Florida International University
FIU Digital Commons

FIU Electronic Theses and Dissertations University Graduate School

3-31-2017

Customized Interfaces for Modern Storage Devices

Leonardo Marmol
Florida International University, lmarm001@fiu.edu

DOI: 10.25148/etd.FIDC001816

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

Part of the Computer Sciences Commons

Recommended Citation
Marmol, Leonardo, "Customized Interfaces for Modern Storage Devices" (2017). *FIU Electronic Theses and Dissertations*. 3165.
https://digitalcommons.fiu.edu/etd/3165

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

CUSTOMIZED INTERFACES FOR MODERN STORAGE DEVICES

A dissertation submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Leonardo Mármol Amador

2017

To: Interim Dean Ranu Jung
College of Engineering and Computing

This dissertation, written by Leonardo Mármol Amador, and entitled Customized Interfaces for Modern Storage Devices, 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.

Jason Liu

Leonardo Babadilla

Ning Xie

Guang Quan

Raju Rangaswami, Major Professor

Date of Defense: March 31, 2017

The dissertation of Leonardo Mármol Amador is approved.

Interim Dean Ranu Jung
College of Engineering and Computing

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

Florida International University, 2017

© Copyright 2017 by Leonardo Mármol Amador

All rights reserved.

DEDICATION

To Nikki-jo, who brings light into my days.

ACKNOWLEDGMENTS

Much needed to happen for this dissertation to be completed, and it would not have been possible without the help, collaboration, and support of many.

I want to thank my friend and life-partner Nikki-jo for all her love and care throughout these years. Without you, this journey would have been much darker and lonelier.

I want to thank my family, who never understood me but accepted me all the same. It is because of you all that I am reminded of what is truly important in life.

I want to thank my advisor and friend Dr. Raju Rangaswami for giving this opportunity, for guiding me throughout this journey, and for believing in me, even in my moments of doubt.

I want to thank my many co-authors, lab-mates, and collaborators from which I learned so much.

I want to thank my dissertation committee for all their feedback that made this work objectively better.

I want to thank all the people in the Computer Science department, in particular Olga, for helping me navigate the bureaucratic jungles of FIU.

Thank you all.

ABSTRACT OF THE DISSERTATION

CUSTOMIZED INTERFACES FOR MODERN STORAGE DEVICES

by

Leonardo Mármol Amador

Florida International University, 2017

Miami, Florida

Professor Raju Rangaswami, Major Professor

In the past decade, we have seen two major evolutions on storage technologies: flash storage and non-volatile memory. These storage technologies are both vastly different in their properties and implementations than the disk-based storage devices that current software stacks and applications have been built for and optimized over several decades. The second major trend that the industry has been witnessing is new classes of applications that are moving away from the conventional ACID (SQL) database access to storage. The resulting new class of NoSQL and in-memory storage applications consume storage using entirely new application programmer interfaces than their predecessors. The most significant outcome given these trends is that there is a great mismatch in terms of both application access interfaces and implementations of storage stacks when consuming these new technologies.

In this work, we study the unique, intrinsic properties of current and next-generation storage technologies and propose new interfaces that allow application developers to get the most out of these storage technologies without having to become storage experts themselves. We first build a new type of NoSQL key-value (KV) store that is FTL-aware rather than flash optimized. Our novel FTL cooperative design for KV store proofed to simplify development and outperformed state of the art KV stores, while reducing write amplification. Next, to address the growing relevance of byte-addressable persistent memory, we build a new type of KV store that is customized and optimized for persistent memory. The

resulting KV store illustrates how to program persistent effectively while exposing a simpler interface and performing better than more general solutions. As the final component of the thesis, we build a generic, native storage solution for byte-addressable persistent memory. This new solution provides the most generic interface to applications, allowing applications to store and manipulate arbitrarily structured data with strong durability and consistency properties. With this new solution, existing applications as well as new "green field" applications will get to experience native performance and interfaces that are customized for the next storage technology evolution.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE PAGE

CHAPTER 1

INTRODUCTION

Storage systems are becoming increasingly important in private, public, and hybrid cloud data centers. Over time, interfaces to storage have evolved from being file oriented, to block oriented, to key-value oriented, and now memory oriented. The block interface established by early disk drives have sustained and dominated storage access for enterprise storage arrays including disk-only, hybrid, and flash-only implementations. Classic storage consuming applications such as SQL database have been increasingly optimized for the block interface over the years even as the underlying implementations have changed. A parallel evolution in the storage stack is that the underlying device technology has changed significantly as well. The industry has almost completely moved from disk to flash-based storage for primary storage in the enterprise.

A recent evolution of the application interface to storage is the key-value interface [mema, Mon14, Roc14, DSL11, MST+14] and a recent evolution in device technology is high-performance flash devices. Key-value stores have become a very popular storage solution for web services, NoSQL applications and distributed environments. The appeal of the interface is the ease of storing and accessing data when associating a minimum amount of semantic information with the data itself. One of the main reasons for the success of key-value stores is their higher performance [BPPP09, DSL10] and scalability [DHJ+07], when compared to traditional storage solutions like relational databases.

It is thus no surprise that modern key-value stores were particularly design for flash storage as a way to provide an even higher performing storage solutions. In doing so, developers often need to make assumptions about the internals of flash storage devices. But these details are often not made available to the public, and optimizations that work well for one vendor do not for others [YPG+14] To improve the situation, we propose that FTL-aware designs, that expose to the applications some of the internal mechanisms of the flash storage, are necessary.

In this thesis, we explore the design of key-value store like solutions for a completely different storage technology, non-volatile memory (NVM). NVM technologies offer a unique duality that combines the byte-addressability and low latency of DRAM with the durability of traditional storage. This eliminates the need to come up clever ways of moving data back and forth between slow persistent storage and fast volatile memory. However, programming directly onto NVM is non trivial, and recycling existing solutions for managing memory or storage are not applicable to NVM. To bridge this gap, we propose and build on top of a new, extended device interface to flash storage that allows a custom library implementation to extract significantly higher performance from the storage device. We empirically demonstrate that the use of this extended interface, that is aware of the underlying flash management mechanisms not only simplify the design and implementation of key-value stores but also significantly improve their performance.

While flash based storage technology has a stronghold on primary storage products for primary data storage, we are currently at the beginning of the next evolution of storage — from flash-based block storage to byte-addressable persistent memory. The next generation of persistent storage devices are going to be memory oriented [Int15a]. 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. 3D-XPoint persistent memory from Intel is currently the most promising and it is expected to provide 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.

Just as the onset of disk-based storage provided the impetus for seek-optimizing workload shaping and flash-based storage have addressed capacity constraints and flash lifetime management, this next evolution of storage technology will usher in entirely new class of applications and new mechanisms for application access to storage. Conventional block I/O processing in the OS has shown to add as much as 300 μs of overhead to each operation and even highly optimized software stacks that bypass the OS for device access add as much as 20 μs of overhead [CME+12, CS13]. The memory-like properties of this new class of persistent memory storage open up the possible of using low-latency `load/store` access. However, store interfaces were design with a much slower medium in mind that operate a much coarse granularity [CLU+15, CME+12, VTS11b, CCA+11b]. Moreover, memory interfaces to storage do not automatically provide the consistency and durability properties that applications expect from traditional storage, and the other hand, When an application consuming storage through a KV interface chooses to utilize a persistent memory device, the existing solutions suffer an impedance mismatch whereby they map the KV implementation to a memory oriented block device (e.g., a RAMDISK) [CLU+15, CNF+09, DKK+14]. Alternatively, custom-built KV stores for memory lose generality. In this thesis, we propose to consume NVM through a native key-value interface with transactions support for updates. With our proposed solution, applications consume NVM without having to understand the details of NVM, and thereby trading high performance for generality.

In the last part of this thesis we focus on a more general approach to NVM in order to enable NVM for all types of applications. While key-value stores are very popular alternative for data storage solutions, many application have requirements that are not satisfy by the simple data model of key vale stores. In particular, having to access all data through a key value interface may impose too much impedance mismatch, and for some applications too much overhead. Additionally, hand-crafting point solutions for storing data within persistent memory could quickly become too costly and time consuming. Then is becomes necessary to have a more generic abstraction that exposes the

What is necessary is a generic abstraction that will expose the richness and performance of the memory interface to the applications and support seamless persistence. To this end, we propose LIBPM: a persistent memory library that implements the *containers* abstraction. The proposed persistent containers are self-contained data hosting units that allow applications to directly store, fetch, and update arbitrarily complex data structures at memory speeds. They provide a simple and light-weight transactional update capabilities that simplify applications development. Additionally, LIBPM simplifies the porting of legacy applications to NVM technologies by automatically discovering and persistent applications' in-memory data based on simple hints.

The next chapter outlines the main claims of this dissertation. It starts by listing the proposed problems to study, followed by descriptions of our proposed solutions and their significance. Each chapter of this dissertation is written to be as self contained as possible. However, in order to better understand the material here presented, readers are encouraged to get familiar with the terms and concepts described in the background chapter 3. The rest of this document dedicates a chapter to each of the systems we have built to address the proposed problems. We conclude the dissertation with a survey of related literature and a detailed description on how our solutions differs from previous ones, concluding statements, and directions for future work.

CHAPTER 2

PROBLEM STATEMENT

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

2.1 Thesis Statement

We propose building native storage solutions to applications that are tailor-made for the underlying storage device technology by:

1. providing native integration between key-value stores and modern flash storage devices,
2. building a key-value store with native, transactional update capabilities when using *non-volatile memory* (NVM) devices, and
3. providing a new, generic application interfaces to NVM devices suitable for both supporting new applications and for porting of legacy applications to NVM.

2.2 Thesis Contributions

This thesis addresses the challenges of building native storage support for applications and makes three distinct contributions.

First, we build native support for accessing today's advanced flash devices via a simple and commonly-used key-value interface. Existing key-value stores use flash storage like any other block device. This approach is convenient because it requires no application modifications, but it does not use the full potential of modern flash storage. Modern Flash Translation Layers (FTL) implement many advanced capabilities that, if exposed to the application layer, could significantly simplify application design and improve performance. Because these interfaces are not exposed to the applications layer, key value stores end up re-implementing some of these capabilities, incurring in additional resources consumption, higher overhead and poorer performance. Examples of these capabilities include mapping from logical to physical addresses, logging, and atomic updates. In this thesis, we propose extending traditional block device interfaces for flash storage, so that these advanced capabilities are made available to the application layer.

Second, we build native support for using next generation *non-volatile memory* (NVM) devices via a key-value interface. With the arrival of NVM device technologies new opportunities for improving the performance of applications become available. Of these options, programming NVM directly is the most effective, yet also the most difficult and error prone. To correctly program NVM, developers would have to become familiar with the many subtleties of NVM that could only be expected from an expert in the subject. For example, even though NVM is persistent, CPU caches are not. So in order to make memory update persistent, CPU caches need to be carefully flushed to guarantee data durability without impacting performance. And depending on the NVM technology and the system architecture, additional instructions may be needed to avoid buffering on the memory controllers. In addition, CPU caches perform out-of-order evictions at any time, making the use memory fences a requirement to preserve applications data consistency. Finally, physical characteristics of competing technologies are often not available to the user, and optimizations for one type of NVM do not apply to others. For these reasons, we propose applications to consume NVM through an indirection layer that hide all the complexities of NVM. To this end, we build a key-value store with support for transactional updates as a viable interface for this indirection layer.

Finally, we go beyond the key-value interface to build native support for accessing NVM similar to how volatile memory is consumed today for increased generality and usage. Using a key-value store to consume NVM trades generality for simplicity and high performance. While the key-value interface is adequate for many applications, it is not good enough for others applications with more complex data requirements. In addition, the key-value store favors the development of new applications without providing a pathway for legacy applications to consume NVM without a significant rewrite of their core logic. To overcome these limitations, we propose LIBPM: a Persistent Memory Library that implements Container as the main abstraction for consuming NVM. Containers are a data hosting unit that can store arbitrarily complex data types while preserving their integrity and consistency. This abstraction facilities the use of NVM without having to navigate all the pitfalls of programming NVM directly. LIBPM provides a simple and high performing transactional update semantics which allows applications to manipulate persistent data at the speed of memory. In addition, LIBPM provides a mechanism to port legacy applications to NVM with automatic data discovery.

2.3 Thesis Significance

This thesis addresses current problems related to developing and deploying applications that consume storage with high-performance and correctness. The solutions discussed in this thesis allow applications to consume storage using high-level API without being concerned about the internals of the underlying storage device and storage software stack implementations. These solutions, in effect, create a sophisticated layer of indirection with native mechanisms for consuming the underlying storage device and exposing simple abstractions to the application developer.

2.3.1 NVMKV

Designing a key-value store that uses the capabilities of moderns FTL layers has multiple benefits. Firstly, it significantly simplifies the design and implementation of the key-value store because many of the internal mechanism needed by the key-value store are already implemented by the FTL layer. Some of these mechanisms include data mapping, compaction/garbage collection, and sparse addressing. Secondly, it improves the performance of the key-value stores by minimizing applications-level write amplifications and reducing the number of IOs per operations. Thirdly, it reduces the resource utilization by the key-value layer because many of the internal in-memory mappings are maintained by the FTL-layer and replaced by simple hashing schemes. Finally, it improves the life time of the device as an consequence of the reduced write amplification and on average one IO per operation.

2.3.2 METRADB

Piggy backing on the popularity of key-value stores, we propose the consumption of NVM through key-value store interfaces with support for transactional updates. This approach makes it easy for modern applications to use NVM without having to worry about the internals of NVM or maintaining application's data in a consistent state. The simplicity of key-value stores interfaces eases the design and implementation of high performing transactional semantics. Additionally, it eliminates many challenges related to using NVM directly such as managing persistent data structure integrity and data sharing. Finally, consuming NVM through an indirection layer facilitates the adoption of future NVM technologies at low development cost because the core application logic remains untouched.

2.3.3 LibPM

To address the limitations of the key-value store interface at address the needs of a lager number of application, LibPM exposes a familiar memory-like interface that operate on Containers. At a high level, Container host persistent data and guarantee the durability and consistent of the data it hosts. Unlike key-value stores, LibPM allows unsupervised access to the data it manages by memory mapping Container directly into the application's address space. This eliminates the impedance of having to access data in persistent memory through a key value stores APIs. Another advantage of LibPM over key value stores is that LibPM can manage arbitrarily complex data types and it's aware of object relations (pointers) which is typically not supported on key value stores. Once a LibPM Container is open, all of is data is readily available for the applications without having to deal with any pointer swizzling. LibPM also provide a simple transactional model that allows developers to make in-memory updates durable with a single atomic API call at the Container granularity. This atomic operations effectively mutates the state of the Container from one consistent point in time to the next. Additionally, LibPM implements an automatic data discovery mechanism based on applications hints that facilitates porting legacy applications to NVM technologies in an incremental fashion and without any changes to their core logic. With LibPM, applications effectively get to consume persistent memory as if it were volatile memory, enabling developers to write applications tailor-made for persistent memory with relative ease.

CHAPTER 3

BACKGROUND

In this chapter we define concepts that are prerequisite for understating the problem described in this thesis as well as the proposed solutions.

3.1 NAND Based Storage Systems

NAND flash memory is a type of non-volatile storage technology that does not require power to retain data. There three types of operations that can be performed on NAND flash: read, write and erase. Reads are typically faster than writes, and writes are faster than erase by about an order of magnitude. Due to the physical characteristics of NAND, data can not be overwritten without first erasing it. This unique characteristic of flash storage has a major impact on how applications use flash storage.

3.2 Flash Translation Layer

NAND Flash storage devices have a software component known as the flash translation layer. This software layer is in charge of presenting the flash storage device to the OS as a regular block device. Additionally, the FTL is in charge or maintaining the mapping between logical to physical addresses, reclaiming unused space in the device (this process is also known as garbage collection), and placing data across the device in a way that avoids uneven wear leveling.

3.3 Non-Volatile Memory

Non-volatile memory (NVM) or persistent memory is a type of memory that like NAND flash does not required power to retain data, but unlike flash it allows data accesses at a byte (8 bits) granularity. NVM is also a random-access memory with latencies for reads and write operations that are comparable to those of DRAM. In this document we will use the terms non-volatile memory and persistent memory interchangeably.

3.4 NVM: Is it memory or storage?

Conventionally, applications have consumed memory and storage through different interfaces. Memory is typically managed by an external library (e.g. libc) which adds new mappings to an application's address space, allowing direct access to data. Storage, on the other hand, is traditionally managed by an external entity like a file system or a database, which provides supervised access to data and guarantees data consistency. Since the existing memory interface guarantees no data durability, it cannot be used *as-is* for NVM. The well-established file interface can also fall significantly short in utilizing the potential of NVM. The large granularity of block-oriented file system induces high software and data transfer overheads making small reads and updates expensive [DKK+14, CME+12, CLU+15]. For these reasons, we believe that in order to fully exploit the potential of NVM a software stack and developer interfaces will need to be created.

CHAPTER 4
NVMKV

In this chapter we examine flash storage devices and how to write applications for this new type of media in a way to utilizes flash full potential. In order to do so, we take a closer looks at the software component of flash storage, the Flash Translation Layer (FTL), and illustrate how exposing some of the FTL internals to user space applications could simplify applications design and improve applications performance. We demonstrate this approach by implementing a high-performance Key-Value Store with transactional update optimized for modern flash storage devices.

4.1 Introduction

Flash-based key-value (KV) stores are becoming mainstream, with the importance of the KV interface to storage and flash technology have been well established through a gamut of implementations [AFK$^+$09, BPPP09, DSL10, DSL11, LFAK11]. However, best utilizing the high-performance flash-based storage to drive the new generation of key-value stores continues to remain a challenge. The majority of the existing KV stores use a logging-based approach which induces significant additional *write amplification* (WA) at the KV software layer in addition to the internal WA caused by the FTL while managing physical flash. The recent LevelDB KV store from Google [GD11] also exhibits rather dramatic auxiliary write amplification. Figure 4.1 reveals a minimum of 2.5*x* auxiliary write amplification for sequential asynchronous writes and a maximum of 43*x* for random synchronous writes.

Modern FTLs offer new capabilities that enable compelling, new design points for KV stores [T1013, Ope14]. Integration with these advanced capabilities results in an optimized *FTL-aware* KV store [MST$^+$14]. First, writing to the flash can be optimized to significantly improve both device lifetime and workload I/O performance. Second, mod-

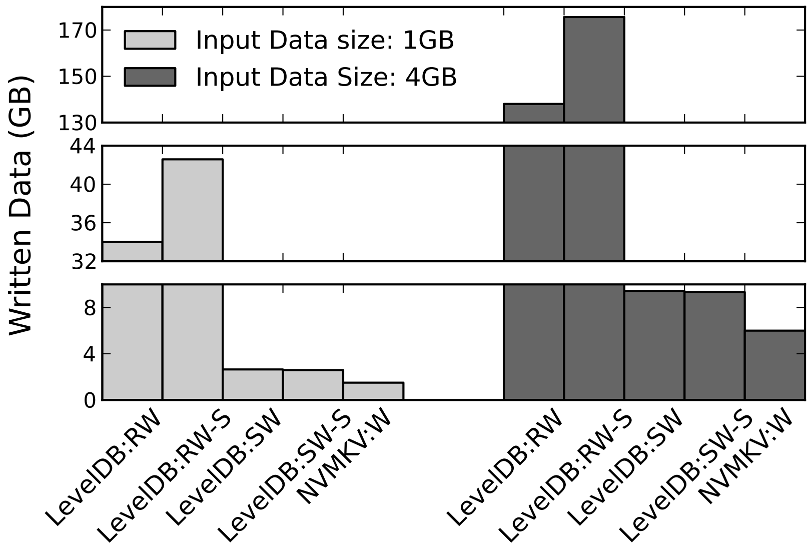

Figure 4.1: A comparison of write amplification. LevelDB variants are RW: Random asynchronous writes, RW-S: random synchronous writes, SW: sequential asynchronous writes, SW-S: sequential synchronous writes.

ern FTLs already perform many functions that are similar to the functionality built into many KV stores such as log-structuring, dynamic data remapping, indexing, transactional updates, and thin provisioning [JBLF12, ONW$^+$11, SSZ12]. Avoiding such replication of functionality can offer significant resource and performance benefits.

Next we present the design, implementation, and evaluation of NVMKV, an FTL-aware KV store. NVMKV has been designed from the ground up to utilize the advanced capabilities found in modern FTLs. It implements a hashing-based design that uses the FTLs sparse address-space support to eliminate all write amplification at the KV layer, improving flash device endurance significantly relative to current KV stores. It is able to achieve single I/O `get/put` operations with performance close to that of the raw device, representing a significant improvement over current KV stores. NVMKV uses the advanced FTL capabilities of *atomic multi-block write*, *atomic multi-block persistent trim*, *exists*, and *iterate* to provide strictly atomic and synchronous durability guarantees for

KV operations.

Two complementary factors contribute to increased collocation requirements for KV stores running on a single flash device. First, given the increasing flash densities, the performance points of flash devices are now based on capacity with larger devices being more cost-effective [YPG$^+$14]. Second, virtualization supports increases in collocation requirements for workloads. A recent study has shown that multiple independent instances of such applications can have a counterproductive effect on the underlying FTL, resulting in increased WA [YPG$^+$14]. NVMKV overcomes this issue by offering a new *pools* abstraction that allows transparently running multiple KV stores within the same FTL. While similar features exist in other KV stores, the FTL-aware design and implementation within NVMKV enables both efficient FTL coupling and KV store virtualization. NVMKV's design also allows for optimized flash writing across multiple KV instances and as a result lowers the WA.

In the quest for performance, KV stores and other applications are trending towards an in-memory architecture. However, since flash is still substantially cheaper than DRAM, any ability to offset DRAM for flash has the potential to reduce Total Cost of Ownership (TCO). We demonstrate how accelerating KV store access to flash can in turn result in similar or increased performance with much less DRAM.

We evaluated NVMKV and compared its performance to LevelDB. We evaluated the scalability of pools, compared it to multiple instances of LevelDB, and also found that NVMKV's atomic writes outperform both async and sync variants of LevelDB writes by up to 6.5x and 1030x respectively. NVMKV reads are comparable to that of LevelDB even when the workloads fit entirely in the filesystem cache, a condition that benefits LevelDB exclusively. When varying the available cache space, NVMKV outperforms LevelDB and more importantly introduces a write amplification of 2x in the worst case, which is small compared to the 70x for LevelDB. Finally, NVMKV improves YCSB benchmark throughput by up to 25% in in comparison to LevelDB.

4.2 Motivation

In this section, we discuss the benefits of better integration of KV stores with the FTL's capabilities. We also motivate other key tenets of our architecture, in particular the support of multiple KV stores on the same flash device and the need to increase performance with smaller quantities of DRAM.

An FTL-aware KV store: A common technique for performance improvement in flash optimized KV stores is some form of log structured writing. New data is appended to an immutable store and reorganized over time to reclaim space [Aer, Roc14, GD11, LFAK11]. The reclamation process, also called garbage collection or compaction, generates *Auxiliary Write Amplification (AWA)* which is the application level WA above that which is generated by the FTL. Unfortunately, AWA and the FTL induced WA have a multiplicative effect on write traffic to flash [YPG$^+$13]. Previous work highlighted an example of this phenomenon with LevelDB, where a small amount of user writes can be amplified into as much as 40X more writes to the flash device [MST$^+$14]. As another example, the SILT work describes an AWA of over 5x [LFAK11]. NVMKV entirely avoids AWA by leveraging native addressing mechanisms and optimized writing implemented within modern FTLs.

Multiple KV stores in the device: The most recent PCIe and SAS flash devices can provide as much as 4-6TB of capacity per device. As density per die increases with every generation of flash driven by the consumer market, the multiple NAND dies required to generate a certain number of IOPs will come with ever increasing capacities as well as reduced endurance [GDS12]. Multiple KV stores on a single flash device become cost effective but additional complexities arise. For instance, recent work shows how applications that are log structured to be flash optimal can still operate in suboptimal ways when either placed above a file system or run as multiple independent instances over a shared FTL [YPG$^+$14]. NVMKV provides the ability to have multiple independent KV workloads share a device with minimal AWA.

Frugal DRAM usage: The ever increasing need for performance is driving the in-memory computing trend [Mem14, SAP14, DFI$^+$13]. However, DRAM cost does not scale linearly with capacity since high capacity DRAM and the servers that support it are more expensive per unit of DRAM (in GB) than the mid-range DRAM and servers. The efficacy of using flash to offset DRAM has also been established in the literature [BP11]. In the KV store context, similar arguments have been made showing the server consolidation benefits of trading DRAM for flash [Aer]. A KV store's ability to leverage flash performance contributes directly to its ability to trade off DRAM for flash. NVMKV operates with high performance and low WA in both single and multiple instance KV deployments.

4.3 Building an FTL-aware KV Store

NVMKV is built using the advanced capabilities of modern FTLs. In this section, we discuss its goals, provide an overview of the approach, and describe its API.

4.3.1 Goals

NVMKV is intended for use within single node deployments by directly integrating it into applications. While it is not intended to replace the scale out key-value functionality provided by software such as Dynamo and Voldemort [DHJ$^+$07, SKG$^+$12], it can be used for single node KV storage within such scale out KV stores. From this point onward, we refer to such single node KV stores simply as KV stores. We had the following goals in mind when designing NVMKV:

Deliver Raw Flash Performance: Convert the most common KV store operations, `GET` and `PUT` into a single I/O per operation at the flash device to deliver close to raw flash device performance. As flash devices support high levels of parallelism, the KV store should also scale with parallel requests to utilize the performance scaling capacity of the device. Finally, when multiple, independent KV instances are consolidated on a single flash device, the KV store should deliver raw flash performance to each instance.

Minimize Auxiliary Write Amplification: Given the multiplicative effect on I/O volume due to WA, it is important to minimize additional KV store writes, which in turn reduces the write load at the FTL and the flash device. Reducing AWA improves KV operation latency by minimizing the number of I/O operations per KV operation as well as improvement of flash device lifetime.

Minimize DRAM Consumption: Minimize DRAM consumption by *(i)* minimizing the amount of internal metadata, and *(ii)* by leveraging flash performance to offset the amount of DRAM used for caching.

Simplicity: Leverage FTL capabilities to reduce code complexity and development time for the KV store.

4.3.2 Approach

Our intent with NVMKV is to provide the rich KV interface while retaining the performance of a much simpler block based flash device. NVMKV meets its goals by leveraging the internal capabilities of the FTL where possible and complementing these with streamlined additional functionality at the KV store level. The high level capabilities that we leverage from the FTL include:

Category	API	Description
Basic	get(...)	Retrieves the value associated with a given key.
	put(...)	Inserts a KV pair into the KV store.
	delete(...)	Deletes a KV pair from the KV store.
Iterate	begin(...)	Sets the iterator to the beginning of a given pool.
	next(...)	Sets the iterator to the next key location in a given pool.
	get_current(...)	Retrieves the KV pair at the current iterator location in a pool.
Pools	pool_exist(...)	Determines whether a key exists in a given pool.
	pool_create(...)	Creates a pool in a given NVMKV store.
	pool_delete(...)	Deletes all KV pairs from a pool and deletes the pool from NVMKV store.
	get_pool_info(...)	Returns metadata information about a given pool in a KV store.
Batching	batch_get(...)	Retrieves values for a batch of specified keys.
	batch_put(...)	Sets the values for a batch of specified keys.
	batch_delete(...)	Deletes the KV pairs associated with a batch of specified keys.
	delete_all(...)	Deletes all KV pairs from a NVMKV store in all pools.
Management	open(...)	Opens a given NVMKV store for supported operations.
	close(...)	Closes a NVMKV store.
	create(...)	Creates a NVMKV store
	destroy(...)	Destroys a NVMKV store.

Table 4.1: **NVMKV API** *The table provides brief descriptions for the NVMKV API calls.*

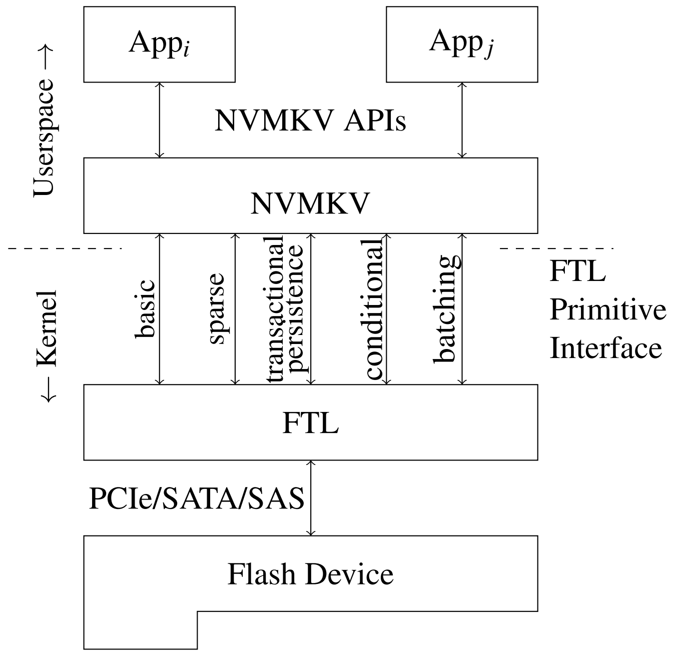

Figure 4.2: NVMKV System Architecture

Dynamic mapping: FTLs maintain an indirection map to translate logical addresses into physical data locations. NVMKV leverages the existing FTL indirection map to the fullest extent to avoid maintaining any additional location metadata. Every read and write operation simply uses the FTL indirection map and thereby operates at raw flash device performance by definition. This approach also reduces the additional DRAM overhead of NVMKV.

Persistence and transactional support: FTLs access data and metadata, and in particular maintain the persistence of indirection maps, at the speed of raw flash. NVMKV leverages this highly tuned capability to reduce the overhead for metadata persistence, logging, and checkpointing operations. Further, as FTLs operate as non-overwriting redirect-on-write stores, they can easily provide high performance transactional write semantics [ONW+11]. NVMKV leverages these capabilities to limit locking and journaling overheads.

Parallel operations: FTLs already implement highly parallel read/write operations while coordinating metadata access and updates. NVMKV leverages this FTL feature to minimize locking, thus improving scalability. We also define batch operations that are directly executed by the FTL to enable parallel KV store requests to be issued with lower I/O stack overhead [VKA12a].

4.3.3 NVMKV Architecture

NVMKV is a lightweight library in user space which interacts with the FTL through a *primitives interface* implemented as IOCTLs to the device driver that manages the flash device. Figure 4.2 shows the architecture of a system with NVMKV. Consumer applications, such as scale out KV stores, communicate with the NVMKV library using the

NVMKV API. The NVMKV API calls are translated to underlying FTL primitives interface calls to be executed by the FTL.

Category	**API**	**Description**
Basic	read(...)	Reads the data stored in the Logical Block Address (LBA).
	write(...)	Writes the data stored in buffer to destination LBA.
	trim(...)	Deletes (or discards) the mapping in FTL for the passed LBA range.
Sparse	exists(...)	Returns the presence of FTL mapping for the passed LBA.
	range_exist(...)	Returns the subset of LBA ranges that are mapped in the FTL.
	ptrim(...)	Persistently deletes the mapping in FTL for the passed LBA range.
	iterate(...)	Returns the next populated LBA starting from the passed LBA.
Transactional Persistence	atomic_read(...)	Executes read for a contiguous LBAs as an ACID transaction.
	atomic_exists(...)	Executes exists for a contiguous LBAs as an ACID transaction.
	atomic_write(...)	Executes write of a contiguous LBAs as an ACID transaction.
	atomic_ptrim(...)	Executes ptrim of a contiguous LBAs as an ACID transaction.
Conditional	cond_atomic_write(...)	Execute the atomic_write only if a condition is satisfied.
	cond_range_read(...)	Returns the data only from a subset of LBA ranges that are mapped in the FTL.
Batching		Operations within each category can be batched and executed in the FTL.

Table 4.2: **FTL Primitive Interface** *Enhanced FTL capabilities that NVMKV builds upon.*

4.3.4 NVMKV Consumer API

NVMKV's consumer applications interact with the library through the NVMKV API. We held discussions with the creators and vendors of several scale out KV stores to identify a set of operations commonly needed in a KV store. These operations formed the NVMKV

API and they fall under five broad categories based on the functionality they provide. The categories are: *basic*, *iterate*, *pools*, *batching*, and *management*.

Table 4.1 presents the overview of the NVMKV API. We leverage the FTL's ability to provide enhanced operations such as *Atomic Writes* to provide transactional guarantees in NVMKV operations. Most existing KV stores do not offer such guarantees for their operations, and adopt more relaxed semantics such as eventual consistency to provide higher performance. On the other hand, we found that our approach enabled us to provide transactional guarantees with no loss of performance. We believe such guarantees can be of use to specific classes of applications as well as for simplifying the store logic contained within such applications. For instance, atomic KV operations imply that applications no longer need to be concerned with partial updates to flash.

The *Basic* and *Iterate* categories contain common features provided by many KV stores today. The *Pools* category interfaces allow for grouping KV pairs into buckets that can be managed separately within an NVMKV store. Pools provide the ability to transparently run multiple KV stores within the same FTL (discussed in more detail in § 4.6). The *Batching* category interfaces allow for group operations both within and across *Basic*, *Iterate*, and *Pools* categories, a common requirement in KV stores [BFPS11]. Finally, the *Management* category provides interfaces to perform KV store management operations.

4.4 Overview and FTL Integration

NVMKV's design is closely linked to the advanced capabilities provided by modern FTLs. Before describing its design in more detail, we provide a simple illustrative example of NVMKV's operation and discuss the advanced FTL capabilities that NVMKV leverages.

4.4.1 Illustrative Overview

To illustrate the principles behind NVMKV's design simply, we now walk through how a `get`, a `put`, and a `delete` operation are handled. We assume the sizes of keys and values are fixed and then address arbitrary sizes when we discuss design details (§4.5).

By mapping all KV operations to FTL operations, NVMKV eliminates any additional KV metadata in memory. To handle `put`s, NVMKV computes a hash on the key and uses the hash value to determine the location (i.e., LBA) of the KV pair. Thus, a `put` operation gets mapped to a write operation inside the FTL.

A `get` operation takes a key as input and returns the value associated with it (if the key exists). During a `get` operation, a hash of the key is computed first to determine the starting LBA of the KV pair's location. Using the computed LBA, the `get` operation is translated to a read operation to the FTL wherein the size of the read is equal to the combined sizes of the key and value. The stored key is matched with the key of the `get` operation and in case of a match, the associated value is returned.

To handle a `delete` operation, the given key is hashed to compute the starting LBA of the KV pair. Upon confirming that the key stored at the LBA is the key to be deleted, a discard operation is issued to the FTL for the range of LBAs containing the KV pair.

In this simplistic example, translating existing KV operations to FTL operations is straightforward and the KV store becomes a thin layer offloading most of its work to the underlying FTL with no in-memory metadata. However, additional work is needed to handle hash collisions in the LBA space and persisting discard operations.

4.4.2 Leveraging FTL Capabilities

We now describe the advanced FTL capabilities that are available and also extended to enable NVMKV. Many of these advanced FTL capabilities have already been used in other applications [DAT+14, JBLF12, ONW+11, SSZ12, YPG+13]. The FTL interface available to NVMKV is detailed in Table 4.2.

Dynamic Mapping

Conventional SSDs provide a dense address space, with one logical address for every advertised available physical block. This matches the classic storage model, but forces applications to maintain separate indexes to map items to the available LBAs. *Sparse address spaces* are available in advanced FTLs which allow applications to address the device via a large, thinly provisioned, virtual address space [ONW+11, SSZ12, YPG+13]. Sparse address entries are allocated physical space only upon a write. In the NVMKV context, a large address space enables simple mapping techniques such as hashing to be used with manageable collision rates.

Additional primitives are required to work with sparse address spaces. `EXISTS` queries whether a particular sparse address is populated. `PTRIM` is a persistent and atomic deletion of the contents at a sparse address. These primitives can be used for individual, or ranges of, locations. For example, `RANGE-EXISTS` returns a subset of a given virtual address range that has been populated. The `ITERATE` primitive is used to cycle through all populated virtual addresses, whereby `ITERATE` takes a virtual address and returns the next populated virtual address.

Transactional Persistence

The transactional persistence capabilities of the FTL are provided by `ATOMIC-WRITE` and `PTRIM` [DAT+14, JBLF12]. `ATOMIC-WRITE` allows a sparse address range to be written as a single ACID compliant transaction.

Optimized Parallel Operations

The FTL is well-placed to optimize simultaneous device-level operations. Two classes of FTL primitives, *conditional* and *batch*, provide atomic parallel operations that are well-utilized by NVMKV. For example, `cond_atomic_write` allows for an atomic write to be completed only if a particular condition is satisfied, such as the LBA being written to is not already populated. This primitive removes the need to issue separate `exists` and `atomic_write` operations. Batch or vectored versions of all primitives are also implemented into the FTL (such as `batch_read`, `batch_atomic_write`, and `batch_ptrim`) to amortize lock acquisition and system call overhead. The benefits of batch (or vector) operations have been explored earlier [VKA12b].

4.5 Design Description

NVMKV implements novel techniques to make sparse addressing practical and efficient for use in KV stores and for providing low-latency, transactional persistence.

4.5.1 Mapping Keys via Hashing

Conventional KV stores employ two layers of translations to map keys to flash device locations, both of which need to be persistent [Mon14, Roc14, AFK$^+$09]. The first layer translates keys to LBAs. The second layer (i.e., the FTL) translates the LBAs to physical locations in flash device. NVMKV leverages the FTLs sparse address space and encodes keys into sparse LBAs via hashing, thus collapsing an entire layer.

NVMKV divides the sparse address into equal sized virtual slots, each of which stores a single KV pair. More specifically, the sparse address space (with addressability through N bits) is divided into two areas: the *Key Bit Range* (KBR) and the *Value Bit Range*

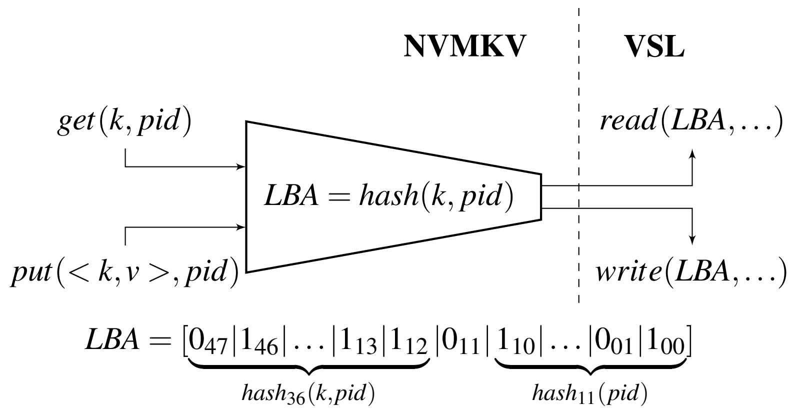

Figure 4.3: Hash model used in NVMKV. The arguments to the functions represent k:key, v:value, and pid:pool_id.

(VBR). This division can be set by the user at the time of creating the NVMKV store. The VBR defines the amount of contiguous address space (i.e., maximum value size or virtual slot size) reserved for each KV pair. The KBR determines the maximum number of such KV pairs that can be stored in a given KV store. In the expected use cases, the sparse virtual address range provided by the KBR will still be several orders of magnitude larger than the number of KV pairs as limited by the physical media.

The keys are mapped to LBAs through a simple hash model as shown in Figure 4.3. User supplied keys can be of variable length up to the maximum supported key size. To handle a `put` operation, the specified key is hashed into an address which also provides its KBR value. The maximum size of the information (Key, Value, metadata) that can be stored in a given VBR is half of the size addressed by the VBR. For example, if the VBR is 11 bits and each address represents a 512B sector, a given VBR value can address 2 MB.

The above layout guarantees the following two properties. First, each VBR contains exactly one KV pair, ensuring that we can quickly and deterministically search and identify KV pairs stored in the flash device. Second, no KV pairs will be adjacent in the sparse address space. In other words, there is always unpopulated virtual addresses between every KV pair. This deliberately wasted virtual space does not translate into unutilized storage since it is in virtual and physical space. These two properties are critical for NVMKV as the value size and exact start location of the KV pair are not stored as part of NVMKV metadata but are inferred via the FTL. Doing so helps in significantly reducing the in-memory metadata footprint of NVMKV. Non-adjacent KV pairs in the sparse address space help in determining the value size along with the starting virtual address of each KV pair. To determine the value size, NVMKV issues a `range_exist` call to the FTL.

A direct consequence of this design is that every access pattern becomes a random pattern, losing any possible order in the key space. The decision to not preserve sequentiality was shaped by two factors: metadata overhead and flash I/O performance. To ensure sequential writes for contiguous keys, additional metadata would be required. This metadata would have to be consulted when reading, updated after writing, and cached in RAM to speed up the lookups. While straightforward to implement, doing so is unnecessary since the performance gap between random and sequential access for flash is ever decreasing; for current high performance flash devices, it is practically non-existent.

4.5.2 Handling Hash Collisions

Hashing variable sized keys into fixed size virtual slots could result in collisions. Since each VBR contains exactly one KV pair, hash conflicts only occur in the KBR. To illustrate the collision considerations, consider the following example. A 1TB Fusion-io ioDrive can contain a maximum of 2^{31} (2 billion) keys. Given a 48 bit sparse address space with 36 bits for KBR and 12 bits for VBR, NVMKV would accommodate 2^{36} keys with a maximum value size of 512KB. Under the simplifying assumption that our hash function uniformly distributes keys across the value ranges, for a fully-utilized 1TB io-

Drive, the chances of a new key insertion resulting in a collision is $1/2^5$ or a little under 3 percent.

NVMKV implicitly assumes that the number of KBR values is sufficiently large, relative to the number of keys that can be stored in a flash device, so that the chances of a hash collision are small. If the KV pair sizes are increased, the likelihood of a collision reduces because the device can accommodate fewer keys while preserving the size of the key address space. If the size of the sparse address space is reduced, the chances of a collision will increase. Likewise, if the size of the flash device is increased without increasing the size of its sparse address space, the likelihood of a collision will increase.

Collisions are handled deterministically by computing alternate hash locations using either linear or polynomial probing. By default, NVMKV uses polynomial probing and up to eight hash locations are tried before NVMKV refuses to accept a new key. With this current scheme, the probability of a `put` failing due to hash failure is vanishingly small. Assuming that the hash function uniformly distributes keys, the probability of a `put` failing equals the probability of 8 consecutive collisions. This is approximately $(1/2^5)^8 = 1/2^{40}$, roughly one failure per trillion `put` operations. The above analysis assumes that the hash function used is well modeled by a uniformly distributing random function. Currently, NVMKV uses the FNV1a hash function [FNV91] and we experimentally validated our modeling assumption.

4.5.3 Caching

Caching is employed in two distinct ways within NVMKV. First, a read cache speeds up access to frequently read KV pairs. NVMKV's read cache implementation is based on LevelDB's cache [GD11]. The read cache size is configurable at load time. Second, NVMKV uses a collision cache to improve collision handling performance. It caches

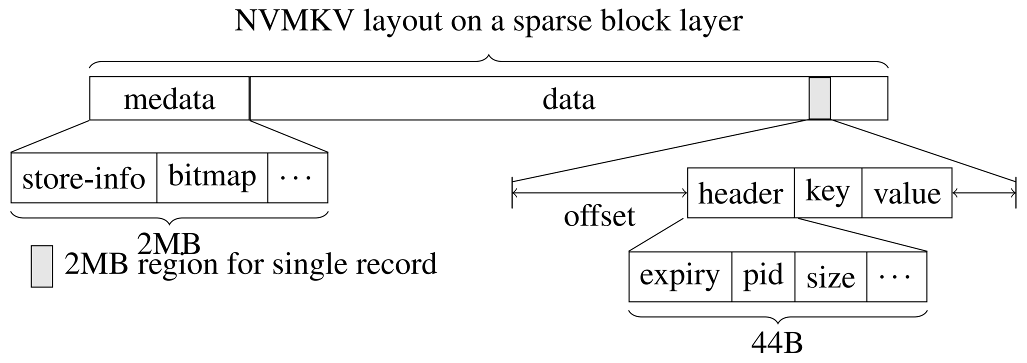

Figure 4.4: NVMKV layout

the key hash (the sparse LBA) along with the actual key which is used during `puts` (i.e., inserts or updates). If the cached key matches the key to be inserted, the new value can be stored in the corresponding slot (the key's hash value). This significantly reduces the number of additional I/Os needed during collision resolution. In most cases, only a single I/O is needed for a `get` or a `put` to return or store the KV pair.

4.5.4 KV Pair Storage and Iteration

KV pairs are directly mapped to a physical location in the flash device and addressable through the FTL's sparse address space. In our current implementation, the minimum unit of storage is a sector and KV pairs requiring less than 512B will consume a full 512B sector. Each KV pair also contains metadata stored on media. The metadata layout is shown in Figure 4.4; it includes the length of the key, the length or the value, pool identifier (to be discussed further in §4.6), and other information.

To minimize internal fragmentation, NVMKV packs and stores the metadata, the key, and the value in a single sector whenever possible. If the size of the KV pair and the metadata is greater than a sector, NVMKV packs the metadata and key into the first sector and stores the value starting from the second sector. This layout allows for optimal storage efficiency for small values and zero-copy data transfer into the users buffer for larger values.

NVMKV supports unordered iteration through all KV pairs stored in the flash device. Key iteration is accomplished by iterating across the populated virtual addresses inside the FTL in order. The iterator utilizes the `ITERATE` primitive in the FTL, which takes in the previously reported start virtual address and returns the start address of the next contiguously populated virtual address segment in the sparse address space. Note that this approach relies on the layout guarantee that each KV pair is located contiguously in a range of virtual addresses, and that there are unpopulated virtual addresses in between each KV pair.

4.5.5 Optimizing KV Operations

NVMKV's design goal is one I/O for a `get` or a `put` operation. For `get`, this is achieved with the KV data layout and the `CONDITIONAL-RANGEREAD` primitive. The layout guarantees that individual KV pairs occupy a single contiguous section of the sparse address space, separated from other KV pairs by unpopulated virtual addresses. Given this, a `CONDITIONAL-RANGEREAD` can retrieve the entire KV pair in one operation without knowing the size of the value up front. Second, collisions induce a minimal number of additional operations. Since `get` and `put` operations for a given key map to hash addresses in a deterministic order, and since `put` places the new KV pair at the first available hash address (that is currently unused) in this order, subsequent `get`s are guaranteed to retrieve the most recent data written to this key. Finally, `DELETE` operations may require more than one I/O per operation, since they are required to read and validate the key before issuing a `PTRIM`. It also needs to check multiple locations to ensure that previous instances of a particular key have all been deleted.

NVMKV is intended to be zero copy and avoid memory comparison operations wherever possible. First, for any value that is large enough to start at its own sector, the data retrieved from a `get` (or written during a `put`) operation will be transferred directly to (or from) the user provided memory buffer. Second, no key comparisons occur unless the key hashes match. Given that the likelihood of collisions is small, the number of key comparisons that fail is also correspondingly small.

4.6 Multiple KV Instances Via Pools

Pools in NVMKV allow applications to group related keys into logical abstractions that can then be managed separately. Besides simplifying KV data management for applications, pools enable efficient access and iteration of related keys. The ability to categorize or group KV pairs also improves the lifetime of flash devices.

4.6.1 Need for Pools

NVMKV as described thus far, can support multiple independent KV stores. However, it would need to either partition the physical flash device to create multiple block devices each with its own sparse address space or logically partition the single sparse address space to create block devices to run multiple instances of KV stores.

Unfortunately, both approaches do not work well for flash. Since it is difficult to predict the number of KV pairs or physical storage needed in advance, static partitioning would result in either underutilization or insufficient physical capacity for KV pairs. Further, smaller capacity physical devices would increase pressure on the garbage collector, resulting in both increased write amplification and reduced KV store performance. Alternatively, partitioning the LBA space would induce higher key collision rates as the KBR would be shrunk depending on the number of pools that need to be supported.

4.6.2 Design Overview

NVMKV encodes pools within the sparse LBA to avoid any need for additional in-memory pool metadata. The encoding is done by directly hashing both the pool ID and the key to determine the hash location within the KBR. This ensures that all KV pairs are equally distributed across the sparse virtual address space regardless of which pool they are in. Distributing KV pairs of multiple pools evenly across the sparse address space not only retains the collision probability properties but also preserves the `get` and `put` performance with pools.

Pool IDs are also encoded within the VBR to optimally search or locate pools within the sparse address space. Encoding pool IDs within the VBR preserves the collision properties of NVMKV. The KV pair start offset within the VBR determines the Pool ID. The VBR size determines the maximum number of pools that can be addressed without hashing, while also maintaining the guarantee that each KV pair is separated from neighboring KV pairs by unpopulated sparse addresses. For example, with a 12 bit VBR, the maximum number of pools that can be supported without pool ID hashing is 1024. If the maximum number of pools is greater than 1024, the logic of `get` is modified to also retrieve the KV pair metadata that contains the pool ID now needed to uniquely identify the KV pair.

4.6.3 Operations

Supporting pools requires changes to common operations of the KV store. We now describe three important operations in NVMKV that have either been added or significantly modified to support pools.

Creation and Deletion: Pool creation is a lightweight operation. The KV store performs a one-time write to record the pool's creation in its persistent configuration metadata. On the other hand, pool deletion is an expensive operation since all the KV pairs of a pool are distributed across the entire LBA space, each requiring an independent `PTRIM` operation. NVMKV implements pool deletion as an asynchronous background operation. Upon receiving the deletion request, the library marks the pool as invalid in its on-drive metadata, and the actual deletion of pool data occurs asynchronously.

Iteration: NVMKV supports iteration of all KV pairs in a given pool. If no pool is specified, all key-value pairs on the device are returned by the iteration routines. Iteration uses the `ITERATE` primitive of the FTL to find the address of the next contiguous chunk of data in the sparse address space. During pool iteration, each contiguous virtual address segment is examined as before. However, the iterator also examines the offset within the VBR of each starting address, and compares it to the pool ID, or the hash of the pool ID, for which iteration is being performed. Virtual addresses are only returned to the KV store if the pool ID match succeeds.

NVMKV guarantees that each KV pair is stored in a contiguous chunk, and that adjacent KV pairs are always separated by at least one empty sector, so the address returned by `ITERATE` locates the next KV pair on the drive (see §4.5.1). When the maximum number of pools is small enough that each pool ID can be individually mapped to a distinct VBR offset, the virtual addresses returned by the `ITERATE` primitive are guaranteed to belong to the pool currently being iterated upon. When the maximum number of pools is larger, the `ITERATE` uses the hash of the pool ID for comparison. In this case, the virtual addresses that match are not guaranteed to be part of the current pool, and a read of the first sector of the KV pair is required to complete the match.

4.7 Implementation

NVMKV is implemented as a stand-alone KV store written in C++ using 6300 LoC. Our current prototype works on top of ioMemory VSL and interacts with the FTL using the IOCTL interface [Fus]. The default subdivision for KBR and VBR used in the current implementation is 36 bits and 12 bits respectively, in a 48 bit address space. The KBR/VBR subdivision is also configurable at KV store creation time. To accelerate pool iteration, we implemented filters inside the `ITERATE/BATCH-ITERATE` FTL primitives. During the iteration of keys from a particular pool, the hash value of the pool is passed along with the IOCTL arguments to be used as a filter for the iteration. The FTL services `(BATCH-)ITERATE` by returning only populated ranges that match the filter. This reduces data copying across the FTL and NVMKV.

4.7.1 Extending FTL Primitives

We extended the FTL to better support NVMKV. `ATOMIC-WRITE` and its vectored forms are implemented in a manner similar to what has been described by Ouyang *et al.* [ONW+11]. Atomic operations are tagged within the FTL log structure, and upon restart, any incomplete atomic operations are discarded. Atomic writes are also not updated in the FTL map until they are committed to the FTL log to prevent returning partial results. `ITERATE` and `RANGE-EXISTS` are implemented as query operations over the FTL indirection map. `CONDITIONAL-READ and CONDITIONAL-WRITE` are emulated within NVMKV in the current implementation.

4.7.2 Going Beyond Traditional KV Stores

NVMKV provides new capabilities with strong guarantees relative to traditional KV stores. Specifically, it provides full atomicity, isolation, and consequently serializability for basic operations in both individual and batch submissions. Atomicity and serializability guarantees are provided for individual operations within a batch, not for the batch itself. The atomicity and isolation guarantees provided by NVMKV rely heavily on the `ATOMIC-WRITE` and `PTRIM` primitives from the FTL.

Each `put` is executed as a single ACID compliant `ATOMIC-WRITE`, which guarantees that no `get` running in parallel will see partial content for a KV pair. The `get` operation opportunistically retrieves the KV pair from the first hash location using `cond_range_read` to guarantee the smallest possible data transfer. In the unlikely event of a hash collision, the next hash address is used. Since the hash address order is deterministic, and every `get` or `put` to the same key will follow the same order, and every write has atomicity and isolation properties, `get` is natively thread safe requiring no locking.

When `ATOMIC-WRITE`s are used, `put` operations require locking for thread safety because multiple keys can map to the same KBR. When a `CONDITIONAL-WRITE` (which performs an atomic `EXISTS` check and `WRITE` of the data in question) is used, `put` operations can also be made natively thread safe. Individual iterator calls are thread safe with respect to each other and to `get`/`put` calls; thus, concurrent iterators can execute safely.

The `ITERATE` primitive is also supported in batch mode for performance. `BATCH-ITERATE` returns multiple start addresses in each invocation, reducing the number of IOCTL calls. For each LBA range returned, the first sector needs to be read to retrieve the key for the target KV pair.

4.8 Evaluation

Our evaluation addresses a new set of questions:

(1) How effective is NVMKV in supporting multiple KV stores on the same flash device? How well do NVMKV *pools* scale?

(2) How effective is NVMKV in trading off DRAM for flash by sizing its read cache?

(3) How effective is NVMKV in improving the endurance of the underlying flash device?

(4) How sensitive is NVMKV to the size of its collision cache?

4.8.1 Workloads and Testbed

We use LevelDB [GD11], a well-known KV store as the baseline for our evaluation of NVMKV. LevelDB uses a logging-based approach to write to flash and uses compaction mechanisms for space reclamation. Our evaluation consists of two parts. We answer question (1) above using dedicated micro-benchmarks for NVMKV and LevelDB. We then answer questions (2), (3), and (4) using the YCSB macro-benchmark [CST+10]. We used the YCSB workloads A, B, C, D, and F with a data set size of 10GB; workload E performs short range-scans and the YCSB Java binding for NVMKV does not support this feature currently. We also used a raw device I/O micro-benchmark which was configured so that I/O sizes were comparable to the sizes of the key-value pairs in NVMKV. Our experiments were performed on two testbeds, I and II. Testbed I was a system with a Quad-Core 3.5 GHz AMD Opteron(tm) Processor, 8GB of DDR2 RAM, and a 825GB Fusion-io ioScale2 drive running Linux Ubuntu 12.04 LTS. Testbed II was a system with a 32 core Intel(R) Xeon(R) CPU E5-2690 0 @ 2.90GHz with 128 GB DDR3 RAM, and a 1.2TB Fusion-io ioDrive2 running Linux Ubuntu 12.04.2 LTS.

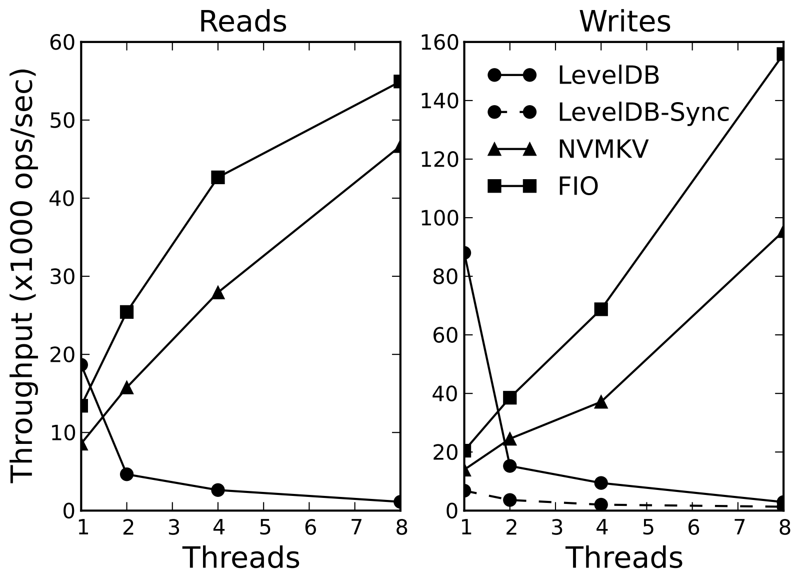

Figure 4.5: Microbenchmark comparing the throughput of LevelDB, NMVKV and Raw Block Device (FIO) for small (1KB) IO operations.

4.8.2 Micro-Benchmarks

Our first experiment evaluates the overhead of the NVMKV stack and LevelDB relative to the raw flash device. Figure 4.5 summarizes the results. `Get` and `put` used 512B values and key sizes ranging from 1 byte to 128 bytes and for these sizes, NVMKV issues I/O operations of size 1KB. The FIO tool was configured to generate 1KB I/Os to the raw device. Figure 4.5 shows that as the thread count increases, the throughput for NVMKV's `get` operations tracks the FIO benchmark's `read` rate. For `put` operations, NVMKV significantly outperforms both the asynchronous and synchronous versions of LevelDB. Additional overheads in NVMKV such as checking for collisions cause performance to be lower than the underlying native device `write` performance extracted by FIO.

Next we ran a single instance of NVMKV and measured the throughput of reads and writes as functions of the number of NVMKV pools. NVMKV also used as many threads as pools. We compared its performance against multiple instances of LevelDB. Both KV stores were configured to use the same workload, sizes of key-value pairs, and accessed a

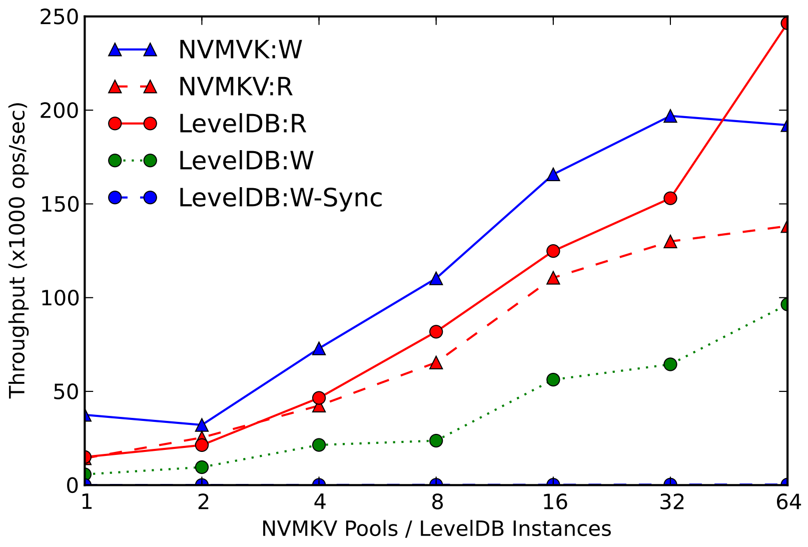

Figure 4.6: Comparing multiple identical instances of LevelDB with a single instance of NVMKV with equal number of pools.

total of 500 MB of data. In addition, LevelDB used both its own user-level cache of size 1GB and the operating system's file system cache as well. On the other hand, NVMKV used neither. LevelDB provides two options for writes, a low-performing but durable *sync* and the high performing *async*, and we include them both here. NVMKV, on the other hand, performs all writes synchronously and atomically, and thus only a synchronous configuration is possible.

Figure 4.6 provides a performance comparison. Due to its low-latency flash-level operations, NVMKV almost equals LevelDB's primarily in-memory performance for up to 32 pools/instances. LevelDB continues scaling beyond 32 parallel threads; its operations continue to be memory-cache hits while NVMKV must perform flash-level accesses (wherein parallelism is limited) for each operation. When writing, NVMKV outperforms LevelDB's *sync* as well as *async versions* despite not using the filesystem cache at all. Even when LevelDB was configured to use async writes, it was about 2x slower than NVMKV in the best case, and about 6.5x slower at its worst. For synchronous writes, a more comparable setup, NVMKV outperforms LevelDB between 643x (64 pools) and 1030x faster (1 pool).

4.8.3 DRAM Trade-off and Endurance

This second experiment using Testbed I addresses questions (2) and (3). NVMKV uses negligible in-memory metadata and does not use the operating system's page cache at all. It implements a read cache whose size can be configured, allowing us to trade-off DRAM for flash, thus providing a tunable knob for trading off cost for performance. To evaluate the effectiveness of the collision cache, we evaluate two variants of NVMKV, one without the collision cache and the other when it uses 64MB of collision cache space. We used the YCSB benchmark for this experiment. We present the results from both the *load phase*, that is common to all workload personalities implemented in YCSB, and the *execution phase*, that is distinct across the workloads.

Figure 4.7 (top) depicts throughput as a function of the size of the application-level read cache available to LevelDB and NVMKV. Unlike NVMKV, LevelDB accesses use the file system page cache as well. Despite this, NVMKV outperforms LevelDB during both phases of the experiment, *load* and *execution*, by a significant margin. Further, the gap in performance increases as the size of the cache increases for every workload. This is because YCSB's workloads favor reads in general, varying from 50%, in the case of workload A, all the way to 100% in the case of workload C. Furthermore, the YCSB workloads follow skewed data access distributions, making even a small amount of cache highly effective.

To better understand these results, we also collected how much data was written to the media while the experiments were running. All workload were configured to use 10GB of data, so any extra data that is written to the media is overhead introduced by NVMKV

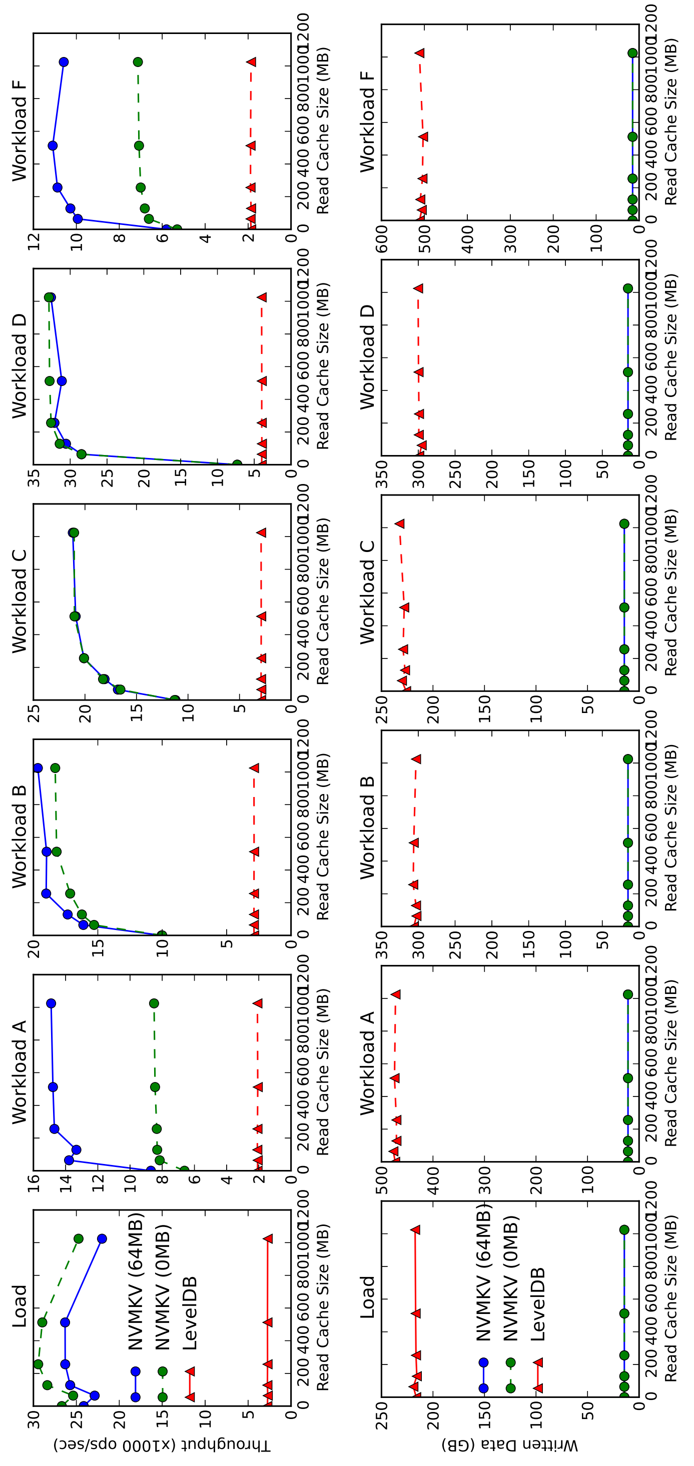

Figure 4.7: Throughput comparison between NVMKV and LevelDB using YCSB workloads (above). Write Amplification Comparison between NVMKV and LevelDB using YCSB workloads (below).

or LevelDB. Figure 4.7 (bottom) depicts the results of the write amplification. By the end of each experiment, LevelDB has written anywhere from 42.5x to 70x extra data to the media. This seems to be a direct consequence of its internal design which migrates the data from one level to the next, therefore copying the same data multiple times as it ages. NVMKV on the other hand, introduces a write amplification of 2x in the worst case. We believe this to be the main reason for the performance difference between the two KV Stores.

Finally, the effect of NVMKV's collision cache is highly sensitive to the workload type. As expected, for read-intensive workloads (i.e., B, C, and D) the presence of collision cache has little to no impact at all. On the other hand, the update heavy workloads (i.e., A and F) benefit significantly from the collision cache, increasing the performance up to 76% and 56% respectively. Surprisingly the load phase is negatively affected by the collision cache, and performance decreases by up to 11%.

4.8.4 Effectiveness of the Collision Cache

We used Testbed II to address question (4). We measured YCSB workload throughput when varying the size of the collision cache in NVMKV. During this experiment, the read cache was disabled to eliminate other caching effects. As shown in Figure 4.8, the presence of the collision cache benefits workloads A and F with a throughput improvement of 28% and 10% respectively. Read-mostly workloads (B, C, and D) do not benefit from the collision cache since the probability of collision is low and a single flash-level read is necessary to service KV `GET` operations. A and F involve writes and these benefit from the collision cache. The collision cache optimizes the handling repeated writes to the same location by eliminating the reading of the location (to check for collisions) prior to the write. Finally, the loading phase does not demonstrate any benefit from the

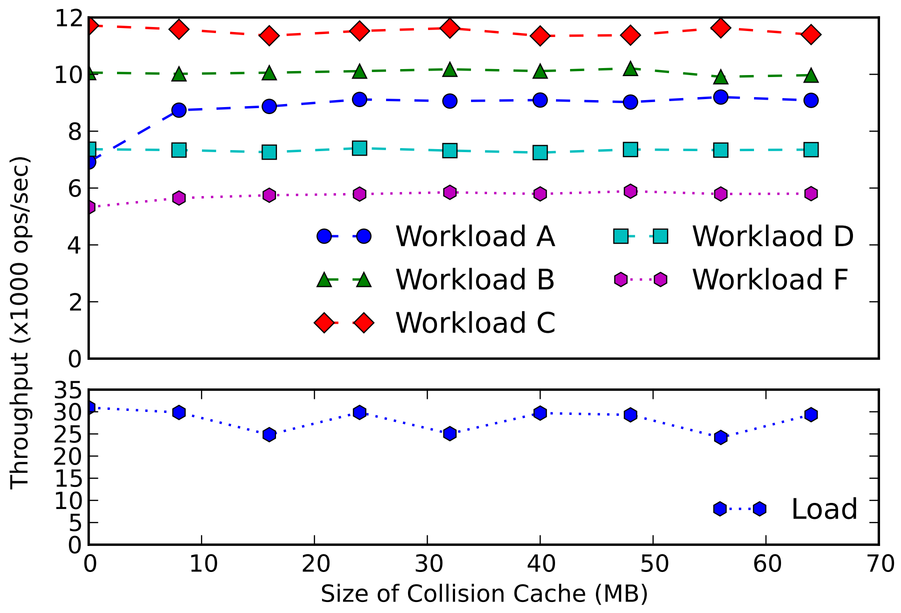

Figure 4.8: Collision cache impact on YCSB workloads.

collision cache mainly because of YCSB's key randomization during inserts.

4.9 Discussion and Limitations

Through the NVMKV implementation, we were able to achieve a majority of our design goals of building a flash-aware lightweight KV store that leverages advanced FTL capabilities. We made several observations through the design and development process.

It is valuable for atomic KV operations, such as those described by Ouyang *et al.* [ONW+11], to be fully ACID compliant. The usage described in Ouyang *et al.*'s work only required the durable writes to have the atomicity property. We found that having *isolation* and *consistency* enables reduced locking and in some cases, fully lock free operation, at the application level. For example, updating multiple KV pairs atomically as a single batch can help provide application-level consistent KV store state without requiring additional locks or logs.

Many primitives required by NVMKV are the same as those required by other usages of flash. FlashTier, a primitives based solid state cache leverages the sparse addressing model, and the EXISTS and PTRIM FTL primitives [SSZ12], as does DirectFS, a primitives based filesystem [Dir, JBLF12].

NVMKV suffers from internal fragmentation for small KV pairs. Since we map individual KV pairs to separate sectors, NVMKV will consume an entire sector (512B) even for KV pairs smaller than the sector size. While this does not pose a problem for many workloads, there are those for which it does. For the second group of workloads, NVMKV will have poor capacity utilization. One way to manage efficient storage of small KV pairs is to follow a multi-level storage mechanism, as provided in SILT [LFAK11], where small items are initially indexed separately and later compacted into larger units such as sectors. We believe that implementing similar methods within the FTL itself can be valuable.

4.10 Summary

Leveraging powerful FTL primitives provided by a flash device allows for rapid and stable code development; application developers can exploit features present in the FTL instead of re-implementing their own mechanisms. NVMKV serves as an example of leveraging and enhancing capabilities of an FTL to build simple, lightweight but highly powerful applications. Through the NVMKV design and implementation, we demonstrated the impact to a KV store in terms of code and programming simplicity and the resulting scalable performance that comes from cooperative interaction between the application and the FTL. We believe that the usefulness of primitives for FTLs will only grow. In time, such primitives will fundamentally simplify applications by enabling developers to quickly create simple but powerful, feature-rich applications with performance comparable to raw devices.

CHAPTER 5
METRADB

In this chapter we propose a new type of key-value stored tailored for NVM. We describe the unique characteristics of this new type of storage technologies and highly their potential for improving applications performance. More importantly, we also identify the challenges associated with programming NVM and propose a solution that offers a tradeoff that favors simplicity and performance over generality.

5.1 Introduction

Non-volatile memory, or NVM, is coming. Several technologies are maturing (FeRAM, ReRAM, PCM, DWM, FJG RAM), and soon we expect products from Intel, Micron, HP, SanDisk, and/or Samsung. Some of these products promise memory density close to flash and performance within a reasonable factor of DRAM. This technology could substantially improve the performance of software systems, especially storage systems.

Unfortunately, using NVM is hard: each technology has its quirks, and the details of products are not yet available. We need a way to integrate NVM into our software systems, without full knowledge of all the NVM product details and without having to redesign every software system for each forthcoming NVM technology.

We advocate the use of *customized* key-value stores. Rather than programming directly on NVM, developers (1) design a key-value store customized for the application, (2) implement the key-value store for the target NVM technology, and (3) program the application using the key-value store. When new NVM products emerge, with similar performance characteristics but different access mechanisms, developers need only modify the key-value store implementation, which is simpler, faster, and cheaper than redesigning the application. Thus, the key-value store serves as a middle layer that hides the details of the NVM technology, while providing a simple and familiar interface to the application. Customization ensures that the design is performant and simple.

We illustrate this idea with an example, METRADB, a key-value store that we customize for VSAN, a distributed storage system offered by VMWARE. Key-value stores vary in functionality. Some provide rich data types; some have variable-length keys; and some support transactions with various assurances. The ideal store for an application depends on its needs. We explain the options, so that developers can decide what they need. In METRADB, we chose these options to satisfy the needs of VSAN.

We report on an early performance evaluation, which compares METRADB against a general solution provided by Intel, namely, the data structures in NVML [Rud]. We find that METRADB's performs 2.2x to 50x better in terms of latency, and its throughput scales well with the number of threads. This performance advantage comes from the customized functionality of METRADB. The trade-off of customization is that developers must build a store for each application. This was not a significant issue for METRADB: it has only 2.3K lines of C code.

5.2 NVM: benefits and challenges

By NVM we mean memory that preserves its contents when power is lost and that can be accessed at a granularity of bytes or words rather than blocks. While there are many NVM technologies, the most mature and promising one is 3D XPoint from Intel and Micron. This technology bring two key benefits relative to DRAM:

- *Non-volatility.* The memory survives power cycles, so software need not resort to slow disks or flash.
- *Density.* DRAM is limited to a few terabytes per machine, but NVM can grow to tens of terabytes.

Relative to disks and SSDs, NVM has two advantages:

- *Performance.* Disks and SSDs can incur large write latencies. NVM has performance closer to DRAM.
- *Fine granularity.* Disks and SSDs operate on 512- or 4096-byte blocks, making them inefficient for small operations; NVM operates on individual words.

However, NVM brings many challenges to developers:

- *Non-persistent caching:* While memory is non-volatile, memory caches are not. When the CPU stores data, it remains in a cache until it is flushed.
- *Out-of-order flushes:* Caches may be flushed without asking and out of order.
- *Torn writes:* Applications may explicitly flush data, but if power is lost during the flush, only some parts will be persistent.
- *Complex interface:* To persist data, applications must follow a ritual of instructions: flushing dirty cache lines, issuing a barrier, committing data, and issuing another barrier (§5.6.3). This ritual is expensive, so developers must worry about how to minimize its use.
- *Non-uniform wear:* As the hardware provides no wear leveling, if a word in NVM changes more often than another, it will wear out more quickly. This can lead to reliability issues if the developer is not careful.
- *Lack of details:* We lack details about cost and performance of NVM.

The first four issues above relate to crash recovery, while the last two issues relate to normal operation. We next describe a middle layer for accessing NVM that can mitigate these issues.

5.3 Main idea and rationale

Instead of programming on NVM directly, we believe developers should introduce a middle layer that hides the complexity of the NVM. When a new NVM technology emerges, developers need not modify the application, just the middle layer. This layer consists of a key-value store library with transactions. A key-value store offers operations to write and read key-value pairs, and possibly more functionality, depending on the store. A key-value store is a good abstraction: it is simple and familiar, it can be implemented easily, and it can perform well.

But a plain key-value store does not suffice to address the above challenges (§5.2); it must also provide *transactions* to avoid torn writes and facilitate crash recovery.

We propose that the key-value store be customized for an application. Many applications do not need all features possible in a key-value store (discussed next), and those features may hinder performance and simplicity.

5.4 Features of key-value stores

Broadly, key-value stores permit users to put, get, and delete key-value pairs, but key-value stores vary in their exact functionality and data models:

- *Key length.* Keys can be fixed- or variable-length, and fixed-length keys can be sparse (e.g., 8-byte integers) or dense (e.g., 0..32K).
- *Value length.* Values can be fixed- or variable-length.
- *Value types.* Values can be typed or opaque. Typed-values can offer richer operations, such as increment for numerical types or set union for set types.
- *Operations.* Besides gets and puts of key-value pairs, the store may support: (a) gets and puts of partial values given by an offset and length, (b) multigets and multiputs of many pairs at once, (c) key enumeration, ordered or not, and (d) read-modify-write.

- *Containers.* If present, containers provide different namespaces to avoid key clashes across users.
- *Transactions.* If present, transactions provide atomicity, isolation, or both. Atomicity protects against crashes and torn writes. Isolation protects against concurrent access.

5.5 Case study: VSAN

We now explain how NVM can be used for a specific application, a VMWARE product called VSAN. VSAN is a distributed storage system that provides logical volumes, where each volume is partitioned and replicated across storage servers. A server stores the volumes on disks or SSDs; in the former case, it can use SSDs as a server-side cache. In addition, clients cache data in memory for efficiency. We envision three uses of NVM in VSAN.

SSD cache metadata. Servers have an SSD block cache; this cache needs a map from SSD to disk LBAs. The map can exceed the DRAM allocated to VSAN. Moreover, reconstructing the map upon recovery can be costly. NVM can avoid these problems. This use case leverages NVM's non-volatility and larger capacity than DRAM.

Client cache. VSAN clients can benefit from a larger cache than fits in DRAM. This use case leverage's NVM's larger capacity.

Checksum storage. VSAN supports 32-byte checksums of 4 KB blocks for reliability. Unfortunately, disk storage is block aligned, making it inefficient to store the checksum. Various schemes are possible, but they cause fragmentation or increase the number of disk operations. NVM solves the problem because it operates on words. This use case leverages NVM's fine access granularity and non-volatility.

In all use cases, NVM accesses require low latency and high throughput since they affect VSAN's performance.

5.6 Key-value store design for VSAN

We now explain how we customize the design of METRADB for VSAN with 3D XPoint as the NVM technology. Our design employs existing techniques or variations; our goal is not to propose new mechanisms but rather to combine known mechanisms in a custom way for a specific application.

5.6.1 Features of METRADB

Key-value stores vary in functionality (§5.4). VSAN needs the following features, which we take as the requirements of METRADB:

Feature	**Choice for VSAN**
Keys	Fixed length
Values	Variable length, untyped
Operations	Get, Put, and Delete only
Containers	Yes
Transactions	Atomicity only, within one container

More precisely, in the VSAN use cases (§5.5), keys are always small binary identifiers, and values can be small values or buffers with generic lengths—hence the choice of fixed-length keys and variable-length untyped values. For the supported operations, it suffices to be able to read, write, and delete keys. Containers are needed because different components of VSAN will use the key value store. Finally, transactions with atomicity facilitate crash recovery, but transactions need not support isolation and need not span multiple containers because each container will be accessed by one thread and each thread will access one container at a time.

5.6.2 Application interface

The interface to the key-value store is below.

Operation	**Description**
$open(name, flags)$	open/create container, get handle
$remove(name)$	remove container
$close(h)$	close a handle
$put(h, k, buf, len)$	put key-value pair
$get(h, k, buf, len)$	get key-value pair
$delete(h, k)$	delete key-value pair
$commit(h)$	commit transaction
$abort(h)$	abort transaction

Broadly, there are container operations (open, remove, close), key-value operations (put, get, delete), and transactional operations (commit, rollback). Opening a container returns a handle to that container, which is later used to put and get key-value pairs. Puts, gets, and deletes are executed in the context of a transaction, which can be later committed or aborted. A transaction is limited to operate on a single container.

5.6.3 Performance considerations

To achieve good performance with 3D XPoint NVM, we must be concerned about three things: cache flushes, memory fences, and NVM commits. Specifically, the CPU caches are volatile; if an application wants its writes to persist, it must flush the dirty cache lines using CLFLUSH, CLFLUSHOPT, or CLWB instructions, then issue a memory fence using

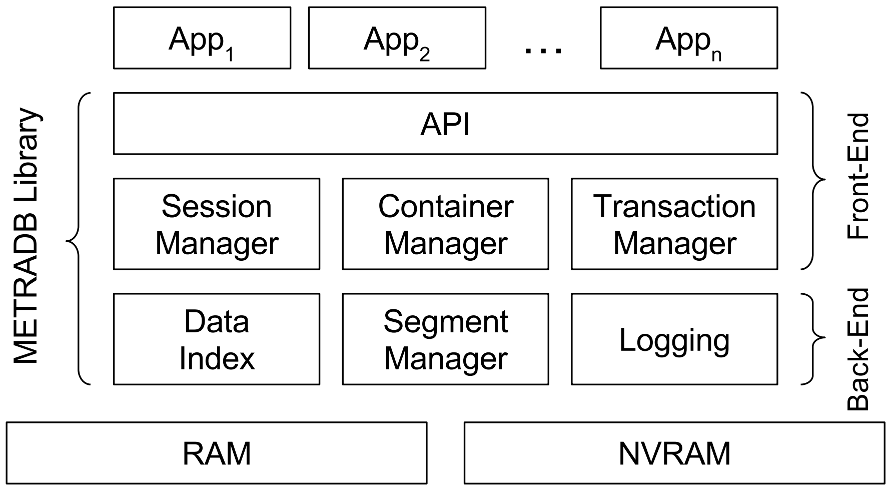

Figure 5.1: METRADB architecture

SFENCE to ensure the flushes are visible, then commit data to NVM using PCOMMIT.[1] To ensure the writes are ordered before subsequent writes, the application then needs to issue an additional fence using SFENCE. The challenge is that the flush, fence, and commit instructions are expensive and must be avoided to obtain good performance.

5.6.4 Architecture

Figure 5.1 shows the architecture of METRADB. METRADB is a library that links to the application. Internally, the library is organized as low-level back-end, which is specific to NVM technology (3D XPoint), and a high-level front-end, which is independent. The back-end includes modules for logging transactions, managing segments for memory allocation, and implementing the data index as a hash table. The front-end components manage the session with the application (open handles, in-memory indexes for performance, etc), keep track of existing containers, and execute transactions.

[1] Asynchronous DRAM Refresh could simplify matters, but it requires a special power supply and its details are not yet available.

Below, we give more details about the data structure (data index), the transaction mechanisms (transaction manager), and memory allocation (segment manager).

5.6.5 Data structure

We store each container in a hash table where each bucket has a head pointer to a doubly linked list. This is a simple and efficient data structure. Inserting a key-value pair requires just two writes: one write to a data buffer containing the key-value pair, one to the previous tail of the linked list. Minimizing writes is important in order to avoid expensive cache flushes. We considered using balanced trees or skip lists, but they were more complex and incurred additional writes.

Each hash table has a fixed size chosen at creation time. This is acceptable because we know the number of keys, within an order of magnitude, of the containers needed for each VSAN use case.

5.6.6 Transactions

We use logging to implement transactions, a well-known technique. There are two types of logs: redo or undo. Redo logs store information to reproduce the writes of a transaction, while undo logs store information to revert a transaction. Undo logs are simpler because transactions can update the underlying data as they execute. However, with NVM, it turns out that undo logs are much less efficient: they require multiple NVM commits per transaction because undo information must be committed to NVM prior to each transactional write (otherwise a power failure may leave the data modified without undo information). In contrast, a redo log need only be committed once to NVM, when the transaction commits. This is much more efficient, so we choose a redo log.

Specifically, as the transaction executes, we append its changes to the log. The log does not store the actual buffers with newly written data, but rather references to the buffers. This avoids subsequent copying overheads. When the transaction commits, we append a commit record, commit the log to NVM, and update the hash table to point to the written buffers.

METRADB uses transactions for two purposes: to provide user-level transactions within a container, and to execute metadata operations on containers and segments (e.g., create container, create segment). There are separate logs for these: one log per container for user-level transactions, and one global log for the metadata operations. We assume there is at most one thread executing a user-level transaction per container; for metadata transactions, we support multithreading using a global lock, which suffices as these transactions need not be efficient.

5.6.7 Memory allocation

We keep track of memory allocations using *slabs* stored in NVM. Each slab has buffers of a given length and a bitmap indicating buffer availability. There are many slabs, to provide buffers of many different sizes. When a transaction executes and allocates a buffer, it creates a DRAM copy of the bitmap of the appropriate slab. This copy is called a *shadow bitmap*. The shadow bitmaps track the tentative allocations done by the transaction. Allocations within a container are done as part of the transaction that allocates the buffer. When the transaction commits, we copy the modified shadow bitmaps into the real slab bitmaps in NVM. Note that a transaction commit need not copy any data buffers.

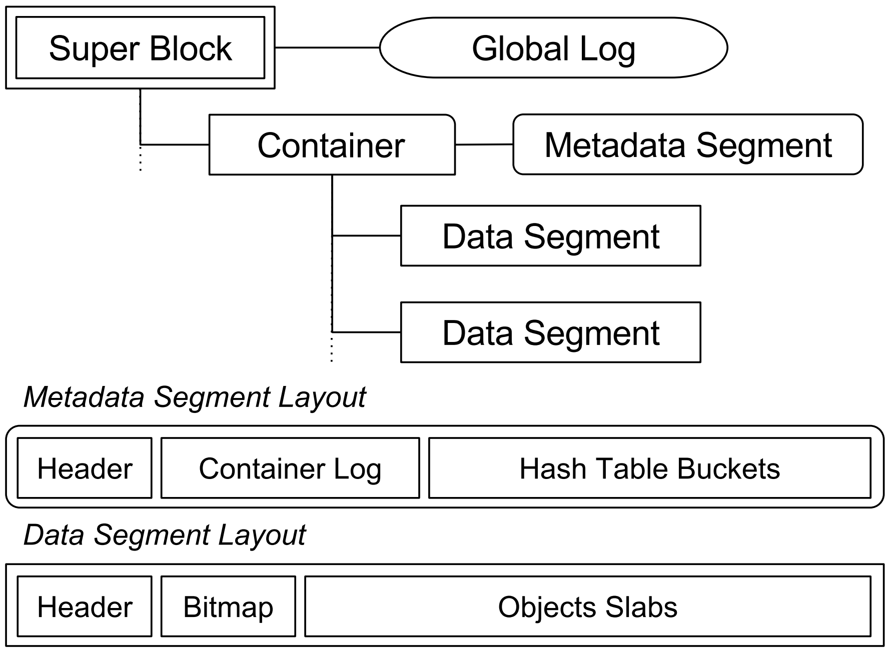

Figure 5.2: Data layout in METRADB

5.6.8 Data layout

Figure 5.2 shows the layout of data in NVM. Broadly, the NVM is organized as a super block, a global log, containers, and segments. The super block indexes all containers and their segments. The global log stores transactional updates on the super block, and the allocation of containers and segments. Every container consists of a metadata segment and several data segments. The metadata segment has a log for the transactions within its container, and the buckets of the container's hash table (pointers to the heads of doubly linked lists). The data segments store the slabs with the actual data of the hash table (elements in the doubly linked list), with each segment holding slabs of a fixed size; a bitmap indicates the allocated slab entries.

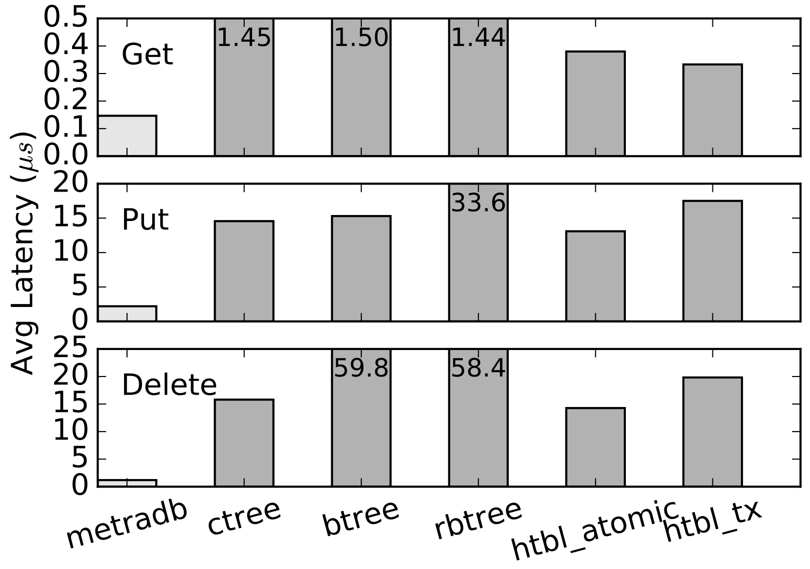

Figure 5.3: Latency of each operation in each system.

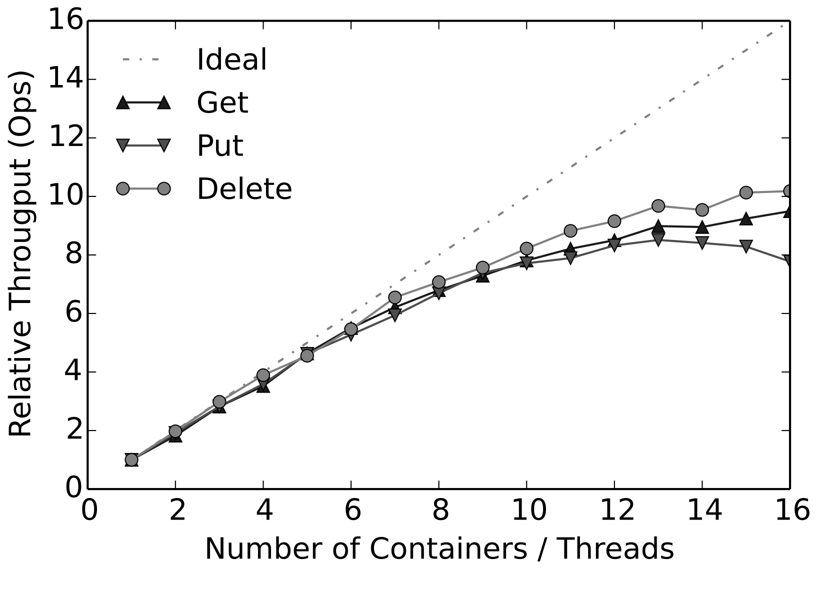

Figure 5.4: Throughput scalability of METRADB

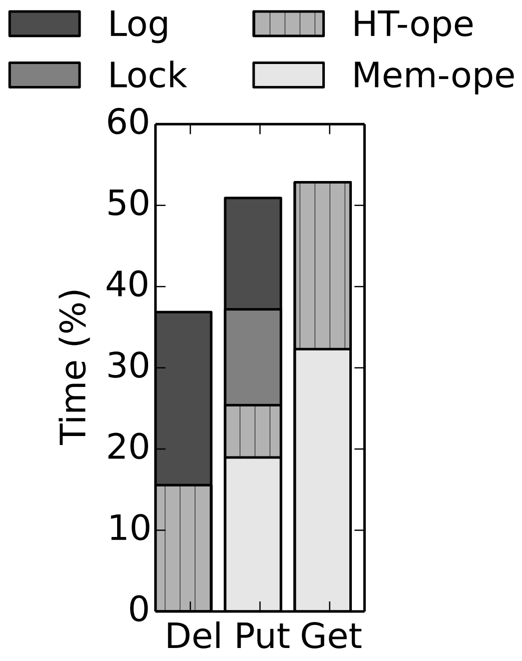

Figure 5.5: METRADB execution breakdown.

5.7 Evaluation

The evaluation of METRADB has three goals: (1) assess the latency of put, get, and delete operations, (2) assess throughput scalability of these operations, and (3) identify any bottlenecks in the design or implementation.

We compare METRADB against Intel's NVML, which is a library for facilitating the programming on NVM. The library provides atomic allocations of persistent objects, concurrent transactional updates, thread synchronization, and persistent pointers. It also contains several persistent data structures, which we use as key-value stores for comparison.

5.7.1 Methodology

We measure operation latency using the benchmark provided by NVML. We configure the benchmark to run lookup (get), insert (put), and delete operations on 250K key-value pairs with 1KB values. We compare METRADB against five data structures provided in NVML: radix-tree (ctree), B-tree (btree), red-black tree (rbtree), and two hash-tables (htbl_atomic and htbl_tx). These provide more features than METRADB (e.g., general transactions), since METRADB is customized to the needs of VSAN (§5.6); our goal is to understand the benefits of customization and the costs of generality.

We also measure throughput scalability of METRADB by running the benchmark with a variable number of threads, each accessing its own container.

To analyze METRADB's behavior, we run the benchmark under a profiler. We classify the execution time into five categories: Log (logging data), Lock (synchronization), HT-ope (hash table operations), Mem-ope (memcpy operation), and Other (remaining executing time).

5.7.2 Testbed

We run experiments on Linux with kernel v4.4, 24 GB of RAM, and an Intel XeonE5-2440 v2 1.90GHz CPU with 8 cores, each with 2 hyper-threads. NVM is not yet available, so instead we flush data (CLFLUSH) to a region of DRAM that is memory-mapped to a file.

5.7.3 Analysis

Figure 5.3 shows the latency of each system. The y-axis indicates average latency per operation in μs. The first bar is METRADB; the other bars are the various NVML data structures. On each operation, METRADB outperforms the best NVML data structure, htbl_atomic. The other NVML data structures are more powerful, but this power is not needed for VSAN, reinforcing the benefit of customization. The advantage of METRADB is greater on put and delete operations: 6.6–15x for puts and 12–50x for deletes, depending on the NVML structure. NVML is worse as it logs and commits to NVM many times per operation/transaction (and more for deletes than puts) due to the use of an undo log, while METRADB can commit the log only once per operation/transaction due to the use of a redo log. Using a redo log is feasible due to METRADB's limited functionality: redo logs in general cause complexity and overheads because the system needs special mechanisms for transactions to see their own updates; with METRADB, however, these mechanisms are straightforward and efficient since METRADB has only two simple update operations—a benefit of customization. For get operations, METRADB is better than NVML by 2.2–10.2x. This difference is due to the use of pointers with an extra level of indirection in NVML; these pointers permit the key-value store to be shared across different address spaces, but this is not needed for VSAN—another benefit of customization.

Figure 5.4 depicts how METRADB's throughput scales as the number of threads increases, each thread accessing its own container. The y-axis is normalized to the throughput of one thread, and the dashed line shows ideal scalability. We can see that METRADB scales almost linearly up to 8 threads, which is the number of CPU cores. Beyond 8, scalability suffers. Profiling information indicates that the main cause is kernel-level locks that synchronize parallel updates to the memory-mapped file.

Figure 5.5 shows a breakdown of the execution of gets, puts, and deletes. For gets, the cost is dominated by memcpy (Mem-ope) and the hash table (HT-ope). Puts spend comparable time on memcpy, locking, and logging; locks are acquired when the operation needs to create new data segments; logging is expensive because it writes to NVM. Deletes do not incur the overheads of memcpy or locking, and therefore are dominated by the hash table and logging. Overall, we see that the inherent cost of memcpy (for gets and puts) is comparable to the overheads imposed by the store for the data structure, locking, and logging. This indicates that there are no obvious bottlenecks in any of the operations.

5.8 Summary

In this chapter we propose that applications consume NVM through a middle layer, rather than programming it directly. Additionally, we argue that this middle layer should be tailored to address the needs of specific types of applications instead of providing a one-size-fits-all solution. In our case, the application in question was VSAN: a hyper-converge storage solution from VMWARE. We concluded that for VSAN, a key-value store interface with support for transactional updates was appropriated. We compared our solution to a more general one [NVM15] and demonstrated that by making the necessary tradeoffs, a higher performing solution could be created in about 2.3K LOC. Even though METRADB was built specifically for VSAN, we believe it is relevant to other systems as well, and we hope that the discussions around why we made certain design decisions help shape future solutions.

METRADB has two main limitations that must be kept in mind. The first of these limitation is related to its transactional model. METRADB only supports one active transaction per container that cannot span multiple containers. While this simple model is adequate for our use case, it may not fit the needs of concurrent applications in the general case. The second relevant limitation relates to the value data type. In particular, METRADB treats all values as opaque arrays of bytes, which makes it unsuitable for applications that require more complex types such as records or relational objects. Finally, the key-value model for storage is not a natural model when managing byte-addressable memory. In particular, applications that desire the flexibility of treating NVM as CPU addressable memory are not well served by METRADB. In the next chapter, we propose a new solution for applications using persistent memory that aims to address several of these limitations.

CHAPTER 6

LIBPM

In the previous chapters we described our efforts to improve the design and performance of key-value stores for flash storage devices by making designs FTL-aware. Then we proposed a key-value store like approach for consuming NVM that hides the complexities of managing NVM behind a simple interface with support for transactional updates. In this chapter, we develop one final system that aims to address the shortcomings of METRADB as the final piece of this dissertation. We provide some more background on NVM, describe the guiding principles that shaped our solutions, list our core assumptions, and evaluate our proposed solution, LIBPM.

6.1 Introduction

The need for high-performance data persistence is a cornerstone concern for application development. Application developers continuously find new and clever ways to move data between *slow large persistent storage* and *fast volatile memory*. All of this changes when the mode of access to the persistent storage changes as is the case with byte-addressable persistent memory (NVM). First, with NVM the dichotomy of byte-addressability and persistence ceases to exist. Second, read/write latencies of NVM are expected to be comparable to those of DRAM. Their unique properties make them suitable candidates for CPU bus attachment and accessibility through `load` and `store` instructions. Persistence mechanisms of applications fundamentally deserve a revisiting.

In this chapter of the dissertation, we build the abstraction of persistent *containers*, a new way for applications to consume NVM. Containers make NVM consumption significantly simple for applications through the use of familiar interfaces and implementing consistency across updates all while retaining the performance benefits of NVM. Applications create containers and use in-memory data structures; a user-level library automatically discovers container-resident data to be made persistent using pointer chasing techniques based on a single, simple application hint. With this model, most of the conventional persistence operations are obviated. As a result, containers radically simplify the porting of existing applications and creation of new applications that use NVM. Container data are mapped to the process address space and data consistency is enforced though a single, simple transactional operation exposed to the application. The container library preserves relationships between the application's objects (via pointers) within a container upon application restart.

Containers are designed to address several goals which are critical to the adoption of any software solution for NVM. First, it implements an interface that is familiar to existing systems by borrowing the simplest elements from both the memory and file interfaces. Second, it creates a simple yet effective transactional model that provided data consistency when using NVM but with low runtime overhead and without requiring CPU ISA changes. And lastly, the porting of existing applications is made extremely simple. Containers thus fill critical software gaps needed to ensure the adoption of NVM technologies in the application marketplace.

6.2 Architecture

LIBPM is logically and architecturally divided into two components, core and back end. These components address functionality visible to applications and interaction with persistent memory management mechanisms within the OS respectively. All of LIBPM's functionality is available to applications via its API. LIBPM's architecture is depicted in Figure 6.1 and further discussed in the remainder of this section.

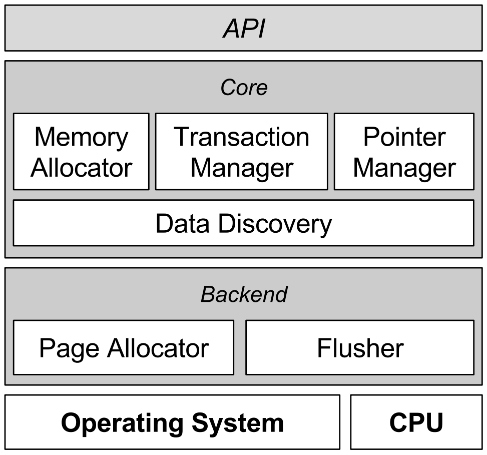

Figure 6.1: LIBPM Architecture.

6.2.1 Application Programming Interface

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

Table 6.1: The LIBPM API.

We believe that a new API for persistent memory will emerge shaped by at least two guiding principles. Foremost, a majority of the application developers will expect the simplicity of the volatile memory API for persistent memory but will expect a user-level library to handle many of the subtleties related to using persistent memory correctly. Thus, the API should ideally provide capabilities to ensure that such access is done safely so as to not corrupt persistent application state after a crash. Second, the library-layer that implements the API must be sufficiently low-overhead so as to preserve the performance advantages of using persistent memory over block-based persistent storage. Guided by these design principles and our experiences from previous work on Software Persistent Memory (SoftPM) [GMC+12], we propose an API that combines a memory like interface with the *container* abstractions.

The proposed API, summarized on Table 6.1, is grouped into four main categories. The first category, Container Management, handles container related operations. It allows programmers to create or restore container with a simple API call, (`open`). Similarly, developer can close an opened container or delete an existing container, thereby reclaiming the NVM used by the container, using (`close`) and (`delete`) API calls respectively. The second category allows developers to create arbitrarily complex data structures within a given container using (`palloc`) and free them from the containers using (`pfree`) with familiar interfaces. It should be noted that in addition to hosting objects, containers also preserve the relations between objects (pointers) by storing additional metadata. This additional information makes it possible for the container to properly restore the persistent objects at random memory locations while maintaining pointer correctness.

The third category illustrates the simplicity of our transactional model. The `commit` operation atomically persists all the changes made to a given container, preserving its consistent state. Alternatively, all the changes could be aborted by issuing a `rollback` call. The last category defines a set of annotation operations on the container. These allow the programmer to set and fetch the container's root using `setroot` and `getroot` respectively. The container root specifies the starting point when computing the memory closure of the container to identify all data that needs to be made persistent.

6.2.2 Core

The LIBPM Core manages the containers that applications create to provide transparent persistent memory allocation and transactional persistence support. Four interconnected components of the LIBPM Core—*Memory Allocator, Transaction Manager, Pointer Manager* and *Discovery*—provide these capabilities. These components, together, implement the LIBPM API.

The *Memory Allocator* is responsible for allocating persistent objects of arbitrary size. It implements a variant of slab allocation [Bon94] whereby objects are grouped together based on their size in order to minimize fragmentation. Besides speeding up allocation, this slab-based allocation technique allows an efficient implementation for freeing persistent memory.

The *Transactions Manager* ensures all-or-nothing persistence semantics for related updates, i.e., all updates to persistent memory that are related from the application's standpoint are made atomically persistent. At a high level, this is accomplished by maintaining two versions of persistent objects: a snapshot version which reflects the previous consistent state and a current version wherein updates are performed in place. Once the container is in a new consistent state after absorbing all necessary modifications into the current version, then the application can either *(i)* make the current version persistent by issuing a commit operation, or *(ii)* revoke all updates to the container by issuing a rollback operation, which would restore the container to its previous, i.e., snapshot, version.

The *Pointer Manager* is responsible to ensuring that the relationship (pointers) between persistent objects get preserved at all times. To do so, it stores metadata that describes the locations and target addresses of persistent pointers so that these pointer values can be corrected when objects are moved in memory, as well as restored correctly when a container is reopened. The latter is particularly important in environments for which LIBPM can not guarantee that the container will be mapped at the same offset within the a process' address space [The].

Finally, the *Discovery* component implements automatic discovery of persistent data based on application-specified annotations for the container's *root* data structure, and automatically annotated information about pointers and memory allocations using static analysis (e.g., CIL [NMRW02]). This component follows the basic approach for memory closure discovery as described in the SoftPM work [GMC+12]. This component simplifies the development of new applications as well as the porting of existing applications to persistent memory.

6.2.3 Backend

LIBPM's *back-end* is in charge of implementing persistence. The back-end is made up of two subcomponents: the Page Allocator and the Flusher. The Page Allocator exports an interface to the upper layers that allows them to allocate and free persistent memory pages. The specific details about how to allocate or free persistent pages are handle by Page Allocator modules. These modules are free to implement the interface whichever way they see fit, allowing them to apply different approaches and optimizations specific to the operating system in use.

The Flusher's primary task is guarantee that writes are durable. Modern systems use multiple levels of caches and memory controllers sitting between the CPU and the memory. Because of these, synchronous updates must be treated with care in order to guarantee that they make it to the persistent media. Based on the available instructions and other details about the hardware, the Flusher customizes the data transfer between the volatile CPU caches and NVM to optimize for performance.

6.3 Design

In this section we describe many of the trade off we made when designing LIBPM. We start by stating assumptions that shaped our design as well as predictions about hardware trends. Then we describe the container abstraction and our transactional model which are at the core of our library. We discuss how persistent memory pointers are managed in LIBPM and contrast it against alternate approaches for managing persistent memory pointers. Finally, we discuss the limitations of LIBPM.

6.3.1 Assumptions

We assume a class of persistent memories with byte addressability and sufficient performance to enable direct CPU access via the memory bus. Further, we assume that traditional CPU architectures for memory access remain fundamentally unchanged. In other words, persistent memory will continue to be managed as pages, which are then managed between the OS and the CPU complex via page tables with hardware accelerated accesses enabled by a hierarchy of volatile CPU caches. The CPU instruction set may evolve to support persistent memory in the future [Cor14, LDK$^+$14, VTS11a]. While these will likely impact the details of the mechanisms we develop, they will not fundamentally change the core research questions that we address in this dissertation. Further, while independently important, concerns such as improving the reliability and durability properties of persistent memory technologies are being addressed by other researchers [ICN$^+$10, SLSB10] and are outside the scope of this dissertation.

6.3.2 The PM Container Abstractions

Our previous work on Software Persistent Memory (SoftPM) [GMC$^+$12] developed a library that allow applications to create data containers for persistently storing arbitrary data. SoftPM *containers* present a powerful abstraction for the developer who simply identifies a single top-level *container root* data structure to persist and all data reachable from this top-level data structure is automatically discovered and atomically persisted by the library infrastructure. The root structure, for instance, may contain pointers to all user-defined data structures in the persistent memory. Listing 6.1 depicts an example of an in-memory Binary Search Tree being migrated to persistent memory using Containers in only few lines of code.

```
struct container_root {
  struct binary_search_tree *node;
} *c_root;
setroot(cid, c_root, sizeof(*c_root));  // cid: container id
c_root->node = tree_root;
commit(cid);
```

Listing 6.1: Migrating RB Tree into persistent memory using containers.

A container can be used to store a portion of an applications in-memory data persistently and guarantee that such data is self-contained. The container provides referential (e.g., pointer-based) integrity and completeness as well as atomic data durability. The primary design goal of SoftPM is to create an easy-to-use interface similar to how volatile memory is used today, transactional consistency for updates to containers, and low-overhead persistence. Application-specified persistence points represent consistent states of application data in persistent memory and allow for straightforward failure recovery by the application.

LIBPM implements SoftPM's container abstraction for a hybrid memory system that is composed of both volatile DRAM and other CPU-addressable persistent memory. In comparison to the prior work, persistence requirements for byte-addressable PM pose requirements that are distinct from those of block-based storage for which SoftPM was built. The ordered durability of data spread across the volatile caches spread across multiple cores of the system needs to be designed in a manner that allows easy developer control over these operations but with minimal developer effort, and optimized for performance. This requires careful use of CPU instructions for enforcing cache line flushes and write barriers, and the use of newer PM-optimized instructions introduced by processor manufacturers (e.g., Intel's recent `clflushopt`, `pcommit`, and `clwb` instructions [Int15b]). Furthermore, the prior work does not support container data sharing across applications, a capability that is important for managing persistent data abstractions in practice.

Segments

A LIBPM container is a collection of segments. Each segments is a set of contiguous pages that get mmaped into the process address space for direct access. There are three types of segments: Metadata, Data, and DLog. Every container has exactly one metadata segment, in which all of the container's self-describing information is stored. The metadata segment also hosts (i) the root pointer, from which all other objects in the container can be found, and (ii) the data log, used to make all updates in the container atomic. The data segments are responsible for hosting all persistent objects. Hence, allocation sizes are limited by the size of data segments; this is a configurable parameter that can be changed at container creation time. As more objects get allocated, more data segments are created, growing the container as needed.

Each data segment hosts objects of a given size, making memory management simple and fast. In order to reduce the number of data segments, memory allocations are rounded up to next multiple of 16 bytes. This is also a configurable property of the container. Finally, there are the DLog segments, which are used to consistent versions of objects stored in data segments for crash recovery purposes. Just like data segments, Dlog segments are allocated on demand. The more changes a transaction makes, the more Dlog segments become necessary and get created. When a transaction is completed successfully, its Dlog segments can be reused or garbage collected.

6.3.3 Transactions

When designing a transactional mechanism for LIBPM we wanted to strike a good balance between performance and ease-of-use. The result is a transaction model that combines memory protection mechanisms with undo logging and operates at a page granularity. This approach eliminates the need for the programmer to be explicit about what data is about to be modified before modifying data, thus vastly simplifying development in comparison to recent efforts in this space [VTS11b, CCA$^+$11b]. LIBPM simply maps segments into the process address space without write permissions and registers a fault handler to track updates. Thus, memory protection allows updates to be logged by LIBPM without developer involvement.

Figure 6.2 illustrates LIBPM handling of updates to containers starting from a recently opened container using segments of size three pages. When a container is opened, all data segments are write protected. First, the application tries to update an object allocated in segment third segment 6.3. Since the page hosting the object is write protected, LIBPM's fault handler gets triggered. Then the entire page gets copy to a non-full Dlog segment 6.4 and a log entry is added 6.5. Finally, the page is made write-able 6.6 and the update proceeds. Subsequent updates to the same page do not trigger additions faults and can proceed without any overhead 6.8,6.9.

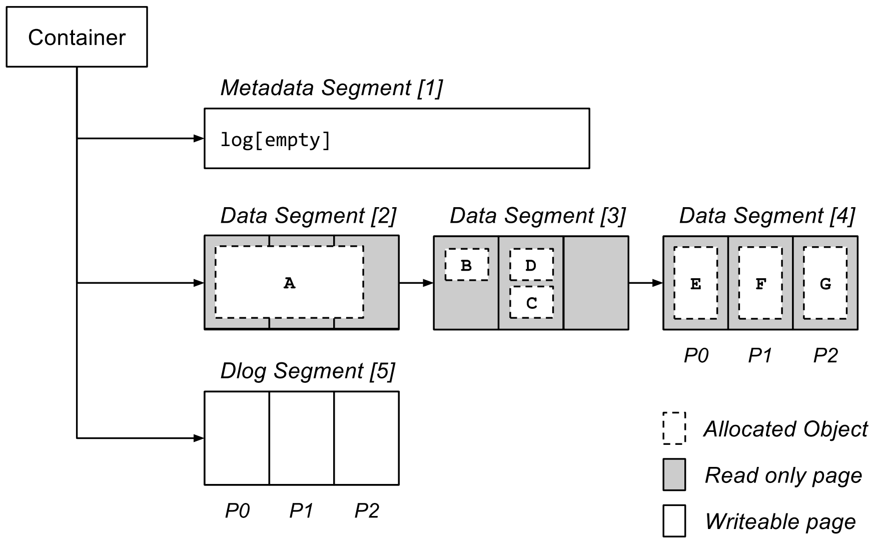

Figure 6.2: LIBPM page logging. State of a container after a successful open.

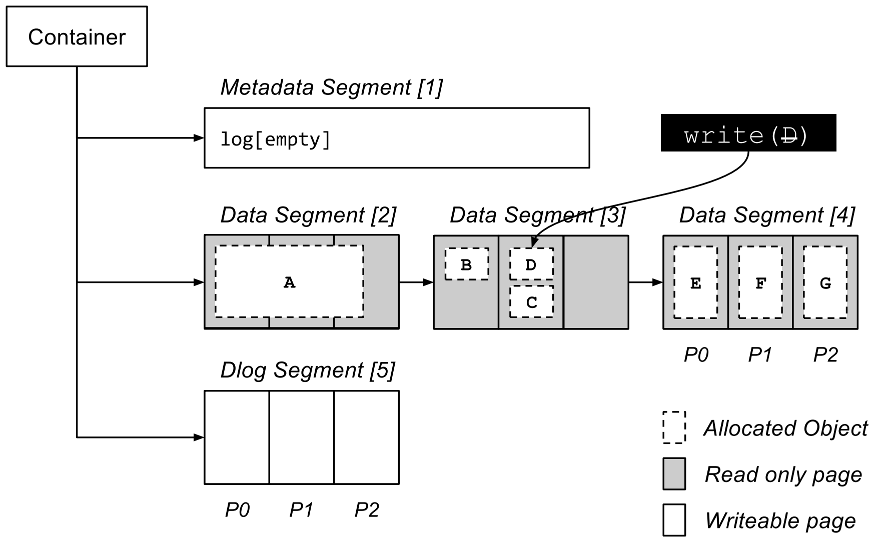

Figure 6.3: LIBPM page logging. The application tries to update object *D* which is mapped read-only.

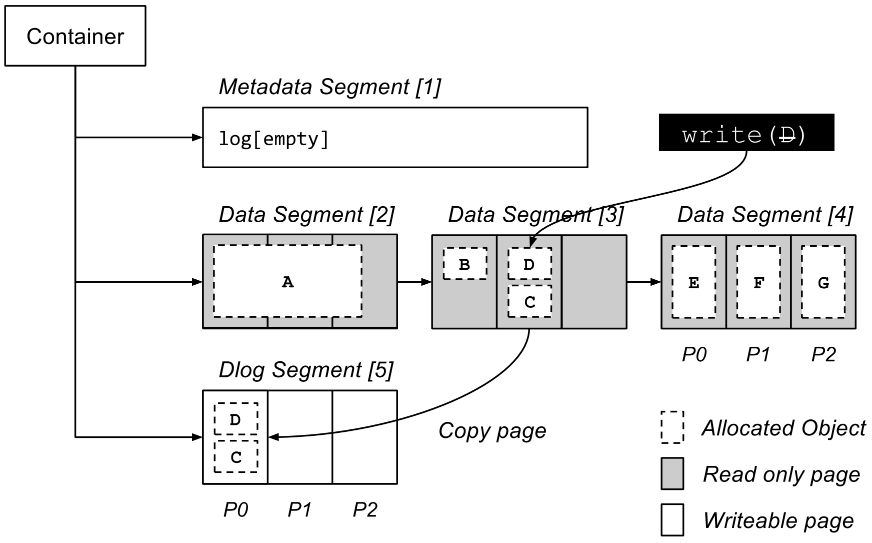

Figure 6.4: LIBPM page logging. The update triggers the fault handler, which initiates the logging mechanism by making a copy of the page containing the faulting address.

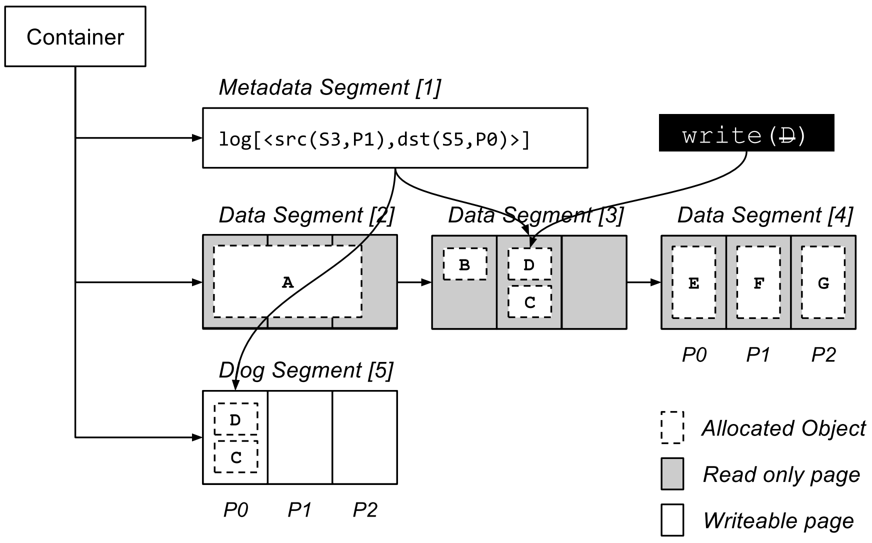

Figure 6.5: LIBPM page logging. Next the update is logged.

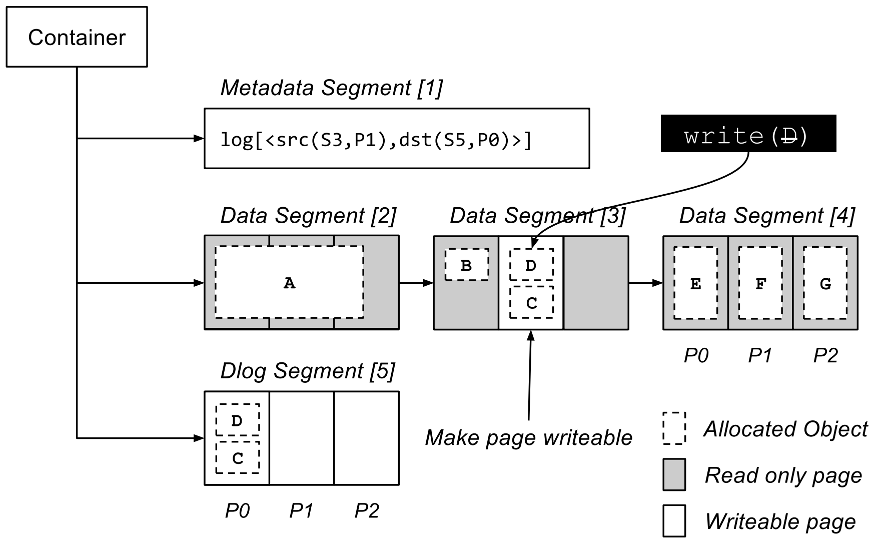

Figure 6.6: LIBPM page logging. Then the page is made writable.

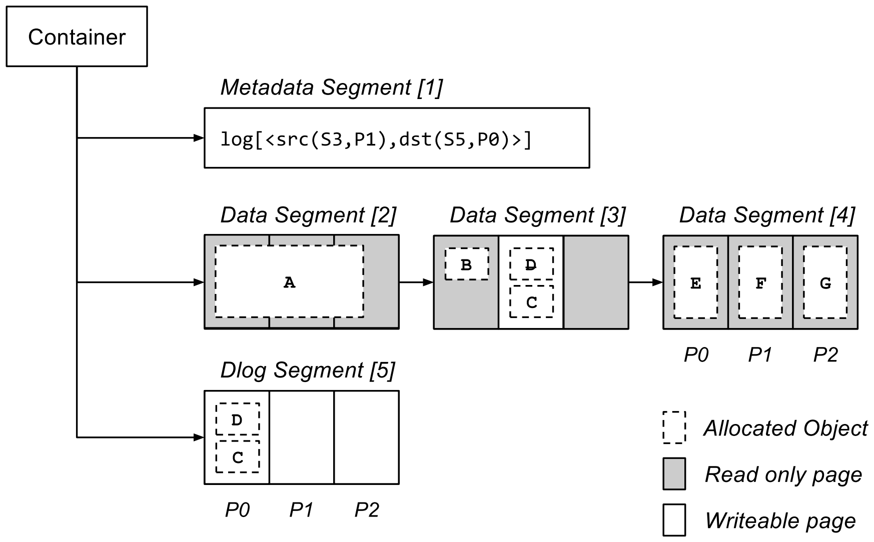

Figure 6.7: LIBPM page logging. Since the page is writeable, the update takes place as intended, all without user intervention.

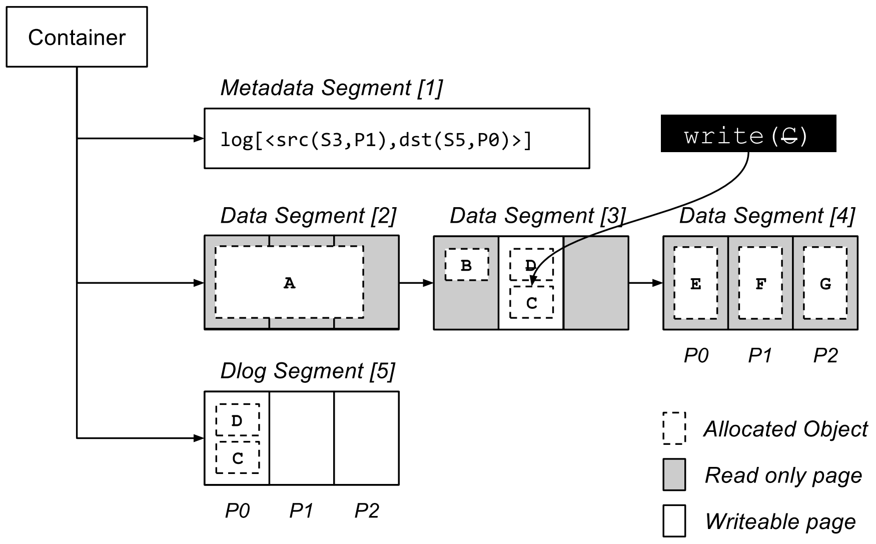

Figure 6.8: LIBPM page logging. Subsequent update to the same page proceed without any overhead.

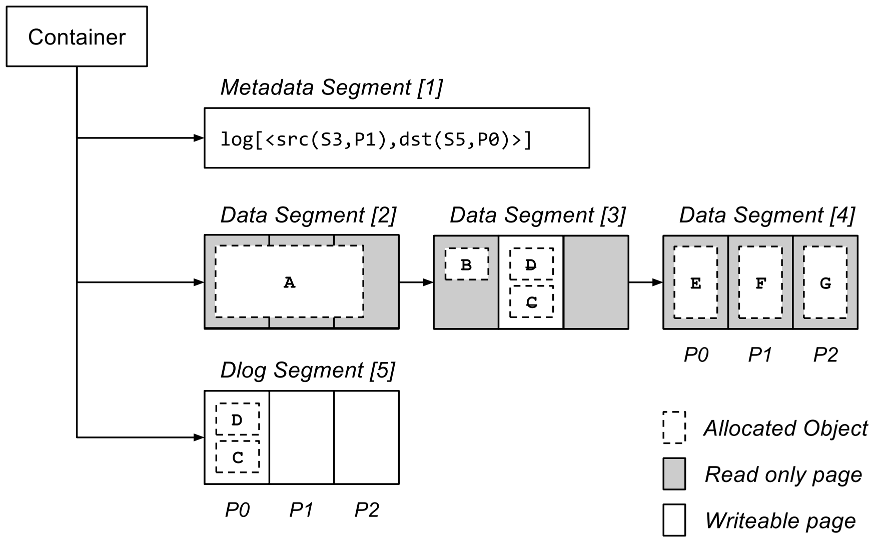

Figure 6.9: LIBPM page logging. State of the container after two updates.

6.3.4 Persistent Memory Pointers

In addition to guaranteeing the consistency and durability of persistent objects, LIBPM's containers also need to preserve the relationships (pointers) among the objects they host.

To better understand what this means, let us consider a container hosting a linked list of m nodes $N_1, N_2, ..., N_m$ such that N_i points to N_{i+1}. When this container is opened, there is no guarantee that the nodes will be `mmap`'ed at the same addresses they were created. Modern systems implement address space randomization (ASLR) by default and therefore it is likely that *next* pointers of the list would be *broken*. To solve this problem, LIBPM stores enough metadata in the container about the objects it hosts so that when a container is opened, all pointers are *fixed* before control is handed to the application. Fixing a persistent pointer simply means updating the target address of the pointer. These updates can be performed in $O(n)$ time in the worst case, where n is the number of persistent pointers in the container. In practice, however, we have seen that this can be avoided by hinting the `mmap` system call about where container segments should be mapped at runtime. Since this is a dynamic operation performed by the LIBPM at runtime, ASLR constraints do not apply for mmap hinting in as much the same way as they would to statically mapped persistent objects.

Previous works have proposed other solutions to the persistent pointer management problem. Most of these can be categorized into three groups:

Fixed-map These solutions simply force all persistent objects to be mapped into the same address space locations, accomplished by disabling ASLR. Disabling ASLR, however, is not recommended since it makes the OS more vulnerable to various types of security attacks.

Pointer swizzling This class of solutions make the programmer responsible for providing serialization methods for every data type that requires persistence. On the downside, these serialization methods must be updated when data types are updated.

Fat pointers With this class of solutions, pointers do not contain the target address. Instead, they contain enough information to compute the target address at runtime. This method has the overhead of runtime pointer dereferencing but it allows persistent objects to be mapped anywhere in the address space.

6.3.5 Limitations

LIBPM's simple design comes with two key limitations.

Single active transaction

LIBPM's transaction model only allows for a single active transaction per container. Whereas this may be enough for many applications, some concurrent applications may require more control on when the updates made by a specific thread of execution are made durable.

Sharing containers across processes

A direct consequence of LIBPM's persistent memory model is that containers cannot be shared across multiple applications. Since the same container could be mapped at different addresses for sharing processes, there is currently no simple way to track and fix pointer locations and pointer values in a manner that is correct for both applications.

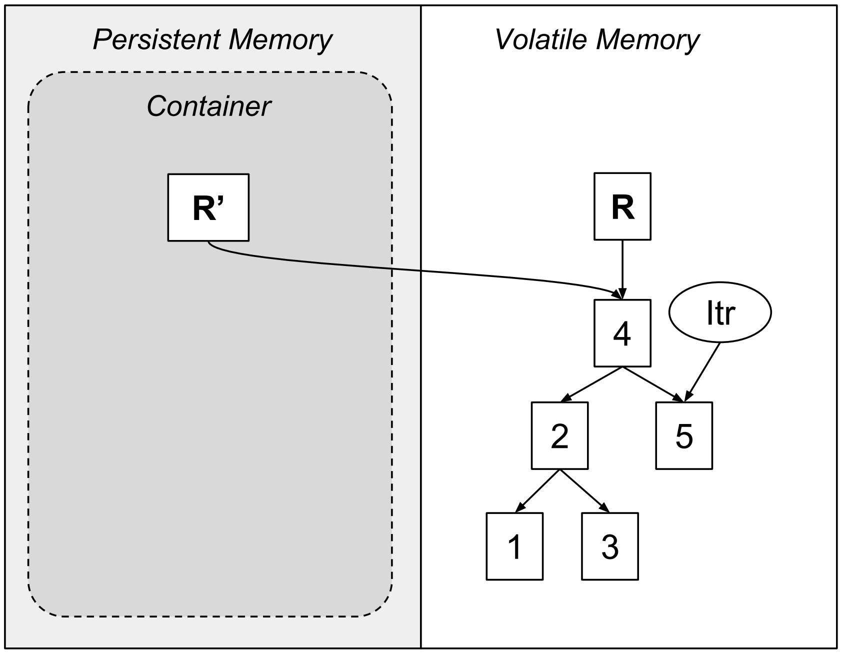

Figure 6.10: Container Closure: Initial state

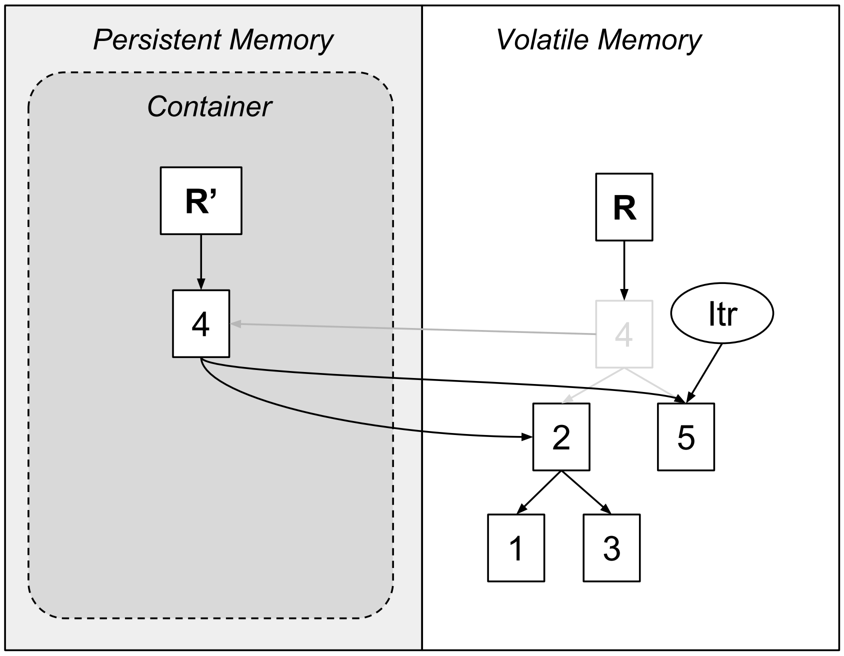

Figure 6.11: Container Closure: Moving all reachable nodes to the Container

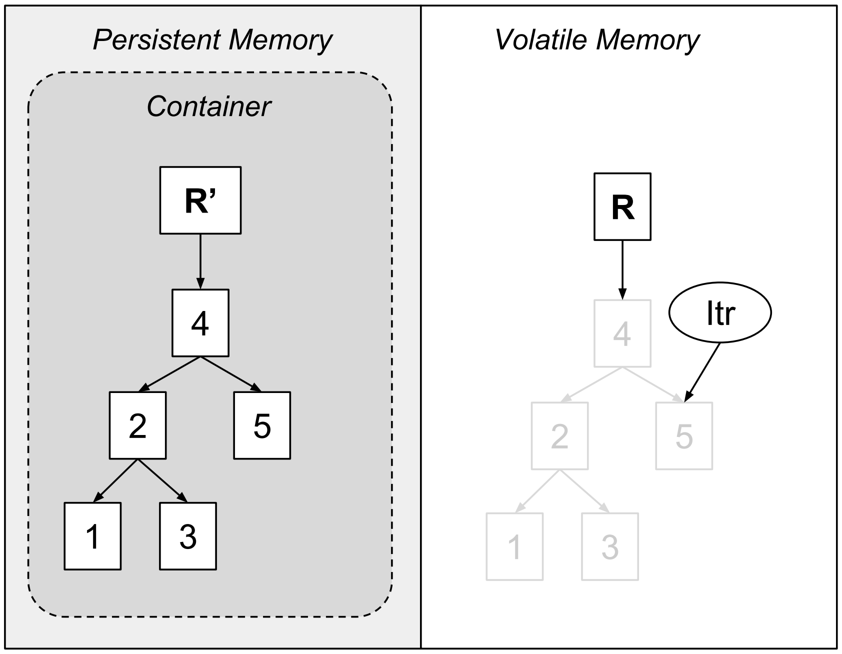

Figure 6.12: Container Closure: All nodes moved

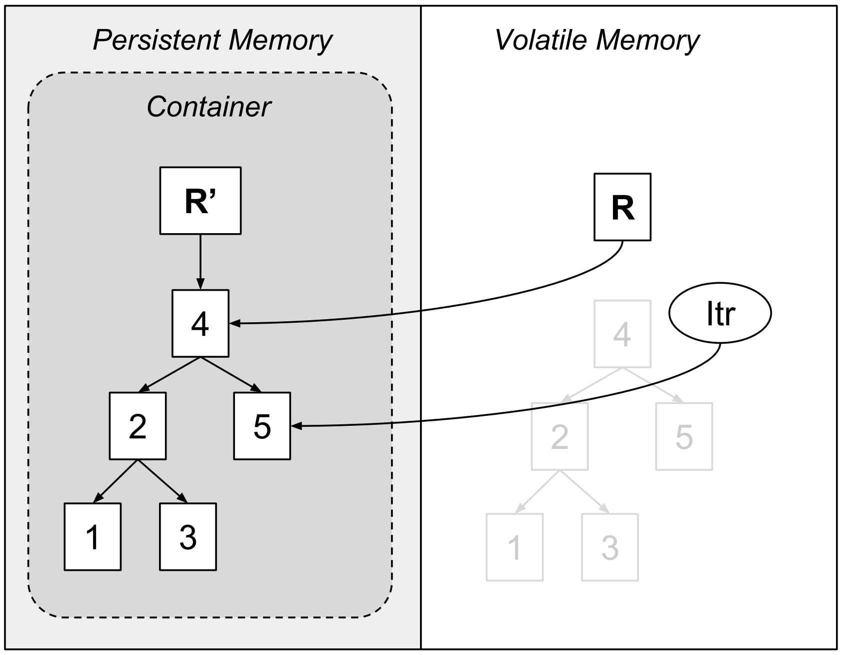

Figure 6.13: Container Closure: Fixing back references

6.4 Porting Legacy Application to NVM

LIBPM provides a mechanism to incrementally port legacy applications to NVM, without having to change the core application logic. To illustrate this process we refer to Figure 6.10, depicting the migration of a Binary Search Tree (BST) from volatile memory to NVM managed by LIBPM. First, the developer creates a new container and allocates within it a root object pointing to the head of the tree. Then, on the next commit operation, LIBPM will add to the container all the objects that are reachable from the root R' hosted in the container. Since node 4 is the first to be reached, it gets moved into the container first, as depicted in Figure 6.11. Following a simple Breadth First Search approach, all nodes reachable from the head of the tree get move into the container eventually, making each one persistent. As nodes get move, their volatile allocations are freed so that the total memory consumption remains the same within a $O(1)$ factor. Figure 6.12 depicts the state in which the entire tree has been made persistent. At this point, the entire tree is in persistent memory and ready to use. However, both the old volatile root and iterator are no longer valid since they point to freed memory. The last step of the container discovery process takes care of fixing these dangling pointers so their target address is updated; now on persistent memory. Figure 6.13 depicts the state of the memory after the `commit` operation is completed. Migrating other data structures can be done in a similar fashion in about the same amount of lines of code. See Listing 6.1 for details.

6.5 Evaluation

The evaluation of LIBPM has three goals: (1) verify the correctness of our implementation, (2) measure the ease-of-use of LIBPM when compared to previous solutions, and (3) evaluate the performance of LIBPM and compare the results with previous solutions.

6.5.1 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$^+$14]) which allows realistic evaluation of software that are intended to run on top of CPU-addressable and byte-addressable NVM devices. The PMEP 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, 8 GB of persistent memory, and a custom version of the Linux kernel v4.1.18.

6.5.2 Correctness

To verify the correctness of LIBPM we used two well known techniques: (1) test coverage analysis and (2) fault injection. Along with the implementation of LIBPM, we also developed a test suite that exercises 100% of the LIBPM code, as reported by Clang/LLVM code coverage tools. This test suite consists of many unit tests as well as several system tests. The unit tests verify the correctness of each individual component whereas the system tests verify inter dependencies among closely related components.

In addition to code coverage, we also performed fault injection at several faulting points contained within critical sections of the transaction and recovery logic. The faults at various faulting points could be triggered both on demand or at random points in time, allowing us to crash the application under examination and verify that the state of the container is always equal to the version found at the most recent consistency point. In every case, LIBPM managed to preserve the consistency of the containers, giving us confidence that our implementation is sound.

6.5.3 Ease of use

Directly programming NVM is challenging. And because of this, previous solutions have all proposed higher level abstractions to hide all the quirks and complexities associated with NVM. However, all of the previous solutions implement a transactional model that makes the user *responsible* for logging updates manually [VTS11b, CCA$^+$11b, Rud]. This model offers adequate control over persistence and performance, but it is also difficult to use correctly which makes it error prone and difficult to debug. For this reason, we steered clear of conventional logging approaches and instead explored an automatic logging technique that requires no user intervention. We quantify the easy of use of our approach by counting the lines of code (LOC) that it takes to implement popular data structures when compared to previous approaches.

Data Structure	LIBPM	NVML
Skiplist	192	264 (+27%)
Hash Table	285	337 (+15%)
RB Tree	387	417 (+7%)
B Tree	428	511 (+16%)

Table 6.2: Counting LOC need to persist popular data structures. Smaller values are desirable.

As we can see from Table 6.2, implementing persistent data structures using NVML requires more LOC than when using LIBPM. For the data structures that we considered, the overhead varied from 7% to 27%. In order to fully understand why this is the case, let us take a closer look at the code. As an example, we examine the `rbtree_rotate` function from the RB Tree implementation using LIBPM and NVML. This function serves as a good example to base our qualitative comparison on because it takes about a dozen lines of non trivial code to create a functional prototype. Furthermore, it illustrates how a non-NVML implementation not just requires additional lines-of-code, this additional code has significant complexity, making the development error-prone.

The formal description of the `rbtree_rotate` operation requirements can be stated as: *Given a node of the tree, and a type of rotation (see Figure 6.14), perform said rotation to adjust the height of the RB Tree in* $O(1)$.

Listing 6.2 presents the code that rotates the persistent RB Tree when using LIBPM. The keen reader will notice that this code would require no modifications if it were used to rotate a volatile RB Tree. This is because LIBPM seamlessly takes care of logging updates transparently and pointer dereferencing integrity without developer involvement. In other words, converting an existing application to use LIBPM as well as writing new applications to use LIBPM is very straightforward given that the manipulation of volatile data structures is a required step for any non-trivial application.

Listing 6.3 shows the same function now implemented using NVML. The first thing to notice is that the signature of the function (lines 2-3) has changed. Developer needs to be aware of persistent data types in a more significant way and use this knowledge continuously during program creation. The use of the macro `TOID` is required to create persistent pointers. Persistent pointers, unlike regular pointers, do not store the value of an absolute offset in the process address space. Instead, they store metadata and a *relative* offset which are used to compute the target of the pointer at runtime. Throughout the rest of the function, other persistent pointers are declared in a similar fashion.

In lines 7 and 8, we see the use of another macro `TX_ADD`, which logs the values of `node` and `child` as part of the ongoing transaction to handle recovery in case the transaction fails to commit. These steps are necessary because both of these variables eventually need to be modified. The user of NVML thus needs to be careful in keeping track of the persistent data that they modify and must make sure that their original values are stored safely in case there is a crash. In line 9 we encounter a new macro `D_RW` used to

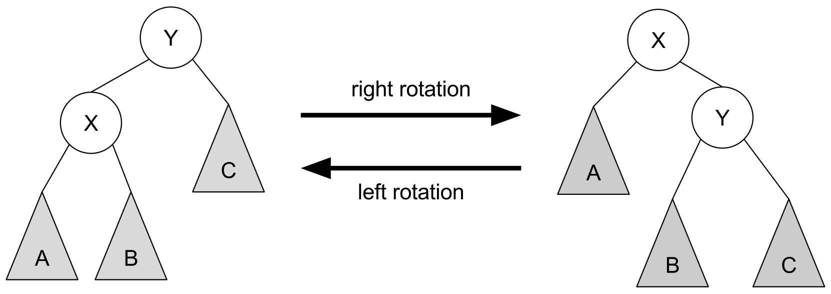

Figure 6.14: Tree rotation

dereference the persistent pointer. At this point, it is safe to make the assignment because the original values have been safely logged. At line 10, we see another dereference used within a comparison operation for two persistent pointers. If the condition is satisfied, then the parent pointer gets updated after its current value is logged using the `TX_SET` macro. For the remainder of the function more assignments and pointer dereferences are made using the same macros.

The complexity of Listing 6.3 is not unique to NVML. It is simply a direct consequence of the persistent pointer and transactional model used by NVML and shared by other previous work [VTS11b, CCA$^+$11b]. While these models do provide higher control and potentially better performance for the advanced user, it also make is very difficult for non experts to use NVM. For these reasons, we believe that our solutions strikes a good balance between performance and programming complexity.

```
static void rbtree_rotate(struct rbtree *tree,
                          struct rbtree_node *node,
                          enum rb_children c)
{
        struct rbtree_node *child = node->slots[!c];
        struct rbtree_node *s = tree->sentinel;
        node->slots[!c] = child->slots[c];
        if (child->slots[c] != s)
                child->slots[c]->parent = node;
        child->parent = node->parent;
        node->parent->slots[NODE_LOCATION(node)] = child;
        child->slots[c] = node;
        node->parent = child;
}
```

Listing 6.2: Implementing RBTree rotate method using LIBPM.

```
static void rbtree_rotate(TOID(struct rbtree) tree,
                          TOID(struct rbtree_node) node,
                          enum rb_children c)
{
        TOID(struct rbtree_node) child = D_RO(node)->slots[!c];
        TOID(struct rbtree_node) s = D_RO(tree)->sentinel;
        TX_ADD(node);
        TX_ADD(child);
        D_RW(node)->slots[!c] = D_RO(child)->slots[c];
        if (!TOID_EQUALS(D_RO(child)->slots[c], s))
                TX_SET(D_RW(child)->slots[c], parent, node);
        child->parent = node->parent;
        TX_SET(node->parent, slots[NODE_LOCATION(node)], child);
        D_RW(child)->slots[c] = node;
        D_RW(node)->parent = child;
}
```

Listing 6.3: Implementing RBTree rotate method using NVML.

6.5.4 Performance

We evaluated the performance of LIBPM and compared it against NVML, a state of the art persist memory library from Intel. NVML provides a set of persistent data structures with transactional support, which we ported to LIBPM as well and compared the latency of each major operation. The data structures we considered were the following:

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

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

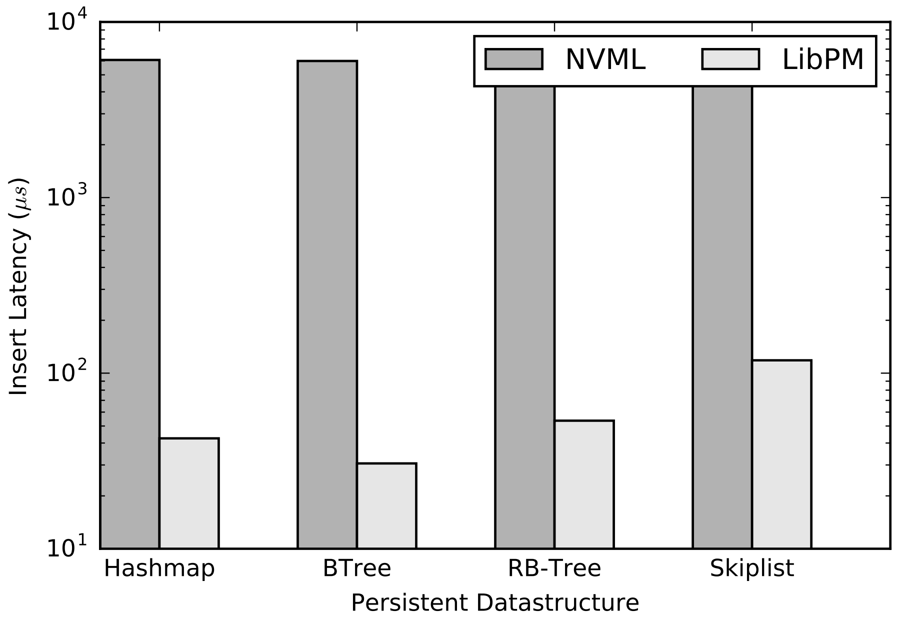

Figure 6.15: NVML vs LIBPM (Inserts)

RBTree A standard Red Back Tree (RBTree).

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

These data structures were modified so that each operation exposed to the users is atomic and durable. Special care was taken so each implementation had the same level of consistency and durability with both libraries: LIBPM and NVML. Each one of these data structures can create a mapping between a 64*bit* integer and a value of an arbitrary size. For these experiments we simply stored a `NULL` value, to better measure the overheads of utilizing the data structures' core logic.

Figure 6.15 depicts the results of comparing the insert operations for all persistent data structures for both LIBPM and NVML. We see that LIBPM outperforms NVML for every data structure, from 51*x* in case of the Skiplist up to 195*x* for the BTree. For lookups, on the other hand, the difference ranges from 1.3*x* in case of the Skiplist to 2.6*x* with the

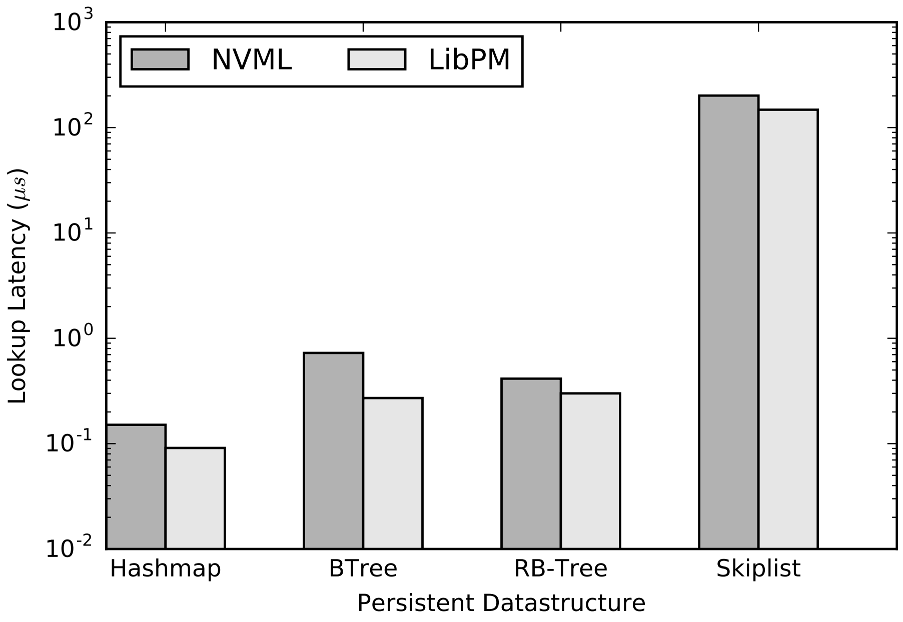

Figure 6.16: NVML vs LibPM (Lookups)

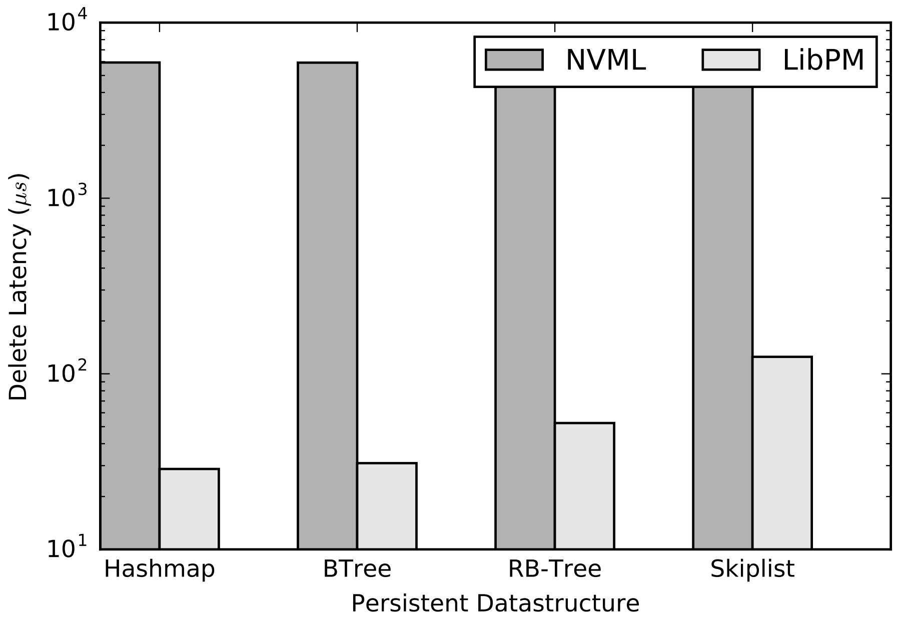

Figure 6.17: NVML vs LibPM (Deletes)

BTree. And finally, the performance difference for delete operations varies from 47*x* to 206*x* for the Skiplist and Hashmap respectively, both in favor of LIBPM.

At a first glance, these results seem counter intuitive. *How is it possible that* LIBPM *outperform* NVML *when* LIBPM *must log much more data for the average transaction?* The answer to this question has three parts: persistent pointer model, shadow data structures, and logging granularity.

```
struct pmemoid {
  uint64_t pool_uuid_lo;
  uint64_t off;
};
```

Listing 6.4: NVML's definition of persistent pointer.

Persistent pointer model

NVML and LIBPM have very different models for persistent pointers. In the case of NVML, a persistent pointer is defined as a type that contains metadata and offset. See details on Listing 6.4. This means that for NVML, a persistent pointer is not simply an offset in the process address space but rather an object with enough information to compute the target address at runtime. Having this layer of indirection allows NVML to `mmap` persistent object anywhere in the address space and more importantly easy sharing of persistent object across multiple processes. For this reason, any workload that involves many pointer dereferencing will take a hit on runtime performance.

LIBPM, on the other hand, has a much simpler approach to persistent pointers. For LIBPM, a persistent pointer is no different from a regular pointer, with the exception that persistent pointers must be fixed before used. The overhead of fixing the pointer is

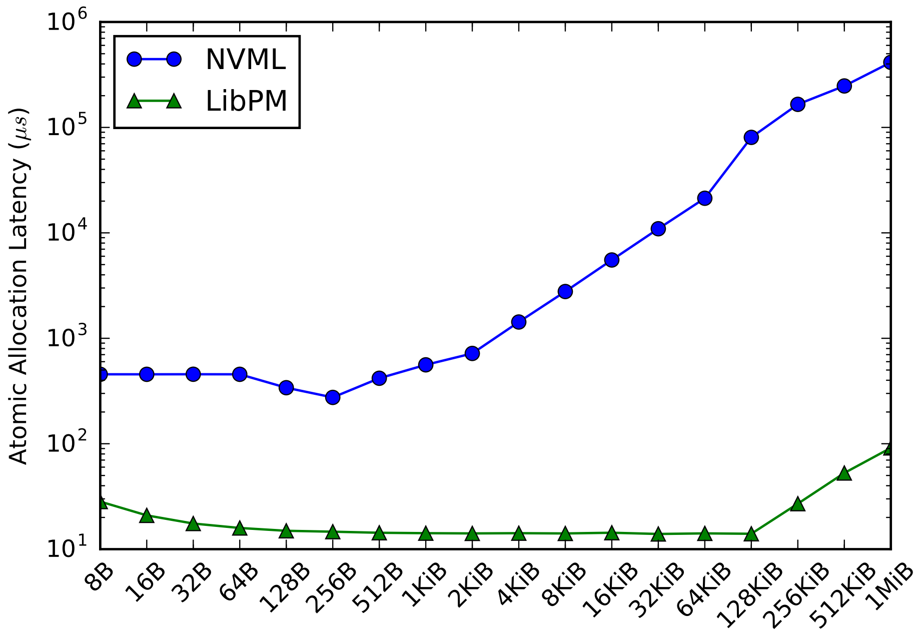

Figure 6.18: Allocation Latency

only paid when the container is opened. After that point, persistent pointer are used no differently than regular pointers.

To measure the overhead of persistent pointer dereferencing, we created a large circular list, several orders of magnitude larger than the CPU cache, and linked the nodes in a way that consecutive nodes are far apart in memory. Then we traverse the list many times while measuring the time that it takes to move from one node to the next. For NVML, latencies were, in average, 2*x* higher than LIBPM.

Shadow data-structures

A memory allocation on persistent memory *must* be performed atomically, so that in case of a crash no memory is leaked. It is important to notice that when persistent memory is leaked, it is leaked persistently. Unlike volatile memory, restarting the application is not sufficient to reclaim a leak on persistent memory. For this reason, both NVML and LIBPM implement atomic memory allocations. NVML makes its memory allocations atomic in the same way it makes memory updates atomic, by logging changes on state before the changes are made. But these log entries must be durable in order to guarantee that the state of the of the memory can be rollback if need be. To make log entries durable, NVML must flush the cache lines containing the entries followed by memory fences, in order to preserve order. Both the flush and the memory fences operation take place during the application critical path, increasing the latency of transactions.

LIBPM implements atomic memory allocation using *shadow bitmaps* and slab allocators. LIBPM packs together persistent object of similar sizes into a slab, and keeps track of slab availability with a persistent bitmap. Additionally, a shadow copy of the bitmap is created on volatile memory that keeps track of all pending allocations and deletions. When a `pmalloc` is called, LIBPM finds an available slab on the shadow bitmap, and returns the correct address for that slab. After that pointer, the use of the LIBPM can freely use the given memory address. The shadow bitmap keeps track of the cache lines containing dirty bits with respect to the persistent bitmap, and flushes this cache lines when a `commit` is issued. In the case of a crash before the commit is completed, nothing need to be done since the persistent bitmap has net been modified. This approach effectively delays cache line flushes and memory fence until the time of commit, reducing the runtime overhead to a minimum.

Logging at a page granularity

Compared to NVML, LIBPM logs and flushes much more data, but in average creates much fewer log entries. This is a direct consequence of logging at a page granularity. However, logging and flushing cache lines is not as expensive as one may think. The new Intel instruction called `CLWB` has similar semantics as `CLFLUSH` and `CLFLUSHOPT` with the significant different that it does not invalidate the cache line it flushes. This has

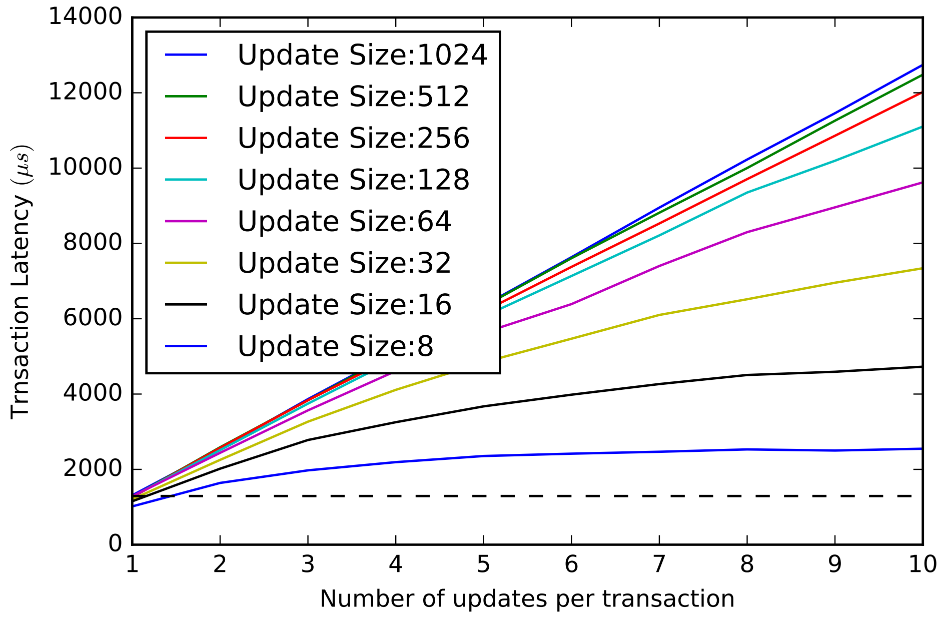

Figure 6.19: Atomic update latency

the advantage that subsequent accesses to the same cache line do not incur a cache miss. Both NVML and LIBPM make use of this optimized instruction but the different lies on the number of memory fences they issue. In order to make log entries durable, in addition to flushing the cache line, once must also issue memory fences to force the flush to take place and preserve the order of operations. The more memory fences are use, the more negative impact it will have on the performance of the application, and since NVML log many more smaller entries, it must also issue many more memory fences. This is the reason for NVML high transaction latency when compared to LIBPM. LIBPM combine several techniques that minimizes the number of memory fences for both updates and memory allocations and deletions.

To better understand this phenomenon, we designed an experiment that measures the latency of transactions while varying the number and size of updates per transaction. Figure 6.19 summarizes the results. As a base line, we used the latency of transactions logging a single entry of one page (4KB) in size. Then we measured the latency of transaction of of multiple updates (1 to 10 updates per transaction) and multiple sizes (from 8 Bytes to 1KB). As we can see, transactions logging a single entry of small updates (e.g. 8 bytes) do have lower latency than those logging an entire page. However, as the number of update per transaction increases, the latency of the transaction also increases, and for as many as two updates, the latencies of transaction with small update sizes is higher.

Temporal locality

Logging at a page granularity has the advantage that updates to near by addresses (within the same page) are very efficient because only the first update trigger the logging logic. Subsequent updates to the same page take place at no extra cost. This is particularly efficient for applications with access pattern characterized by temporal locality. However, for transactional systems based on log-what-change do not benefit from temporal locality.

6.6 Summary

In this chapter we propose a new approach for consuming persistent memory based on the container abstraction. Containers allow applications to program persistent memory very much the same way they program volatile memory. This simplicity is achieved through a combination of key design ideas and a simple-to-use API. First, containers are mapped directly to the process address space, so that applications access persistent memory without any operating system supervision. Second, our proposed transactional model uses memory protection mechanisms to automatically track the pages that are updated at runtime. This approach frees the programmer from the error prone task of explicitly logging every byte before it is updated, as has been used in the recent open-source project, NVML. Third, containers are fully self-describing entities; they maintain enough information about the data it hosts so that they can be migrated to a different machine. This chapter also describes in detail the architecture and design of our solution, LIBPM, and compares its performance against NVML, a state-of-the-art persistent memory library from Intel. The results show an improvement of performance for persistent data structures of up to 195*x* for write intensive workloads and 2.6*x* for read intensive workloads.

CHAPTER 7

RELATED WORK

In this chapter we examine the body of related work for NVMKV, METRADB and LIBPM. In each subsection, we list the most relevant work for each of the systems propose on this thesis and describe the difference between our solutions and previous works, highlighting our unique contributions.

7.1 NVMKVS

Most previous work on FTL-awareness has focused on leveraging FTL capabilities for simpler and more efficient applications, focusing on databases [ONW+11], file systems [JBLF12] and caches [SSZ12]. NVMKV is the first to present the complete design and implementation of an FTL-aware KV store and explains how specific FTL primitives can be leveraged to build a lightweight and performant KV store. NVMKV is also the first to provide support for multiple KV instances (i.e., *pools*) on the same flash device. Further, NVMKV trades-off main memory for flash well as evidenced in the evaluation of a read cache implementation. Finally, NVMKV extends the use of the FTL primitives in a KV store to include conditional-primitives and batching.

There is substantial work on scale-out KV stores and many of the recent ones focus on flash. For example, Dynamo [DHJ+07] and Voldemort [SKG+12] both present scale out KV stores with a focus on predictable performance and availability. Multiple local node KV stores are used underneath the scale out framework and these are expected to provide `get`, `put`, and `delete` operations. NVMKV complements these efforts by providing a lightweight, ACID compliant, and high-performance, single-node KV store.

Most flash-optimized KV stores use a log structure on block-based flash devices [Roc14, AFP+08, BPPP09, BFPS11, DSL10, LFAK11]. FAWN-KV [AFP+08] focused on power-optimized nodes and uses an in-memory map for locating KV pairs at they rotate through the log. FlashStore [DSL10] and SkimpyStash [DSL11] take similar logging-based approaches to provide high-performance updates to flash by maintaining an in-memory map. SILT [LFAK11] provides a highly memory optimized multi-layer KV store, where data transitions between several intermediate stores with increasing compaction as the data ages. Unlike the above mentioned systems, NVMKV eliminates an entire additional layer of mapping along with in-memory metadata management by utilizing the FTL mapping infrastructure.

There are several popular disk optimized KV stores [memb, Mon14, GD11]. Memcachedb [memb] provides a persistent back end to the in-memory KV store, memcached [mema], using BerkeleyDB [OBS99]. BerkeleyDB, built to operate on top a black-box block layer, caches portions of the KV map in DRAM to conserve memory and incurs read amplification on map lookup misses. MongoDB, a cross-platform document-oriented database, and LevelDB, a write-optimized KV store, are HDD based KV stores. Disk-based object stores can also provide KV capabilities [GMC+12, LLOW91, LAC+96, SMK+93]. Disk-based solutions do not work well on flash because the significant AWA that they induce reduces the flash device lifetime by orders of magnitude [MST+14].

Finally, we examine the role of consistency in KV stores in the literature. Anderson *et al.* analyze the consistency provided by different KV stores [ALS+10]. They observe that while many KV stores offer better performance by providing a weaker form of (eventual) consistency, user dissatisfaction when violations do occur is a concern. Thus, while many distributed KV stores provide eventual consistency, others have focused on strong transactional consistency [SSR08]. NVMKV is a unique KV store that leverages the advanced capabilities of modern FTLs to offer strong consistency guarantees and high-performance simultaneously.

7.2 METRADB

We are not the first to propose a key-value store over NVM. Prior work includes Echo [BHC+13], CDDS [VTRC11], wB+Trees [CJ15], and the data structures provided with Intel's NVML [Rud].

These are general solutions with broad scope, while we advocate a custom solution for each application. Similarly, other ideas to facilitate the use of NVM include persistent regions [VTS11b], persistent objects [CCA+11b, GMC+12], file systems [CNF+09, DKK+14], persistent transactions [GDV15, VTS11b], memory management [MAK+13], and durability of lock-based code [CBB14]. Again, these ideas target generality. Other works propose relaxed ordering of writes to NVM [PCW14, LSSM14], non-volatile caches [ZLY+13], and other software-hardware architectures [GDV13]. These idea are orthogonal to and motivate our work, since they facilitate access to NVM by proposing alternative hardware.

7.3 LIBPM

Observing that writes to persistent memory should ideally be unsupervised for best performance, several persistent heap abstractions have been proposed for block stores [BP09, SMK+93, GMC+12, LC97, LLOW91, ver, JLR+94, WKRP02, ZW94]. Recent efforts have applied the persistent heap approach to expose persistent memory to applications [CCA+11a, VTS11a, ZYMS15]. We identify two significant drawbacks that stall their widespread adoption. First is the significant reliance on the developer to annotate memory structures (e.g., variables, allocations, pointers etc.) as persistent; these requirements have been reported to be the source of most programmer bugs when using RVM, a seminal work on creating a persistent memory abstraction [MSSL97]. Second, the proposed abstractions also rely on a static mapping reducing their portability across executables and making heap evolution (such as shrinking and expansion) problematic, besides being too rigid for contemporary systems that demand flexibility in managing address spaces [LNBZ08, The]. We believe that the simplicity of the Container abstraction and its transactional model will help developers overcome these limitations.

CHAPTER 8

CONCLUSIONS

In this thesis we studied two types of storage technologies: NAND flash and persist memory. These new storage media are fundamentally different from their predecessors, the hard disk, for which the current I/O software stack was tailored made. Because of this legacy, it is not easy for application developers to take full advantage of the potential that these technologies have to offer. To address these issues, we proposed new interfaces for both NAND flash and persistent memory, and showcase their merits by building systems that use these new interfaces effectively.

We began by making that case that the traditional `read` and `write` interface of the block layer is quite limiting for modern flash devices. In particular, modern FTLs implements many advanced capabilities that are useful for user space applications as well. Some of these capabilities include log structuring, creating and maintaining dynamic mappings from logical to physical addresses, indexing, transactional updates, and thin provisioning, among others [JBLF12, ONW$^+$11, SSZ12]. Exposing these capabilities to applications simplifies the design and development of applications, as well as improves their performance. Moreover, having access to a richer interface allows applications to make more informed decisions that reduces write amplification at the device and thereby increases the life span of the devices. We built NVMKV as an example of what could be done when better interfaces are available to applications.

Next, we focused our attention on persistent memory. Specifically, we studied how persistent memory could be used to improve the performance of storage systems. We proposed a solution that consumes persistent memory through a library layer that decouples the application logic from the details of the programming persistent memory. For our case study, we developed a key value store with transactional support that consumes persistent memory and is designed specifically for VSAN, a hyper-converged storage solution from VMware. We argued that application tailored solutions could achieved higher performance than more generic solutions and supported our case with an example key value store we built called METRADB.

Lastly, we addressed some of the limitations of consuming persistent memory through a customized key-value store interface by proposing a more general solution based on persistent containers. The container abstraction allows developers to allocates persistent objects using a familiar memory-like interface and automatically guarantees the durability and consistency of the objects it hosts through simple-to-use atomic transactions. Additionally, containers allow unsupervised access to the data they host, making the programming of persistent memory very much like that of traditional volatile memory. We described and evaluated LIBPM, which is our persistent memory library and compared it to NVML, a state of the art library for persistent memory developed by Intel.

BIBLIOGRAPHY

[Aer] Aerospike: High performance KV Store use cases. http://www.aerospike.com/.

[AFK$^+$09] D. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. In *Proc. of ACM SOSP*, 2009.

[AFP$^+$08] David G. Andersen, Jason Franklin, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A fast array of wimpy nodes. Technical Report CMU-PDL-08-108, Carnegie Mellon University Parallel Data Laboratory, May 2008. This version is stale; please use the SOSP 2009 version.

[ALS$^+$10] Eric Anderson, Xiaozhou Li, Mehul A. Shah, Joseph Tucek, and Jay J. Wylie. What consistency does your key-value store actually provide? *HotDep*, 6, october 2010.

[BFPS11] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele. Many-core key-value store. In *Green Computing Conference and Workshops (IGCC), 2011 International*, pages 1 –8, july 2011.

[BHC$^+$13] Katelin A. Bailey, Peter Hornyack, Luis Ceze, Steven D. Gribble, and Henry M. Levy. Exploring storage class memory with key value stores. In *Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, 2013.

[Bon94] Jeff Bonwick. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *USENIX Summer*, pages 87–98, 1994.

[BP09] Anirudh Badam and Vivek S. Pai. Ssdalloc: Hybrid ssd/ram memory management made easy. In *Proc. of NSDR*, 2009.

[BP11] Anirudh Badam and Vivek S. Pai. Ssdalloc: Hybrid ssd/ram memory management made easy. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, 2011.

[BPPP09] Anirudh Badam, KyoungSoo Park, Vivek S. Pai, and Larry L. Peterson. Hashcache: cache storage for the next billion. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, NSDI'09, 2009.

[CBB14] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: leveraging locks for non-volatile memory consistency. In *OOPSLA*, 2014.

[CCA$^+$11a] 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.

[CCA$^+$11b] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *ASPLOS*, 2011.

[CJ15] Shimin Chen and Qin Jin. Persistent B+-trees in non-volatile main memory. *VLDB*, 8(7):786–797, February 2015.

[CLU$^+$15] Daniel Campello, Hector Lopez, Luis Useche, Ricardo Koller, and Raju Rangaswami. Non-blocking writes to files. In *Proceedings of the USENIX Conference on File and Storage Technologies*, February 2015.

[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 *Proceedings of ASPLOS*, 2012.

[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.

[Cor14] Intel Corporation. Intel architecture instruction set extensions programming reference, October 2014.

[CS13] Adrian M. Caulfield and Steven Swanson. Quicksan: A storage area network for fast, distributed, solid state disks. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, 2013.

[CST$^+$10] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, , and R. Sears. Benchmarking cloud serving systems with YCSB. June 2010.

[DAT$^+$14] Dhananjoy Das, Dulcardo Arteaga, Nisha Talagala, Torben Mathiasen, and Jan Lindström. Nvm compression—hybrid flash-aware application level compression. In *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 14)*, Broomfield, CO, October 2014.

[DFI$^+$13] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, 2013.

[DHJ$^+$07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[Dir] Native Flash Support for Applications. http://www.flashmemorysummit.com/.

[DKK$^+$14] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *EUROSYS*, 2014.

[DSL10] Biplob Debnath, Sudipta Sengupta, and Jin Li. Flashstore: high throughput persistent key-value store. *Proc. of VLDB*, 3(1-2):1414–1425, September 2010.

[DSL11] Biplob Debnath, Sudipta Sengupta, and Jin Li. Skimpystash: Ram space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, 2011.

[FNV91] FNV1a Hash. http://www.isthe.com/chongo/tech-/comp/fnv/index.html, 1991.

[Fus] Fusion-io, Inc. ioMemory Virtual Storage Layer (VSL). http://www.fusionio.com/overviews/vsl-technical-overview.

[GD11] Sanjay Ghemawat and Jeff Dean. LevelDB. https://code.google.com/p/leveldb/, 2011.

[GDS12] Laura M. Grupp, John D. Davis, and Steven Swanson. The bleak future of nand flash memory. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, 2012.

[GDV13] Ellis Giles, Kshitij Doshi, and Peter Varman. Bridging the programming gap between persistent and volatile memory using WrAP. In *ACM International Conference on Computing Frontiers*, October 2013. Article 30.

[GDV15] Ellis Giles, Kshitij Doshi, and Peter Varman. SoftWrAP: A lightweight framework for transactional support of storage class memory. In *MSST*, 2015.

[GMC$^+$12] Jorge Guerra, Leonardo Marmol, Daniel Campello, Carlos Crespo, Raju Rangaswami, and Jinpeng Wei. Software Persistent Memory. In *Proc. of USENIX ATC*, 2012.

[ICN$^+$10] Engin Ipek, Jeremy Condit, Edmund B. Nightingale, Doug Burger, and Thomas Moscibroda. Dynamically replicated memory: Building reliable systems from nanoscale resistive memories. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, 2010.

[Int15a] Intel Corporation. 3D XPoint^{TM} Technology Revolutionizes Storage Memory. http://www.intel.com/content/www/us/en/architecture-and-technology/3d-xpoint-technology-animation.html, 2015.

[Int15b] Intel Corporation. Intel architecture instruction set extensions programming reference, August 2015.

[JBLF12] William Josephson, Lars Bongo, Kai Li, and David Flynn. Dfs: A file system for virtualized flash storage. In *Usenix Conference on File and Storage Technologies, February 2010*, February 2012.

[JLR$^+$94] H. V. Jagadish, Daniel F. Lieuwen, Rajeev Rastogi, Abraham Silberschatz, and S. Sudarshan. Dali: A high performance main memory storage manager. In *Proc. of VLDB*, 1994.

[LAC$^+$96] Barbara Liskov, Atul Adya, Miguel Castro, Sanjay Ghemawat, Umesh Gruber, Robert a nd Maheshwari, Andrew C. Myers, Mark Day, and Liuba Shrira. Safe and Efficient Sharing of Persistent Objects in Thor. In *Proc. of ACM SIGMOD*, 1996.

[LC97] David E. Lowell and Peter M. Chen. Free Transactions with Rio Vista. In *Proc. of the ACM symposium on Operating Systems Principles*, pages 92–101, 1997.

[LDK$^+$14] Philip Lantz, Subramanya Dulloor, 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.

[LFAK11] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, October 2011.

[LLOW91] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore Database System. *Commun. ACM*, 34:50–63, October 1991.

[LNBZ08] Vitaliy B. Lvin, Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Archipelago: trading address space for reliability and security. *SIGARCH Comput. Archit. News*, 36(1):115–124, 2008.

[LSSM14] Youyou Lu, Jiwu Shu, Long Sun, and Onur Mutlu. Loose-ordering consistency for persistent memory. In *IEEE International Conference on Computer Design*, 2014.

[MAK$^+$13] Iulian Moraru, David G. Andersen, Michael Kaminsky, Niraj Tolia, Parthasarathy Ranganathan, and Nathan Binkert. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In *TRIOS*, November 2013.

[mema] memcached. http://memcached.org/.

[memb] memcachedb. http://memcachedb.org/.

[Mem14] MemSQL. http://www.memsql.com, 2014.

[Mon14] MongoDB. http://mongodb.org, 2014.

[MSSL97] Henry M. Mashburn, Mahadev Satyanarayanan, David Steere, and Yui W. Lee. RVM: Recoverable Virtual Memory, Release 1.3. *http://www.coda.cs.cmu.edu/doc/html/rvm_manual.html*, September 1997.

[MST$^+$14] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, Raju Rangaswami, Sushma Devendrappa, Bharath Ramsundar, and Sriram Ganesan. Nvmkv: A scalable and lightweight flash aware key-value store. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, Philadelphia, PA, June 2014. USENIX Association.

[NMRW02] G. C. Necula, S. Mcpeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction*, Lecture Notes in Computer Science, 2002.

[NVM15] NVM Library team at Intel Corporation. pmem.io: Persistent memory programming. http://pmem.io/, 2015.

[OBS99] Michael A. Olson, Keith Bostic, and Margo I. Seltzer. Berkeley db. In *USENIX Annual Technical Conference, FREENIX Track*, 1999.

[ONW$^+$11] Xiangyong Ouyang, David Nellans, Robert Wipfel, David Flynn, and Dhabaleswar K. Panda. Beyond block i/o: Rethinking traditional storage primitives. In *High Performance Computer Architecture, Feb 2011*, Feb 2011.

[Ope14] NVM Primitives Library. http://opennvm.github.io/, 2014.

[PCW14] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. In *ISCA*, 2014.

[Roc14] RocksDB. http://rocksdb.org, 2014.

[Rud] Andy Rudoff. NVM library. http://pmem.io.

[SAP14] What is SAP HANA? http://www.saphana.com/community/about-hana, 2014.

[SKG$^+$12] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. Serving large-scale batch computed data with project voldemort. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, FAST'12, 2012.

[SLSB10] Stuart Schechter, Gabriel H. Loh, Karin Straus, and Doug Burger. Use ecp, not ecc, for hard failures in resistive memories. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, 2010.

[SMK$^+$93] Mahadev Satyanarayanan, Henry H. Mashburn, Puneet Kumar, David C. Steer, and James J. Kistler. Lightweight Recoverable Virtual Memory. *Proc. of ACM SOSP*, December 1993.

[SSR08] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. Scalaris: reliable transactional p2p key/value store. In *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*, ERLANG '08, pages 41–48, New York, NY, USA, 2008. ACM.

[SSZ12] Mohit Saxena, Michael Swift, and Yiying Zhang. Flashtier: A lightweight, consistent and durable storage cache. In *European Conference on Computer Systems, April 2012*, April 2012.

[T1013] SBC-4 SPC-5 Atomic writes and reads. http://www.t10.org/cgi-bin/ac.pl?t=d&f=14-043r2.pdf, 2013.

[The] The PaX Team. PaX Address Space Layout Randomization (ASLR). *Available online at: http://pax.grsecurity.net/docs/aslr.txt*.

[ver] Versant. http://www.versant.com/.

[VKA12a] Vijay Vasudevan, Michael Kaminsky, and David G. Andersen. Using vector interfaces to deliver millions of iops from a networked key-value storage server. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 8:1–8:13, New York, NY, USA, 2012. ACM.

[VKA12b] Vijay Vasudevan, Michael Kaminsky, and David G. Andersen. Using vector interfaces to deliver millions of iops from a networked key-value storage server. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 8:1–8:13, New York, NY, USA, 2012. ACM.

[VTRC11] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *FAST*, 2011.

[VTS11a] Haris Volos, Andres Jaan Tack, and Michael Swift. Mnemosyne: Lightweight persistent memory. In *Proc. of ASPLOS*, 2011.

[VTS11b] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *ASPLOS*, 2011.

[WKRP02] An-I Andy Wang, Geoffrey H. Kuenning, Peter Reiher, and Gerald J. Popek. Conquest: Better Performance Through a Disk/Persistent-RAM Hybrid File System. *Proc. of the USENIX Annual Technical Conference*, June 2002.

[YPG$^+$13] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, Swaminathan Sundararaman, and Robert Wood. Hec: Improving endurance of high performance flash-based cache devices. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, 2013.

[YPG$^+$14] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, and Swaminathan Sundararaman. Don't stack your log on my log. In *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 14)*, Broomfield, CO, October 2014. USENIX Association.

[ZLY$^+$13] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. In *MICRO*, 2013.

[ZW94] W. Zwaenepoel and M Wu. eNVy: A Non-Volatile Main Memory Storage System. In *Proceedings of the Sixth Symposium on Architectural Support for Programming Languages and Operating Systems*, 1994.

[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.

VITA

LEONARDO MARMOL
marmol@cs.fiu.edu

August 2011	B.S. Computer Science Florida International University Miami, FL
September 2012 - Present	Graduate Research Assistant Florida International University Miami, FL
September 2015 - April 2016	Intern VMware Palo Alto, CA
June 2015 - September 2015	Intern Intel Labs Santa Clara, CA
May 2014 - August 2015	Intern Intel Labs Santa Clara, CA
May 2013 - August 2013	Intern Fusion-io San Jose, CA
May 2012 - August 2012	Intern Fusion-io San Jose, CA

PUBLICATIONS AND PRESENTATIONS

- Non-Volatile Memory Through Customized Key-Value Stores
  USENIX Workshop on Hot Topics in Storage and File Systems June 2016
  Authors: Leonardo Mármol, Jorge Guerra, Marcos K. Aguilera

- NVMKV: A Scalable, Lightweight, FTL-aware Key-Value Store
  USENIX Annual Technical Conference July 2015
  Authors: Leonardo Mármol, Swaminathan Sundararaman, Nisha Talagala, Raju

Rangaswami

- NVMKV: A Scalable and Lightweight Flash Aware Key-Value Store
  USENIX Workshop on Hot Topics in Storage and File Systems May 28, 2014
  Authors: Leonardo Mármol, Nisha Talagala, Swaminathan Sundararaman, Raju Rangaswami, Bharath Ramsundar, Sriram Ganesan

- Write Policies for Host-side Flash Caches
  USENIX Conference on File and Storage Technologies 2013
  Authors: Ricardo Koller, Leonardo Mármol, Raju Rangaswami, Swaminathan Sundararaman, Nisha Talagala, Ming Zhao

- Software Persistent Memory
  USENIX Annual Technical Conference 2012
  Authors: Jorge Guerra, Leonardo Mármol, Daniel Campello, Carlos Crespo, Jinpeng Wei