and update them safely with provided language or library support. For example, Mnemosyne [Vol11] proposes adding a new keyword to the C language, namely *persistent*, that allows some data to be placed in persistent memory regions or segments, and it uses software transactional memory to atomically update them. Furthermore, these persistent segments must be mapped to the same part of the virtual address space in all processes that use them. This is problematic, in the general case, since it may be difficult to know in advance all the segments that will be mapped in a program *a priori*. Furthermore, it hinders the use of Address Space Layout Randomization, an affordable and prevalent security mechanism.

NVHeaps [Cob11] provides object level abstractions that allow programmers to construct non-volatile objects by deriving from a predefined base class, called *NVObject*, and then accessing them through special NV-pointers. Persistent Memory Development Kit [Pme] builds on the advances of NVHeaps and Mnemosyne; a programmer translates persistent ObjectIDs to access data. It is widely adopted in the industry and academia and supports existing various systems. It provides interfaces to write persistent programs with transactions and multi-threaded supports. In this dissertation, we propose techniques that are built on a system similar to PMDK.

There are more prior works, like [AH87; Lam91; Sin92; Car94; Atk96; Lis96], that provide persistent

programming to database systems. We believe that while the implementation of our design is based on PMDK, our approach can be applied in a variety of existing persistent programming libraries that provide dynamic persistent memory allocation and atomicity primitives.

Prior capability-based systems [Lev84] and segmented-memory architectures, like the Burrough's 5500 and Intel iAPX 432, were also described as object-based systems, and they supported object-based operations and references. In such systems, all of memory was viewed as an object, and operations on objects were encapsulated and protected. That is not the case in our system. ObjectIDs are only used to reference persistent objects and hardware is added only to support efficient translation of persistent object addresses. All other operations on persistent objects are performed in software without kernel help or encapsulation. However, some of the features of capability-based systems may be worth revisiting in future systems with persistent memory.

### 6.1.1 Address Translation

In Chapter 3, we propose a hardware optimization on PMDK [Pme] and similar programming model. To our knowledge, there are no prior work has considered the overhead of supporting relocatable persistent objects nor proposed hardware or software optimizations to enhance performance of translation from ObjectIDs to virtual addresses. Concurrent with our work and highly related, Chen *et. al* [Che17] describe compiler and language level mechanisms to reduce the overhead of supporting position independence. However, their designs show higher overheads in the presence of many pools and may benefit from hardware translation to reduce that overhead.

### 6.1.2 Permission Checking

In Chapter 4, we provide permission checks, which is paid little attention by prior works. Mnemosyne [Vol11] does not use the concept of relocatable pools so the persistent data can never be shared between programs and their kernel extension can make sure all the data owned by the program is mapped to the address space. NV-heaps [Cob11] are similar in concept to the pool in this paper, and required programmers to open them with names before accessing them, potentially incurring the large overheads we observed. PMDK [Pme] also leaves the responsibility to programmers to open pools in advance. If a programmer fails to open the pool, translation will cause a program failure. The pool open interface performs permission check and denies any illegal requests to open a pool. Recent work [Che17] and Chapter 3 has sought to reduce ObjectID translation overheads in NVML through software and hardware support. Chen *et al.* [Che17] evaluated various low-level software mechanisms while our work in Chapter 3 proposed to treat ObjectID as a persistent address space and provided hardware support for translating ObjectIDs. Both works assume programmers need to open the pools manually to ensure the pool is valid to access.

Manual calls to `pool_open` in these prior works can perform permission checks, but we argue

that it increases the programming burden a lot to call `pool_open` before any dereference to an object in an un-opened pool especially when considering shared pools between programs. Instead, our paper proposes hardware support to automatically perform permission checks while translating ObjectIDs. Programmers do not need to worry about opening pools in advance. At the same time, our design ensures permissions are obeyed.

**Capabilities.** Our notion of permissions checks are similar to concepts championed by capability architectures [Woo14; Lev84]. Capabilities can not only enforce permissions across users but also between objects created by the same program. Capabilities [Woo14], in general, are stronger than the permissions checks provided by our design.

### 6.1.3 Persistent Data Retention Model

We found no literature that discusses a similar concept as Persistent Data Retention Models. However, there is recognition that such support is needed. For example, all the libraries we examined (Mnemosyne [Vol11], NV-Heaps [Cob11], NVM-direct [Ora15] and PMDK [Pme]) can support Reset Model. And some libraries, like NV-Heaps [Cob11], NVM-direct [Ora15] and PMDK [Pme] can provide different levels of Manual Model, while there's no prior work proposing similar concept to Automatic Model. More detailed comparison of the existing library can be found in Section 5.1.2.

## 6.2 NVMM File Systems and Persistent Stores

Prior works including BPFS [Con09], PMFS [Dul14], SCMFS [WR11], Aerie [Vol14], NOVA [XS16] and its variant NOVA-Fortis [Xu17] propose persistent file systems on non-volatile memory (NVM). Persistent file systems provide direct access to NVMM with atomicity support and write-ordering primitives to ensure correctness and they serve as the foundation of existing persistent programming models [Vol11; Cob11; Zho16]. The file-based abstraction is adopted in persistent programming models because the file API provides natural ways to create, delete, resize and rename persistent regions [Rud13].

Our works in this dissertation focus on the problems related with direct access (DAX) to NVMM. Our works can be built on these novel NVMM file system, or any file systems that can provide direct access to NVMM to programmers. In Chapter 3, we focus on providing hardware support for address translation. In Chapter 4, we focus on providing hardware support for pool permission checks. The SPOT design doesn't replace any file system data structure. It merely serves as a permission check and physical address mapping table that can be added to facilitate fast operations on persistent objects in the file system. Chapter 5 discusses more about high-level persistent programming models that was made with the assumption of DAX.

## 6.3 Durability and Failure Safety

Many prior works have pointed out the problem of *durability*, the process of ensuring that modified data has been written back to non-volatile storage, and *failure-safety*, the larger problem of how to update a persistent data structure so that it is always in a consistent state or recoverable to one. To address durability, works like [Pel14; Lu14b; Kol16a] discusses issues of ordering the timing of "persist", an operation of when NVMM writes becomes durable. Other works like [Jos15b; Shi17a] thrives to break persist barriers in order to make it faster or transparent from programmers. To address failure-safety, many works propose better logging schemes, either write-ahead undo-logging or redo-logging, or provide transactional semantics when persisting data. Works like [Shi17b; Jos17] provides hardware support for logging. Works like [Kol16c; Liu17] focus on providing more efficient transactions. And other works focus on providing faster failure safety or making it transparent to the programmers [Ren15].

Our works in this dissertation doesn't consider to improve either durability or failure safety of NVMM. However we are targeting real NVMM systems so we need durability and failure safety to guarantee correctness, but our techniques don't rely them to work.

### 6.3.1 Dynamic Software Update (DSU)

Another technique that bears resemblance to PDRM in Chapter 5 is Dynamic Software Updating (DSU) [HN05]. DSU, at its heart, is also about retaining data from one version of software to another. But our work has different goals. Their work focuses on upgrading a running program, like a server that cannot be brought down. However, PDRM focuses on giving programmers a means of reasoning about outcomes when developing programs with persistent data. Another difference is that DSU requires programmers to provide both versions of the program and a dynamic patch in order for compilers to fix the memory during compilation, whereas PDRM doesn't require full source code of previous versions and only needs the layout of the data that changes.

### 6.3.2 Struct Splitting

The LEDS we proposed in Chapter 5 bears similarity to structure splitting [HT05; GZ07]. In order to have better cache behavior for structs, the compiler can split a struct into two smaller ones with fields that are often accessed together. Sometimes, it needs to add a pointer field in one sub-struct in order to find another sub-struct. The LEDS can be viewed as a split struct with retained fields placed in one sub-struct and new fields placed in another. However, the goal of LEDS is to extend the struct, which adds overhead instead of optimizing it.

## CHAPTER

# 7

# CONCLUSION

Emerging NVM technology is still at an early stage, but it has potential to make significant changes to the future design of computer systems. Deployment of NVM technology creates many interesting research challenges. In this dissertation, we mainly focus on addressing issues when providing direct access (DAX) to programmers and using NVM as part of the main memory. We hope the discoveries in this dissertation can help make NVM more programmable, so that in the future, NVM can be widely used as a center-piece in future computer systems.

The first two Chapters, Chapter 3 and Chapter 4 strives to reduce the overhead of using DAX of NVM and remove programming burden from programmers. In Chapter 3, we address the problem of how programmers can efficiently manipulate persistent data. With hardware-supported address translation, we can treat ObjectIDs as a new persistent memory address space and a program can use load and store instructions to directly access persistent data using ObjectIDs, and these new instructions can reduce the programming complexity of this system. The results show that the *Pipelined* implementation has an average speedup of 1.58× on an out-of-order processor, respectively, over the baseline system. In Chapter 4, we address another problem of how to ensure correctness when accessing data across different persistent regions. The permission check of a pool, in prior works, was performed at the time of pool open, just as what we do with memory-mapped files. We envision in the future, the file abstraction of pool might be broken. ObjectIDs provides an opportunities to remove the programming burden since all access to persistent data needs to go through ObjectID, where the pool ID is contained within. We propose System Persistent Object

Table (SPOT) to organize pools created in the whole system and perform automatic permission check along with hardware-supported translations. The evaluation shows it offers a compelling  $3.3\times$  speedup on average for the microbenchmarks we studied for RANDOM pattern, when 32 pools are used in the workload.

The last Chapter, Chapter 5, addresses another aspect that is also important to make persistent programming universal. When persistent programs become widely adopted, they can be developed and evolved as any program nowadays in software engineering. We need to answer a simple yet important question before using it: what happens to persistent data in memory when the source code changes. We propose Persistent Data Retention Model (PDRM) that explains how persistent data is *retained* as programs are modified and change over time. Discarding the persistent data when program changes (Reset Model) could be what people should assume without any supports. Manual Model is what we do with file-based persistent programming nowadays but it introduces more programming burden in persistent programming with NVMM. Automatic Model can reduce the burden from programmers but requires language and compile supports and may increase overhead. We propose Lazily Extendable Data Structures (LEDS) as a type of Automatic Model, with simple extensions to C/C++ language and shows it introduce an overhead of 0.45% to 10.27% on average depending on the workloads.

Lastly, let's imagine the implications of this dissertation. Consider a future system with all the designs in this dissertation included. Programmers would be able to write persistent programs that read and write persistent data in a pool, as if they were transient variables thanks to hardware-supported address translations, discussed in Chapter 3. They would not need to worry about checking permissions on the persistent data thanks to automatic permission checks, discussed in Chapter 4, as long as they have permissions to access the pool. And when they need to modify the program or its data structures, either fixing a bug or implementing new features, they can easily define extensions for the modified structures and retain the existing persistent results from previous runs (thanks to LEDS in Chapter 5).

## 7.1 Future Directions

There are several interesting research directions this dissertation left unexplored. Chapter 3 discusses the *Parallel* design, which saves an extra cycle but it proves to be tricky to implement on out-of-order processors. Future works could provide speculation to address this issue. Chapter 4 introduces SPOT to provide permission check but we narrow the discussion down to one SPOT access at the same time. Concurrency needs to be supported on modern systems. We have to solve the problem when multiple processes/users try to access SPOT at the same time. Meanwhile, current systems do not allow owners to change the permission of a pool once it's created. What if it is allowed? Then there's possible concurrent *writes* to the SPOT, too. How to ensure the ordering of reads and writes to the

SPOT? And we also need to define the behaviors of the system when the owner changes permission. Do we need to revoke the access of other processes immediately? What happens if a process is in the middle of a transaction, or not in a transaction? How to update translations in the POLB and POT? These questions need to be investigated.

Chapter 5 provides concepts of Persistent Data Retention Model and implementations of Manual Model and Automatic Model (LEDS). There are many other possible implementations, especially with Automatic Model. Further efforts are needed to ease the process of converting persistent data. Meanwhile, future works can seek more compiler-based optimizations. For example, a field place in an extension structure may require multiple indirections to access, so compilers can reorder the struct by moving hot fields out of the extensions with some profiling data. Also, future work can address the problem of deleting a field with some better solutions.

Overall, there are many interesting topics on NVM programming to investigate in the future. It's an exciting area that needs full-stack solutions, from programming language to architectures. Hardware and software co-design will lead to better solutions. Also, nowadays, one must be an expert to write a persistent program, with its complex and error-prone interfaces. Programming interfaces need to evolve beyond the PMDK-like C-based interfaces. Efforts are needed to make it easier for everyday programmers to write persistent programs. Future programming interfaces might not only provide interfaces in an existing language, because all existing languages assume volatile memory. Maybe a complete redesign can lead to a cleaner and simpler language that suits persistent programming well. A new language, or some major changes can lead to compiler research to provide language support and optimization as well as architecture supports.

This dissertation explores multiple problems at different layers of the stack design to improve persistent programming. We hope it will inspire more research in this area, and eventually lead to a future computer system where persistent memory is naturally deployed and easier to program than today.

## BIBLIOGRAPHY

- [AS10] Akinaga, H. & Shima, H. “Resistive random access memory (ReRAM) based on metal oxides”. *IEEE, Vol. 98, Issue: 12*. 2010.
- [AH87] Andrews, T. & Harris, C. “Combining Language and Database Advances in an Object-oriented Development Environment”. *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*. OOPSLA '87. Orlando, Florida, USA, 1987, pp. 430–440.
- [Atk96] Atkinson, M. P. et al. “An Orthogonally Persistent Java”. *SIGMOD Rec.* **25.4** (1996), pp. 68–75.
- [Bed04] Bedeschi, F et al. “An 8Mb demonstrator for high-density 1.8 V phase-change memories”. *VLSI Circuits, 2004. Digest of Technical Papers. 2004 Symposium on*. IEEE. 2004, pp. 442–445.
- [Bha03] Bhatkar, S. et al. “Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits”. *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*. SSYM'03. Washington, DC: USENIX Association, 2003, pp. 8–8.
- [Car94] Carey, M. J. et al. “Shoring Up Persistent Applications”. *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*. SIGMOD '94. Minneapolis, Minnesota, USA: ACM, 1994, pp. 383–394.
- [Car14] Carlson, T. E. et al. “An Evaluation of High-Level Mechanistic Core Models”. *ACM Transactions on Architecture and Code Optimization (TACO)* (2014).
- [Che17] Chen, G. et al. “Efficient Support of Position Independence on Non-Volatile Memory”. *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM. 2017.
- [Cob11] Coburn, J. et al. “NV-heaps: making persistent objects fast and safe with next-generation, non-volatile memories”. *International conference on Architectural support for programming languages and operating systems*. 2011.
- [Con09] Condit, J. et al. “Better I/O through byte-addressable, persistent memory”. *ACM Symposium on Operating Systems Principles*. 2009.
- [Dul14] Dulloor, S. R. et al. “System Software for Persistent Memory”. *Proceedings of the Ninth European Conference on Computer Systems*. EuroSys '14. Amsterdam, The Netherlands: ACM, 2014, 15:1–15:15.
- [FW08] Freitas, R. F. & Wilcke, W. W. “Storage-class memory: The next storage system technology”. *IBM Journal of Research and Development* **52.4.5** (2008), pp. 439–447.

- [Gan05] Gandhi, A. et al. “Scalable Load and Store Processing in Latency Tolerant Processors”. *International symposium on Computer architecture*. 2005.
- [GZ07] Golovanevsky, O. & Zaks, A. “Struct-reorg: current status and future perspectives”. *Proceedings of the GCC Developers’ Summit*. 2007, pp. 47–56.
- [HT05] Hagog, M. & Tice, C. “Cache aware data layout reorganization optimization in gcc”. *Proceedings of the GCC Developers’ Summit*. 2005, pp. 69–92.
- [HN05] Hicks, M. & Nettles, S. “Dynamic Software Updating”. *ACM Trans. Program. Lang. Syst.* **27.6** (2005), pp. 1049–1096.
- [Ins17] InstLatX64. *x86, x64 Instruction Latency, Memory Latency and CPUID dumps*. 2017.
- [Int16] Intel. *Intel 64 and IA-32 Architectures Developer’s Manual: Vol. 3A*. Intel, 2016.
- [IM15] Intel & Micron. *Intel and Micron Produce Breakthrough Memory Technology*. 2015.
- [Int] *Intel 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A: System Programming Guide, Part 1*. Intel.
- [Jos17] Joshi, A. et al. “ATOM: Atomic Durability in Non-volatile Memory through Hardware Logging”. *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2017, pp. 361–372.
- [Jos15a] Joshi, A. et al. “Efficient persist barriers for multicores”. *International Symposium on Microarchitecture*. 2015.
- [Jos15b] Joshi, A. et al. “Efficient Persist Barriers for Multicores”. *Proceedings of the 48th International Symposium on Microarchitecture*. MICRO-48. Waikiki, Hawaii: ACM, 2015, pp. 660–671.
- [Kaw07] Kawahara, T. et al. “2Mb Spin-Transfer Torque RAM (SPRAM) with Bit-by-Bit Bidirectional Current Write and Parallelizing-Direction Current Read”. *IEEE International Solid-State Circuits Conference. Digest of Technical Papers*. 2007.
- [Kol16a] Kolli, A. et al. “Delegated persist ordering”. *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press. 2016, p. 58.
- [Kol16b] Kolli, A. et al. “High-Performance Transactions for Persistent Memories”. *International conference on Architectural support for programming languages and operating systems*. 2016.
- [Kol16c] Kolli, A. et al. “High-performance transactions for persistent memories”. *ACM SIGOPS Operating Systems Review* **50.2** (2016), pp. 399–411.
- [KK09] Kryder, M. H. & Kim, C. S. “After Hard Drives—What Comes Next?” 2009, pp. 3406–3413.

- [Kul13] Kultursay, E. et al. “Evaluating STT-RAM as an energy-efficient main memory alternative”. *IEEE International Symposium on Performance Analysis of Systems and Software*. 2013.
- [Lam10] Lam, C. H. “Storage class memory”. *Solid-State and Integrated Circuit Technology (IC-SICT), 2010 10th IEEE International Conference on*. IEEE. 2010, pp. 1080–1083.
- [Lam91] Lamb, C. et al. “The ObjectStore Database System”. *Commun. ACM* **34**.10 (1991), pp. 50–63.
- [Lee10] Lee, B. C. “Phase change technology and the future of main memory”. *IEEE Micro, Vol. 30, Issue: 1*. 2010.
- [Lev84] Levy, H. M. *Capability-Based Computer Systems*. Newton, MA, USA: Butterworth-Heinemann, 1984.
- [Lis96] Liskov, B. et al. “Safe and Efficient Sharing of Persistent Objects in Thor”. *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*. SIGMOD '96. Montreal, Quebec, Canada: ACM, 1996, pp. 318–329.
- [Liu17] Liu, M. et al. “DudeTM: Building Durable Transactions with Decoupling for Persistent Memory”. *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '17. Xi'an, China: ACM, 2017, pp. 329–343.
- [Lu14a] Lu, Y. et al. “Loose-Ordering Consistency for persistent memory”. *Computer Design, 2014 32nd IEEE International Conference on (ICCD'14)*. 2014.
- [Lu14b] Lu, Y. et al. “Loose-ordering consistency for persistent memory”. *Computer Design (ICCD), 2014 32nd IEEE International Conference on*. IEEE. 2014, pp. 216–223.
- [Lib] *Man page for the libpmemobj library in Persistent Memory Development Kit (PMDK) in Linux, <http://pmem.io/pmdk/manpages/linux/master/libpmemobj/libpmemobj.7.html>*.
- [Mar17] Marathe, V. J. et al. “Persistent Memcached: Bringing Legacy Code to Byte-Addressable Persistent Memory”. *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*. Santa Clara, CA: USENIX Association, 2017.
- [Min08] Minh, C. C. et al. “STAMP: Stanford transactional applications for multi-processing”. *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*. IEEE. 2008, pp. 35–46.
- [Mor13] Moraru, I. et al. “Consistent, durable, and safe memory management for byte-addressable non volatile main memory”. *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems (TRIOS'13)*. 2013.

- [Nal17] Nalli, S. et al. “An Analysis of Persistent Memory Use with WHISPER”. *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’17. Xi’an, China: ACM, 2017, pp. 135–148.
- [Ora15] Oracle. *NVM Direct Library*. <http://www.oracle.com/technetwork/oracle-labs/open-nvm-download-2440119.html>. 2015.
- [PLS05] P. Leach, M. M. & Salz, R. *A Universally Unique Identifier (UUID) URN Namespace*. <https://tools.ietf.org/html/rfc4122>. 2005.
- [Pel14] Pelley, S. et al. “Memory persistency”. *International symposium on Computer architecture*. 2014.
- [Pme] *Persistent Memory Programming*. <http://pmem.io>. NVM Library Team at Intel, 2016.
- [Raj14] Rajachandrasekar, R. et al. “MIC-Check: A distributed check pointing framework for the Intel many integrated cores architecture”. *International symposium on High-performance parallel and distributed computing*. 2014.
- [Ren15] Ren, J. et al. “ThyNVM: Enabling software-transparent crash consistency in persistent memory systems”. *Proceedings of the 48th International Symposium on Microarchitecture*. ACM. 2015, pp. 672–685.
- [Rot05] Roth, A. “Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization”. *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*. ISCA ’05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 458–468.
- [Rud13] Rudoff, A. “Programming models for emerging non-volatile memory technologies”. *login*: **38.3** (2013).
- [Shi17a] Shin, S. et al. “Hiding the long latency of persist barriers using speculative execution”. *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*. IEEE. 2017, pp. 175–186.
- [Shi17b] Shin, S. et al. “Proteus: A flexible and fast software supported hardware logging approach for NVM”. *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM. 2017, pp. 178–190.
- [Sin92] Singhal, V. et al. “Texas: Good, Fast, Cheap Persistence for C++”. *Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications (Addendum)*. OOPSLA ’92. Vancouver, British Columbia, Canada: ACM, 1992, pp. 145–147.
- [Str08] Strukov, D. B. et al. “The missing memristor found”. *nature* **453**.7191 (2008), pp. 80–83.
- [Swal17] Swanson, S. *A Vision of Persistence*. <https://www.sigarch.org/a-vision-of-persistence/>. 2017.

- [Tpc] *TPC Benchmark C*. Transaction Processing Performance Council (TPC), 2010.
- [Vil10] Villa, C. et al. “A 45nm 1Gb 1.8 V phase-change memory”. *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*. IEEE. 2010, pp. 270–271.
- [Vol11] Volos, H. et al. “Mnemosyne: Lightweight Persistent Memory”. *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVI. Newport Beach, California, USA: ACM, 2011, pp. 91–104.
- [Vol14] Volos, H. et al. “Aerie: Flexible file-system interfaces to storage-class memory”. *Proceedings of the Ninth European Conference on Computer Systems*. ACM. 2014, p. 14.
- [Wan15] Wang, C. et al. “How to Be Consistent with Persistent Memory? An Evaluation Approach”. *IEEE International Conference on Networking, Architecture and Storage (NAS’15)*. 2015.
- [Wan17] Wang, T. et al. “Hardware Supported Persistent Object Address Translation”. *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM. 2017.
- [Wan18] Wang, T. et al. “Hardware Supported Permission Checks on Persistent Objects for Performance and Programmability”. *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2018.
- [Woo14] Woodruff, J. et al. “The CHERI Capability Model: Revisiting RISC in an Age of Risk”. *Proceeding of the 41st Annual International Symposium on Computer Architecture*. ISCA ’14. Minneapolis, Minnesota, USA: IEEE Press, 2014, pp. 457–468.
- [WR11] Wu, X. & Reddy, A. “SCMFS: a file system for storage class memory”. *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM. 2011, p. 39.
- [XS16] Xu, J. & Swanson, S. “NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories.” *FAST*. 2016, pp. 323–338.
- [Xu17] Xu, J. et al. “NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System”. *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP ’17. Shanghai, China: ACM, 2017, pp. 478–496.
- [Zha13] Zhao, J. et al. “Kiln: Closing the performance gap between systems with and without persistence support”. *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM. 2013, pp. 421–432.
- [Zho16] Zhou, J. et al. “NVHT: An Efficient Key-Value Storage Library for Non-Volatile Memory”. *2016 IEEE/ACM 3rd International Conference on Big Data Computing Applications and Technologies (BDCAT)*. 2016, pp. 227–236.