


3-28-2018

Rethinking the I/O Stack for Persistent Memory

Mohammad Ataur Rahman Chowdhury
Florida International University, mchow017@fiu.edu

DOI: 10.25148/etd.FIDC006534

Follow this and additional works at: <https://digitalcommons.fiu.edu/etd>

 Part of the [Computer and Systems Architecture Commons](#), [Data Storage Systems Commons](#), [OS and Networks Commons](#), [Software Engineering Commons](#), and the [Systems Architecture Commons](#)

Recommended Citation

Chowdhury, Mohammad Ataur Rahman, "Rethinking the I/O Stack for Persistent Memory" (2018). *FIU Electronic Theses and Dissertations*. 3572.

<https://digitalcommons.fiu.edu/etd/3572>

This work is brought to you for free and open access by the University Graduate School at FIU Digital Commons. It has been accepted for inclusion in FIU Electronic Theses and Dissertations by an authorized administrator of FIU Digital Commons. For more information, please contact dcc@fiu.edu.

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

RETHINKING THE I/O STACK FOR PERSISTENT MEMORY

A dissertation submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Mohammad Chowdhury

2018

To: Dean John L. Volakis
College of Engineering and Computing

This dissertation, written by Mohammad Chowdhury, and entitled Rethinking the I/O Stack for Persistent Memory, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.

Giri Narasimhan

Jason Liu

Leonardo Bobadilla

Gang Quan

Raju Rangaswami, Major Professor

Date of Defense: March 28, 2018

The dissertation of Mohammad Chowdhury is approved.

Dean John L. Volakis
College of Engineering and Computing

Andrés G. Gil
Vice President for Research and Economic Development
and Dean of the University Graduate School

Florida International University, 2018

© Copyright 2018 by Mohammad Chowdhury

All rights reserved.

DEDICATION

I dedicate this dissertation to Dr. Muhammad Ali Choudhury, who has been the inspiration behind all my academic achievements.

ACKNOWLEDGMENTS

I would like to thank my advisor, Professor Raju Rangaswami for supporting me during these past five years. I am very grateful to him for his advice, all the insightful discussions, and suggestions. I consider myself very lucky for getting him as my PhD supervisor, as his unequivocal support for my research has always boosted my confidence in the periods of frustration.

I thank all my dissertation committee members: Professor Giri Narasimhan, Professor Jason Liu, Professor Leonardo Bobadilla, and Professor Gang Quan for their insights and valuable feedback.

I also thank all my friends in Miami who helped me throughout my stay. I am very much indebted my friend Kishwar Ahmed, who has been my constant companion for the last six years. Thank you!

Last but not the least, I want to thank my family who never lost their faith in me. Special thanks to my wife Tashfia, for being a great friend, partner, and a wonderful mother. You have pulled me up when I was down, made me smile when I was sad, and most importantly you were my constant reminder that there is life other than PhD. Having you and Yusra in my life is a beautiful blessing.

ABSTRACT OF THE DISSERTATION
RETHINKING THE I/O STACK FOR PERSISTENT MEMORY

by

Mohammad Chowdhury

Florida International University, 2018

Miami, Florida

Professor Raju Rangaswami, Major Professor

Modern operating systems have been designed around the hypotheses that (a) memory is both byte-addressable and volatile and (b) storage is block addressable and persistent. The arrival of new Persistent Memory (PM) technologies has made these assumptions obsolete. Despite much of the recent work in this space, the need for consistently sharing PM data across multiple applications remains an urgent, unsolved problem. Furthermore, the availability of simple yet powerful operating system support remains elusive.

In this dissertation, we propose and build The Region System – a high-performance operating system stack for PM that implements usable consistency and persistence for application data. The region system provides support for consistently mapping and sharing data resident in PM across user application address spaces. The region system creates a novel IPI based PMSYNC operation, which ensures atomic persistence of mapped pages across multiple address spaces. This allows applications to consume PM using the well understood and much desired memory like model with an easy-to-use interface. Next, we propose a metadata structure without any redundant metadata to reduce CPU cache flushes. The high-performance design minimizes the expensive PM ordering and durability operations by embracing a minimalistic approach to metadata construction and management.

To strengthen the case for the region system, in this dissertation, we analyze different types of applications to identify their dependence on memory mapped data usage, and

propose user level libraries LIBPM-R and LIBPMEMOBJ-R to support shared persistent containers. The user level libraries along with the region system demonstrate a comprehensive end-to-end software stack for consuming the PM devices.

TABLE OF CONTENTS

CHAPTER	PAGE
1. INTRODUCTION	1
2. PROBLEM STATEMENT	7
2.1 Dissertation Statement	7
2.2 Dissertation Contribution	7
2.3 Dissertation Significance	8
2.3.1 Minimize Cache Flush Requirements	8
2.3.2 Direct and Consistent Access to Mapped Data	9
2.3.3 Simplified and Shared Application Development	10
2.4 Summary	10
3. BACKGROUND	12
3.1 Persistent Memory	12
3.2 Consistency Requirements for PM	13
3.2.1 The Importance of Instruction Ordering	14
3.2.2 Comparison of Cache Flush Instructions on PMEP	16
3.3 Summary	17
4. REGION SYSTEM: ARCHITECTURE AND INTERFACE	18
4.1 Assumptions	18
4.2 Application Requirements from PM Software Stack	19
4.2.1 Persistent Namespaces	19
4.2.2 Mapped Data Consistency	19
4.2.3 Consistent Sharing	20
4.2.4 Simple Memory-like Interface	21
4.2.5 Arbitrary and Unordered PM Allocation	22
4.3 The OS Memory/Storage Stack for PM	23
4.3.1 Elements of the Stack	23
4.3.2 PM Use Cases	24
4.4 The Region System Interface	25
4.4.1 Example	27
4.5 Architecture	27
4.5.1 Persistent Metadata in the Region System	28
4.5.2 Volatile Metadata in the Region System	31
4.6 Summary	32
5. REGION SYSTEM: DESIGN	33
5.1 Region System Operations	33
5.1.1 Persistent Metadata Operations	34
5.1.2 Persistent Data Operation	38

5.1.3	Pmsync	39
5.1.4	Recovery	42
5.2	Summary	43
6.	REGION SYSTEM: IMPLEMENTATION AND EVALUATION	44
6.1	Implementation	44
6.1.1	Kernel Modifications	44
6.1.2	Kernel Interaction for Region System Operations	46
6.1.3	Memory Management	48
6.2	Evaluation	49
6.2.1	Methodology	49
6.3	Microbenchmarks	50
6.3.1	Pmsync Comparison with EXT4-DAX	53
6.3.2	Pmsync Comparison with PMEM.IO	54
6.3.3	Cost of Pmsync	56
6.4	Summary	58
7.	USER LEVEL LIBRARIES	59
7.1	Introduction	59
7.2	Background	61
7.2.1	Contemporary Sharing Mechanisms	61
7.2.2	Transparent Sharing	62
7.2.3	Shared Atomic Durability	63
7.3	Persistent Containers	63
7.3.1	Challenges of Sharing Persistent Containers	64
7.4	LIBPM-R: Fixed Map Shared Containers	65
7.4.1	Architecture	66
7.4.2	Shared Atomic Transactions	67
7.5	LIBMEMOBJ-R: Location Independent Shared Containers	76
7.5.1	Architecture	76
7.6	Analysis of Performance	77
7.7	Summary	80
8.	RELATED WORK	81
8.1	Application usage of PM	81
8.2	Native OS support for PM	83
8.2.1	PM as a block device	83
8.2.2	File systems	83
8.2.3	Memory Mapping	85
8.2.4	Other PM-optimized OS features:	85
8.2.5	PM optimized architectures and data structures	86
8.3	Summary	86

9. CONCLUSIONS AND FUTURE WORK	88
BIBLIOGRAPHY	91
VITA	98

LIST OF FIGURES

FIGURE	PAGE
3.1 PM and DRAM on memory bus.	13
3.2 CPU, caches and memory controller layout.	14
3.3 Comparisons of cache flush instructions.	17
4.1 Applications requirements for PM usage.	20
4.2 A PM augmented memory/storage stack.	22
4.3 Region System Tree.	28
4.4 Region System Root.	29
4.5 Region Root - rnode.	30
5.1 Region Create: Required ordering of updates to the PM metadata.	35
5.2 Copy-on-Write propagation elimination.	42
6.1 Simplified diagram of kernel and region system interaction.	45
6.2 Pmsync Example.	47
6.3 Comparison of region system interface with ext4-dax using microbenchmarks.	51
6.4 Average latency of region system operations relative to EXT4-DAX.	52
6.5 Pmsync comparison with EXT4-DAX msync.	53
6.6 Normalized average libpmem and libpmem-DAX latency with respect to libpmem-region.	54
6.7 Comparison of libpmem, libpmem-dax and libpmem-region latency with respect to number of dirty pages.	55
6.8 Pmsync breakdown.	56
6.9 Possible write orderings for region system and PMEM.IO.	57
7.1 Mapping two persistent containers to Process A's address space.	64
7.2 Mapping two persistent containers to Process B's address space fails.	65
7.3 LIBPM-R Architecture.	66
7.4 LIBPM-R transaction: Container state after a successful open. Pages are write protected.	68

7.5	LIBPM-R transaction: The application tries to update object <i>C</i> which is mapped read-only.	69
7.6	LIBPM-R transaction: The update triggers the fault handler in the region system, which initiates the write by making a copy of the page containing the faulting address.	70
7.7	LIBPM-R transaction: The page is granted write permission, and the write to <i>C</i> goes through.	71
7.8	LIBPM-R transaction: The write to <i>D</i> takes place without any overhead. . .	72
7.9	LIBPM-R transaction: A commit is issued.	73
7.10	LIBPM-R transaction: The page is write protected, and the snapshot pointer now points to current page, and the old snapshot page is deleted.	74
7.11	LIBPMEMOBJ-R Architecture.	77
7.12	LIBPMEMOBJ-R vs LIBPM-R (Inserts).	78
7.13	LIBPMEMOBJ-R vs LIBPM-R (Lookups).	79
7.14	LIBPMEMOBJ-R vs LIBPM-R (Deletes).	79

CHAPTER 1

INTRODUCTION

Memory and storage have been managed as separate entities within operating systems (OS) because of their uniquely different properties. Whereas memory is byte-addressable, volatile, and fast, storage is block addressable, persistent, and slow. The emergence of byte-addressable persistent memory (PM) hardware, such as ReRAM, STT-MRAM, PCM, and 3D-XPoint present a combination of properties of both memory and storage. The current OS software stack, which was not designed to exploit the unique properties of PM, thus requires a rethink.

Persistent memory raises two fundamental challenges for building future systems. First, while the relatively significant software overheads can be reluctantly tolerated today with μ s latency flash devices to accommodate legacy software, entirely new approaches to device access would become inevitable with persistent memory that is three orders of magnitude faster. The latency of PM access affects not just application latency but also system resource consumption. The longer an operation executes it consumes host resources within the application, libraries, and the OS, thereby increasing resource provisioning requirements to address a certain level of application load. Thus, it is critical for PM access and management software to minimize its footprint; making accesses purely memory-oriented and highly efficient will become inevitable. Second, working with persistent memory is significantly different than block storage since it is directly exposed to the CPU. Working with it correctly can introduce significant complexity to development work flow. Radically new approaches for exposing persistent memory to applications, and simplifying the role of the developer would become critical in the near future.

Current OS support for PM involves reusing the abstractions and interfaces of the file or memory subsystems. Conventional file systems expose persistent storage by presenting a file abstraction to applications and using block-oriented access to persist data. For byte-

addressable PM, accessing PM in large blocks slows down both read and write operations significantly, owing to higher data software stack and data transfer overheads [CDC⁺10, CME⁺12] as well as the *read-before-write* requirements [UKRV11, CLK⁺15]. On the other hand, while memory management systems support byte-granularity access via mapping physical addresses, they do not support persistent namespaces nor the notion of consistency or durability of memory updates. What is necessary is a PM-tailored OS software stack that can address the unique needs of applications when using PM and simplifying their development without sacrificing the performance benefits of using PM.

Recent work has tackled the above impedance mismatch by building PM-aware file systems [CNF⁺09, WR11, DKK⁺14, XS16], programming abstractions [CCA⁺11, VTS11, GMC⁺12], PM-optimized block devices [CDC⁺10, CME⁺12], RDMA-based PM file system back-ends [ZYMS15], and user-level PM libraries [pmea]. Some of these address the issue of PM data consistency when being updated by a single application [ZYMS15, XS16, OS16, ZHL⁺16]. However, the issue of consistently sharing PM data across multiple applications, as file systems today allow, remains a visible, unsolved problem. Recent file systems have introduced techniques like epochs, short-circuit-shadow paging [CNF⁺09], atomic-in-place updates [DKK⁺14, XS16], and fine grained metadata journaling [CYW⁺16] to optimize metadata updates. However, these solutions do not minimize the number of metadata updates necessary for a given operation. For instance, appending to or increasing the size of a file may require an addition of a new data block, which could result in multiple updates to the inode including rewriting the file size, initializing data pointers, and free-space bitmap updates. In fact, the existing PM specific solutions all store redundant metadata for ensuring the consistency of data stored in PM, a design that has seamlessly percolated from legacy file systems designed for block storage. The more pieces of metadata a PM managing layer maintains, the greater the burden of *ordering* these updates, which in turn impacts perfor-

mance [CCA⁺11, CNF⁺09, GMC⁺12, VTS11, DKK⁺14, XS16]. The file systems build on the well-established POSIX interface, utilize PM’s byte-addressability, and provide durability guarantees, but they do not address the consistency of updates to file data – mapped and shared between applications. Consequently, applications are required to implement custom mechanisms for ensuring the consistency of their PM-resident data, a difficult task given the nuanced treatment necessary for ordering operations to PM without loss of performance.

Solutions for consuming PM target supporting the vast majority of applications that consume storage using the file `read/write/fsync` interface. This interface is not best suited for PM which can support direct `load/stores`. When using the `mmap` interface to make this possible, most existing solutions does not provide consistency and atomic durability guarantee. The only exception here is *failure-atomic msync* [PKS13] which proposed a journaling based solution to address the issue for conventional block-based file systems. Thus surprisingly, despite being a very important usage model for PM, the topic of mapped data durability remains largely unexplored. Furthermore, how such data would be shared across multiple processes or applications remains unaddressed.

Our thesis is that persistent memory will drive new applications — applications that use it not just for its persistence but also its memory like properties. Such applications would ideally want to map PM space within their address spaces for direct access. Further, they would require arbitrary and unordered allocation and deallocation of PM space similar to how memory is used today. Finally, they would want a simple interface that atomically persists a group of updates to in-memory state. To fill this need, we propose the *region system*, a new PM-specific OS software stack that exposes a persistent namespace with memory-like operations augmented with transactional consistency.

The *region system* is both lightweight and low-overhead; it minimizes the amount of metadata it maintains and eschews redundancy to simplify durability and consistency op-

erations. To support mapping PM space within process address spaces for direct access, the *region system* provides a persistent *msync* operation **pmsync** which provides atomicity and gives full control on mapped data persistence to the applications. The *region system* supports mapping of PM pages within multiple application address space at the same time. To achieve transparent yet consistent sharing across the sharing processes, updates are reflected across all processes upon invocation of *pmsync* by one of the processes. The *region system* uses an inter-processor interrupt (IPI) based solution which ensures that invocation of *pmsync* by any one of the cooperating processes gets immediately reflected within the address spaces of all processes sharing the region. Synchronizing region updates and persistence operations is done by the cooperating processes which share such pages. We believe that these semantics provide the necessary simplicity in sharing data within persistent memory without sacrificing logical functionality.

The *region system* also presents a novel *dual-pointer* mechanism at in how it manages the internal metadata which prevents copy on write amplification throughout the *region system* tree. The *region system* supports the creation and management of *regions* which allows for unordered and arbitrary allocation of PM space at the *page* granularity. This is a necessary requirement for applications to fully benefit from memory like usage of PM. Maintaining consistency of PM requires careful ordering of updates which involves flushing dirty cache lines to the PM adding significant cost to the overall process [DKK⁺14, LDK⁺14a]. As a remedy, the *region system* employs a non-redundant metadata architecture requiring only atomic 8-byte updates to ensure durable PM operation.

The final part of the dissertation explores the new interactions of the applications with the PM devices. Applications find new and interesting ways to store data in PM devices. Customized key-value stores [MSTR15, MGA16, LHZS17, ZSLH16] have been proposed as popular options to optimize PM usage. Though key-value stores provide a simple

interface for data storage it might not be sufficient for applications that employ complex object models to store data. It is possible that forcing the applications to use PM optimized key value stores would force applications to change their traditional work flow as well as data access pattern. Multiple persistent object stores [CCA⁺11, VTS11, pmea] have been proposed to access persistent memory in conventional memory like way. PMEM.IO is the latest development in this space which provides persistent object store (container) support with the LIBPMEMOBJ library. However, these libraries do not allow the containers to be shared across multiple applications as they are limited by their underlying consistency mechanism. Besides that, developers are required to make specific annotations for each of the PM objects and the transactions involving those objects. Specifying each transactions separately to ensure consistency of the operations is a cumbersome task for the application developers. We propose a way to provide strong consistency guarantees for PM specific libraries sans the development complexities at the application level. We show that, applications can achieve immediate data persistence consistently by using the *region system* based persistent container solution LIBPM-R which also facilitates transparent sharing of the persistent containers. LIBPM-R is only restricted by the requirement to map the containers at the same virtual address across all applications. However, we demonstrate that the existing persistent container solutions such as PMEM.IO can be easily modified to use *region system*, and propose LIBPMEMOBJ-R to build shared persistent containers that can be mapped independently across multiple applications.

In this dissertation, we tailor a complete solution for consuming PM devices by different layers of the current software stack. Our contributions and the significance of the dissertation is clearly outlined in Chapter 2. Chapter 3 provides an overview of the persistent memory technology. In Chapter 4, we discuss the requirements from applications point of view for ensuring a seamless and consistent use of PM as storage class memory. We also highlight the limitations of the current I/O stack in getting the best out of the

PM devices. A new OS subsystem - “*the region system*” is introduced along with the interfaces in this chapter. Chapter 5 delves into the details of persistent *region system* operations to elaborate how the region system minimizes cache flush cost by introducing non-redundant metadata structure. We also discuss and demonstrate the *pmsync* interface to achieve atomic durability of shared mapped data in a shared environment. In the following chapter, we provide details on *region system* implementation and provide comparative analysis of *region system* with respect to the contemporary solutions. In Chapter 7, we present LIBPM-R, a persistent memory library to support fixed-map shared persistent containers by making use of the *region system* interface. In addition to that, we also present a LIBPMEMOBJ-R to achieve a map-anywhere shared persistent containers solution. In Chapter 8, we discuss and analyze the contemporary research on PM usage before concluding our dissertation.

CHAPTER 2

PROBLEM STATEMENT

This chapter introduces the proposed research problems and their significance. In addition, this chapter also identifies the major challenges associated with the problems under consideration and outline our unique contributions.

2.1 Dissertation Statement

We propose to build “Region System”, a kernel subsystem, to support persistent memory to achieve the following goals:

- Minimizing unwanted latency in the persistent memory access path by eliminating metadata redundancy.
- Provide users with direct and consistent access to shared persistent memory areas across multiple applications.
- Provide developers with tools to simplify application development by supporting transparently shared persistent containers.

2.2 Dissertation Contribution

This dissertation provides solutions to integrate persistent memory management into the current system software. First, we analyze and establish the requirements of applications using PM. Beside that, we identify the road blocks to achieve full utilization of PM devices. Based on our findings, we designed “*The Region System*” – a kernel subsystem, which guarantees direct and consistent access to PM resident shared data. The “*region system*” uses a non-redundant metadata structure to reduce cache flush cost aimed to lower the cost of achieving consistency. We also reduce COW amplification by introducing a

novel dual-pointer mechanism for the data pages. Finally, we present LIBPM-R, a persistent memory library that supports consistent sharing of persistent containers. LIBPM-R simplifies the development complexity inherent to the contemporary persistent container based solutions. We also present a *region system* augmented version of PMEM.IO, which makes it possible for the applications to map the persistent containers anywhere in the address space.

2.3 Dissertation Significance

The emergence of high-speed and byte-addressable persistent memory devices makes the existing OS abstractions obsolete for PM usage. While the memory subsystem does not provide namespace support, the file system abstraction falls short on providing consistent direct access to PM. Contemporary solutions, as a whole, are built in the mold of existing file systems which eventually means the same storage usage model is extended to PM. In this dissertation, we propose a new kernel subsystem to achieve maximum benefit without forfeiting consistency guarantees. With the support provided by the *region system*, we present two versions of persistent containers that can be shared with different degree of flexibility with strong consistency guarantees.

2.3.1 Minimize Cache Flush Requirements

If multiple inter-dependent objects reside in a system, the updates to those objects require to happen simultaneously to maintain data integrity. With file systems for instance, the file *inode* and free-space bitmap contain redundant information, which need to be atomically written out to maintain consistency. Conventionally, such redundant information is maintained for performance reasons; loading all file inodes to reconstruct the free-space bitmap was considered expensive because of increased I/O requirements and thus the redundant

persistent versions. However, the redundancy also adds complexity to file system design and overhead during runtime. Furthermore, updates to PM requires careful ordering of instructions, and the problems of not doing so are well-described in contemporary PM research literature [CCA⁺11, CNF⁺09, GMC⁺12, VTS11, DKK⁺14, XS16]. Additionally, costly cache line flushes has to be enforced to maintain the order of updates to PM. The “*region system*” does not keep any redundant metadata. The absence of redundant metadata significantly lowers the ordering requirements for PM updates, which eventually lowers the required number of cache line flushes contributing to better performance.

2.3.2 Direct and Consistent Access to Mapped Data

Previous studies have shown that with PM, the current storage stack contributes 97% of the access overhead [CS13] and that direct CPU loads/stores can significantly lower latency and improve the CPU efficiency of applications [Fus]. PM-specific solutions, either as OS optimizations or development of new programming abstractions, fall short on providing comprehensive support for mapped application data management. Existing support provided by the file system and memory interfaces through traditional `mmap-async` interface does not provide any consistency guarantee to the updates to the file backed memory mapped data. This deficiency of consistency guarantee makes the interface even unsuitable for PM. However, not being able to use memory mapping interface forces programmers to use complex transactional mechanism which requires specifying each individual transactions. The *region system*'s `pmmmap-pmsync` interface provides strong consistency guarantee for mapped data which enables applications to make all the changes atomically durable rather than specifying individual transactions. This direct access mechanism to PM mapped areas completely eliminates the software stack overhead and provides the users with an easy-to-use model.

2.3.3 Simplified and Shared Application Development

The emergence of PM devices requires a careful consideration of the existing applications concerning PM usage. Most of the existing applications follow the read-write-fsync model to do smaller writes to the PM. Though, this model works well for the existing slow storage, it does not fit well to the PM. Characteristics of the PM creates new opportunities for the existing applications, hints at new class of applications. This class of applications are the ones requiring persistence of dynamically allocated data. PMEM.IO's LIBPMEM-OBJ, the most recent persistent memory library, has proposed transactional consistency of data structures. However, similar to previous solutions [VTS11, CCA⁺11], the persistent object stores can not be shared by multiple applications and the manipulation of the objects requires specific annotations. Annotating every operation by the developers is cumbersome as well as error prone from the developers perspective. In this regard, we propose a persistent memory library which removes the need for annotation and simplifies development complexity. The LIBPM-R library provides a simplified transactional model allowing atomic durability of transactions in a shared environment. Applications can share persistent object stores (containers) only being restricted by the fact that the containers have to be mapped to the exact same address across the sharing applications. To overcome this limitation, we present a portable persistent container library LIBPMEMOBJ-R, which combines the strict durability guarantee of the *region system* with the indirect persistent pointers PMEMOID's to achieve location independent persistent containers.

2.4 Summary

We have introduced the dissertation statement in this section briefly outlining the significance and the contributions of the dissertation. In the next chapter, we provide a necessary overview of the PM devices, explain the PM integrated system architecture, and point out

the consistency issues arising from the integration. We also present a comparison of different cache flush instructions to identify the best available option.

CHAPTER 3

BACKGROUND

We have stated the significance and contributions of our dissertation in the previous chapter. However, in this dissertation we extensively refer to the new and upcoming persistent memory devices which are not yet introduced to the mainstream systems. In this chapter, introduce persistent memory devices and the characteristics of these devices in comparison with the traditional memory and storage. We also discuss the architecture of a PM integrated system as well as the complexities of achieving consistency for these systems. Finally, we discuss the instructions that help to achieve PM consistency and provide our reasoning for the instruction we choose to use throughout our implementation.

3.1 Persistent Memory

The technology curve for storage is at a tipping point wherein within a years time we will witness a new class of storage devices that are astronomically faster than todays state-of-the-art flash-based storage (as shown in Table 3.1). Such a drastic and unilateral performance improvement for a single computing resource has not been witnessed in the recent past. Simply put, the new disruptive storage technology will challenge all that we know about how systems should be built.

	DRAM	3D Xpoint	NAND
Endurance (P/E Cycles)	10^{15}	10^7	10^3
Read latency	Nanoseconds	10s of Nanoseconds	10 - 100 Microseconds
Normalized read latency	1	10	$10^4 \sim 10^5$

Table 3.1: Comparison of Memory technology

While a range of persistent memory technologies including ReRAM, STT-MRAM, and PCM, have been discussed in the literature, Intel’s 3D-XPoint is the most recently announced, assumed to be the most promising, and expected to be available within the

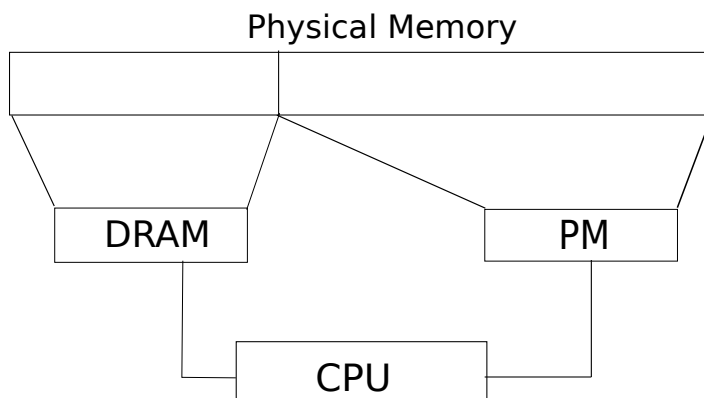


Figure 3.1: PM and DRAM on memory bus.

span of one year. 3D-XPoint promises storage access latencies in the order of tens of nanoseconds for 512-byte accesses. This is three orders of magnitude faster than state-of-the-art flash and immediately brings into focus the overheads contained in the storage stack. Conventional block I/O processing in the OS has shown to add as much as $300\mu s$ of overhead to each operation and even highly optimized software stacks that bypass the OS for device access add as much as $20\mu s$ of overhead.

These devices will be sharing the memory bus alongside conventional DRAM and constitute the total physical memory of a system as shown in Figure 3.1. This byte-addressability combined with persistence presents new challenges to the operating system components as well as application developers to achieve best performance from the new technology. However, persistence at the memory bus also requires additional measures to be taken to achieve consistency.

3.2 Consistency Requirements for PM

Traditionally, being volatile in nature, memory (DRAM) is considered merely a placeholder for intermediate data in the traditional I/O stack. File data are made durable by writing to persistent storage by means of read-write-fsync system calls or to mapped

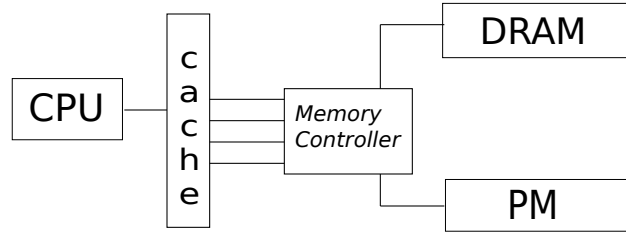


Figure 3.2: CPU, caches and memory controller layout.

data by the `msync` interface. Persistent memory, on the other hand, is directly exposed to the CPU `load/stores`. Persistence at the memory level makes this narrow interface unsuitable as the operating system or applications need to think about intricate details of cache flushes to the memory as well as the ordering of `load/stores` to the memory [LDK⁺14b, Mck05].

3.2.1 The Importance of Instruction Ordering

To understand the implications of persistent memory on the memory bus, we need to delve into details of the interactions among the CPU, the caches, and the memory controller. Figure 3.2 presents typical architecture of a system with PM on the memory bus.

The three places for data to reside in a PM based system are the CPU caches, the memory controller, and the memory (persistent and volatile). When a store instruction is executed the cache contents for the address is modified. Eventually, the cache lines may be evicted to be written to the memory which goes through the memory controller. However, the memory controllers hold their own small storage to optimize the writes to the memory. As a result, the store instructions that are executed by the CPU might not reach memory in the same order of execution. In case of a system or power failure, out of order writes can corrupt the system.

To ensure the ordering of writes a special cache line flush instruction (`CLFLUSH/CLFLUSHOPT/CLWB`) followed by a barrier (`MFENCE/SFENCE`) has to be issued to ensure

a cache line is flushed immediately from the cache. This combination does not influence the contents of the memory controller which requires separate mechanism to flush the contents from there to the memory. Intel has recently deprecated the PCOMMIT instruction which was proposed to clear the contents of the memory controller in favour of ADR. We present some of the CPU instructions that are relevant to our dissertation.

CLFLUSH: Invalidates the cache line that contains the linear address specified with the source operand from all levels of the processor cache hierarchy (data and instruction). The invalidation is broadcast throughout the cache coherence domain. If, at any level of the cache hierarchy, the line is inconsistent with memory (dirty) it is written to memory before invalidation. The source operand is a byte memory location.

CLFLUSHOPT: Similar to CLFLUSH, but with ordering optimizations

CLWB: Writes back to memory the cache line (if modified) that contains the linear address specified with the memory operand from any level of the cache hierarchy in the cache coherence domain. The line may be retained in the cache hierarchy in non-modified state. Retaining the line in the cache hierarchy is a performance optimization (treated as a hint by hardware) to reduce the possibility of cache miss on a subsequent access. Hardware may choose to retain the line at any of the levels in the cache hierarchy, and in some cases, may invalidate the line from the cache hierarchy. The source operand is a byte memory location.

ADR: ADR stands for Asynchronous DRAM Refresh. ADR is a feature supported on Intel chipsets that triggers a hardware interrupt to the memory controller which will flush the write-protected data buffers and place the DRAM in self-refresh. This process is critical during a power loss event or system crash to ensure the data is in a safe state when the NVDIMM takes control of the DRAM to backup to Flash. Note that ADR does not flush the processor cache. In order to do so, an NMI routine would need to be executed prior to ADR.

3.2.2 Comparison of Cache Flush Instructions on PMEP

We wanted to find out which of the cache flush instructions perform best in a real PM augmented environment. However, PM devices are not readily available at present. Fortunately, we had access to a physical machine that closely emulates PM behavior. We deployed and evaluated our system on an instance of PMEP– Persistent Memory Emulator Platform [LDK⁺14b, DKK⁺14], built by Intel. According to the developers, PMEP partitions the available DRAM memory into emulated PM and regular volatile memory, emulates configurable latencies and bandwidth for the PM range, allows configuring pm wbarrier latency (default 100ns), and emulates the optimized cache flush operation. PMEP is implemented on a dual-socket Intel[®] Xeon[®] processor-based platform, using special CPU microcode and custom platform firmware. Each processor runs at 2.6GHz, has 8 cores, and supports up to 4 DDR3 Channels (with up to 2 DIMMs per Channel). The custom BIOS partitions available memory such that channels 2-3 of each processor are hidden from the OS and reserved for emulated PM. Channels 0-1 are used for regular DRAM. NUMA is disabled for PM channels to ensure uniform access latencies. In our machine, PMEP has 32GB DRAM and 32GB PM, for a 1:1 capacity ratio.

CLFLUSH and CLFLUSHOPT instructions invalidate the cache lines while flushing the cache lines, which adds additional penalty to the overall process besides the cost of maintaining ordering of instructions. However, CLWB does not incur that cost resulting in better performance. We have done a simple test where we write 8 bytes to each of 2048 pages (4KB pages) and plotted the average per page latency of flushing those 8 bytes to the PM. CLWB, performs 10X better than the CLFLUSH and CLFLUSHOPT as shown in Figure 3.3. Therefore, in our implementation we decided to use CLWB for better performance.

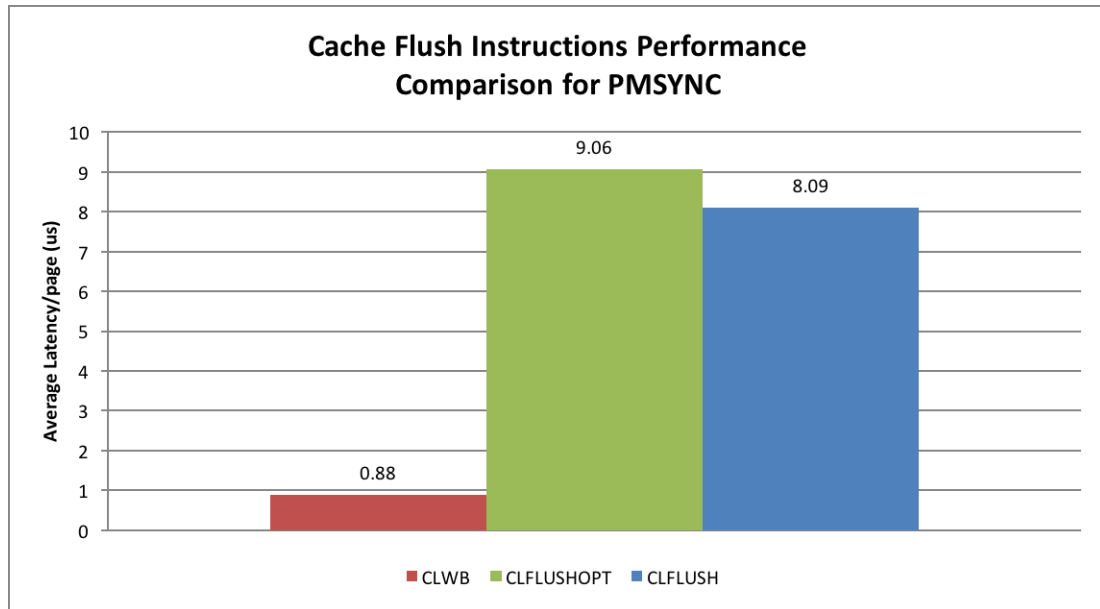


Figure 3.3: Comparisons of cache flush instructions.

3.3 Summary

In this chapter, we introduced the features of PM devices and described the challenges associated with achieving consistency when PM is added to the conventional system. We have introduced the persistent memory emulator platform PMEP and also explained the reasoning behind our selection to use CLWB as the cache flush instruction for implementing the *region system*. In the next chapter, we first analyze the requirements from the system for PM based application developer. Then, we introduce a new kernel subsystem “*the region system*”, layout the interface, and provide the architecture of the system.

CHAPTER 4

REGION SYSTEM: ARCHITECTURE AND INTERFACE

In the previous chapter, we have presented a short overview of PM devices and an introduction on the ordering requirements of the PM integrated systems. In this chapter, we will discuss the architecture and the interface of the *region system* putting emphasis on the design decisions important to minimize cache flush and to support consistent sharing of PM mapped areas.

We first layout the assumptions and application usage of the PM followed by an analysis on the requirements of PM application developers from a PM integrated IO stack. Then we present the *region system* interface that fulfills those requirements, and conclude the chapter with a description of the persistent and volatile metadata architecture of the system.

4.1 Assumptions

Taking note of the current industry developments and predictions about PM devices we make the following assumptions:

1. PM devices will share the same address space with DRAM and they will be added to memory bus as shown in Figure 3.1 to achieve lowest possible latency and direct CPU access. In other words, volatile and persistent memory will co-exist in the I/O stack.
2. The CPU memory access mechanisms in PM systems will remain fundamentally same as they are today.
3. PM will continue to be managed in hardware as pages, and by the OS via page tables with hardware accelerated accesses enabled by a hierarchy of volatile CPU caches.

4. The CPU instruction set may evolve to support PM in the future [VTS11, LDK⁺14a, DKK⁺14].
5. The management of new memories remains a task for the OS along with the resource management and security enforcement of the hardware device.

4.2 Application Requirements from PM Software Stack

The characteristics of PM devices blurs the conventional boundary of storage and memory, and exposes the limitations of the existing solutions in managing PM optimally. We believe that PM access interfaces within the OS should be tailored to expose the unique properties of PM devices so applications can exploit the full potential of this new technology with *ease*. In this section, we justify a minimum set of requirements (as shown in Figure 4.1) that the OS should meet to satisfy both the support of new PM devices and its use by applications.

4.2.1 Persistent Namespaces

As with conventional storage, applications using PM will require the ability to identify previously stored data and distinguish it from unrelated data stored by other applications.

4.2.2 Mapped Data Consistency

Unlike block-based persistent storage of today, PM devices can be accessed directly by the CPU. To utilize this new, powerful capability, it is valuable to expose PM directly within a process' virtual address space. Previous studies have shown that with PM, the current storage stack contributes to 97% of the access overhead [CS13] and that direct CPU loads/stores can significantly lower latency and improve the CPU efficiency of ap-

plications [Fus]. Thus, the conventional memory mapping approach used for volatile DRAM and files becomes very valuable with PM. However, the possibility of corruption increases as the PM can contain uncommitted data after a system failure or crash. Thus, it is necessary to have a mechanism to achieve atomic durability of mapped data and to revert back to a previously application defined consistent state in case of a failure to achieve atomic durability.

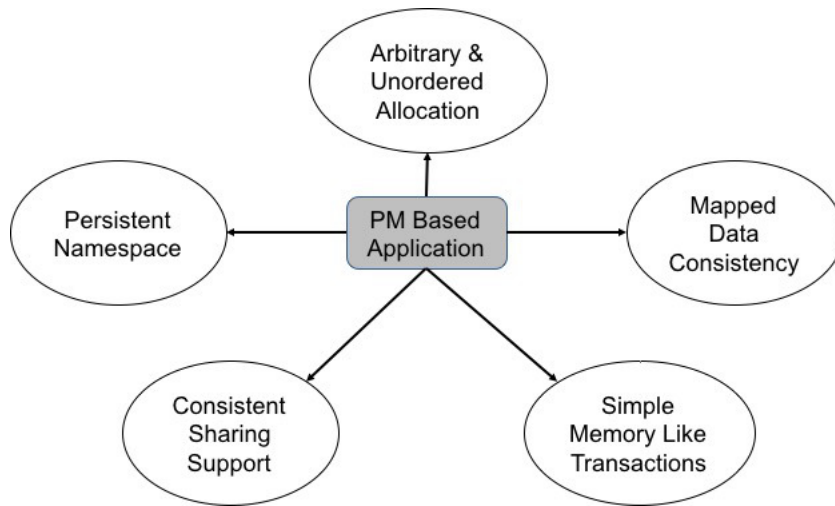


Figure 4.1: Applications requirements for PM usage.

4.2.3 Consistent Sharing

Direct exposure of PM to CPU load/stores provides an unique opportunity to reflect all the changes made to a particular PM location visible to multiple applications simultaneously. The current PM-specific solutions does not support any notion of shared data consistency. The file mapping mechanism either supports private copies (MAP_PRIVATE) of the data, or shared copies (MAP_SHARED) which might not be transparent across the applications at any given time. We posit that the applications which map the same PM area have some motive to do so, and they should be able to transparently make the updates visible to all concerned parties. However, the applications may decide durability points by synchro-

Table 4.1: A summary of recent research on PM-specific software solutions

	Namespace	Mapped Data Consistency	Consistent Sharing	Memory Like Transactions	Arbitrary and Unordered Allocation
File systems [CNF ⁺ 09, WR11, DKK ⁺ 14, ZHL ⁺ 16, OS16]	✓	✗	✗	✗	✗
Memory subsystem	✗	✗	✗	✓	✓
Block devices [CDC ⁺ 10, CME ⁺ 12, CMH14]	✓	✗	✗	✗	✗
Persistent Heaps [CCA ⁺ 11, VTS11, pmea]	✓	✓[transactional]	✗	✗	✓
NOVA [XS16]	✓	✓[private]	✗	✓	✗
Mojim [ZYMS15]	✓	✓[replicated]	✗	✓	✗
Atomic Msync [PKS13, VMP ⁺ 15]	✓	✓[non-PM]	✗	✓	✗

nizing amongst themselves. The OS, in this case, should provide the basic support for transparent atomic durability of shared PM areas across sharing applications.

4.2.4 Simple Memory-like Interface

Mapping the PM directly to applications address space only to update the PM areas using a complex transactional mechanism would bring little benefit to the application developers. Current transactional mechanisms [pmea, CCA⁺11, VTS11] require specific set of steps to start, end, or persist a transaction. In some cases, the applications have to go through the cumbersome task of identifying each of the PM resident objects. We believe that this approach does not yield full benefit of direct PM access, and makes the development process harder for the developers. Our proposal is that applications should be able to continue their current approach of using in-memory objects and should not have to worry about individually ensuring each objects durability. They should be able to make changes to objects in a PM area and persist the updates with a simple call like `msync` at a single point in time. The changes that were made durable at a certain time should be recoverable until the application issues durability for a second set of modifications to the same region.

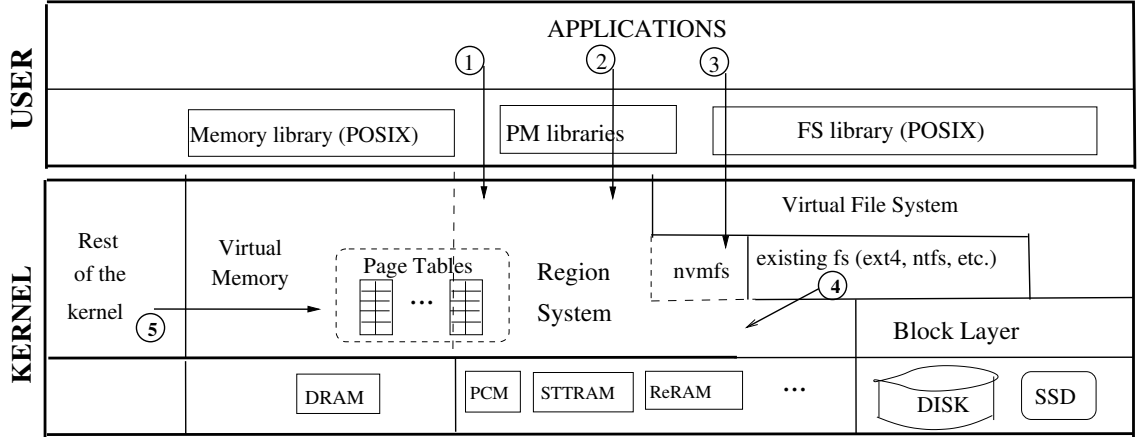


Figure 4.2: A PM augmented memory/storage stack.

4.2.5 Arbitrary and Unordered PM Allocation

Let's assume an use case where an application wants to construct and manipulate a persistent B-tree to store some data. The application would allocate memory for the B-tree internal nodes as well as data nodes and these could have different sizes. The allocated nodes can be deleted in arbitrary order depending on the applications requirements. The memory subsystem can easily handle the use case by allocating chunks of memory for the B-tree nodes, which can later be de-allocated irrespective of the order of allocation. The only issue here is that the memory subsystem does not support associating a persistent namespace to the allocations. This capability is also not supportable using the file interface, where files are sequential byte streams that do not support arbitrary and unordered allocation of PM. Some file systems support punching holes in a file, but the support is offset by the complexity of managing arbitrary chunk sizes. We postulate that, for future PM consuming applications, adding and removing PM areas of arbitrary sizes within a defined namespace in an unordered manner would be a primary requirement.

4.3 The OS Memory/Storage Stack for PM

In this section, we discuss an augmented OS memory/storage stack that addresses PM devices. We then identify a wide spectrum of interactions that are possible with PM devices and the OS support that make such interactions possible. These PM interaction alternatives address existing persistent storage interfaces that applications today are familiar with as well as new ones that are specific to the properties of byte-addressable PM devices. Our goal here is not to recommend one approach over the other, but rather to identify possible approaches and discuss the potential use-case for each.

4.3.1 Elements of the Stack

Figure 4.2 presents an augmentation of the current memory and persistent storage software stack that addresses the new PM devices. Boxes indicate software or hardware components in the system and dashed lines indicate that the component code bases are tightly coupled, i.e., designed to work together.

The OS contains a new data management entity, the *Region system*, besides the conventional virtual memory system and file system. The most important characteristic of this entity is that the region system is tightly coupled with the virtual memory manager and exposes PM areas directly to processes via appropriate page table mappings. The region system would also be tightly integrated with PM specific file system implementations illustrated by the NVMFS component. Further, we anticipate that an entirely new suite of user-level PM libraries will become available to facilitate application interaction with PM. Finally, since this work is focused on OS support, we do not discuss or preclude other alternatives to accessing PM devices directly by bypassing the OS partially or entirely [CME⁺12].

4.3.2 PM Use Cases

We identify five types of interactions that are possible with the region system, either directly or indirectly, each of which lead to a different use case for PM. These interactions are indicated using numbered arrows in Figure 4.2:

1. Applications interact directly with the region system using an PM-tailored system call interface (discussed in §4.4). This use case has been proposed recently [sni].
2. Applications interact with PM specific libraries that provide advanced support for allocating and using persistent memory similar to volatile memory. The libraries in turn can use the OS interface exported by the region system. Several recent proposals exist that follow this approach [CCA⁺11, GMC⁺12, VTRC11, VTS11].
3. Applications use the familiar file system interface (with potential PM-specific extensions to POSIX) to interact with an underlying PM specific file system. This use case supports legacy applications as well as newer applications that rely on the familiar file abstraction. Recent proposals on file systems specifically designed for PM follow this approach [CNF⁺09, WR11].
4. Applications interact with an existing file system (such as ext4, ntfs) and the file system itself interacts with the region system to implement PM specific bindings. Recent proposals for extending current file systems for PM follow this approach [LBN13].
5. Finally, the region system can also provide persistent memory access to the rest of the kernel. For instance, process and network management subsystems could store their state persistently to resume services upon system restarts.

Combinations of these use cases can also exist. For example, PM libraries can interact with the region system or with file systems that export PM.

Class	Name	Description
1,5	<code>region_d open(char *region_name, flags f);</code>	Open/Create a persistent region; returns a valid region descriptor on success.
	<code>int close(region_d rd);</code>	Close an open region with descriptor <code>rd</code> ; returns success or failure.
1,5	<code>int delete(char *region_name);</code>	Delete a persistent region <i>region_name</i> freeing any associated unshared persistent pages; returns success or failure.
2	<code>ppage_number alloc_ppage(region_d rd);</code>	Given a valid region descriptor, allocate a persistent page within the region; returns persistent page descriptor for the new page.
	<code>int free_ppage(region_d rd, ppage_number ppn);</code>	Free a persistent page from a region; persistent page can now be allocated again to other regions; returns success or failure.
3,5	<code>vaddr pmmmap(vaddr va, region_d rd, ppage_number ppn, int nbytes, flags f);</code>	Map a persistent page to the process virtual address space at address <code>vaddr</code> (hint); flags specify the type of mapping; returns mapped virtual address <code>vaddr</code> .
	<code>int pmunmap(vaddr va);</code>	Unmap the persistent page mapped at the given virtual address; returns success or failure
4	<code>int pmsync(vaddr va);</code>	Atomically persist all modifications to mapped pages of the region which is mapped at <i>va</i> so that the contents of the region are always consistent with the latest <code>pmsync</code> .

Table 4.2: A system call interface to persistent memory. *Syscalls address one or more of the following classes of functions: (1) Namespace management, (2) Allocation, (3) Mapping, (4) Consistency, and (5) Legacy support.*

4.4 The Region System Interface

In Table 4.2, we suggest a possible OS interface to the region system. We had the following goals in mind: (i) expose PM areas directly within a process address space, (ii) provide support for safely and consistently modifying/sharing persistent memory, and (iii) reuse familiar notions of file system namespace and memory mapping. While the suggested OS interface is not intended to be comprehensive, we chose components to highlight both the unique potential and challenges when working with PM. Besides these PM-specific system calls, classic file system calls (e.g., `read`, `write`) can be supported via PM-specific

file systems or PM-specific bindings in existing file systems that in turn interact with the region system.

The region system exports a namespace management capability similar to file systems, allowing the creation of named regions and region directories (`open`, `close`, `delete`). This allows for straightforward integration with conventional POSIX file system interface in the future as necessary. Applications can also allocate/free persistent memory pages (`alloc_ppage`|`free_ppage`) within a specified region. Similar to current memory mapping support for files, they can map/unmap persistent pages within a region to their address space (`pmmmap`|`pmunmap`). We believe that the key distinction between this and existing POSIX file interfaces is the `pmsync` interface that implements the transactional persistence of regions, allowing an application to atomically persist all modifications to mapped pages of a region. This simplifies application development when using PM whereby the developer does not need to worry about temporarily inconsistent versions of their data structures in PM mapped memory pages; they retain the flexibility to specify `pmsync` operations once the in-memory data is deemed consistent from the application's viewpoint.

Regions: The region system creates the `region` abstraction for using persistent memory. A region is an unordered collection of persistent pages (`ppages`) identified by a unique region descriptor. `ppages`, which are distinguished by page numbers, can be added to a region and deleted later in arbitrary sequence irrespective of the sequence in which they were allocated. The size of the `ppages` are 4KB, which is same as the kernel page size. Unlike conventional file systems, there is no `read/write` access to regions; memory-mapping of `ppages` is the only mechanism to consume PM. The `pmmmap` system call allows the application to map `ppages` to the process address space, while `pmunmap` reverses the mapping. By invoking `pmsync`, applications can make all the changes to a regions' mapped `ppages` atomically durable.

4.4.1 Example

The code snippet in Listing 4.1 illustrates how an application would consume PM through the region system interfaces. The similarity with existing POSIX interface makes the region system interface very easy to use.

Listing 4.1: Region system usage illustration

```
#define PAGE_SIZE 4096

int rd = open_region("__region_1");
int ppage_no = alloc_ppage(rd);
void *data = mmap(NULL, rd, ppage_no, PAGE_SIZE,
    MAP_SHARED);
/* write something to data */
pmsync(log);
pmunmap(log);
close_region(rd);
```

4.5 Architecture

In this section, we present the *region system* architecture. The architecture of the persistent metadata is self sufficient, which means the *region system* metadata can be rebuilt using the information stored in itself. However, *region system* also uses some volatile metadata to interact with the kernel and to optimize the metadata operations. The separation of volatile and persistent metadata is achieved so that the volatile metadata does not have any impact on consistency.

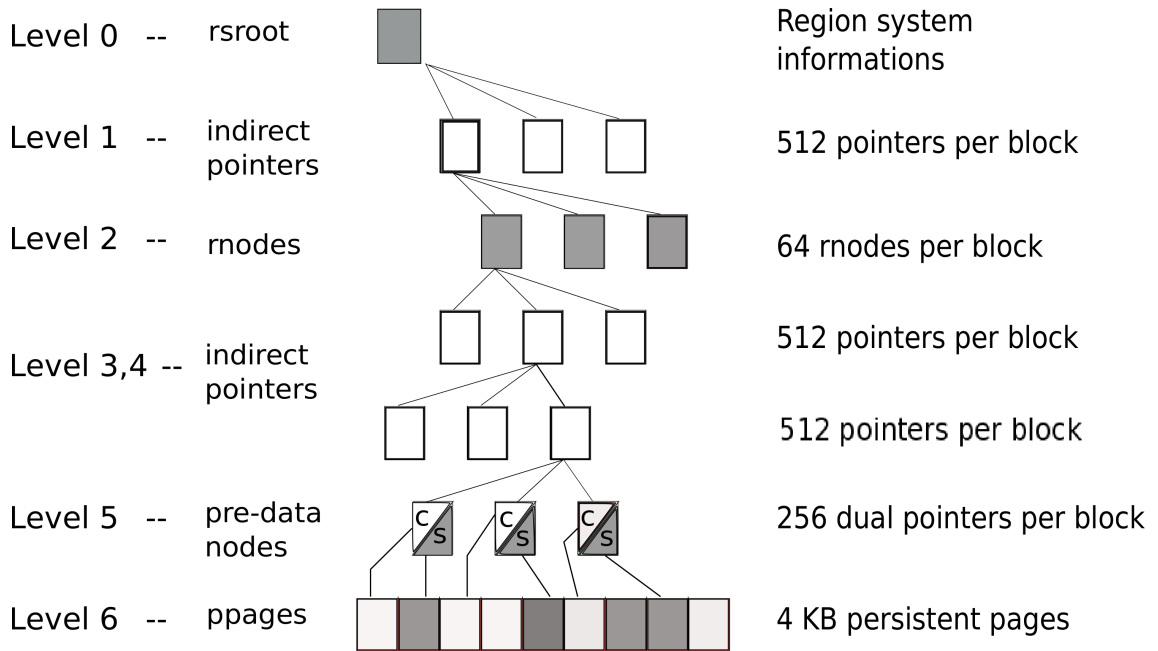


Figure 4.3: Region System Tree.

4.5.1 Persistent Metadata in the Region System

Region system design is partially inspired by the WAFL [HLM94](Write Anywhere File Layout) file system. However, region system is designed to hold no redundant metadata to avoid additional cache flushes. This design choice leads to a fixed layout for the region system metadata. The metadata is stored in the persistent memory physical pages. Figure 4.3 shows the layout of the region system.

The region system can be best thought as a tree of 4KB pages. For the rest of the discussion, we will identify these 4KB pages as blocks. There are 5 types of blocks – `rsroot`, `rnode`, `indirect blocks`, `pre-data`, and `data blocks`. `rsroot` contain the information for a *region system* instance, which includes the name, permission, etc. `Rnode` contains the region specific information such as name, id, flags, etc. The `indirect blocks` each contain 512 pointers to the next blocks. `Data blocks` are `ppages`, and those are the leaf nodes. However, the `pre-data` blocks has two pointers for each `ppage` – one for the snapshot, another for the current version of data, thus limiting the

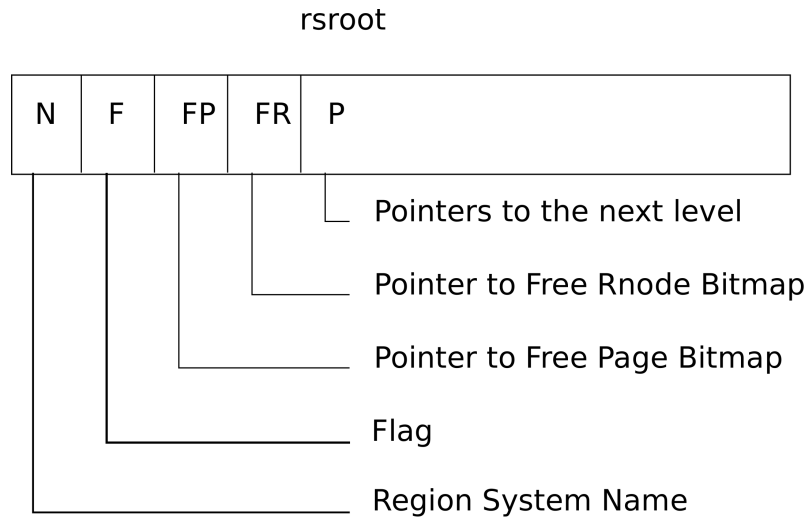


Figure 4.4: Region System Root.

number of pointers to 256 per block.

Rsroot

`Rsroot` is the root of the *region system* instance. It contains the necessary information to traverse the *region system* tree. The starting physical address and the size of the region system have to be known beforehand to mount the region system instance. The `rsroot` contains an identifier, a flag, and pointers the next indirect layer as shown in Figure 4.4. The pointers, FP and FR, in conjunction with the flag ensure a faster mount time in case of a clean shutdown. The flag value can be either `RS_VALID` or `RS_CLEAN_SHUTDOWN`. The flag is checked during the mount operation to verify if a *region system* instance already exists at the provided address with the same name, and to identify the status of the region if it exists. Presence of any of the above mentioned flags confirm the existence of a region system instance. If the flag is `RS_CLEAN_SHUTDOWN`, the free space bitmaps can be readily used. However, flag value `RS_VALID` indicates that a complete traversal of the recovery process has to be done in order to generate the bitmaps.

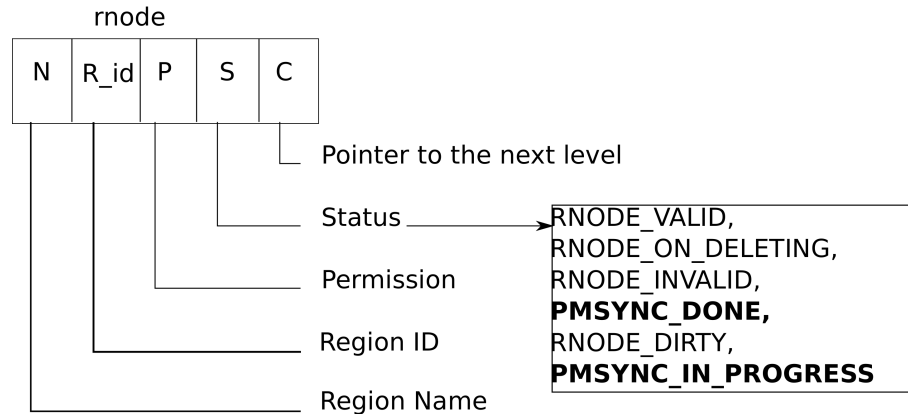


Figure 4.5: Region Root - rnode.

Rnode

The `rnode` is the root of a *region*. A simplified version with important fields is shown in Figure 4.5. It contains the name, id, and pointers to the next indirect blocks for the *region*. However, the most important part of the `rnode` is the `STATUS` field which helps to ensure idempotent `PMSYNC` which is further discussed in chapter 5. The `rnodes` are 64 bytes in size, which means one `rnode` block contains 64 `rnodes` once created.

Indirect Pointers

Indirect pointers are the simplest metadata block which contain 512 8-byte pointers to the next level.

Pre-data Nodes

Pre-data nodes reside at the last level of the metadata. Instead of one pointer to the data pages, we store two pointers – one for the `snapshot` and one for the `current` version of the data. The contents of the pointers denote a state for the `ppage` in concern as shown in Table 4.3.

Table 4.3: Ppage states derived from pre-data current and snapshot pointers

Current	Snapshot	State
0	0	No Ppage
0	y	Invalid – There can not be a snapshot without a current
x	0	Un-synced page, mapped to the address space
x	y	x == y, Page in synced state
		x != y, Page in un-synced state, “y” is the consistent version

Data Pages

Data pages are simple 4KB blocks memory page-aligned data. The `current` version of the page is mapped to the process address space during a mapping. A “synced” page is mapped read-only. In case of an “un-synced” page, the `current` version is mapped with read-write privilege, and the `snapshot` version is kept in PM for potential recovery.

Pointer Values

The pointers in the metadata tree are not actual physical addresses. The *region system* does the physical to kernel virtual address conversion when it requires to map the pages.

4.5.2 Volatile Metadata in the Region System

Region system keeps volatile metadata carefully separated from the persistent metadata.

The volatile metadata at the region level includes PM free page bitmap, free rnode lists, a red-black tree to quickly lookup open regions, and a red-black tree to help quick lookup of existing regions. These trees are generated during the mount, and they grow depending on the operations on regions within the region system. Region level volatile metadata includes a region free page list, which is persisted on a successful `region close` operation.

4.6 Summary

In this chapter, we proposed the *region system* interface after laying out the assumptions of PM usage alongside possible PM use cases. We have also presented the on-PM architecture of the *region system* as well as the volatile metadata. The separation of persistent and volatile metadata in the *region system* design is intended to make the consistency operations independent of the volatile metadata.

In the next chapter, we explore the design of the *region system* volatile and persistent operations based on the architecture. Furthermore, we delve into the details of atomic data persistence of mapped data.

CHAPTER 5

REGION SYSTEM: DESIGN

In the previous chapter, we described the inadequacy of the current OS software stack to fully support PM application development. We proposed a new kernel subsystem – *region system*, and presented the architecture of the system along with the interfaces that are exposed by the system. In this chapter, we initially elaborate how we minimize cache flushes by designing optimized *region system* operations on the aforementioned architecture. Finally, we discuss how *region system* architecture supports atomic durability of mapped data in a shared environment.

5.1 Region System Operations

To support low-latency operations, the region system adopts a lightweight and low overhead approach to managing PM. First, it carefully avoids keeping any redundant data persistently. Avoiding the use of redundant metadata eliminates the need for atomic updates to inter-dependent metadata, simplifying the task of keeping metadata consistent. Only a single version of the metadata necessary to reconstruct the region system state is kept up-to-date on the PM device. More importantly, the region system metadata operations has been designed to preserve the atomicity of the operations. In this section, we distinguish the region system operations by their nature and delve deep into the discussion of how the atomicity of operations are maintained minimizing the redundant metadata.

We carefully consider the impact of the system calls from Table 4.2 on the region system metadata and define two classes of syscalls. The first class, which is of most importance, is the persistent syscalls. Syscalls like `create_region`, `delete_region`, `allocate_ppage`, `deallocate_ppage`, `pmsync` are defined as persistent syscalls as they modify the persistent metadata tree. Other syscalls do not modify the metadata tree,

but rely on the interactions of the region system volatile metadata and kernel data structures; we identify these as volatile syscalls. Based on their scope of operation, we further divide the persistent operations into two categories – persistent metadata operations and persistent data operation.

5.1.1 Persistent Metadata Operations

For the persistent syscalls, it is very crucial to identify the updates going to be made to the persistent metadata and the order of those updates. The concept of durability point is associated with all the syscalls. A durability point is the place where the last update in a set of updates is done to the PM for that operation. Until the durability point is reached, the syscalls are not complete. The same general principle holds for all the system calls—all region metadata get updated atomically to implement atomicity of the system calls themselves. CLWB and SFENCE instructions are used to ensure the proper ordering of writes to the PM. Noting that the previously introduced PCOMMIT instruction is now deprecated [pco], we rely on the recommended alternative technique, ADR (Asynchronous DRAM Refresh), to flush the write pending queue of the memory controller. The non-redundant metadata design minimizes the number of updates requiring memory ordering. Next, we outline the updates to the PM, the order of those updates, and the durability point for each operation.

Create

Figure 5.1 illustrates the *region system* metadata which are updated or written into for creating a region. The circles show the where the metadata is being updated, and the numbers inside the circles indicate the ordering of the update. Upon a `region_create` invocation, the rnode free list is generated by allocating a rnode page (64 rnodes) if the

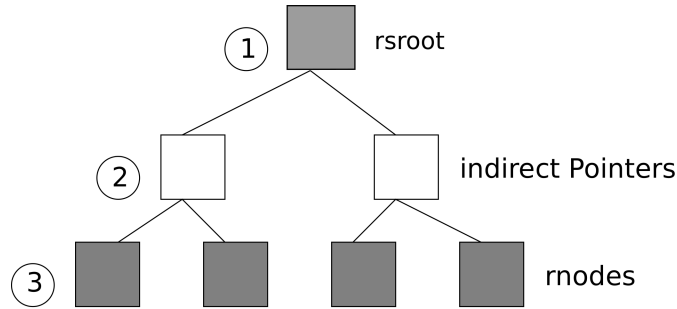


Figure 5.1: Region Create: Required ordering of updates to the PM metadata; circled numbers show the order of the update.

list is empty, otherwise a rnode is picked from the list. If a rnode indirect slot is available then the indirect pointer is updated to point to the rnode (at line 13 of the Algorithm 1, and ② in the figure). Otherwise a new indirect block is allocated (512 indirect pointers) and the address of the page is saved inside the `rs_root` (at line 10 of the Algorithm 1, and ① in the figure). Finally, the rnode picked from the list is updated with necessary information including the region status to achieve consistency. In case of a system failure, the `rnode_status` helps to identify if a rnode visited while traversing the region system tree from `rs_root` during recovery should go to the free list or if the `r_node` is for a valid region. A persistent barrier (`CLWB + SFENCE`) is executed after `rnode_status` is set to valid, which marks the completion of the syscall (at line 15 of the Algorithm 1, and ③ in the figure). This is the durability point for the create system call. The updates at ① and ② are forced to complete before this point as the `CLWB+SFENCE` combination is invoked to force the updates out of the cache lines, so that there is no inconsistency if ③ failed.

Delete

Region delete (Algorithm 2) operation also requires ordering of updates and careful set up of rnode flag. We have to deal with the volatile metadata and then persistent metadata. The volatile metadata management involves clearing the free lists and removing the region from the red black trees. The allocated pages for the region needs to be free and added

Algorithm 1: Algorithm: Region Create

Data: Region name
Result: Region id after creating region

- 1 **if** *free rnode not available in free_rnode_list* **then**
- 2 | Allocate new rnode block;
- 3 | Add rnodes to the *free_rnode_list*;
- 4 **else**
- 5 | Pick the next available rnode from the list;
- 6 Calculate the indirect pointer address for that rnode;
- 7 **if** *indirect block not available* **then**
- 8 | Allocate indirect block;
- 9 | Update appropriate rsroot pointer for the indirect block;
- 10 | CLWB(RSROOT_POINTER)+SFENCE
- 11 **else**
- 12 | Update the appropriate indirect block pointer for the rnode;
- 13 | CLWB(INDIRECT_POINTER)+SFENCE;
- 14 Update rnode status;
- 15 CLWB(RNODE)+SFENCE;
- 16 Return Region Id ;

to the global free page list. However, the most important task is to set `rnode status` to `REGION_ON_DELETE` to mark the start of the deletion process. Once the flag is set, the region delete operation will always complete even in the case of a power failure. This is the durability point for the delete operation. Upon completion, the rnode is added to the free rnode list and rnode status set to `RNODE_INVALID`.

Allocate

Allocation of a ppage is also done atomically. The durability point of the syscall is when the `current pre-data pointer` is updated with the address of a newly allocated page, and the update is persisted with a `CLWB+SFENCE` combination. Another cache flush operation is done before updating the pre-data current pointer if any of the indirect blocks were also allocated. Algorithm 3 shows how the pre-data snapshot pointer is set to zero to set the state of the page to unsynced state as described in Table 4.3.

Algorithm 2: Algorithm: Region delete

Data: Region name

Result: success or failure

```
1 if Region not present in rs_region_rb_tree then
  | // region not present
2 | return failure;
3 Set rnode status RNODE_ON_DELETING;
4 CLWB(RNODE)+SFENCE;
5 Clear the volatile metadata for the region including the dirty page lists;
6 if Region had pages allocated then
7 | De-allocate all the pages;
8 | De-allocate indirect pointers;
9 | Add the above to PM free page list;
10 Add rnode to PM free rnode list;
11 Set rnode status RNODE_INVALID;
12 CLWB(RNODE)+SFENCE;
13 Return success;
```

Algorithm 3: Algorithm: Allocate ppage

Data:

Result: ppage

```
1 Get next free page from region_free_page_list;
2 Calculate the 1st indirect pointer, 2nd indirect pointer, and pre-data pointer address;
3 foreach indirect pointers upto last level do
4 | if Any pointer block do not exist then
5 | | Allocate block and update previous pointer;
6 CLWB(POINTER ADDRESS)+SFENCE;
7 Allocate 4KB page;
8 Update pre-data current pointer with the page address;
9 Update pre-data snapshot pointer with 0;
10 CLWB(PRE-DATA POINTER)+SFENCE;
11 Return ppage_no;
```

De-allocate

For this system call, we need to free the ppage assigned for the ppage number. However, there can be one or two pages assigned for the ppage number according to the state of the page. If the page is in synced state, only one page has to be freed as both `snapshot` and `current` pointers point to the same page. If the page is unsynced but has a valid `snapshot` pointer, two pages are freed. For an unsynced page only the `current` version of the page needs to be deleted as shown in Algorithm 4.

Algorithm 4: Algorithm: Deallocate ppage

Data: ppage number
Result: success or failure

- 1 Calcualte pre-data address;
- 2 Save snapshot and current pointer values;
- 3 Set pre-data `snapshot = 0` and `current = 0`;
- 4 `CLWB(PRE-DATA POINTER)+SFENCE`;
- 5 **if** `snapshot == current OR snapshot == 0` **then**
- 6 Free the page pointed by `current`;
- 7 **if** `snapshot ≠ current` **then**
- 8 Free both `current` and `snapshot`;
- 9 Add freed page/s to the Region free page list;
- 10 Return success;

5.1.2 Persistent Data Operation

Traditionally, The `mmap-msync` based POSIX interface is not atomic. Most of the PM-specific file systems ignore the benefits of the directly mapped data as they follow the traditional `read-write-fsync` interface which disallows the applications to get complete benefit of using PM. Mapping the PM directly to the process address space minimizes the OS intervention, hence maximizing the performance. However, there are multiple complexities with writing to PM directly. First, if every update is not a desired unit of

atomic write, the PM content is bound to be inconsistent if a failure occurs. Secondly, today's multi-processor system supports multiple processes executing at the same time. All the contents of the different cache lines associated with the mapped page do not reach the PM voluntarily when one process issues an `msync`. Third, additional writes can infiltrate the PM when a sync process is ongoing, polluting the desired content of the PM. To ensure proper and consistent use of PM three capabilities are absolutely necessary – 1) maintain some kind of state for PM mapped data, 2) implement mechanism to reflect cache contents of all CPUs to the PM immediately at the time of the sync operation, 3) keep mapped data protected from further writes during the syncing process.

The `pmsync` interface achieves all of the above goals and simplifies the consistent management of durable data mapped into an application's address space. We now discuss how the `pmsync` operation provides an atomic durability mechanism for shared `ppages` across multiple processes.

5.1.3 Pmsync

Region system manages data in the form of `ppages`, which are mapped to the applications address space. With `pmsync`, an application can choose when it wants to make any of its in-memory data durable. *Region system* provides the support by encapsulating states to the `ppages` as depicted in Table 4.3 in conjunction with the `rnode` status flags. The `rnode status` flags shown in Figure 4.5 is very relevant to the `pmsync` discussion.

Initially, when a `ppage` from a `region` is mapped to an application's address space, updates to the page are not made durable until a `pmsync` is invoked. At this point, the region is in unsynced state. On `pmsync`, all updates to the page since the previous `pmsync` are made atomically durable. Upon a `pmsync` completion the current and snapshot pointer both point to the same `ppage`, and the `rnode` status is set to PM-

SYNC_COMPLETE. Any updates to ppages after the latest `pmsync` get discarded in case of a system crash or failure. If the rnode status is not `PMSYNC_IN_PROGRESS` after a system restart, it means that the changes to the current pages are not final. Hence, those pages can be discarded, and the snapshot pages are recovered to be the current page. The rnode flag makes sure the states are clearly transitioned. Regardless of the number of pages mapped to a process' address space, all updates to a region are always made atomically durable.

Protecting the Sync

CPUs are free to execute load/store instructions to mapped ppages at any time. Furthermore, multiple applications may share ppages of a single region. A trivial use case is applications accessing a single data structure stored in the ppages of a particular region. As a result, ppages from the same region can be mapped to address spaces of processes running in different CPUs. It is imperative that not only the invoking CPU but all CPUs are restricted access to the region pages upon a pmsync invocation. However, CPUs may not be deprive other tasks from executing for more than a short period of time to preserve system responsiveness to ongoing tasks. Thus, a mechanism that bars all CPUs from accessing the mapped region pages upon pmsync invocation and sets up the protection of the mapped pages quickly is necessary.

To implement pmsync, the region system first issues an Inter-processor Interrupt (IPI) to all the CPUs to put them in a non-maskable interrupt state. IPIs are intended to stop all the access to the region while the regions dirty pages are write protected for all the processes. For efficiency, the region system keeps a list of dirty pages which contains all the necessary information for write protecting the pages and setting up `copy-on-write` (CoW) operations. The invoking CPU's IPI-handler sets up the mapped pages for CoW. The other CPU's IPI-handlers wait for the completion of the invoking CPU's IPI handler

before they become ready for normal execution, thus ensuring that no memory pages get modified while the invoking CPU's IPI handler is executing. The invoking CPU's IPI handler write protects all the mapped pages of the region. Upon this handler's completion, processes are disallowed access to the mapped pages; the `pmsync` operation acquires a region-wide lock which page fault handlers as well as other persistent operations are required to acquire before accessing the region.

System Wide Cache Flush

Upon returning from the IPI, the active CPU also flushes the dirty cache lines for the region using the kernel virtual address. At this point CPU cache snooping mechanisms make the pages transparent across all the processes. After cache flush, an idempotent execution of `pmsync` is initiated. First, the region `r_node` status is updated to `PM-SYNC_IN_PROGRESS`. Dual pointers, used in combination with the `r_node` flags `PM-SYNC_IN_PROGRESS`, `PMSYNC_COMPLETE`, ensure an idempotent `pmsync`. After the `r_node` is set to `PMSYNC_IN_PROGRESS`, `pmsync` is guaranteed to finish successfully, even after a system crash. Finally, the snapshot pointer is modified to point to the current page and the previous snapshot if there was any is deleted, before setting the region status to `PMSYNC_COMPLETE`. The modifications to the `r_node` status are protected by the `(clwb+sfence)` persistent barrier to enforce proper ordering of updates.

Eliminate CoW Propagation

During normal operation, any access to a mapped page within a `pmsync`'ed region results in a copy of the current page, and the snapshot pointer is changed to point to the copied page while the current (old) page is made accessible to the user. Existing snapshot mechanisms allow the `copy-on-write` (CoW) update to propagate to intermediate layers of metadata, even up to the root. The region systems novel `dual pointer` technique

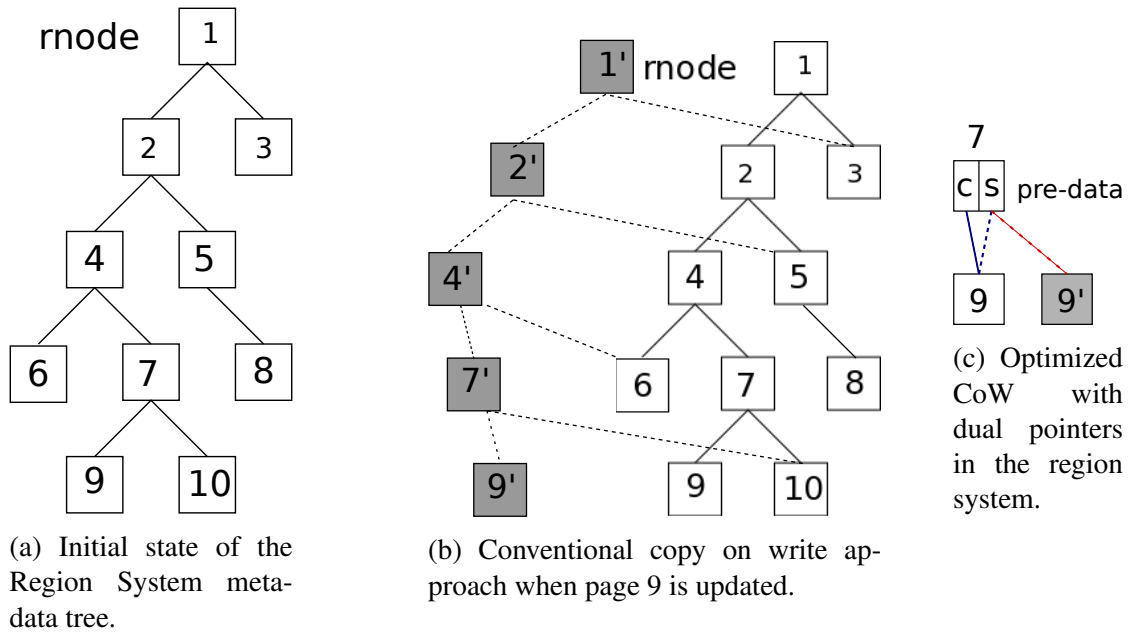


Figure 5.2: Copy-on-Write propagation elimination.

eliminates such CoW amplification by limiting recursive updates to the lowest metadata layer in the region system metadata hierarchy.

An example can be seen in Figure 5.2, where a write attempt to page 9 would force 5 blocks in the upper layer copied for the region system architecture. However, because of the pre-data dual pointers, only one block gets copied.

5.1.4 Recovery

The persistent metadata stored by the region system has sufficient information to recover from a crash. During startup, the region system looks for a `rs_root` in a particular physical location which is provided as a kernel boot parameter. If the `rs_root` is found at that location, the region system can be mounted by following the steps below.

In case of an unclean shutdown, the entire metadata for the region system can be rebuilt by traversing the tree starting from `rs_root` in case the system is recovering from an unclean shutdown. The volatile metadata structures are also reconstructed during the

recovery process. While reconstructing, the `r_node status` provides support for completing region wide operations such as `pmsync` and `delete`. To ensure proper recovery, other operations on the region system are suspended until the region system is reconstructed.

During a normal (clean) shutdown or clean `region_close` operation, system wide and region wide free page bitmaps are persisted to ensure a faster boot. The `rs_root` contains a pointer to the PM free page list and free rnode list for the region system, while the `r_node` holds a pointer to the region wide free page list. However, after a crash or power failure they are reconstructed by walking the region system metadata structures depending on the `rs_root` or `r_node` status.

5.2 Summary

In this chapter, we have classified the *region system* operations into two categories – persistent and volatile operations. The persistent operations are further divided into data and metadata operations. We have described the process of achieving atomic persistence for metadata operations by carefully planning the order of updates to the PM and by identifying the durability point for each operation. We have also outlined the pre-requisites of maintaining consistency of shared PM areas, and provided the solution by means of `pmsync`.

In the next chapter, we will delve into our implementation, provide a visual example of `pmsync` operation, and finally present some comparison with the contemporary PM specific solutions.

CHAPTER 6

REGION SYSTEM: IMPLEMENTATION AND EVALUATION

In the previous chapter we have described how the design of *region system* operations optimize the PM usage. We also outlined the steps required to achieve atomically durability of mapped data as the durability points of the persistent metadata operations. However, to verify the validity of our design, in this chapter, we discuss the implementation of *region system* as a kernel module and evaluate with the state-of-the-art file system and persistent memory library.

6.1 Implementation

Region system is implemented as a Linux kernel module for Linux version 3.10.14. The core module contains almost 4000 lines of code. However, the modification to the kernel is less than 100 lines of code. The kernel modifications, though small in quantity are very significant to combine region system to the I/O stack. Next, we provide a high level overview of the kernel modifications and their relations with *region system* persistent and volatile metadata.

6.1.1 Kernel Modifications

Region system runs with minimal modification to the Linux kernel code base . There are several critical data structures which had to be modified to support region system operations. Figure 6.1 shows a simplified version of the kernel and region system interaction, and separation of persistent and volatile memory.

The `task_struct` holds an array of all regions opened by that process by means of `nvm_regions` which is an additional data structure to support the *region system*. The

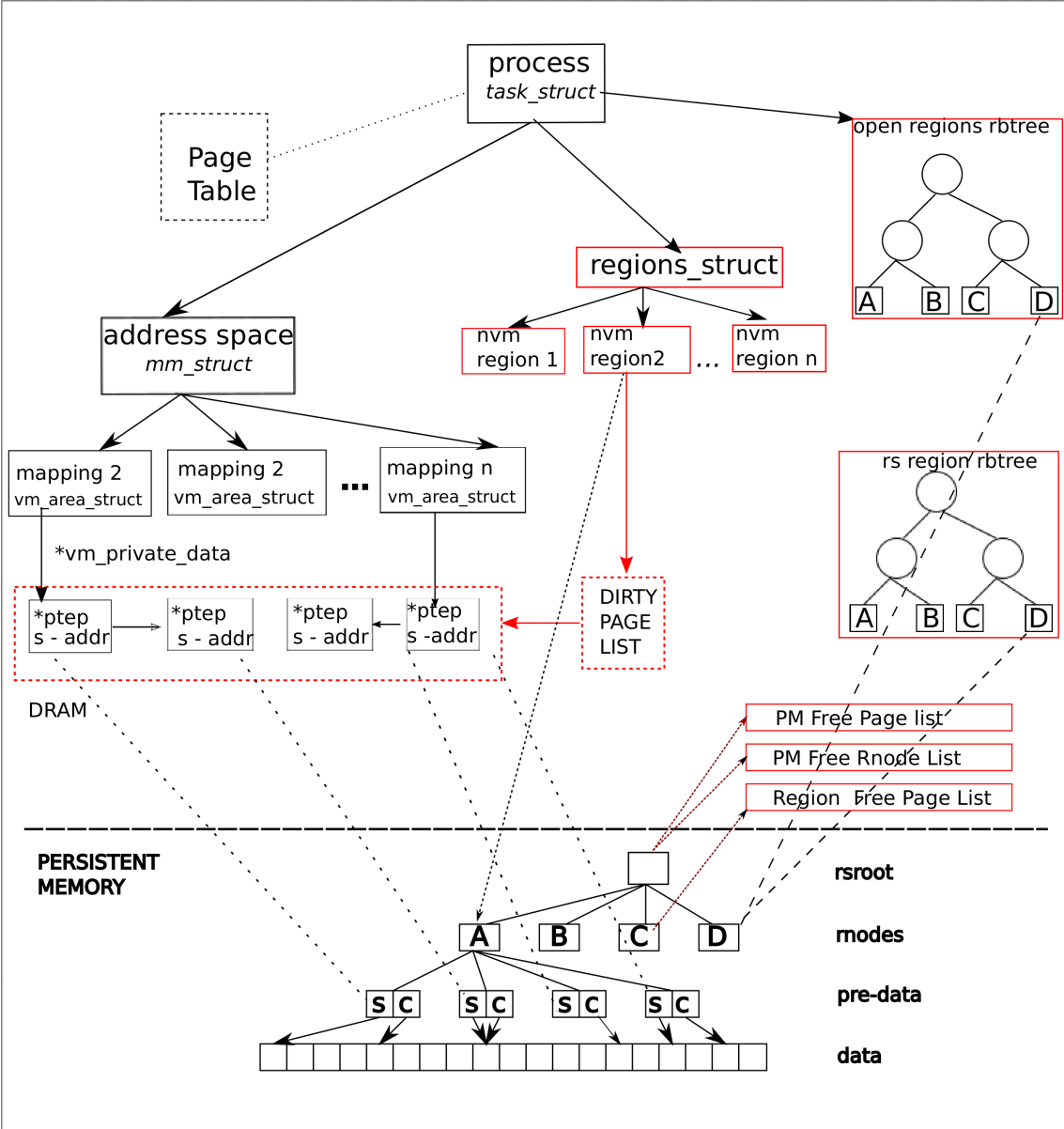


Figure 6.1: Simplified diagram of kernel and region system interaction.

`nvm_regions` struct contains a pointer to the `rnode` as well as a dirty list of pages. The list elements, which holds the address of the snapshot pages for that particular mapping are also referenced by the `vma_area_struct` by the `vm_private_data` pointer. There could be multiple mapping for an address space. The red-black trees serve the purpose of optimized searching for existing regions for the region system and open regions of the process. The important volatile metadata's are shown in Figure 6.1 alongside the persistent metadata. The data structures bounded by red boxes are added by the *region system*.

6.1.2 Kernel Interaction for Region System Operations

The volatile operations consist of `open`, `close`, `pmmmap`, and `pmunmap`. The `open` syscall involves setting up the `regions_struct` for a process. As shown in Figure 6.1, the `regions_struct` contains an array to store pointers to the `nvm_regions`. An instance of the `nvm_region` data structure is created for each open region which contains the count of processes that opened the region. The counter inside the `nvm_region` is incremented during the create operation. The `nvm_region` also contains a pointer to the PM resident `rnode` which is used to get the `rnode` features such as name and status for the other region system operations such as `close`, `delete`, `pmmmap`, `pmsync`, etc. Once the region is opened, the `rnode` is added to the `open_regions_rb_tree` to help quick lookup of the region.

`pmmmap` and `pmsync` fall in the category of persistent operations. However, both of these operations rely on some volatile metadata to maintain atomicity. The `pmmmap` syscall involves recording the mapped `pte` entry, `mm_struct`, and kernel virtual address of the `snapshot_pointer` for the pages per mapping. A list of the above mentioned combination is also kept per region for an easier traversal during `pmsync`, which we will discuss

shortly. The nature of the volatile metadata is such that, after a system crash the *region system* can be returned to a consistent state without these information. These metadata are only stored to speed up the `pmsync` process. To provide a complete understanding of the process an example of the `pmsync` operation is given below.

Pmsync Example

The steps involved in `pmsync` are described in Figure 6.2 with the aid of an example. It illustrates `pmsync` using a simplified version of the region system metadata structure (sans indirect blocks).

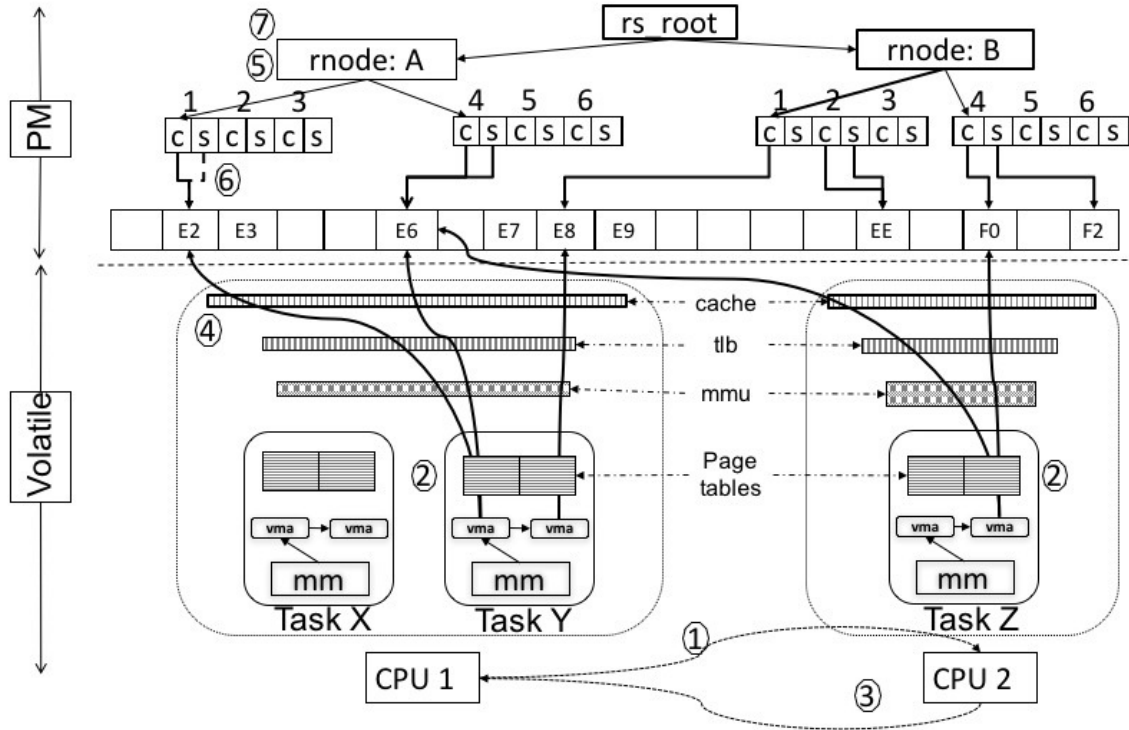


Figure 6.2: Simplified `pmsync` example.

Initial state: Region A has 2 pages mapped, page 1 (physical address: E2) is in unsynced state, and mapped by Task Y (CPU 1). Ppage 4 (E6) is in synced state and mapped by Task Y (CPU 1) and Task Z (CPU 2). Region B has 3 ppages 1, 2, and 4,

among which 2 (EE) is not mapped to any process. Ppage 1 (E8) (unsynced) is mapped to Task Y (CPU 1), ppage 4 (F0)(previously synced) is mapped to Task Z.

On pmsync invocation for rnode A the following steps occur:

1. CPU 1 sends IPI to CPU 2.
2. IPI handler in CPU 1 write protects all the dirty pages mapped to each process belonging to region A at this point. The IPI handler in CPU 1 write protects user mapped page for ppage 1 (E2), nothing is done of ppage 4 (E6) as the page is not dirty, already write protected. CPU 2's IPI handler waits for CPU 1's IPI handler to complete. Ppages from Region B are untouched.
3. CPU 2's IPI handler finishes.
4. CPU 1 flushes dirty cache lines of ppage 1 (E2).
5. The rnode A flag set to PMSYNC_IN_PROGRESS protected by CLWB+SFENCE. At this point even after a system crash or power failure pmsync will complete on next mount of RS.
6. ppage 1's snapshot pointer points to E2. At this step all the current and snapshot pages of region A point to same ppage.
7. The rnode A flag is set to PMSYNC_COMPLETE protected by CLWB+SFENCE.

6.1.3 Memory Management

Persistent memory allocation and de-allocation are managed by the *region system* rather than the kernel memory management subsystem.. It is important to restrict the kernel memory manager from accessing the subsystem so that it cannot treat these physical pages (PM) as regular DRAM pages. The reasoning is quite simple – decoupling the PM management from DRAM management so that other subsystems can not affect PM

by accident. As an example, it would not be ideal for the PM pages to be swapped out or replaced by other pages brought in from storage. Hence, the PM is reserved so that only region system can modify this portion of physical memory.

We deploy a simple 4KB page allocation policy, and maintain a PM free page bitmap for book keeping. The garbage collection is also done by the region system module.

6.2 Evaluation

For evaluation, we compare region system operations with regular state-of-the-art file system ext4 (DAX enabled), alongside contemporary persistent memory library PMEM.IO. Both ext4-DAX `msync` and PMEM.IO flushing primitives provide weak or no atomic durability guarantees. On that note, no contemporary solution provide similar atomic durability and consistency guarantee like region system.

6.2.1 Methodology

Given that persistent memory hardware is currently unavailable for purchase in the marketplace, we rely on both hardware and software emulation. All our experiments were conducted on Intel's persistent memory emulation platform (PMEP [LDK⁺14b]) which allows realistic evaluation of software that are intended to run on top of CPU-addressable and byte-addressable persistent memory devices. The PMEPE environment also allows for fine-grained latency control over PM access times and supports Intel's advanced x86 instructions specifically developed for PM. The machine was equipped with a Intel(R) Xeon CPU E5-4620 v2 @ 2.60GHz, 32 GB of volatile memory, 32 GB of persistent memory, and a custom version of the Linux kernel v4.1.18. The *region system* is installed as a module on kernel version 3.10.14 under the same conditions.

We separate the evaluation into two parts; first we compare the metadata operations followed by the comparison of pmsync operation.

6.3 Microbenchmarks

The region system exports a minimal interface to the applications for mmap based usage of PM. The system calls in Table 4.2 are designed based on the POSIX standard file system calls in terms of functionality. Writing applications using this interface is straightforward. We wrote several simple applications to compare the performance of similar operations of region system to the ext4-DAX file system. The DAX code avoids the extra copy in the page cache by performing reads and writes directly to the storage device, which is similar in principle to region system. DAX also primarily supports block sizes of kernel “PAGE_SIZE” which is also the case for region system. However, DAX does not provide any of the atomic durability for data as provided by the region system. The main purpose of this evaluation is thus an evaluation of the cost of the additional features provided by the region system relative to the state-of-the-art DAX performance features.

We compare the latency of similar operations across these systems in Figure 6.3. We measure performance in terms of operation latency for up to 2K operations. The basic operations are open (6.3f), close (6.3g), create (6.3a), delete (6.3b) a region or a file, and allocating (6.3c) and deallocating (6.3d) multiple of 4KB PM chunks/pages for files or regions. Other operations include mapping pages of the file/region into the user address space (6.3e). We also measure the latency of punching holes in the file or region system (6.3h); this operation allows simple deallocation of file space when using file-mapped PM space.

The operations can be grouped into two categories – `persistent` and `volatile`. The `persistent` operations require updates to PM resident metadata and data; create,

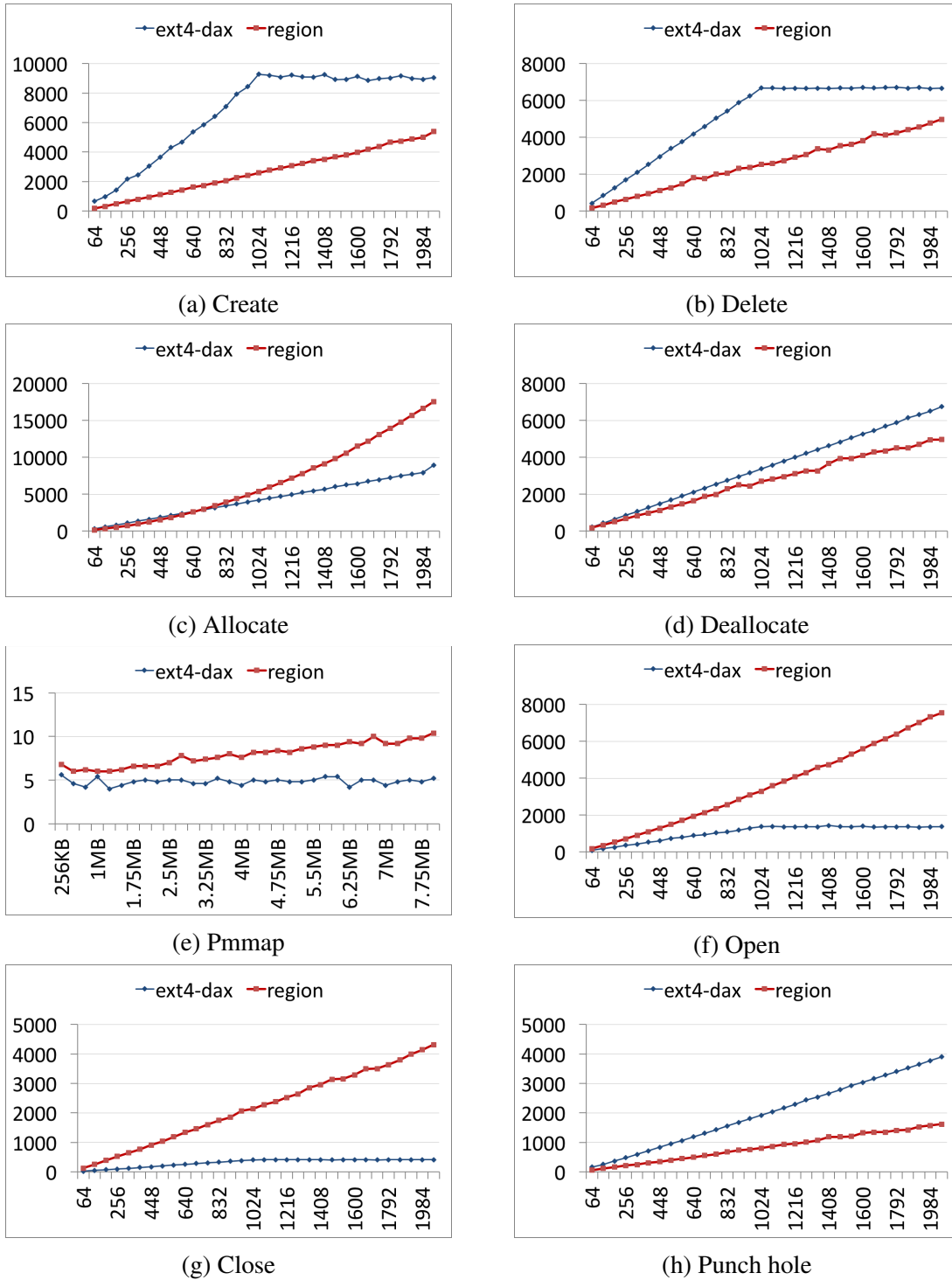


Figure 6.3: Comparison of region system interface with ext4-dax using microbenchmarks. X-axis shows the number of operations (file size in case of pmmap), Y-axis shows latency in microseconds.

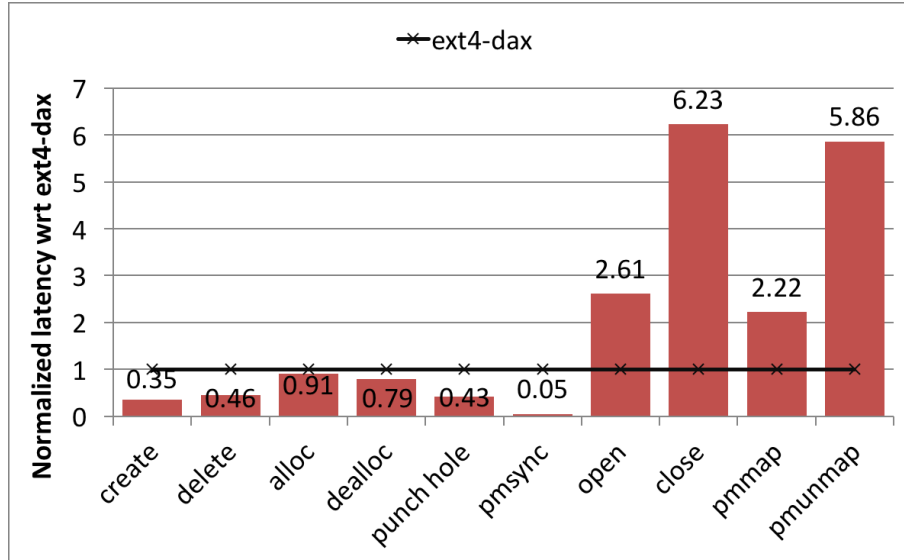


Figure 6.4: Average latency of region system operations relative to EXT4-DAX.

delete, allocate, deallocate, pmsync, and punching holes fall into this category. Other operations fall into the volatile category, where no PM-resident data or metadata is updated. We note from Figures 6.3a, 6.3b, and 6.3d that the region system performs better than the ext4-dax counterpart by a significant margin. This is achieved because of low metadata flush cost since the region system does not contain any redundant persistent metadata. The region system’s allocate operation does not outperform ext4-DAX `fallocate` because of the volatile metadata dependency on the page allocation process. The volatile operations perform worse than DAX due to the additional book-keeping required for various volatile data structures.

Punching holes: The region system is designed to support unordered and arbitrary allocation-deallocation. Though traditional POSIX file systems do not support this mechanism, this feature is rather useful to support arbitrarily ordered allocations and deallocations of data within files as necessary with memory-like usage of PM. Ext4-DAX supports punching holes via the `fallocate` system call. We compare the performance of deallocating every alternate pages after a file/region is initially allocated. Figure 6.3h shows that region system performs much better than ext4-DAX in punching holes in the allo-

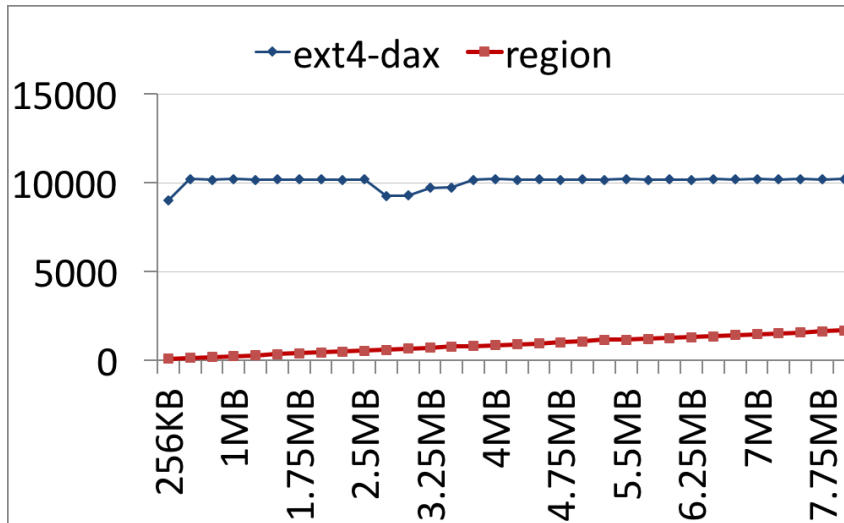


Figure 6.5: Pmsync comparison with EXT4-DAX msync.

cated PM space.

6.3.1 Pmsync Comparison with EXT4-DAX

The ext4 file system has recently enabled support for direct access to the underlying device. By eliminating page cache, the DAX code removes the extra copy (page cache) by performing reads and writes directly to the storage device. For file mappings, the storage device is mapped directly into userspace. Ext4 file system has two modes of msync operation – SYNC and ASYNC. SYNC mode waits until all the changes are made durable to the media by the file system, while the ASYNC mode returns instantly without waiting for the msync to complete. None of the modes guarantee atomic durability of the mapped data.

Figure 6.5 shows the latency of performing `pmsync` on files ranging from 256KB to 8MB, where all the pages are made dirty by writing 8 bytes to them before invoking `pmsync`. Interestingly, `pmsync` performs better than ext4-DAX’s msync despite msync not providing any data consistency guarantee like region system for smaller flush sizes.

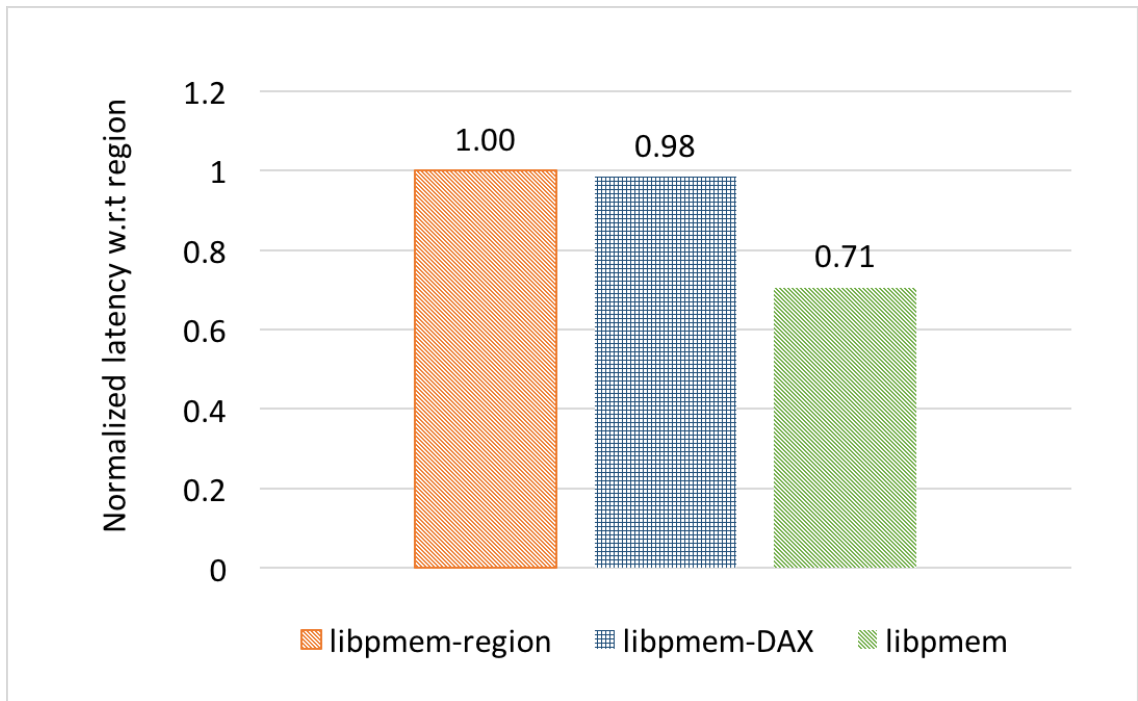


Figure 6.6: Normalized average libpmem and libpmem-DAX latency with respect to libpmem-region.

However, `pmsync` latency increases with number of pages to be flushed and `ext4-DAX` `msync` latency remains almost constant. We provide our reasoning for the slower performance for larger size in Section 6.3.3 For this benchmark, `pmsync` performs better than `msync` for up to 13248 dirty pages of a file sized 56MB.

6.3.2 Pmsync Comparison with PMEM.IO

PMEM.IO [pmea] is a suite of persistent memory libraries developed for making persistent memory programming easier. PMEM.IO's LIBPMEM provides an interface to map PM via the underlying PM specific file system (e.g., `ext4-DAX`). However, it implements PM specific functions such as `pmem_flush`, `pmem_drain`, etc. in user space without involving the underlying file system. LIBPMEM can also be configured to use `ext4-DAX`'s `msync` instead of the user level flushing functions. We call the former vari-

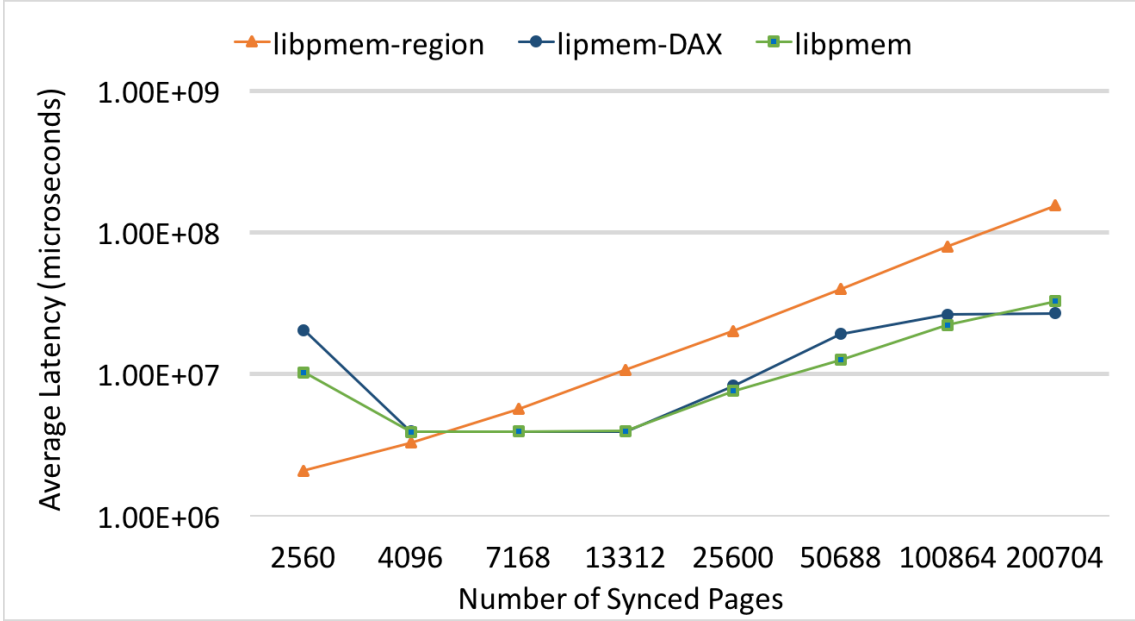


Figure 6.7: Comparison of libpmem, libpmem-dax and libpmem-region latency with respect to number of dirty pages.

ation as LIBPMEM and the latter variation as LIBPMEM-DAX. Neither variant provides support for transactional consistency of the data stored using LIBPMEM. We created a variant of LIBPMEM which is built to consume regions implemented by the region system and supports mapping ppages to the user address space. We call this variant LIBPMEM-REGION. By using LIBPMEM-REGION, pending data updates are made transactionally durable when using pmsync.

We compared the LIBPMEM-REGION which provides atomic durability of PM changes with the non transactional LIBPMEM and LIBPMEM-DAX. We ran the LIBPMEM library’s `pmem_flush` benchmark [pmeb] for all three systems. This benchmark writes a single byte to several pages of the mapped file/region and flushes the contents later. We ran the tests for different sizes of files/regions (12MB to 786MB) while `pmembench` executed multi-threaded (1 to 16). To determine how the region system performs on an average relative to these two variants we calculated their normalized average latency over the same dataset. Pmsync performs within 2% of LIBPMEM-DAX `msync`, and 30% of LIBP-

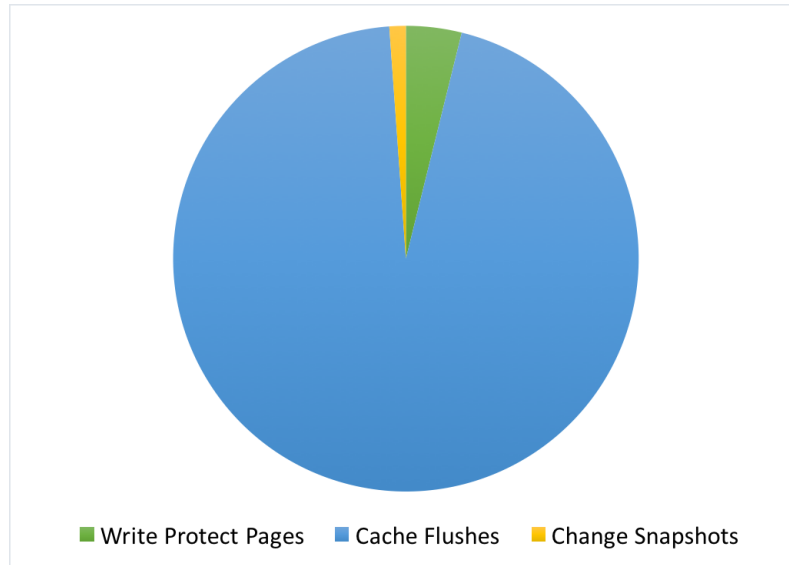


Figure 6.8: Pmsync breakdown.

MEM `pmem_flush`. The normalized average latency of LIBPMEM and LIBPMEM-DAX are shown in Figure 6.6 along with a comparison of latency with respect to increasing number of dirty pages in Figure 6.7. For an application that dirties a small number of pages between consecutive syncs, the performance of LIBPMEM-REGION is competitive relative to its less transactional variants.

6.3.3 Cost of Pmsync

To further investigate `pmsync` behaviour, we have done an analysis on the average time taken by the different sub tasks of the `pmsync` operation. In Figure 6.8, we have plotted the percentage of time taken for flushing the cache lines, to write protect the pages, and to change the snapshot pointer to the current data page for a `pmsync` operation on a region with 6400 dirty pages.

We can see that, cache flushing takes up 95% of the total `pmsync` time, which should be the common cost with `ext4-DAX msync`. However, the costs of write protecting the pages and changing the snapshots are associated with achieving atomic durability of the

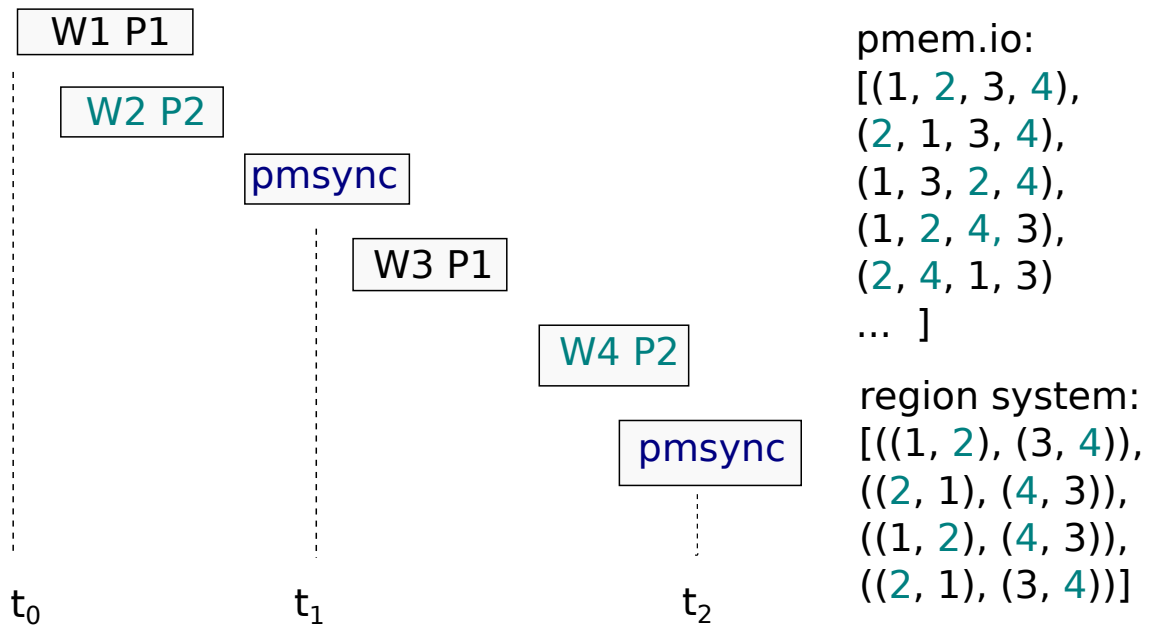


Figure 6.9: Possible write orderings for region system and PMEM.IO.

region. The *region system* has to do two additional passes on the dirty page list to ensure an *idempotent pmsync*. The first one is to write protect the ppages, the second pass to flush the cache lines, and the third pass to change the snapshots. As a result, the `pmsync` operation experiences a linear increase of cost directly proportional to the number of dirty pages. On the other hand, the total cost of cache line flushes may reach a plateau when the total size of dirty data is more than the CPU cache size. Unfortunately, *region system* can not avoid traversing the dirty ppage list thrice to ensure an idempotent sync operation.

Furthermore, the *region system* `pmsync` maintains ordering of writes across all the processors. The `pmsync` is designed in such a way that all the updates made after a `pmsync` call will be made durable only after the previous `pmsync` is completed and a consistent state is achieved. However, for DAX file systems as well as PMEM.IO-LIBPMEM, that is not the case. In Figure 6.9, we demonstrate how the writes from different processors can be intermingled between two invocation of `msync` in case of LIBPMEM. LIBPMEM only maintains ordering among the writes issued from a single CPU. On the other hand,

region system maintains a strict ordering of write between two different pmsync across multiple CPU's, which ensures writes from a later state (in this case t_1-t_2) can not corrupt a previously issued pmsync ($t_0 - t_1$) or vice versa. In case of *region system*, the writes that are issued during an ongoing pmsync operation are delayed until the pmsync is finished. Furthermore, the subsequent writes are made durable in strict order by virtue of page protection and copy-on-write mechanism. To achieve consistency, *region system* has to forgo durability of simultaneous writes and maintain strict sequential order across multiple CPU's. For this reason, pmsync performance is not as good as ext4-DAX or LIBPMEM when the number of dirty pages is high.

6.4 Summary

In this chapter, we discussed the *region system* implementation as a kernel module which can be installed alongside a patch of less than 100 lines of code. We have also examined the interaction between the *region system* operations and kernel entities. Besides that, we evaluated *region system* operations with mainstream ext4-DAX file system and PMEM.IO persistent memory library. In our findings, *region system* performs better than ext4-DAX for most of the persistent operations. Besides that, despite providing strict atomic durability guarantees, *region system-pmsync* performs competitively compared to PMEM.IO flush. In the next chapter, we introduce two versions of shared persistent containers built on top of *region system* which either provide simplicity of application development or flexibility of location independent mapping.

CHAPTER 7

USER LEVEL LIBRARIES

In the previous chapters we described the design of OS level support for integrating PM into the I/O stack via the *region system*. The region system uses a non-redundant metadata architecture to minimize cache flushes, the concept of regions that allows arbitrary and unordered allocation of PM, and finally an interface that supports shared and transparent atomic durability of mapped data. In this chapter, we discuss the shortcomings of contemporary designs that support persistent memory containers of data and present two solutions - LIBPM-R and LIBPMEMOBJ-R to achieve shared persistent memory containers with different degree of flexibility with regards to mapping the containers into the applications address space.

7.1 Introduction

The possibility of having a universally consistent view of persistent storage for multiple applications across multiple CPU's is yet unexplored due to the hardware barrier imposed by the traditional storage stack. Memory, being the temporary placeholder for the data brought in from disk, only provides a local view of the data to the applications which might be inconsistent with the on-disk version. On the other hand, persistent memory (PM) devices can be mapped directly to the applications address space and could be accessed simultaneously by the CPUs with the changes being transparently visible across the system. The composite nature of the PM devices allowing byte-addressable-persistence opens the door for *transparent sharing* of durable data. It is thus timely to investigate the possibility of consistently sharing persistent objects in real time across multiple applications which could alter the way applications work today and how future applications get built. Now that the hardware limitations are overcome, we envision that in the near future applications would be willing to share large data structures residing

in PM to minimize persistent memory usage while maintaining consistency of the data across the applications.

Contemporary persistent containers or object stores [CCA⁺11, VTS11, pmea, Mar17] provide support for working with in-memory persistent objects. Specifically PMEM.IO, a persistent memory library supported by Intel, has been focusing on developing primitives for programming using PM. PMEM.IO's transactional object store – LIBPMEMOBJ – converts EXT4-DAX persistent files into persistent object stores along with the support for annotating persistent objects, operations, and transactions. However, there are a few drawbacks of the system. First, the developers have to carefully annotate all the persistent operations including operations that are part of an atomic transactions. Second, in some cases one operation on an object could internally require a sequence of multiple operations on the data structure before a desired consistent state is reached. An insertion or deletion of a node into an AVL tree would be such an operation where multiple additional nodes have to be rearranged inside the parent data structure. In such cases programmers have to rewrite the complex data structures to capture such occurrences and to retrofit the PM library, whereas it would have been easier to have a single instruction to commit all the changes at once. SoftPM [GMC⁺12], and subsequently LIBPM [Mar17] overcomes the complexity of specific annotation of objects by providing an easy-to-use persistent container model with atomic persistent sync support. However, both of them are limited by the fact that these containers can not be shared by different processes simultaneously.

Based on the experience of consistently sharing PM regions [CR17], we explore the possibilities of sharing persistent containers across multiple applications consistently and simultaneously. In this chapter, we propose two solutions for consistently sharing PM containers or object stores across multiple applications based on the shared atomic durability capabilities of *region system*. First, we propose LIBPM-R and show that persistent containers can be maintained consistently with the simplicity of memory like transac-

tions alongside the ability to share the containers transparently among multiple processes but with the restriction that the containers are mapped to the same virtual address within each applications address space. Second, we propose LIBMEMOBJ-R, and demonstrate that with *region system* support PMEM.IO persistent object stores can be shared across multiple processes, and additionally overcome the fixed map constraint of the previous solution.

7.2 Background

In this section, we discuss different sharing techniques used by the contemporary OS subsystems, and briefly re-examine transparent sharing and shared atomic durability of mapped data.

7.2.1 Contemporary Sharing Mechanisms

Sharing data helps to reduce pressure on storage and memory. Consider a situation where the C library is not shared and has to be loaded separately to the memory for every program that is using it. That would be a disaster. The same rule applies for storage – when multiple applications read from a file, multiple copies are not generally mapped to the physical memory. However, file and memory interfaces achieve sharing differently.

File Backed Mappings

File backed mappings mirror the contents of an existing file. The contents of those pages are initialized by reading the contents of the file being mapped, at the appropriate offset for that page. Writes to file mappings created with the MAP_SHARED flag update the page cache pages, making the updated file content immediately visible to other processes using the file, and eventually the cached pages will be flushed to disk, updating the on-disk

copy of the file. Flushing the content of cached pages can be forced by issuing a `msync` call. However, the cache being flushed to the storage is not guaranteed to be atomic as some of the cache pages from a set of changes may reach the storage while others reside in memory when a failure occurs.

Writes to file mappings created with the `MAP_PRIVATE` flag result in a copy-on-write operation, allocating a new local copy of the page to store the changes. These changes are not made visible to other processes, and do not update the on-disk copy of the file.

Anonymous mappings may be created by passing the `MAP_ANONYMOUS` flag to `mmap()`. Anonymous memory is a memory mapping with no file or device backing it and therefore not persistent at all. Instead, it provides a mechanism for programs to allocate memory from the operating system to implement temporary structures such as a process' stack and heap.

Sharing Memory

The Unix processes share the address space of the parent process when a child is forked. This mapping is mostly copy-on-write which helps speedier creation of processes. However, Unix threads share heap data and other shared libraries. Normally the shared heap is not saved after the threads complete their execution. PM based heap changes this assumption and opens new dimensions for the applications to achieve shared data persistence at the application level.

7.2.2 Transparent Sharing

In conventional block-based storage, the applications need to communicate between themselves to get the latest updates to a shared file. One way to do that is to issue `msync()` synchronously and notify the concerned applications to remap the file after the comple-

tion of the synchronization operation. However, for DAX mapped PM where the PM is directly mapped to the applications address space, any updates should be visible across the shared applications immediately. We characterize this outcome as real-time transparent sharing of persistent storage.

7.2.3 Shared Atomic Durability

As we have discussed in Chapter 5, while direct CPU access to the PM can easily achieve transparent sharing, it provides no support with regards to maintaining PM consistency and thus is practically useless for sharing by itself. The region system [CR17] supports atomic durability of mapped data by propagating all the in-flight changes to the PM all while keeping the mapped region isolated from further modifications during this process by using Inter-Processor Interrupts (IPI). It also implements a novel dual-pointer technique to manage a consistent snapshot version of the current mapping in the address space. The region system supports a universal access usage model where any process can make changes to the mapped region at any given time, and “all” modifications across the applications are made durable if any of them decides to persist the changes.

7.3 Persistent Containers

The need for application data persistence and reusing the data without requiring serialization lead to multiple solutions [CCA⁺11, VTS11, GMC⁺12, pmea, Mar17] which offer persistence of software data for traditional storage as well as PM. However, the proposed solutions do not recognize the concept of sharing for persistent object stores nor do they provide support for it.

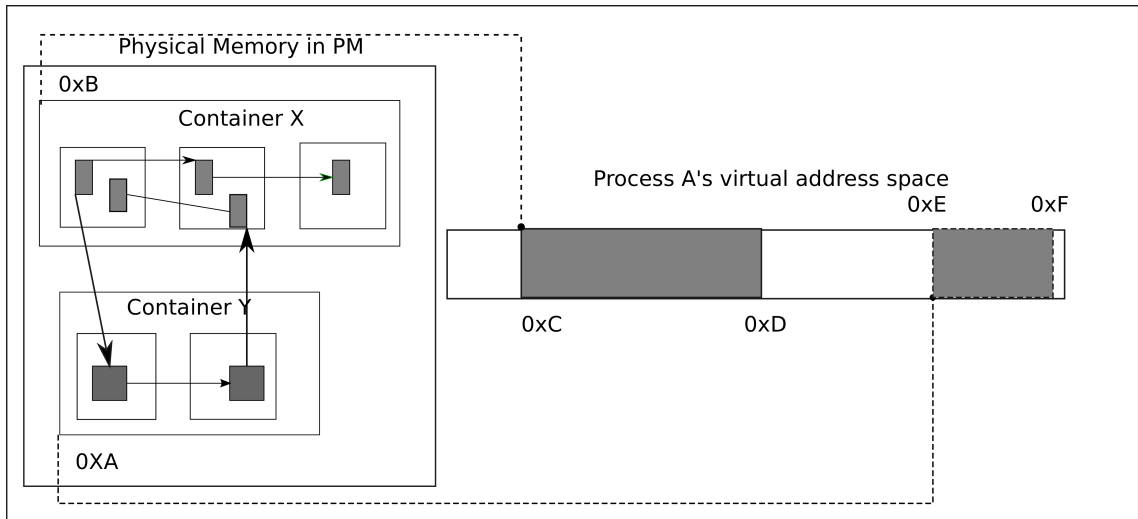


Figure 7.1: Mapping two persistent containers to Process A's address space.

7.3.1 Challenges of Sharing Persistent Containers

Though sharing object stores among multiple application address spaces appeared infeasible due to the hardware limitations until now, with the addition of PM to the IO stack, it will be possible for multiple applications to access objects from a persistent container simultaneously. However, there are a few obstacles in sharing persistent containers. First, when the containers are mapped to a process address space it is possible that the mapped address in the process virtual address space would not be same as the virtual address where the container was created. In this case, while mapping, the pointers to in-container objects need to be updated to match the target virtual address space of the process. All the objects need to be traversed, and the pointers have to be fixed before the control is handed back to the application.

Second, the containers might have pointers to other objects either in the same container or to a different container. In both the cases, if the containers are not mapped to the same location in different applications, the pointers will be corrupt. In the latter case, all the inter-related containers (which have pointers across each other) have to be mapped at the exact same virtual addresses within all applications at all times. Figure 7.1 depicts

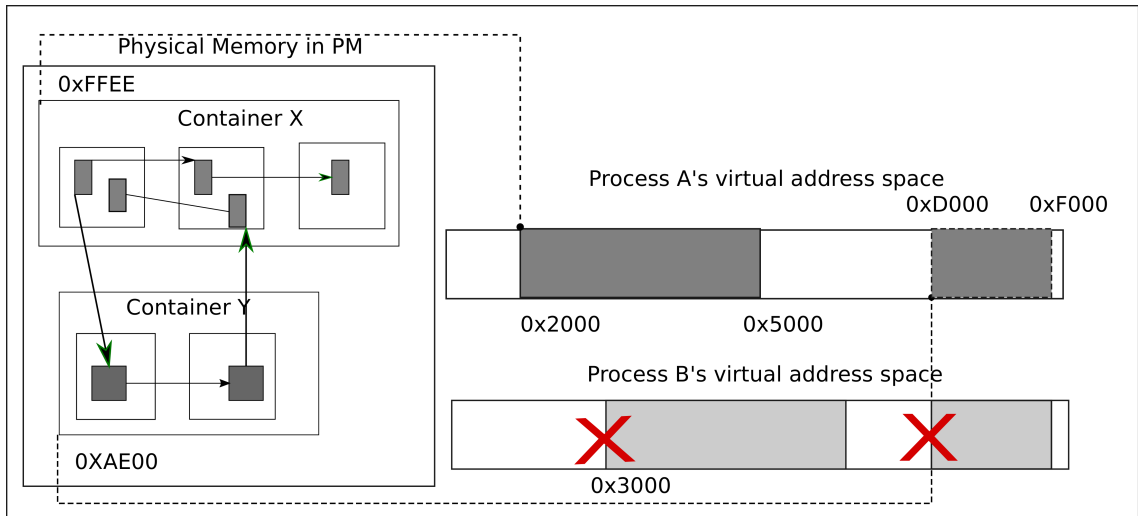


Figure 7.2: Mapping two persistent containers to Process B’s address space fails.

two containers *X* and *Y*, where *X* has some outgoing pointers to the *Container Y* besides in-container pointers. *Container Y* has one outgoing pointer to *Container X*. As shown in Figure 7.2, when the same virtual address of process B is not available to map *Container X*, even if *Container Y* is free to map at the desired address, the entire mapping can not be complete because of the dependency.

Next, we explore two possible solutions to achieve shared persistent containers with regions. First, we present LIBPM-R, which supports shared persistent containers with a simple memory like interface. Second, we propose LIBMEMOBJ-R which is built on the support provided by the *region system* interface to overcome the limitation of LIBPM-R, although providing a less developer-friendly interface.

7.4 LIBPM-R: Fixed Map Shared Containers

We propose a persistent container library - LIBREGION - which facilitates the sharing of object stores. LIBREGION exports API’s similar to SoftPM and LIBPM [GMC⁺12, Mar17], and provides a simple transactional model in comparison to the other [VTS11,

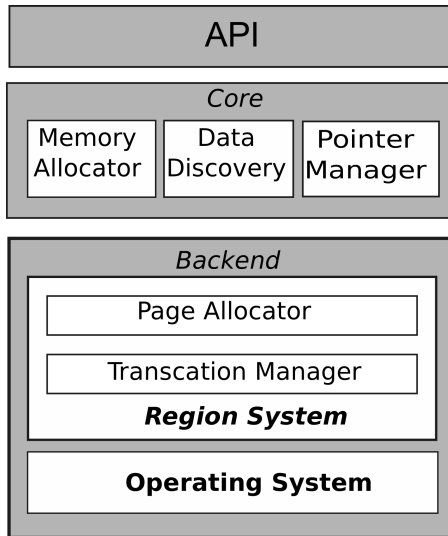


Figure 7.3: LIBPM-R Architecture.

pmea, CCA⁺11] solutions. These containers present a powerful abstraction for the developer who simply identifies a single top-level container root data structure to persist. All data reachable from this top-level data structure is automatically discovered and atomically persisted by the library infrastructure. The library uses regions as the data store rather than using regular files. The `pmsync` operation exported by the region system kernel subsystem is issued while committing changes to the object store. This usage model allows applications to make a batch of changes and then commit all changes at once rather than flushing and logging every modification to the container.

7.4.1 Architecture

LIBPM-R is built around the design principles of SoftPM [GMC⁺12]. Figure 7.3 presents a simplified logical diagram of the LIBPM-R architecture. The core provides the support for container memory allocations, pointer management, and data discovery. The core component does not need any effort in terms of transaction management due to the transactional consistency support provided by *region system*. We present a short description of

Category	API	Description
Container Management	open(...) close(...) clear(...) delete(...)	Open / create container. Close open container. Clear container contents. Delete all container data and metadata (persistently).
Memory Allocation	pmalloc(...) pfree(...)	Allocate persistent object in container. Free persistent objects.
Transactional Updates	commit(...) rollback(...)	Atomically apply all pending updates to the container. Undo all updates to the container.
Annotation	setroot(...) getroot(...) exclude(...)	Set object as the root of the container. Get address of root object. Do not follow pointer during container closure.

Table 7.1: The LIBPM-R API.

API's followed by brief discussion on the different components of LIBPM-R. The API table presented in Table 7.1 provides a concise summary of the LIBPM-R functionality. The three groups containing memory allocation, container management, and annotation provide support spanning from complex object management and namespace management to annotation support for containers. The transactional support for commit and rollback are directly linked to the *region system* interface depicted in Table 4.2 .

7.4.2 Shared Atomic Transactions

LIBPM-R's transactional mechanism provides shared atomic durability of transactions across multiple processes. The transaction model works at a page granularity and all the implementation heavy lifting is performed by the *region system* at the OS level. LIBPM-R has two types of segments – metadata and data. The data segments consist of ppages allocated from a region. The containers and regions have one-to-one mapping, which means all the data segments of a container are part of the same region.

Figure 7.4 illustrates LIBPM-R's handling of updates to containers starting from a recently opened container using segments of size three pages. The container is opened by

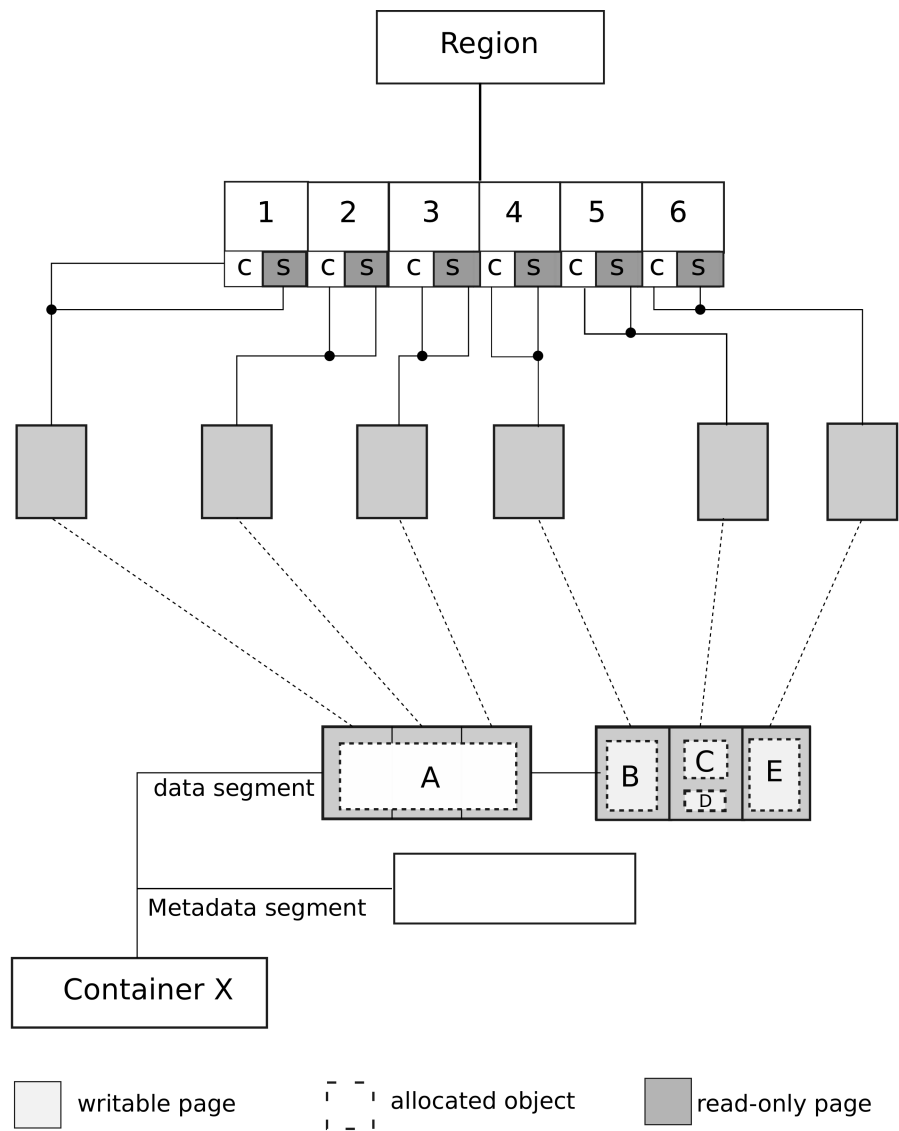


Figure 7.4: LIBPM-R transaction: Container state after a successful open. Pages are write protected.

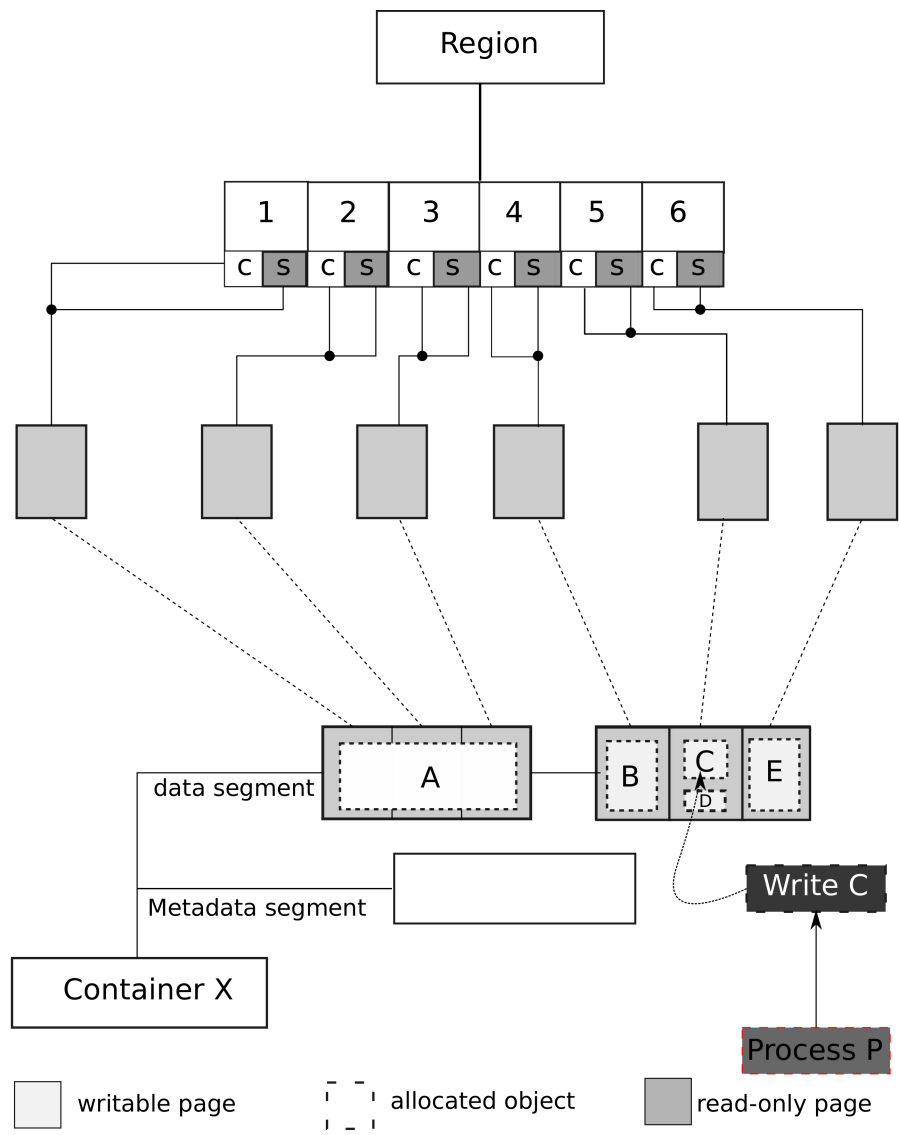


Figure 7.5: LIBPM-R transaction: The application tries to update object C which is mapped read-only.

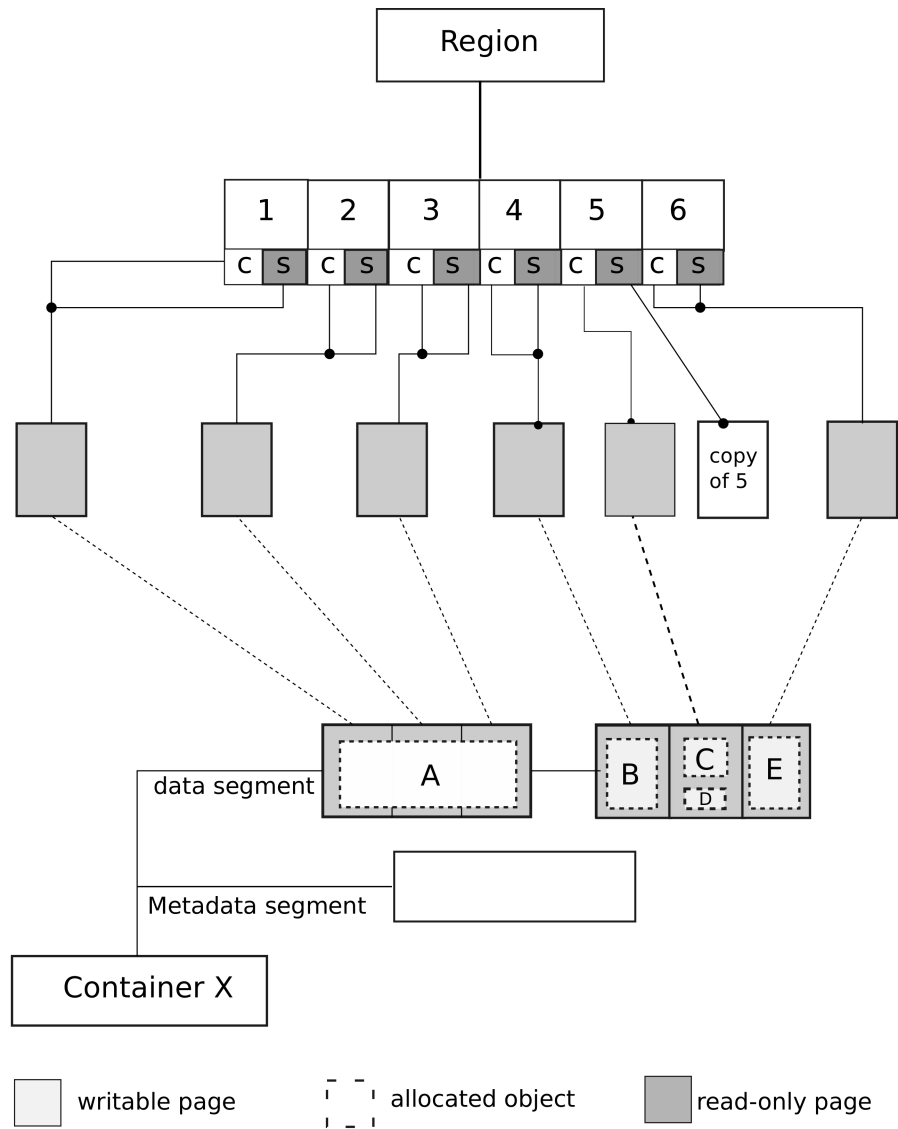


Figure 7.6: LIBPM-R transaction: The update triggers the fault handler in the region system, which initiates the write by making a copy of the page containing the faulting address.

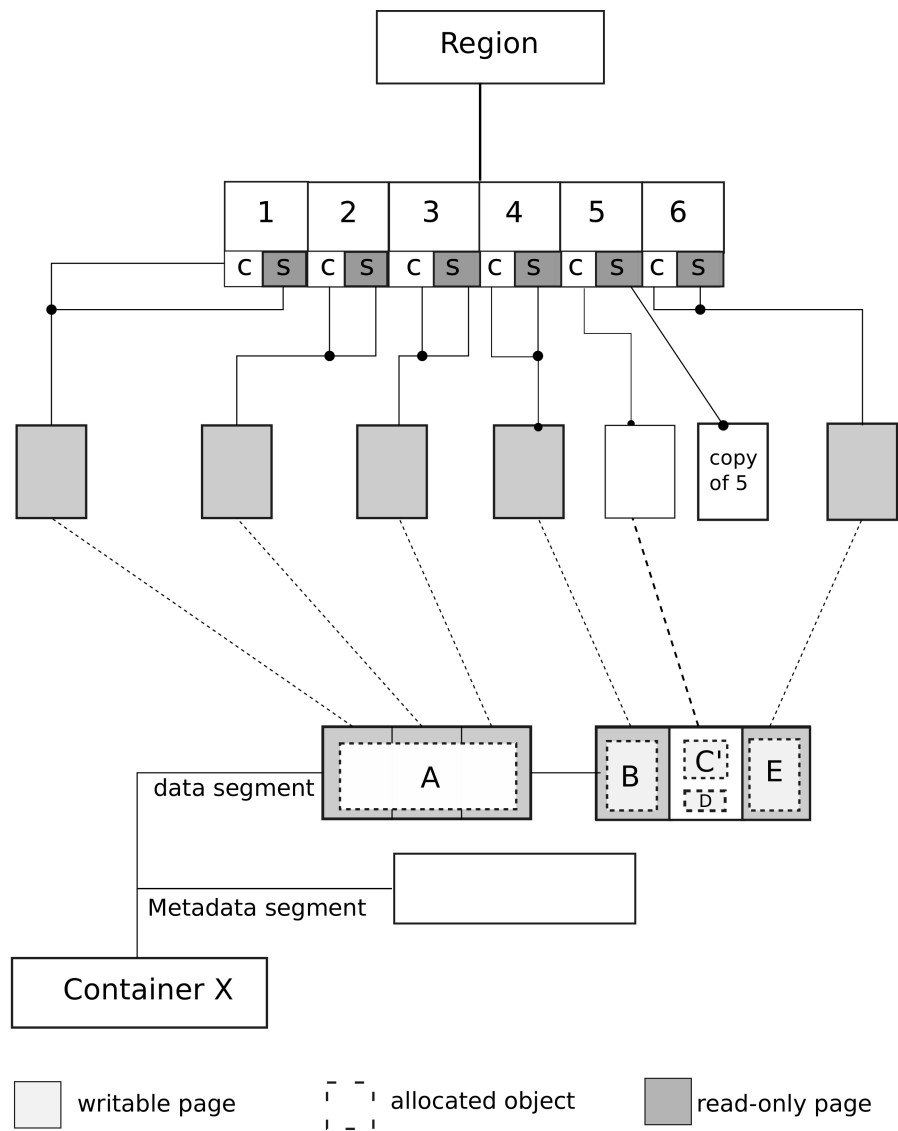


Figure 7.7: LIBPM-R transaction: The page is granted write permission, and the write to C goes through.

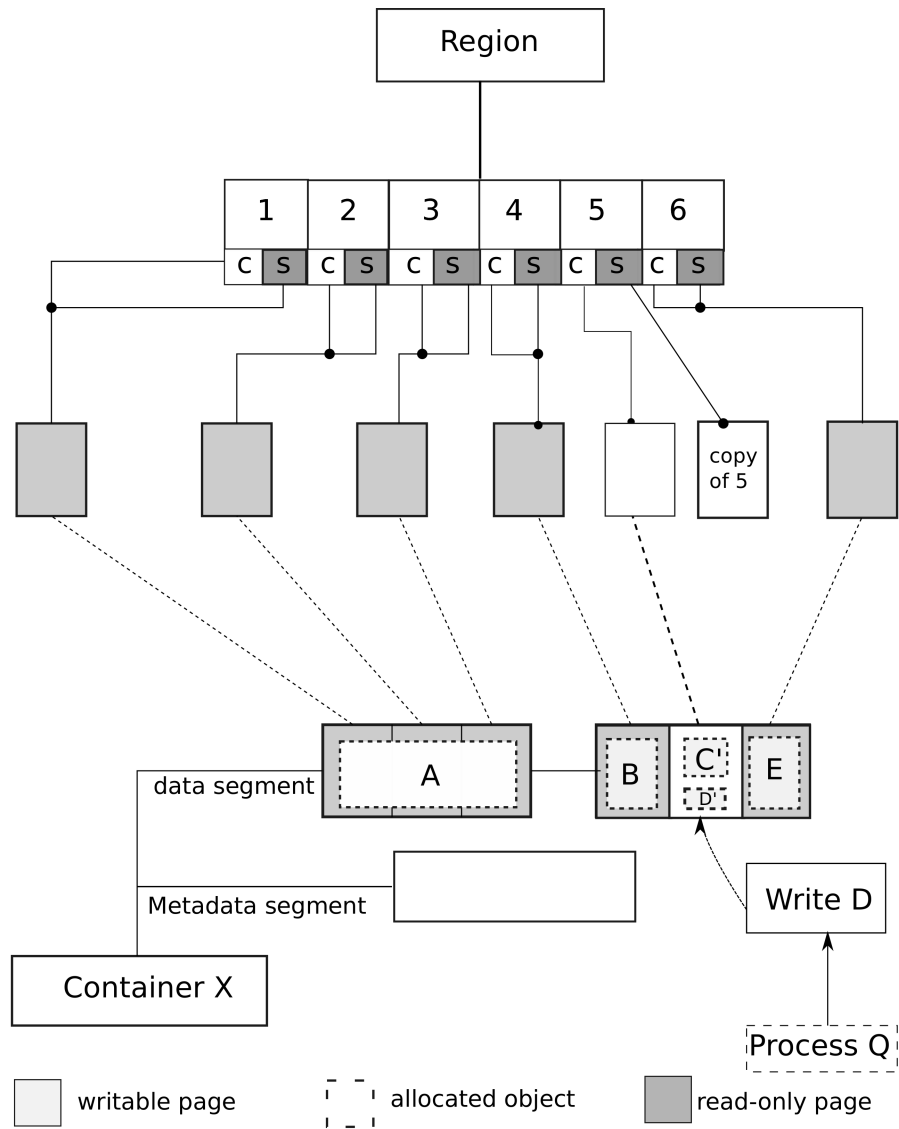


Figure 7.8: LIBPM-R transaction: The write to *D* takes place without any overhead.

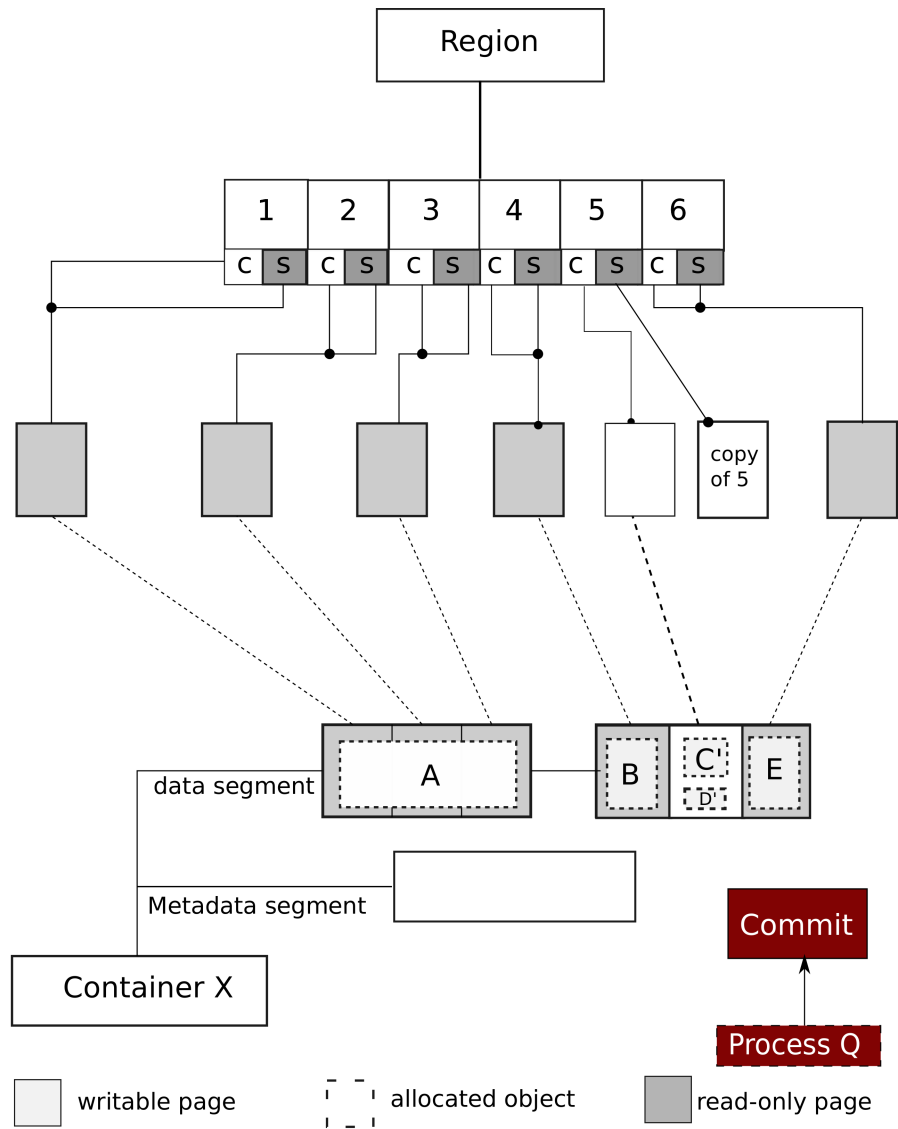


Figure 7.9: LIBPM-R transaction: A commit is issued.

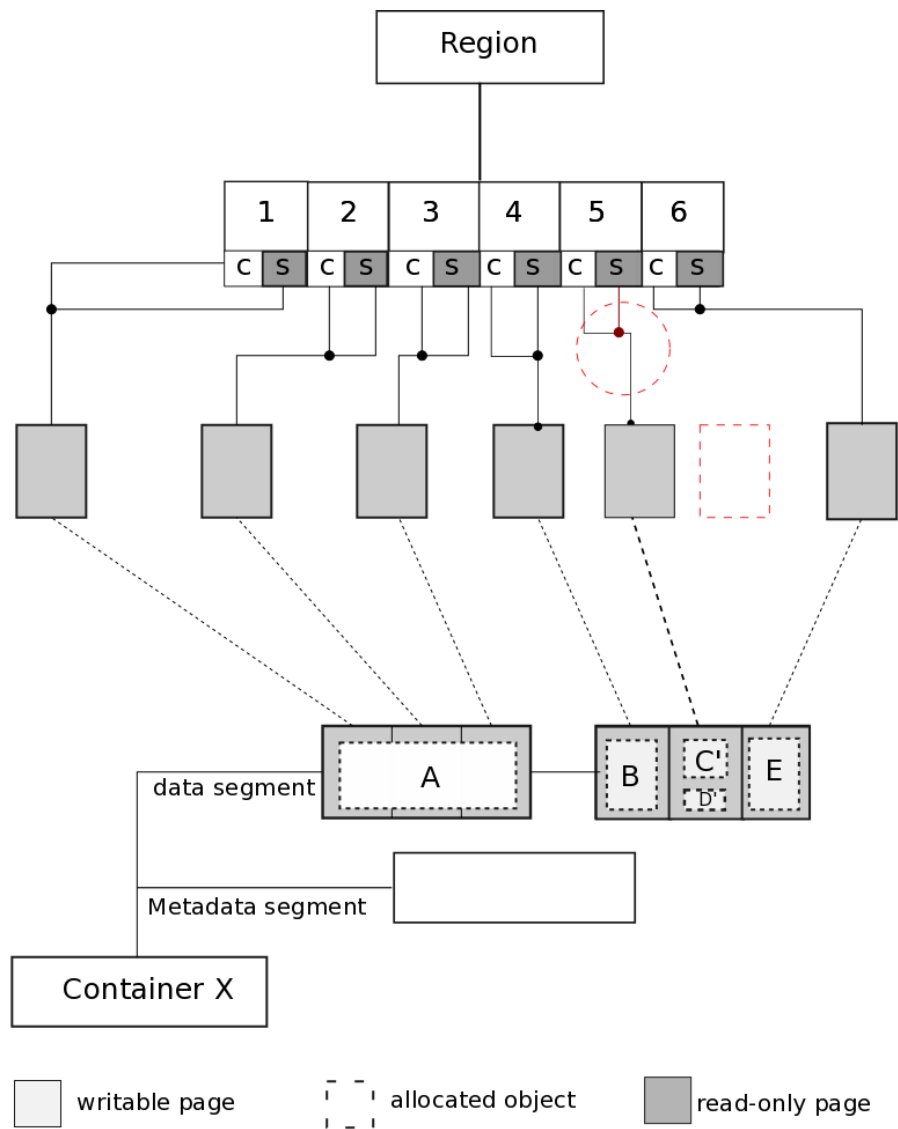


Figure 7.10: LIBPM-R transaction: The page is write protected, and the snapshot pointer now points to current page, and the old snapshot page is deleted.

two processes P and Q ; LIBPM-R maps the container at the same location in the address space of both the processes. When a container is opened, all data segments are write protected. First, process P tries to update an object C [Figure 7.5]. Since the page hosting the object is write protected, *region system* takes control of the write process. Then the entire page gets copied and the snapshot pointer is updated to point to the copied page [Figure 7.6]. Finally, the page is made writeable [Figure 7.7] and the update proceeds. Subsequent updates to the same page by process P do not trigger additional faults and can proceed without any overhead [Figure 7.8]. During a commit by process Q , the current page is made read-only and the old snapshot is deleted after the snapshot pointer is updated to point to the current page [Figure 7.10].

The key to achieving shared atomicity of transactions is that, the state of the pages are maintained by the *region system* rather than the user level library. The *region system* module always maps the *current* pages to the application's address space, and the mapping never changes even when the page permissions are changed. Additionally, as the snapshots are maintained by the *region system*, there are no local copies for each process ensuring that all the processes look at the same page at any given time.

The limitation of LIBPM-R lies in the fact that the containers need to be mapped at the same location across all the process address spaces as the LIBPM-R library uses a fixed pointer model. In other words, we can say that, LIBPM-R provides supports for fixed map shared containers. However, as the OS is designed to return random addresses to prevent security threats to the system, the desired mapping to a specific address might fail. This restriction can be overcome by providing hints for mapping and disabling Address Space Layout Randomization (ASLR). The former is successful in most of the cases in a 64 bit architecture, however disabling ASLR will impose security threats to the system. In the next section, we discuss how we can achieve location independent shared containers using fat pointers.

7.5 LIBMEMOBJ-R: Location Independent Shared Containers

Persistent memory pointers can be managed in several ways. LIBPM-R deals the pointers as fixed map pointers. Other techniques like pointer swizzling can also be used to manage persistent objects where the programmer provides serialization methods for the data types. More relevant to our discussion are the *fat pointers* – which do not store the actual addresses but provide a mechanism to calculate the target addresses from the stored information. The downside of the fat pointers is that they need to be dereferenced at runtime. However, fat pointers can be used to achieve location independent shared containers. To overcome the limitations of fixed-map shared containers, we identified that if the pointers can be stored as an offset to the root of the container, then the applications are free to map the containers at any location available. Conveniently, LIBMEMOBJ does something similar.

7.5.1 Architecture

LIBMEMOBJ is part of the PMEM.IO nvmdk suite [pmea]. It presents interfaces to create *persistent memory pool*, in other word persistent containers. Rather than storing absolute pointers, it stores PMEMOID's, which are a combinations of the `id` of the container and `offset` from the root of the container for each object. The library provides some mechanism to store and lookup virtual addresses from the stored `PMEMoids`.

```
1 struct pmemoid {  
2     uint64_t pool_uuid_lo;  
3     uint64_t off;  
4 };
```

Listing 7.1: PMEM.IO's definition of persistent pointer.

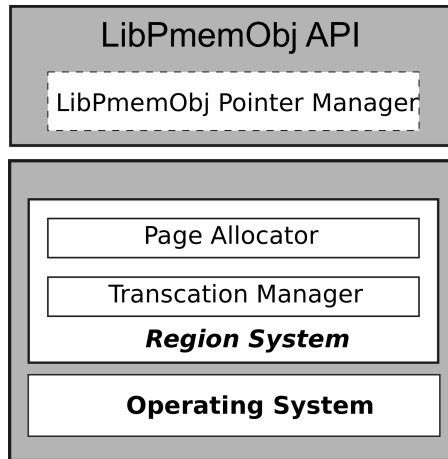


Figure 7.11: LIBPMEMOBJ-R Architecture.

However, the consistency method used by PMEM.IO is neither failure atomic nor does it support consistent shared mappings. We replaced the consistency primitives of LIBPMEMOBJ, and redesigned it to work with regions instead of files. Consequently, the LIBPMEMOBJ-R library now supports shared atomic durability inherited from region system. The method of persisting objects is changed to batch durability from a per-object persist operation requirement. Rather than persisting each modification to the container, applications now have the flexibility to decide when to make the containers durable by communicating between themselves. Figure 7.11 presents a simple schematic diagram of the modified architecture.

7.6 Analysis of Performance

PMEM.IO provides a set of persistent data structures with transactional support, which we ported to LIBPM-R and we compared the latency of each major operation. The data structures we considered were the following:

Hashmap A hashtable that deals with collisions by linking entries together in the same bucket.

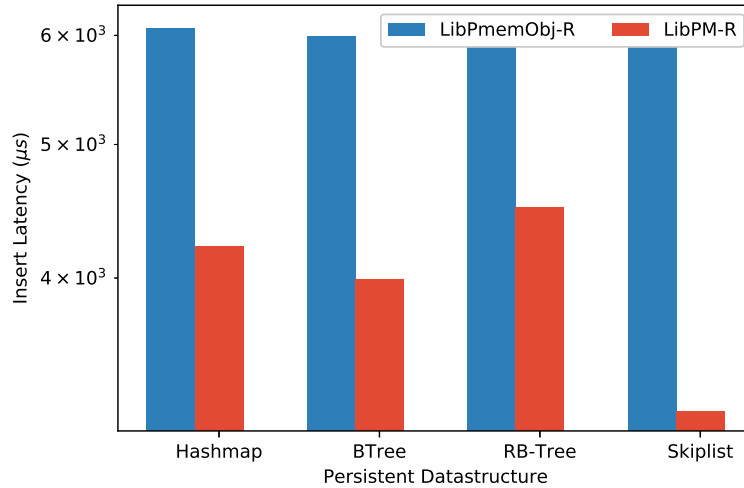


Figure 7.12: LIBPMEMOBJ-R vs LIBPM-R (Inserts).

BTree An in-memory version of a regular BTree with branching factor of eight.

RBTree A standard Red Back Tree (RBTree).

Skiplist A skiplist with a single value per node and eight levels.

The ease of use and simplified development complexity of fix mapped persistent containers over fat pointer based containers are well established by LIBPM [Mar17]. In our experiments, we tried to verify if the same data structures can be mapped by different processes. In all our experiments, same persistent containers were safely and consistently shared across multiple applications.

Each one of these data structures can create a mapping between a *64bit* integer and a value of an arbitrary size. Figure 7.12 depicts the results of comparing the insert operations for all persistent data structures for both LIBPM-R and LIBPMEMOBJ-R. We see that LIBPM-R outperforms LIBPMEMOBJ-R for every data structure consistently for all the operations from 1.2x to 1.9x. The delete operation in Figure 7.14 also show similar trend where LIBPM-R does better than LIBPMEMOBJ-R by 1.3x to 1.8X.

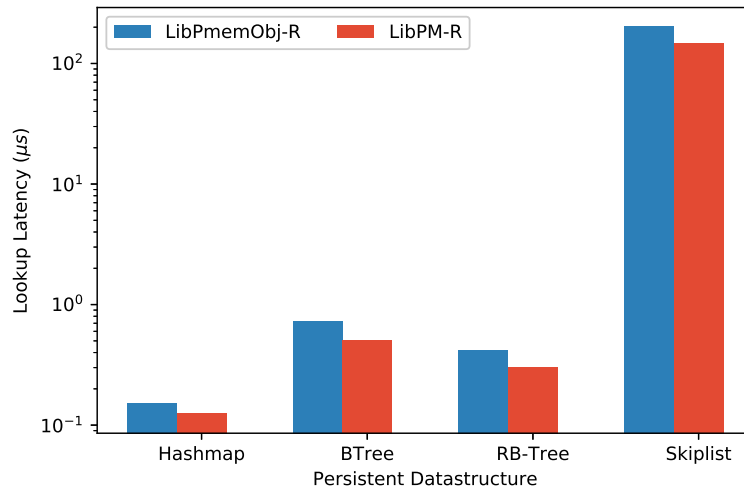


Figure 7.13: LIBPMEMOBJ-R vs LIBPM-R (Lookups).

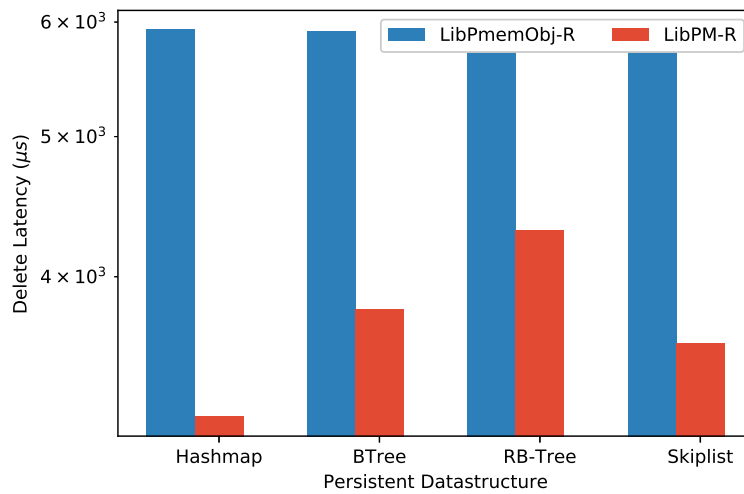


Figure 7.14: LIBPMEMOBJ-R vs LIBPM-R (Deletes).

7.7 Summary

In this chapter, we first described the challenges in achieving shared persistent containers. Next, we proposed two shared persistent container solutions based on fixed pointers and fat pointers. Finally, we analyzed that fixed map containers outperform fat pointer based containers by a small margin. In our experience, we find that LIBPM-R provides simpler transactional model and comparative ease of application development. However, LIBMEMOBJ-R, with consistency guarantees of *region system*, provides a safer environment for development. Persistent memory application developers can choose any of these two solutions to achieve shared persistent containers depending on the requirements of the product they are developing.

In the next chapter, we talk about contemporary PM solutions related to our work, and finally conclude our dissertation.

CHAPTER 8

RELATED WORK

In this section, we look at the PM-centric solutions for a better understanding of the scope of the research.

Research on PM-centric software stacks has mainly addressed two areas: (i) application usage of the persistent memory, and (ii) native OS support for PM. As we discuss below, these two classes of solutions have been developed as silos and solutions that span the concerns of both areas remain unexplored. While we discuss individual solutions in the remainder of this section, Table 8.1 summarizes the recent research in this space evaluated against several requirements of PM-specific software solutions.

8.1 Application usage of PM

The work on application usage of persistent memory has focused on new programming abstractions and models [CCA⁺11, VTS11, GMC⁺12]. NV-Heaps [CCA⁺11] and Mnemosyne [VTS11] propose persistent heap-based solutions for PM. Both utilize the *mmap()* system call to provide the abstraction of persistent heaps. NV-heaps [CCA⁺11] are treated as regular files internally within the PM-mounted file system. On the other hand, Mnemosyne [VTS11] introduces *persistent regions* mapped directly to the address space of processes; a user level library explicitly manages the region-file mappings for each process. The persistent memory library effort [pmea] provides object-based transactional support (`libpmemobj`) built on top of low level persistent memory library (`libpmem`). All these solutions rely on memory transactions to make PM updates atomic and durable. Thus, while these solutions support consistent updates and provide transactional consistency, they require that applications explicitly specify individual updates to PM, a cumber-

Table 8.1: A summary of recent research on PM-specific software solutions

Class	Solution	Persistent namespace	Arbitrary and unordered allocation & deallocation	User space mapping	Mapped data consistency Guarantee	Mapped data consistency mechanism	Consistency implementation	sharing support
File systems	BPFS [CNF ⁺ 09]	Y	N	N	N	-	-	N
	SCMFS [WR11]	Y	N	Y	N	-	-	N
	PMFS [DKK ⁺ 14]	Y	N	Y	N	-	-	N
	NOVA [XS16]	Y	N	Y	Y	duplicate page mapping	copy-on-write	N
	HVMFS [ZHL ⁺ 16]	Y	N	Y	N	-	-	N
	FCFS [OS16]	Y	N	Y	N	-	-	N
Persistent Heaps	Mnemosyne [VTS11]	Y (File backed)	Y	Y	Y	Transactional	Compiler support + User level library	N
	NVHeaps [CCA ⁺ 11]	Y (File backed)	Y	Y	Y	Transactional	User level library	N
Replication	Mojim [ZYMS15]	Y (File backed)	N	Y	Y (replicated) N (unreplicated)	Mirroring based atomic msync (region-wide) & gmsync(multi-region)	OS	N
Atomic Msync	Msync [PKS13, VMP ⁺ 15]	Y (File backed)	N	Y	Y	File-wide atomic msync	File-system specific	N
Block devices	Moneta+ [CDC ⁺ 10, CME ⁺ 12]	Y (combined with file systems)	N	N	-	-	-	-
	PMBD [CMH14]	Y (Combined with file system)	N	N	-	-	-	-
Region System	Regions	Y	Y	Y	Y	Region-wide atomic pmsync	OS	Y

some task. Finally, there is lack of support for sharing data stored in PM across processes in these solutions.

8.2 Native OS support for PM

The second class of solutions address optimizing the current OS stack for PM [CDC⁺10, CME⁺12, WR11, CNF⁺09, DKK⁺14, YMH12, LBN13, BCLN13, ZYMS15, ZHL⁺16]. These solutions fill critical gaps in the software stack as we discuss below. However, none of these solutions empower applications with simple-to-use consistency primitives for updates to PM-mapped areas of the application's address-space.

8.2.1 PM as a block device

PM can be exported as a block device to provide backwards compatibility with existing storage stacks [CMH14]. Pointing out the software stack overheads, Moneta [CDC⁺10] proposes a optimized software stack to use PM as block device, while Moneta-D [CME⁺12] improves the performance further by carefully eliminating OS interference during an IO operation. However, at a fundamental level, the interface is not ideally suited for PM because it does not support byte-addressability. Application reads of a few bytes, that can be efficiently supported by PM, get translated to much larger block granularity accesses to the device; additionally, writes may incur read-modify-write overheads [CLK⁺15].

8.2.2 File systems

Recent file systems such as PMFS [DKK⁺14], BPFS [CNF⁺09], SCMFS [WR11], HVM-FS [ZHL⁺16], FCFS [OS16] provide PM-specific file management solutions. These solutions all provide persistent namespace and support mapping portions of files to pro-

cess address spaces. BPFS supports durable consistency guarantees in the form of epoch barriers [CNF⁺09] but does not allow application access to PM mapped areas directly. Neither PMFS nor SCMFS guarantee the ordering or consistency of updates to the PM pages mapped to the process address space. FCFS provide applications failure consistency by using transactional mechanism to encapsulate file system writes to the PM using the `write()` syscall into transactions. HVMFS proposes a snapshotting mechanism primarily triggered by `fsync` for keeping multiple consistent versions of a file. The Nova file system [XS16] has addressed minimizing the cache flush requirements by optimizing the log, while the work of Chen *et al.* does so using fine grained metadata journalling [CYW⁺16]. However, none of the above-discussed PM-specific file systems support arbitrary and unordered allocation/deallocation as well as mapped data consistency.

The Nova [XS16] file system proposed support for atomic-mmap with strong consistency guarantees. However, the atomic-mmap primitive does not present actual file pages to application address space as it maps copies of the persistent pages. Upon invocation of `msync`, a non-temporal write operation is performed to write to the original pages from the “copy” pages. Though this version of atomic-mmap achieves consistency, it forfeits the basic advantages of sharing mapped pages across multiple applications. As the mapped pages are not actual physical pages, it disables multiple applications to share and have a coherent view of the page. When actual physical pages are mapped by different applications running in different cores concurrently, processor snooping mechanism can make sure that all the CPU caches and the applications have a coherent view of the shared pages – which is not possible in this case.

8.2.3 Memory Mapping

Though byte-addressability makes PM devices readily accessible by CPU load/stores, few solutions address consistent access to PM mapped areas. Mojim [ZYMS15] proposed a set of replication schemes with varying degrees of reliability and consistency guarantees. Mojim exposes file-backed PM regions to the user address space for direct loads and stores. A system configuration file is used to specify the *Mojim regions* that can be replicated across mirror and the backup nodes. The OS service provides the applications with simpler interfaces for creating *sync points* for the data area by calling specific system calls (such as `msync` or `gmsync`). Mojim provides availability, reliability, and consistency guarantees via replication to a peer node. However, no consistency guarantees are provided for the application-mapped data areas for the general un-replicated case.

Other researchers have explored the consistency of memory-mapped data. The importance of failure-atomic `msync` when using file systems has been articulated well by Park and Verma *et al.* [PKS13, VMP⁺15]. Though their proposed solutions are independent of the underlying storage type, they are designed for specific file systems. However, safely mapping shared PM data within multiple processes cannot be dealt by the applications on their own. At any consistency point, updates to the PM need to be reflected to all the different user-space mappings for that portion of data. To that end, in designing the *region system* we seek to provide native support to ensure that synced data is shared transparently across multiple mappings.

8.2.4 Other PM-optimized OS features:

OS behavior has been revisited for PM devices by several researchers. Due to the reduced latency of the PM devices, synchronous polling mechanism has been suggested for better performance over interrupt driven I/O. [YMH12]. Significant improvement in I/O perfor-

mance is also offered by optimizing the buffer cache [LBN13]. Baek *et al.* [BCLN13] propose a unified memory hierarchy which supports file objects and memory objects residing on the PM device as well as interchanging the objects without copy overhead. This allows the applications to associate a namespace to a memory object by converting it to a file and vice-versa. However, the solution does not support atomic durability of multiple updates to the memory objects.

8.2.5 PM optimized architectures and data structures

A lot of work has been done on how to combine persistent memory into the current software stack [LIMB09, ZZYZ09, QSR09, WZ94]. Apart from that, significant effort has been focused on memory persistence [PCW14, DSST89, CNC⁺96, NH12, SMK⁺93, LSS16]. These systems focus on either part on whole system persistence with the help of persistent memory. Integrating persistent memory introduces a lot of new questions on how to maintain a consistent system. Researchers have analyzed the effects of ordering [CSADAD12, CI08], logging [KKB⁺16, CCV15], and proposed solutions to achieve crash consistency [RZK⁺15, SKB⁺17, PKS13]. Other group of research focus on PM optimized data structures such as B+ tree [CJ15, CLX14, LLS13, OLN⁺16, YWC⁺15, HKWN18, MAK⁺13], radix tree [LLS⁺17], other generalized data structures [VTRC11].

8.3 Summary

In this section we looked at the different classes of contemporary solutions for consuming PM. Our finding is that the consistently sharing PM data among applications is not considered as a viable option because of absence of proper solutions. In our dissertation, we have focused on bridging the gap and enabling applications with the memory like usage model by supporting consistent sharing of PM data by introducing the *region system*.

While doing so, we have reduced the cost of PM interaction by eliminating redundant metadata from the system. Finally, we proposed two viable alternatives for achieving shared persistent containers with the help of the *region system*.

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

Persistent memory enables a new breed of applications — applications that are both stateful and extremely high performant. Current operating system abstractions are not well-suited to deliver the capabilities of this new class of memory-like storage devices to applications whereby developers experience the simplicity and flexibility of the memory interface and the durability of storage. Consequently, developers today must either compromise on transactional durability properties of weaker abstractions or take on significant additional development complexity to build these capabilities themselves.

In the previous chapter (Chapter 8), we have presented a comprehensive summary of the existing solutions that tackle the problem of PM integration into the I/O stack. The existing solutions concentrate on different classes of applications but fail to guarantee atomic durability of mapped data in a multi-processor system. Besides that, transparent sharing of persistent objects across multiple applications was not addressed by these previous solutions. In this dissertation, we have presented *region system*, a new kernel subsystem for managing PM consistently and exposing it directly within application address spaces. We have proposed the *region system* interface based on the requirements from an application developers point-of-view. A fundamental driving principle behind the *region system* is to reduce development complexity significantly when using PM to develop powerful stateful applications. Developers benefit from a simple durability interface that makes a group of application updates to PM-resident data atomically durable. This dissertation highlights the ease of use, sharing capabilities, and strong consistency and data durability guarantees provided by the *region system* for mmap based applications. The *region system* also supports arbitrary and unordered allocation/deallocation of data within regions at the page granularity. This capability allows developers to use PM

as they would use DRAM within their code, thus drastically simplifying development of applications.

In Chapters 4 and 5 of this dissertation, we have presented the architecture, interface, and design of the *region system*. An overview of the implementation of the *region system* is provided in Chapter 6 followed by the evaluation. We have evaluated our system with a contemporary PM file system (i.e. Ext4-DAX) and a state-of-the-art PM library PMEM.IO [pmea]. In our experiments, for persistent operations, where persistent metadata of the system are modified, the *region system* performs $1.1\times$ to $2.85\times$ better than EXT4-DAX. These results validate that minimizing cache flushes by designing a non-redundant metadata architecture reduces latency of persistent PM operations. We have evaluated the atomically durable `pmsync` operation with the non-transactional PMEM.IO `pmem_flush` operation, and observed that the strongly consistent `pmsync` operation performs competitively (within 30%) when compared against the PMEM.IO library's weakly consistent `pmem_flush` operation. The performance penalty for attaining atomic durability and providing strong consistency guarantee are likely to be acceptable for applications that have PM data consistency requirements.

The design of *region system* opens a new window for existing mmap based applications to migrate to using PM effectively and easily. Based on the guarantee of shared atomic durability of mapped regions, we proposed two viable implementations of shared persistent containers, LIBPM-R and LIBPMEMOBJ-R, in Chapter 7. These container types facilitate seamless application development when the data structures are shared across multiple processes, system-wide. Most importantly, LIBPM-R allows applications to be developed without worrying about complex transactional mechanisms, while the modified version of LIBPMEMOBJ-R supports location independent sharing of containers. In our experiments, LIBPM-R consistently performs better than LIBPMEMOBJ-R by $1.2\times$ to $1.9\times$. However, LIBPM-R is susceptible to security attacks due to dis-

abling ASLR (Address Space Layout Randomization). LIBMEMOBJ-R, on the other hand, provides the flexibility of mapping the container data anywhere within an application's address space without compromising its security. Developers can choose between the two versions of containers based on the requirements for security and performance withing their applications.

The concept of shared persistent containers can be readily applied to contemporary storage architectures. These containers could be used as the building blocks for large scale object-based file systems, or these could also be used to alleviate the resource constraints in multi-tenant database systems where resources such as PM would be in high demand. Besides that, these containers can be also used to consistently store intermediate results in a parallel pipelined computation platform. The above mentioned scenarios would be worthy candidates for extending the applicability of the shared persistent containers in future research. A current limitation of our proposed region-based container solutions is such that, the containers they support must use data structures which would eventually reside completely in the PM. Supporting shared hybrid containers where data can reside in persistent as well as volatile memory is an interesting area for future work as well.

In this dissertation, we have proposed a complete solution for integrating PM devices to the traditional I/O stack. The *Region System* provides support to the applications for directly accessing PM resident data consistently while minimizing the latency of the PM access path by providing a non-redundant metadata architecture. The shared persistent containers provide the ability to effortlessly share persistent objects in contemporary multi-processor systems. We expect that, the dissemination of our research would facilitate finding optimal solutions for using PM devices across multiple layers of the storage stack.

BIBLIOGRAPHY

- [BCLN13] Seungjae Baek, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Energy-efficient and high-performance software architecture for storage class memory. *ACM Trans. Embed. Comput. Syst.*, 12(3), April 2013.
- [CCA⁺11] Joel Coburn, Adrian Caulfield, Ameen Akel, Laura Grupp, Rajesh Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *ASPLOS*, 2011.
- [CCV15] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D. Viglas. Rewind: Recovery write-ahead system for in-memory non-volatile data-structures. *Proc. VLDB Endow.*, 8(5):497–508, January 2015.
- [CDC⁺10] Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollow, Rajesh K. Gupta, and Steven Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '43*, pages 385–395, Washington, DC, USA, 2010. IEEE Computer Society.
- [CI08] Nathan Chong and Samin Ishtiaq. Reasoning about the arm weakly consistent memory model. In *Proceedings of the 2008 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness: Held in Conjunction with the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08), MSPC '08*, pages 16–19, New York, NY, USA, 2008. ACM.
- [CJ15] Shimin Chen and Qin Jin. Persistent b+-trees in non-volatile main memory. *Proc. VLDB Endow.*, 8(7):786–797, February 2015.
- [CLK⁺15] Daniel Campello, Hector Lopez, Ricardo Koller, Raju Rangaswami, and Luis Useche. Non-blocking writes to files. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, 2015.
- [CLX14] Ping Chi, Wang-Chien Lee, and Yuan Xie. Making b+-tree efficient in pcm-based main memory. In *Proceedings of the 2014 International Symposium on Low Power Electronics and Design, ISLPED '14*, pages 69–74, New York, NY, USA, 2014. ACM.

- [CME⁺12] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. Providing safe, user space access to fast, solid state disks. In *ASPLOS*, 2012.
- [CMH14] Feng Chen, M.P. Mesnier, and S. Hahn. A protected block device for persistent memory. In *Mass Storage Systems and Technologies (MSST), 2014 30th Symposium on*, 2014.
- [CNC⁺96] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher Aycock, Gurushankar Rajamani, and David Lowell. The rio file cache: Surviving operating system crashes. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS VII*, pages 74–83, New York, NY, USA, 1996. ACM.
- [CNF⁺09] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Doug Burger, Benjamin Lee, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *SOSP*, 2009.
- [CR17] Mohammad Chowdhury and Raju Rangaswami. Native os support for persistent memory with regions. In *Proceedings of 33rd International Conference on Massive Storage Systems and Technology, MSST '17*, 2017.
- [CS13] Adrian Caulfield and Steven Swanson. QuickSAN: A Storage Area Network for Fast, Distributed Solid State Disks, March 2013.
- [CSADAD12] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Consistency without ordering. pages 9–9, 2012.
- [CYW⁺16] Cheng Chen, Jun Yang, Qingsong Wei, Chundong Wang, and Mingdi Xue. Fine-grained metadata journaling on nvm. In *IEEE 32nd International Conference on Massive Storage Systems and Technology, MSST '16*, 2016.
- [DKK⁺14] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems, EuroSys '14*, pages 15:1–15:15, New York, NY, USA, 2014. ACM.

- [DSST89] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent, February 1989.
- [Fus] Fusion-io directFS and ACM. <http://www.fusionio.com/blog/blurring-the-line-between-memory-and-storage-introducing-filesystem-support-for-persistent-memory/>.
- [GMC⁺12] Jorge Guerra, Leonardo Marmol, Daniel Campello, Carlos Crespo, Raju Rangaswami, and Jinpeng Wei. Software persistent memory. In *USENIX ATC*, 2012.
- [HKWN18] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable transient inconsistency in byte-addressable persistent b+-tree. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 187–200, Oakland, CA, 2018. USENIX Association.
- [HLM94] Dave Hitz, James Lau, and Michael Malcolm. File system design for an nfs file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, WTEC'94, pages 19–19, Berkeley, CA, USA, 1994. USENIX Association.
- [KKB⁺16] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. Nvwal: Exploiting nvram in write-ahead logging. *SIGPLAN Not.*, 51(4):385–398, March 2016.
- [LBN13] Eunji Lee, Hyokyung Bahn, and Sam H. Noh. Unioning of the Buffer Cache and Journaling Layers with Non-Volatile Memory. In *FAST*, 2013.
- [LDK⁺14a] Philip Lantz, Subramanya Dullloor, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. Yat: A validation framework for persistent memory software. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014.
- [LDK⁺14b] Philip Lantz, Subramanya Dullloor, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. Yat: A validation framework for persistent memory software. In *Proceedings of the USENIX Annual Technical Conference*, ATC '14, June 2014.
- [LHZZ17] H. Liu, L. Huang, Y. Zhu, and Y. Shen. Librekv: A persistent in-memory key-value store. *IEEE Transactions on Emerging Topics in Computing*, PP(99):1–1, 2017.

- [LIMB09] Benjamin Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting Phase Change Memory as a Scalable DRAM Alternative. In *ISCA*, 2009.
- [LLS13] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. The bw-tree: A b-tree for new hardware platforms. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, pages 302–313, Washington, DC, USA, 2013. IEEE Computer Society.
- [LLS⁺17] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. Wort: Write optimal radix tree for persistent memory storage systems. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies, FAST'17*, pages 257–270, Berkeley, CA, USA, 2017. USENIX Association.
- [LSS16] Youyou Lu, Jiwu Shu, and Long Sun. Blurred persistence: Efficient transactions in persistent memory. *Trans. Storage*, 12(1):3:1–3:29, January 2016.
- [MAK⁺13] Iulian Moraru, David Anderson, Michael Kaminsky, Parthasarathy Ranganathan, Niraj Tolia, and Nathan Binkert. From Filesystem Designer to Persistent Memory Data Structure Designer: Enabling Safe Memory Management for Byte Addressable NVRAM, March 2013.
- [Mar17] Leonardo Marmol. Customized interfaces for modern storage devices. 2017.
- [Mck05] Paul E. Mckenney. Memory ordering in modern microprocessors. *Linux Journal*, 30:52–57, 2005.
- [MGA16] Leonardo Mármol, Jorge Guerra, and Marcos K. Aguilera. Non-volatile memory through customized key-value stores. In *Proceedings of the 8th USENIX Conference on Hot Topics in Storage and File Systems, HotStorage'16*, pages 101–105, Berkeley, CA, USA, 2016. USENIX Association.
- [MSTR15] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. NVMKV: A scalable, lightweight, ftl-aware key-value store. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 207–219, Santa Clara, CA, 2015. USENIX Association.
- [NH12] Dushyanth Narayanan and Orion Hodson. Whole-system persistence, March 2012.

- [OLN⁺16] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD '16*, pages 371–386, New York, NY, USA, 2016. ACM.
- [OS16] Jiaxin Ou and Jiwu Shu. Fast and failure-consistent updates of application data in non-volatile main memory file system. In *IEEE 32nd International Conference on Massive Storage Systems and Technology, MSST '16*, 2016.
- [pco] Deprecating the PCOMMIT Instruction. <https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction>.
- [PCW14] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecture, ISCA '14*, pages 265–276, Piscataway, NJ, USA, 2014. IEEE Press.
- [PKS13] Stan Park, Terence Kelly, and Kai Shen. Failure-atomic msync(): A simple and efficient mechanism for preserving the integrity of durable data. In *Proceedings of the 8th ACM European Conference on Computer Systems, EuroSys '13*, 2013.
- [pmea] Persistent Memory Programming. <http://pmem.io>.
- [pmeb] Pmem.io Benchmarks. <https://github.com/pmem/nvml/tree/master/src/benchmarks>.
- [QSR09] Moinuddin Qureshi, Viji Srinivasan, and Jude A. Rivers. Scalable High-Performance Main Memory System using Phase Change Memory Technology. In *Proceedings of the ISCA, 2009*.
- [RZK⁺15] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. Thynvm: Enabling software-transparent crash consistency in persistent memory systems. In *Proceedings of the 48th International Symposium on Microarchitecture, MICRO-48*, pages 672–685, New York, NY, USA, 2015. ACM.
- [SKB⁺17] Jihye Seo, Wook-Hee Kim, Woongki Baek, Beomseok Nam, and Sam H. Noh. Failure-atomic slotted paging for persistent memory. *SIGPLAN Not.*, 52(4):91–104, April 2017.

- [SMK⁺93] Mahadev Satyanarayanan, Henry H. Mashburn, Puneet Kumar, David C. Steer, and James J. Kistler. Lightweight Recoverable Virtual Memory. *Proc. of the ACM Symposium on Operating Systems Principles*, December 1993.
- [sni] SNIA NVM TWG. <http://www.snia.org/nvmsummit>.
- [UKRV11] Luis Useche, Ricardo Koller, Raju Rangaswami, and Akshat Verma. Truly non-blocking writes. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'11, 2011.
- [VMP⁺15] Rajat Verma, Anton Ajay Mendez, Stan Park, Sandya Srivilliputtur Manarswamy, Terence P. Kelly, and Charles B. Morrey III. Failure-atomic updates of application data in a linux file system. In *Proc. of the USENIX Conference on File and Storage Technologies (FAST 15)*, February 2015.
- [VTRC11] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proc. of USENIX FAST*, February 2011.
- [VTS11] Haris Volos, Andres Jaan Tack, and Michael Swift. Mnemosyne: Lightweight persistent memory. In *Proc. of ASPLOS*, 2011.
- [WR11] Xiaojian Wu and A. L. Narasimha Reddy. Scmfs: a file system for storage class memory. In *Proc. of SC*, 2011.
- [WZ94] Michael Wu and Willy Zwaenepoel. envy: A non-volatile, main memory storage system. *SIGPLAN Not.*, 29(11):86–97, November 1994.
- [XS16] Jian Xu and Steven Swanson. Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, February 2016. USENIX Association.
- [YMH12] Jisoo Yang, Dave B. Minturn, and Frank Hady. When Poll is Better than Interrupt. In *FAST*, 2012.
- [YWC⁺15] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. Nv-tree: Reducing consistency cost for nvm-based single level systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 167–181, Berkeley, CA, USA, 2015. USENIX Association.

- [ZHL⁺16] Shengan Zheng, Linpeng Huang, Hao Liu, Linzhu Wu, and Jin Zha. Hmvfs: A hybrid memory versioning file system. In *IEEE 32nd International Conference on Massive Storage Systems and Technology, MSST '16*, 2016.
- [ZSLH16] J. Zhou, Y. Shen, S. Li, and L. Huang. Nvht: An efficient key-value storage library for non-volatile memory. In *2016 IEEE/ACM 3rd International Conference on Big Data Computing Applications and Technologies (BDCAT)*, pages 227–236, Dec 2016.
- [ZYMS15] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. Mojim: A reliable and highly-available non-volatile memory system. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15*, 2015.
- [ZZYZ09] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. A Durable and Energy Efficient Main Memory using PCM Technology. In *ISCA*, 2009.

VITA

MOHAMMAD ATAUR RAHMAN CHOWDHURY
mchow017@fiu.edu

December 2017	Masters in Computer Science Florida International University Miami, Florida
October 2009	B.S. Computer Science & Engineering Bangladesh University of Engineering and Technology Dhaka, Bangladesh
June 2013 - Present	Graduate Research Assistant System Research lab Florida International University Miami, FL
May 2017 - August 2017	Software Engineering Intern Datos IO San Jose, CA
May 2014 - August 2014	Research Intern Fusion-io San Jose, CA
September 2012 - May 2013	Graduate Teaching Assistant Florida International University Miami, FL
October 2009 - August 2012	Software Engineer IMS Health Dhaka, Bangladesh

PUBLICATIONS AND PRESENTATIONS

- Native OS Support for Persistent Memory with Regions
33rd International Conference on Massive Storage Systems and Technology
(MSST2017)
Authors: Mohammad Chowdhury, Raju Rangaswami

- Shared Atomic Consistency for Persistent Memory Containers
9th Annual Non-Volatile Memories Workshop (NVMW 2018)
Authors: Mohammad Chowdhury, Raju Rangaswami
- LibPM: Simplifying Application Usage of Persistent Memory
Submitted to Transaction of Storage (Under revision)
Authors: Leonardo Marmol, Mohammad Chowdhury, Raju Rangaswami
- Location aware code offloading on mobile cloud with QoS constraint
2014 IEEE 11th Consumer Communications and Networking Conference (CCNC),
Las Vegas, NV, 2014, pp. 74-79.
Authors: S. Tasnim, M. Chowdhury, K. Ahmed, N. Pissinou and S. S. Iyengar