UNIVERSITY OF CALIFORNIA SAN DIEGO

Usable Interfaces for Emerging Memory Technologies

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

in

Computer Science

by

Suyash Mahar

Committee in charge:

- Professor Steven Swanson, Chair
- Professor Amy Ousterhout
- Professor Paul Siegel
- Professor Jishen Zhao

2025

The Dissertation of Suyash Mahar is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2025

DEDICATION

To anyone reading this.

EPIGRAPH

! Paragraph ended before `\HyPsd@@ProtectSpacesFi` was complete.
<to be read again>`\par \end{document}`

—pdfTex (Tex Live 2024)

So, I switched to Typst.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ASLR	Address Space Layout Randomization
CXL	Compute Express Link
DAX	Direct Access
DDR	Double Data Rate
DRAM	Dynamic Random Access Memory
FAMS	Failure atomic msync()
KV	Key value
NT	Non Temporal
NVM	Non-volatile Memory
PCIe	Peripheral Component Interconnect Express
PMDK	Persistent Memory Development Kit
PMEM	Persistent Memory
RPC	Remote procedure call
SSD	Solid State Disk
TLB	Translation Lookaside Buffer
WAL	Write Ahead Log
YCSB	Yahoo! Cloud Serving Benchmark

ACKNOWLEDGEMENTS

I have really enjoyed my PhD experience.

This dissertation would not have been possible without the contribution of several people. I am deeply thankful to my advisor, Steven Swanson for all his support, guidance and fun conversations during my PhD.

I would also like to thank Terence Kelly who helped with several technical discussions, feedback on my writing, and advice on how to handle different aspects of my PhD life. Special thanks to Professor Joseph Izraelevitz, who helped me shape my research project when I was just starting out my PhD. I would also like to thank my collaborators from several industry internships that helped me get a better understanding of commercial implications of my research. Thanks to Abhishek Dhanotia and Hao Wang for providing insights into needs and operations of hyperscalars. Thanks to Professor Samira Khan and Professor Baris Kasikci for their guidance during my internship at Google.

I am also deeply indebdted to people who helped me get into research during my undergraduate which led me to where I am today. Professor Saugata Ghose and Dr. Rachata Ausavarungnirun who have helped not only spark my interest in Computer Archicture but have helped me over the years. Thanks to Professor Ran Ginosar and Professor Leonid Yavits who helped me get started in the field of Persistent Memories an a wonderful summer in Haifa.

I am grateful to all the fun times I had at UCSD. Ziheng for being a great friend and roommate. Mingyao for all his restaurant recommendations! Nara for all the support. Zixuan for helping me figure out various processes at UCSD and life. Yanbo for being the perfect travel companion! Seungjin, Ehsan, Zifeng, YJ, and Theo for all the fun memories.

A special thanks to Professor Samira Khan for providing me opportunities during my undergrad to make significant contributions to research projects, helping me with my PhD applications, and providing support and guidance over the years. I would also like to thanks everyone at SHIFTLAB who helped me when I was getting started with my research, especially, Korakit Seemakhupt and Yasas Seneviratne.

Finally, I would like to thank my parents without who provided me support and encouragement during my PhD. My girlfriend, Caleigh who has always been supportive and encouraging.

The introduction contains material from: "Snapshot: Fast, Userspace Crash Consistency for CXL and PM Using msync," by Suyash Mahar, Mingyao Shen, Terence Kelly, and Steven Swanson, which appears in the Proceedings of the 41st IEEE International Conference on Computer Design (ICCD 2023) of which the dissertation author is the primary investigator and first author. "Puddles: Application-Independent Recovery and Location-Independent Data for Persistent Memory," by Suyash Mahar, Mingyao Shen, TJ Smith, Joseph Izraelevitz, and Steven Swanson, which appears in the Proceedings of the 19th European Conference on Computer Systems (EuroSys 2024) of which the dissertation author is the primary investigator and first author. "MemRPC: Fast Shared Memory RPC For Containers and CXL," by Suyash Mahar, Ehsan Hajyjasini, Seungjin Lee, Zifeng Zhang, Mingyao Shen, and Steven Swanson, which is under review of which the dissertation author is the primary investigator and first author.

Chapter Chapter 2 contains material from "Snapshot: Fast, Userspace Crash Consistency for CXL and PM Using msync," by Suyash Mahar, Mingyao Shen, Terence Kelly, and Steven Swanson, which appears in the Proceedings of the 41st IEEE International Conference on Computer Design (ICCD 2023). The dissertation author is the primary investigator and first author of this paper.

Chapter 3 contains material from "Puddles: Application-Independent Recovery and Location-Independent Data for Persistent Memory," by Suyash Mahar, Mingyao Shen, TJ Smith, Joseph Izraelevitz, and Steven Swanson, which appears in the Proceedings of the 19th European Conference on Computer Systems (EuroSys 2024). The dissertation author is the primary investigator and first author of this paper.

Chapter 4 contains material from "MemRPC: Fast Shared Memory RPC For Containers and CXL," by Suyash Mahar, Ehsan Hajyjasini, Seungjin Lee, Zifeng Zhang, Mingyao Shen, and Steven Swanson, which is under review. The dissertation author is the primary investigator and co-first author of this paper.

Thanks! 01 Jan 2025

VITA

2020	Bachelor of Technology, Electronics and Communication Engineering, Indian Institute of Technology Roorkee
2025	Doctor of Philosophy, Computer Science, University of California San Diego

PUBLICATIONS

Suyash Mahar, B. Ray, S. Khan, *PMFuzz: Test Case Generation for Persistent Memory Programs* (Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems).

L. Yavits, L. Orosa, Suyash Mahar, J. D. Ferreira, M. Erez, R. Ginosar, O. Mutlu, WoLFRaM: *Enhancing Wear-Leveling and Fault Tolerance in Resistive Memories Using Programmable Address Decoders* (2020 IEEE 38th International Conference on Computer Design (ICCD), 187-196).

D. Saxena, Suyash Mahar, V. Raychoudhury, J. Cao, Scalable, *High-Speed On-Chip-Based NDN Name Forwarding Using FPGA* (Proceedings of the 20th International Conference on Distributed Computing and Networking).

Suyash Mahar, M. Shen, T. Smith, J. Izraelevitz, S. Swanson, *Puddles: Application-Independent Recovery and Location-Independent Data for Persistent Memory* (Proceedings of the Nineteenth European Conference on Computer Systems, 575-589).

Suyash Mahar, H. Wang, W. Shu, A. Dhanotia, *Workload Behavior Driven Memory Subsystem Design for Hyperscale* (arXiv preprint arXiv:2303.08396).

Suyash Mahar, E. Hajyasini, S. Lee, Z. Zhang, M. Shen, S. Swanson, *Telepathic Datacenters: Fast RPCs Using Shared CXL Memory* (under review).

Suyash Mahar, M. Shen, T. Kelly, S. Swanson, *Snapshot: Fast, Userspace Crash Consistency for CXL and PM Using* `msync()` (2023 IEEE 41st International Conference on Computer Design (ICCD), 495-498).

Suyash Mahar, S. Liu, K. Seemakhupt, Y. Young, S. Khan, *Write Prediction for Persistent Memory Systems* (2021 30th International Conference on Parallel Architectures and Compilation).

Z. Wang, Suyash Mahar, L. Li, J. Park, J. Kim, T. Michailidis, Y. Pan, T. Rosing, *The Hitchhiker's Guide to Programming and Optimizing CXL-Based Heterogeneous Systems* (arXiv preprint arXiv:2411.02814).

FIELD OF STUDY

Major Field: Computer Science

Studies in Computer Science and Engineering (Computer Systems)
Professor Steven Swanson, Chair

ABSTRACT OF DISSERTATION

Usable Interfaces for Emerging Memory Technologies

by

Suyash Mahar

Doctor of Philosophy in Computer Science

University of California San Diego, 2025

Professor Steven Swanson, Chair

Several new memory technologies have emerged in the recent past to address the growing need for memory capacity and bandwidth. These memory technologies include Intel 3D X-Point-based DC-PMM and CXL-based memory expanders, memory semantic SSDs, and shared memory pools. However, these new memory technologies are significantly more expensive to deploy as they require specialized hardware and software support. Using these memory technologies requires new programming methodologies, however, they are often supported using legacy memory or storage interfaces.

To solve these challenges, in thesis, we focus on three different aspects to better utilize these new memory technologies. First, we adapt existing storage interface for newer non-volatile memories using Snapshot. Next, with Puddles, we reimagine the interface for accessing high-performance, byte-addressable non-volatile memories. Finally, we explore how to use upcoming Compute Express Link (CXL)-based shared memory to build high-performance RPC framework.

Crash consistent programming for persistent memory is challenging as frameworks available today require programmers to use transactions and manual annotations. While the failure atomic `msync()` (FAMS) presents a simpler interface for crash-consistency by holding off writes to the backing media until the application calls `msync()`, it suffers from significant performance overheads. To overcome these limitations, we propose Snapshot, a userspace implementation of FAMS. Snapshot uses compiler-based annotations to efficiently track and sync updates to the backing media on a call to `msync()`. Snapshot offers between 1.2× to 8× better performance than PMDK across a range of workloads and evaluation platforms.

While Snapshot is able to significantly improve application performance while using the legacy interface, applications are still limited by their idiosyncrasies. Interfaces available are limited in one or more of their abilities. For example, no persistent memory programming framework enables application to easily map PM data into its address space while being able to share data between process, ship data between machines, and have it be consistent after a crash, and use native, 64-bit pointers. To support these features, we propose Puddles, a new persistent memory abstraction. Puddles provide application independent recovery after a crash, even before the application writing the last time to the data has started again. Puddles support native pointers and are thus compatible with legacy software, all while supporting

sharing and shipping PM data between processes and machines without expensive serialization and deserialization.

Lastly, using RPCool, we show how these emerging memory technologies provide new use cases like high-performance shared memory remote procedure calls (RPCs) using CXL 3.0-based shared memory. Today, datacenters often rely on RPCs for inter-microservice communication. However, RPCs require slow and inefficient serialization and compression to communication over a serial network link like TCP. To address these limitations, we propose RPCool, an RPC framework that uses shared memory to pass pointers to data and avoid serialization. In addition to providing high-performance shared memory RPCs, RPCool also provide isolation similar to traditional networking by preventing invalid pointers and preventing sender from manipulating shared data while the receiver is processing it. Further, to overcome limited range of CXL 3.0, RPCool can automatically fallback to RDMA-based distributed shared memory. Overall, RPCool reduces round-trip latency by 2.2× compared to the state-of-the-art RDMA framework, and 6.3× compared to CXL-based RPC framework.

INTRODUCTION

In the past few years, application working set sizes and datasets growth have far outpaced memory systems. This is commonly known as the memory wall. As a result, we now have a range of technologies that provide large memory capacities with a byte-addressable interface to addresses this challenge: Intel's now-discontinued DC-PMM [48], CXL-based memory expanders [88], memory boxes [85], and Memory Semantic SSDs [29].

First, these new memory technologies are often significantly more expensive to deploy as they require specialized hardware support, and often have increased operational costs. Further, they often require a complete software-rewrite to adapt to their device-specific interfaces.

Second, these memory technologies often make compromises and tradeoffs compared to the traditional DRAM-based memory. For example, all memory technologies listed above have significantly different performance characteristics compared to DRAM, requiring software modifications to fully exploit the potential of these memories.

Finally, unlike the DRAM's interface that we have been using for decades, and are familiar with, these memory technologies requires the programmer to use specialized interfaces to get the most out of them. For example, programming for non-volatile memory requires programmers to use abstractions like transactions for achieving crash consistency. These interfaces often leave a lot to be desired in terms of their ease of use.

Thus, in this thesis, we propose the need for a set of carefully crafted abstractions to help applications achieve the full potential of these new memory technologies. To solve these challenges, I will present my three research works.

First we address the complexity and challenges of achieving crash consistency. Persistent memory programming libraries requires programmers to use complex transactions and manual annotations for applications to be crash consistent. In contrast, the failure-atomic `msync()` (FAMS) [81] interface is much simpler as it transparently tracks updates and guarantees that modified data is atomically durable on a call to the failure-atomic variant of `msync()`. However, FAMS suffers from several drawbacks, like the overhead of `msync()` and the write amplification from page-level dirty data tracking.

To address these drawbacks while preserving the advantages of FAMS, we propose Snapshot in Chapter 2, an efficient userspace implementation of FAMS. Snapshot uses compiler-based annotation to transparently track updates in userspace and syncs them with the backing byte-addressable storage copy on a call to `msync()`. By keeping a copy of application data in DRAM, Snapshot improves access latency. Moreover, with automatic tracking and syncing changes only on a call to `msync()`, Snapshot provides crash-consistency guarantees, unlike the POSIX `msync()` system call.

While snapshot makes crash consistency on non-volatile memory devices easier for the programmer, it still programmers have to rely on traditional memory or block-storage interfaces to access it. In Chapter 3 we reimagines the persistent memory programming interfaces and solve the major challenges with interfaces available today using puddles.

We argue that current work has failed to provide a comprehensive and maintainable in-memory representation for persistent memory. PM data should be easily mappable into a process address space, shareable across processes, shippable between machines, consistent after a crash, and accessible to legacy code with fast, efficient pointers as first-class abstractions. While existing systems have provided niceties

like `mmap()`-based load/store access, they have not been able to support all these necessary properties due to conflicting requirements.

We propose Puddles, a new persistent memory abstraction, to solve these problems. Puddles provide application-independent recovery after a power outage; they make recovery from a system failure a system-level property of the stored data rather than the responsibility of the programs that access it. Puddles use native pointers, so they are compatible with existing code. Finally, Puddles implement support for sharing and shipping of PM data between processes and systems without expensive serialization and deserialization.

Finally, in Chapter 4 we take a new approach to the shared memory technology and build a high-performance RPC framework on top of CXL-based shared memory. Datacenter applications often rely on remote procedure calls (RPCs) for fast, efficient, and secure communication. However, RPCs are slow, inefficient, and hard to use as they require expensive serialization and compression to communicate over a packetized serial network link. Compute Express Link 3.0 (CXL) offers an alternative solution, allowing applications to share data using a cache-coherent, shared-memory interface across clusters of machines.

RPCool is a new framework that exploits CXL's shared memory capabilities. RPCool avoids serialization by passing pointers to data structures in shared memory. While avoiding serialization is useful, directly sharing pointer-rich data eliminates the isolation that copying data over traditional networks provides, leaving the receiver vulnerable to invalid pointers and concurrent updates to shared data by the sender. RPCool restores this safety with careful and efficient management of memory permissions. Another significant challenge with CXL shared memory capabilities is that they are unlikely to scale to an entire datacenter. RPCool addresses this by falling back to RDMA-based communication.

Chapter 5 concludes the thesis.

Chapter 1

Emerging Memory Technologies

To understand the different memory technologies available today to overcome the memory wall, this chapter first provides an overview of byte-addressable devices and then provides the details of the non-volatile memories like Intel's DC-PMM and finally the CXL interconnect and device types enabled using it.

Table 1.1: Byte-addressable devices.

Device	Interface	Technology
Optane PM	Mem. Bus	PM & Internal caches [101]
Mem. Semantic SSDs [29]	CXL 1.0+	Flash + Large DRAM cache
Memory Expander [88]	CXL 1.0+	DRAM
Memory Box [85]	CXL 2.0+	DRAM
NV-DIMMs [96]	Mem. Bus	DRAM
Embedded NVM [47]	Internal Bus	ReRAM

1.1 Byte-addressable Storage Devices

Recent advances in memory technology and device architecture have enabled a variety of storage devices that support byte-addressable persistence. These devices communicate with the host using interfaces like CXL.mem [1], DDR-T [52], or DDR-4 [96] and rely on flash, 3D-XPoint, or DRAM as their backing media, as shown in Table 1.1.

These devices share a few common characteristics: (1) they offer byte-level access to data, (2) they improve on existing DDR-based memory either in storage capacity, bandwidth, or are non-volatile, and (3) they are generally slower than DRAM.

1.2 Non-Volatile Memory (NVM)

Non-volatile memory, like a disk, does not lose data on a power failure. However, unlike disks, non-volatile memories are byte-addressable which allows the applications to access them using the processor's load-store interface just like how they would access traditional DDR-based memory.

Examples of non-volatile memories include Intel's DC Persistent Memory Modules [48] and embedded NVMs [47]

1.2.1 Non-Volatile Memory Programming

While accessing non-volatile memory using the load-store interface resembles DDR-based memories, ensuring data is consistent after an unexpecting power loss requires programmers to use much more complex programming interface.

This is because, when an application issues a store to the Persistent Memory, CPU might buffer the data in its caches, preventing it from reaching the Persistent Memory media. Thus, data written to the Persistent Memory is not guaranteed to reach the persistent media unless explicitly flushed from the caches. To solve this problem, Intel and other CPU vendors have introduced special CPU instructions that flush the data from volatile caches into the non-volatile domain. On x86, the `clwb` instruction writes back a cacheline from the CPU caches, and an `sfence` instruction enforces ordering among `clwb` instructions [40]. Other platforms, e.g., ARM has similar instructions (`DC CVAP` and `DSB`) to ensure the data has reached the persistence domain [43].

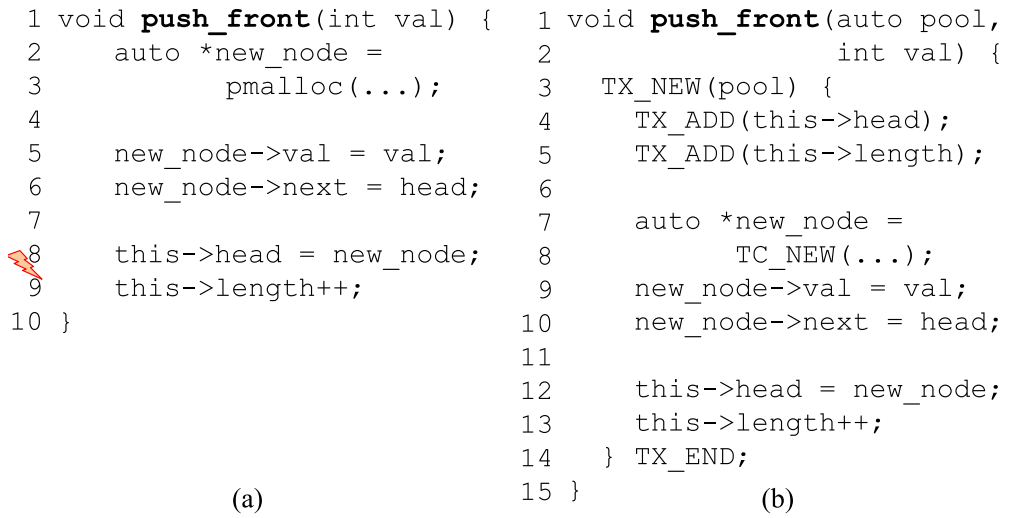


Figure 1.1: Code example showing a push function for a linked list data structure. Function in (a) will have inconsistent memory state after a crash between line 8 and 9 (shown with a lightning bolt), (b) is a crash consistent program using crash-consistent transactions.

Using these instructions requires the programmer to carefully order the instructions to ensure the data on the Persistent Memory is always in a consistent state. Consider an example where the application needs to insert a new node to the head of a persistent linked list. As shown in Figure 1.1, the application would first construct a new node on the PM (line 2), set the node as the head (line 8), and finally increment the node count (line 9). If the system crashes between the line 8 and 9, on restart, the linked list is in an inconsistent state. Since the program did not write the new length to the memory, the length field after restart is off by one. To solve this problem, the application can roll back the allocation on line 2 and the update on line 8. Rolling back these changes makes the node count consistent with the actual number of nodes in the linked list.

To simplify ordering requirements for PM, libraries such as Intel's PMDK provide a transactional syntax, marked by `TX_BEGIN` and `TX_END`. All updates performed in a transactions are atomic with respect

a crash. That is, if the application crashes during a transaction, either all or none of the updates will surivive the crash. Figure 1.1 implements the same linked list, but uses PMDK to first backup all the data modified using `TX_ADD` in the transaction (line 4 and 5) before updating them. In case the system crashes during transaction, the application would undo the changes on restart. This is referred to as undo-logging. Similarly, an application might choose to use redo-logging, where it will log the new values for the logged locations and hold-off the actual updates until the end of the transaction.

1.3 Compute Express Link (CXL)

Compute Express Link (CXL) is a new PCIe-based interconnect that enables novel host-device, device-device and host-host communications at byte granularity. CXL enables several use cases like memory expansion and memory sharing while enabling cache coherent connection among CXL-connected devices and hosts.

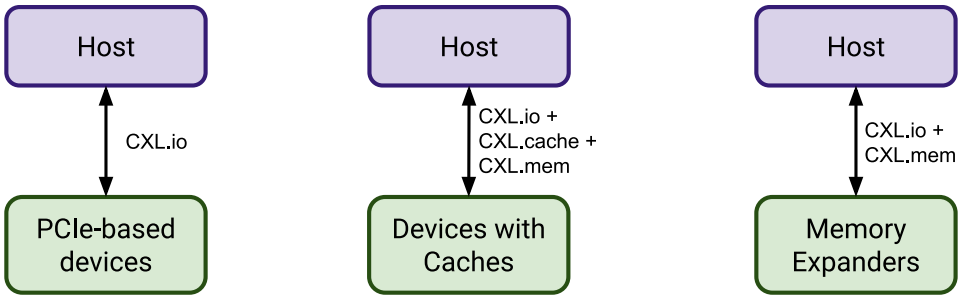
1.3.1 Protocols

CXL is built on-top of the PCIe physical layer and supports three access protocols for different use case: CXL.io, CXL.cache, and CXL.mem.

1. **CXL.io** is a PCIe-compatible protocol for discovery, configuration, management, and PCIe IO transactions. CXL.io accesses do not rely on hardware-based cache coherency.
2. **CXL.cache** provides support for CXL devices to coherently access and cache host memory on device. This enables CXL-connected devices to access cached host memory with low-latency compared to PCIe IO transactions or DMAs.
3. **CXL.mem** provides support for host to coherently access and cache device memory.

1.3.2 Device Types

Using a combination of CXL protocols, CXL defines three device types in its specification (Figure 1.2).



(a) CXL Type 1 (b) CXL Type 2 (c) CXL Type 3

Figure 1.2: CXL device types.

1. **CXL Type 1**: CXL Type 1 devices are typically accelerators which use CXL.io and CXL.cache to cache host memories. Hosts offload workload to the device and the device can coherently access data directly from the host's memory and process it.
2. **CXL Type 2**: CXL Type 2 devices are accelerators with on-device memory which uses CXL.io, CXL.cache, and CXL.mem, allowing the device to cache host memory locally on the accelerator without requiring explicit copies between host and device.
3. **CXL Type 3**: CXL Type 3 device uses CXL.io and CXL.mem to enable hosts to expand memory capacity or bandwidth using a CXL-attached memory expander.

1.3.3 Memory Sharing

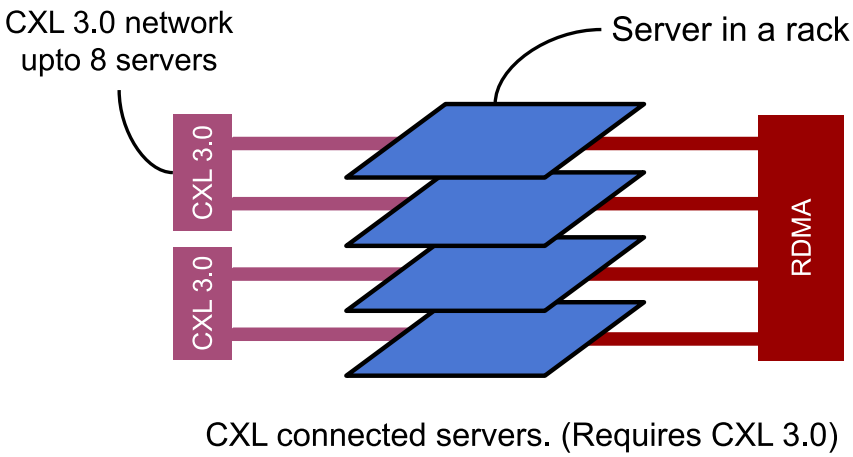


Figure 1.3: CXL shared memory scenarios.

Using CXL 3.0+, multiple hosts can can connect to the same CXL-attached memory expander or pool. This enables multiple hosts to share the same region of memory and access it coherently without explicity software-based synchonization.

Figure 1.3 shows an example of how CXL 3.0-based shared memory could be deployed in a datacenter. A small collection of hosts (a pod) are connected using CXL 3.0 and can access a shared region of memory, while beyond a pod, hosts communicate over traditional networks like RDMA.

Chapter 2

Snapshot: High-performance `msync()`-based crash consistency.

Make sure the chapter introductions are more specifc than the thesis introduction and do not repeat the same stuff.

Recent memory technologies like CXL-based memory semantic SSDs [29], NV-DIMMs [96], Intel Optane DC-PMM, and embedded non-volatile memories [47] have enabled byte- addressable, non-volatile storage devices. However, achieving crash consistency on these memory technologies often requires complex programming interfaces. Programmers must atomically update persistent data using failure-atomic transactions and carefully annotated LOAD and STORE operations, significantly increasing programming complexity [69]. The `msync()` system call offers a simpler interface for durability. The programmer maps a file from the persistent media into the virtual memory and calls `msync()` to make any changes durable.

The `msync()` interface, however, makes no crash-consistency guarantees. The OS is free to evict dirty pages from the page cache before the application calls `msync()`. A common workaround to this problem is to implement a write-ahead-log [6, 30, 53] (WAL), which allows recovery from an inconsistent state after a failure. However, crash consistency with WAL requires an application to call `msync()`/`fsync()` multiple times to ensure application data is always recoverable to a consistent state after a crash.

Park et al. [81] overcome this limitation by enabling failure- atomicity for the `msync()` system call. Their implementation, FAMS (failure atomic `msync()`), holds off updates to the backing media until the application calls `msync()` and then leverages filesystem journaling to apply them atomically. FAMS is implemented within the kernel and relies on the OS to track dirty data in the page cache. OS-based implementation, however, suffers from several limitations:

1. Write-amplification on `msync()`: The OS tracks dirty data at page granularity, requiring a full page writeback even for a single-byte update, wasting memory bandwidth on byte- addressable persistent devices.
2. Dirty page tracking overhead: FAMS relies on the page table to track dirty pages; thus, every `msync()` requires an expensive page table scan to find dirty pages to write to the backing media.
3. Context switch overheads: Implementing crash consistency in the kernel (e.g., FAMS) adds context switch overhead to every `msync()` call, compounding the already high overhead of tracking dirty pages in current implementations.

In this work, we address the shortcomings of FAMS with Snapshot, a drop-in, userspace implementation of failure atomic `msync()`. Snapshot transparently logs updates to memory-mapped files using compiler-generated instrumentation, implementing fast, fine-grained crash consistency. Snapshot tracks all updates in userspace and does not require switching to the kernel to update the backing media.

Snapshot works by logging `STORE`s transparently and makes updates durable on the next call to `msync()`. During runtime, the instrumentation checks whether the store is to a persistent file1and logs the data in an undo log.

Snapshot's ability to automatically track modified data allows applications to be crash-consistent without significant programmer effort. For example, Snapshot's automatic logging enables crash consistency for volatile data structures, like shared-memory allocators, with low-performance overhead.

Snapshot makes the following key contributions:

1. Low overhead dirty data tracking for `msync()`. Snapshot provides fast, userspace-based dirty data tracking and avoids write-amplification of the traditional `msync()`.
2. Accelerating applications on byte-addressable storage devices. Snapshot enables porting of existing `msync()`-based crash consistent applications to persistent, byte-addressable storage devices with little effort (e.g., disabling WAL-based logging) and achieves significant speedup.
3. Implementation space exploration for fast writeback. We study the latency characteristics of NT-stores and `clwb`s and use the results to tune Snapshot's implementation and achieve better performance. These results are general and can help accelerate other crash-consistent applications.
4. For b-tree insert and delete workloads running on Intel Optane DC-PMM, Snap- shot performs as well as PMDK and outperforms it on the read workload by 4.1×. Moreover, Snapshot outperforms non-crash-consistent `msync()` based implementation by 2.8× with 4 KiB page size and 463.8× with 2 MiB page size for inserts.
5. For KV- Store, Snapshot outperforms PMDK by up to 2.2× on Intel Op- tane and up to 10.9× on emulated CXL-based memory semantic SSD. Finally, Snapshot performs as fast as and up to 8.0× faster than Kyoto Cabinet's custom crash-consistency implementation.

2.1 Background and Motivation

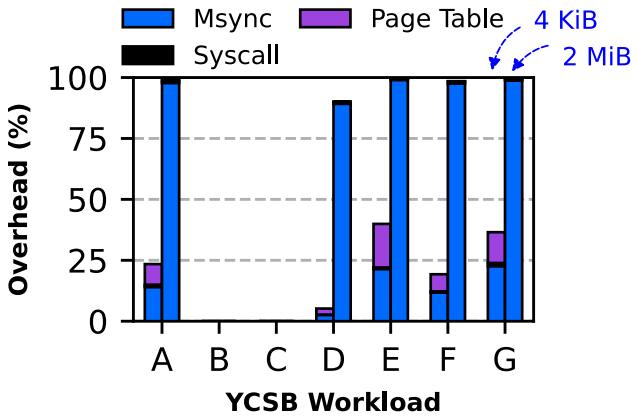


Figure 2.1: `msync`-based KV-Store perf. breakdown (4 KiB and 2 MiB pages).

2.1.1 Filesystem-based Durability and msync

The POSIX `msync()` system call guarantees persistency but provides no atomicity. Part of the dirty data can reach the storage device before the application calls `msync()`. To achieve atomicity, applications often use a write-ahead-log (WAL) [7, 30] to write the modified data to a log and then change the memory location in place. Applications (e.g., Kyoto Cabinet [30]) issue two `msync()` every time they need to atomically update a mapped file, one to persist the write-ahead- log and the second to persist the application updates. Once the data is updated and durable with `msync()`, the application can drop the log. Although applications using `msync()` and WAL to achieve crash consistency directly run off of DRAM (when memory-mapped), they suffer from the overhead of context switches, page table scanning (for finding dirty pages), and TLB shootdowns (to clear access/dirty bit for the page table). This overhead is negligible compared to the access latency of disks and SSDs, but when running on NVM or memory semantic SSDs, the overhead of performing an `msync()` dominates the application's runtime. Figure 2.1

shows the % of runtime spent on the `msync()` call, the context switch overhead for the `msync()` call, and the TLB shootdown overhead across the YCSB workloads for PMDK's KV-Store, modified to use `msync()`. For 2 MiB pages, `msync()`'s overhead is up to 100% of the execution runtime. With DAX-mapped files, although application `LOAD`s and `STORE`s are directly to the storage media (e.g., Optane) and filtered through caches, the `msync()` system call still provides no atomic durability guarantee. On `msync()`on a DAX-mapped file, the kernel simply flushes the cachelines of all dirty pages to the persistent device.

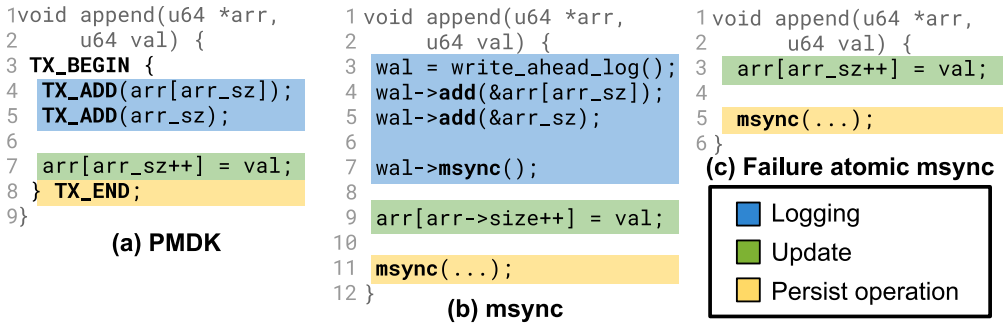


Figure 2.2: Comparison of PM programming techniques using an append() function that appends a new entry at the end of a persistent array. `msync()` calls in (b) and (c) persist the entire write-containing memory range.

Programmers can use a userspace transactional interface (e.g., PMDK) to avoid performance bottlenecks of the `msync()` system call. While this helps with the software overhead, PMDK requires programmers to carefully annotate variables, wrap operations in transactions, and use non-standard pointers. These additional requirements make crash-consistent programming with PMDK hard [44] and error-prone [37, 68–70, 78].

2.1.2 Programming with FAMS

FAMS fixes shortcomings of the POSIX `msync()` and simplifies crash consistency with a failure-atomic `msync()` interface. FAMS lets the programmer call `msync()` to guaran- tee that the updates since

the last call to `msync()` are made atomically durable. Despite FAMS's simpler programming model, its kernel-based implementation suffers from perfor- mance overheads. Figure 2.2 compares PMDK, traditional `msync()`-based WAL [7, 30], and FAMS using an `append()` method for an array. Unlike PMDK or `msync()`, FAMS does not require the programmer to manually log updates either using an undo- log or a write-ahead log. In the FAMS variant (Figure 2.2c), the application maps a file into its address space. Next, the application updates the mapped data using `LOAD`s and `STORE`s (Line 3) and, finally, calls `msync()` when the data is in a consistent state (Line 5). FAMS ensures that the backing file always reflects the most recent successful `msync()` call, which contains a consistent state of application data from which the application may recover. FAMS implements failure-atomicity for a file by disabling writebacks from the page cache. When an application calls `msync()` on a memory-mapped file, FAMS uses the JBD2 layer of `Ext4` to journal both metadata and data for the file. In contrast, PMDK and WAL-based crash consistency require programmers to annotate updated memory locations manually. Lines 4-5, Figure 2.2a for PMDK, and Lines 3-7, Figure 2.2b for WAL-based crash consistency. While PMDK's C++ interface does alleviate some of these limitations, it still requires the use of annotations and fat-pointers, e.g., `persistent_ptr<MyClass> obj`. Despite FAMS's simpler programming interface, applications still suffer from kernel-based durability's performance overhead (e.g., context switch overhead, page table scanning, etc.).

2.2 Overview

Snapshot overcomes FAMS's limitations by providing a drop-in, userspace implementation of failure atomic , resulting in a significant performance improvement for crash- consistent applications. To provide low overhead durability, Snapshot introduces a compiler-based mechanism to track dirty data in

userspace. Snapshot records these updates in an undo that is transparent to the programmer. On `msync()`, Snapshot updates the persistent storage locations recorded in the log. In case of a failure, Snapshot can use the log to roll back any partially durable data. Since Snapshot is implemented in userspace, it avoids the overhead of switching to the kernel and managing TLB coherency. Using Snapshot, the application synchronously modifies data only on the DRAM, speeding up the execution. At the same time, Snapshot maintains a persistent copy on the backing media and automatically propagates all changes to the persistent copy on an . As Snapshot is built on the interface, programmers can port any conventional application written for - based crash consistency with minimal effort to benefit from automatic dirty-data tracking while significantly improving runtime performance. Snapshot's userspace implementation enables legacy disk-based applications to take advantage of faster access times and direct-access (DAX) storage without requiring extensive application rewrites.

2.3 Implementation

Snapshot is implemented as a combination of its compiler pass and a runtime library, `libsnapshot`. Snapshot's com- piler instruments every store instruction that can write to the heap using a call to an undo-log function. The runtime library, `libsnapshot`, provides runtime support for Snapshot: implementing the logging function and Snapshot's `msync()`. Next, we will discuss how the programming interface and logging for Snapshot are implemented, followed by the various optimizations possible in Snapshot to improve its performance.

2.3.1 Logging, Instrumentation, and `msync()`

Snapshot tracks updates to persistent data by instrumenting each `STORE` in the target application with a call to the logging function (instrumentation). During runtime, the instrumentation takes the

STORE's address as its argument, checks if the STORE is to a persistent file, and logs the destination memory location to an undo log.

Logging and Recovery. Snapshot's undo log lives on persistent media to enable recovery from crashes during an msync() call. When an application calls msync(), Snapshot reads the addresses from the undo log entries and copies all modified locations from DRAM to the backing media. The call to msync() only returns when all the modified locations are durable. If the system crashes while copying the persistent data, on restart, Snapshot uses its undo-log to undo any changes that might have partially persisted. While Snapshot maintains per-thread logs, calls to msync() persist data from all threads that have modified data in the memory range. Snapshot maintains a thread-local log to keep track of modified locations and their original values. Snapshot provides limited crash-consistency guarantees for multithreading, similar to PM transactional libraries like PMDK. For example, PMDK prohibits programmers from modifying shared data from two threads in a single transaction. Similarly, in Snapshot, the program should not modify a shared location from two threads between two consecutive msync()s.

Log Format. Logs in Snapshot are per-thread and store only the minimal amount of information needed to undo an interrupted msync(). Logs hold their current state, that is, whether a log holds a valid value. The log also maintains a tail that points to the next free log entry and the size of the log for use during recovery. Each log entry in the log is of variable length. The log entry consists of the address, its size in bytes, and the original value at the address.

While Snapshot tracks all store operations to the memory- mapped region, POSIX calls such as memcpy(), and memmove() are not instrumented as they are part of the OS-provided libraries. To solve this, libsnapshot wraps the calls to memcpy(), memmove(), and memset() to log them directly and then calls

the corresponding OS-provided function. While Snapshot catches some of these functions, applications relying on other functions, e.g., `strtok()`, would need to recompile standard libraries (`glibc`, `muslc`, etc.) with Snapshot to be crash-consistent.

Logging Design Choices. Despite implementing undo- logging, Snapshot only needs two fences per `msync()` to be crash-consistent, as it does not need to wait for the undo-logs to persist before modifying the DRAM copy. This contrasts with PMDK (which also implements undo-logging), where every log operation needs a corresponding fence to ensure the location is logged before modifying it in place. Redo-logging persistent memory libraries eliminate this limitation and only need two fences per transaction. Redo logging, however, requires the programmer to interpose both the loads and stores to redirect them to the log during a transaction, resulting in higher runtime overhead. Snapshot, on the other hand, only interposes store instructions which always write only to the DRAM and avoids any redirection.

2.3.2 Optimizing Snapshot

Snapshot includes a range of optimizations to maximize its performance. In particular, it must address challenges related to the cost of range tracking and reducing the required instrumentation.

2.3.2.1 Low-cost Range Tracking

Since Snapshot's compiler has limited information about the destination of a `STORE`, on every call, the instrumentation checks if the logging request is to a memory-mapped persistent file. Snapshot simplifies this check by reserving virtual address ranges for DRAM (DRAM range) and the backing memory (persistent range) when the application starts. Reserving ranges on application start makes checks for write operations a simple range check. In our implementation, we reserve 1 TiB of virtual address space for both ranges to map all persistent-device-backed files. This range is configurable and is limited only by

the kernel's memory layout. Further, copying a location from DRAM to the backing media now only needs simple arithmetic, i.e., copy from offset in the DRAM range to the same offset in the persistent range.

2.3.2.2 Fewer Instrumentations

Instrumenting stores indiscriminately results in useless calls for stores that cannot write to persistent locations (e.g., stack addresses). Snapshot reduces this overhead by tracking all stack allocations in a function at the LLVM IR level during compilation. Next, Snapshot instruments only those stores that may not alias with any stack-allocated addresses, reducing the amount of unnecessary instrumentation.

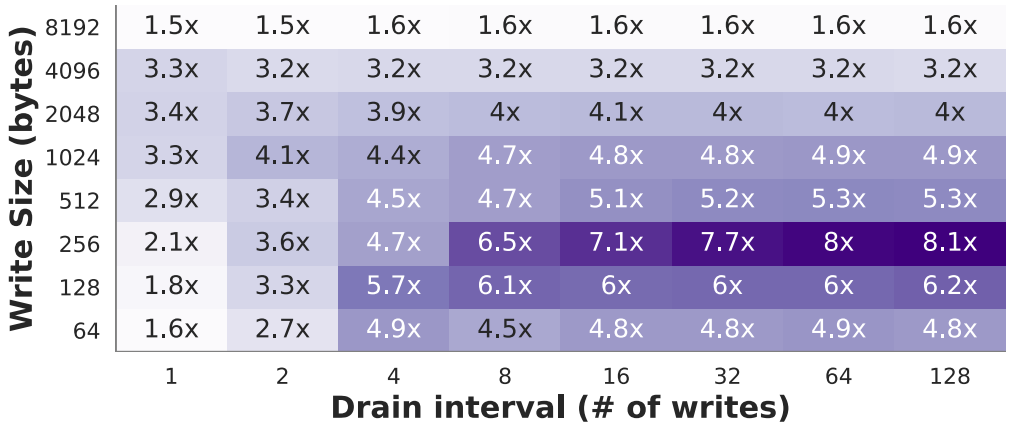


Figure 2.3: Speedup of NT-stores over `clwb` instructions for PM writes. NT-stores always outperform write+`clwb`.

2.3.3 Optimizing Backing Memory Accesses

Flush and fence instructions needed to ensure crash consistency add significant runtime overhead. To understand and reduce this overhead, we study the relative latency of write+`clwb` vs. NT-Store instructions and find that non- temporal stores, particularly those that align with the bus's transfer size, result in considerable performance improvement over the `clwb` instruction.

While we perform the experiments on Intel Optane DC-PMM, we expect the results and methodology to be similar on other storage devices as they have similar memory hierarchies (volatile caches backed by byte-addressable storage devices). Figure 2.3 shows the latency improvement from using NT-Stores to update PM data vs. using writes followed by `clwb`s. The heatmap measures the latency of the operation while varying both the write size, that is, the amount of data written, and the frequency of the fence operation (`sfence`).

From Figure 2.3, we observe that NT-Stores consistently outperform store+`clwb`. Moreover, the performance gap between `clwb`s and NT-Stores increases as the fence interval increases. In contrast, when the write size is increased, the performance only increases until the write size matches the DDR-T transaction size (256 B). Since the CXL protocol uses 64 B packet size for v1-2 and 256 B for CXL v3, we expect to see maxima for those sizes for CXL-based byte-addressable storage devices.

Based on this observation, Snapshot always uses NT-Store instructions to copy modified cache-lines from the DRAM copy to the persistent copy on a call to `msync()`.

Finally, to find modified locations to write to the persistent copy from the DRAM copy, Snapshot iterates over its undo log. As this log is stored on the backing memory, it can be slow to access. This is a result of the log design where each log entry is of variable length, thus, accesses to sequential log entries result in variable strides and poor cache performance. As a result, Snapshot has to waste CPU-cycles traversing the entries. To reduce read traffic to the backing memory and mitigate this additional overhead, Snapshot keeps an additional, in-DRAM list of the updated addresses and their sizes, thus avoiding accessing the backing memory. While it is possible to split the log to separate log entry's data

into a different list, log entries would then require more instructions to flush them, adding overhead to the critical path.

2.3.4 Memory Allocator

While Snapshot provides a failure atomic `msync()`, applications need to allocate and manage memory in a memory-mapped file. Shared memory allocators, like `boost.interprocess` [2], provide an easy way to man- age memory in a memory-mapped file by providing `malloc()` and free()-like API. These operations, while enable memory management, are not crash-consistent. However, Snapshot's ability to automatically log all updates to the persistent memory, enables applications to use volatile shared memory allocators for allocating memory in a crash- consistent manner. To demonstrate Snapshot's utility, we use Snapshot to enable `boost.interprocess` to function as a persistent memory allocator. `boost.interprocess` allocates objects from a memory-mapped file, provides API to access the root object as a pointer, and allocate/free objects while Snapshot tracks updates and makes changes atomically durable on an `msync()`.

2.3.4.1 Decoupling Memory Allocator and Logging

Unlike traditional PM programming libraries that couple memory allocators and logging techniques, Snapshot permits any combination of a logging technique and a memory allocator. For example, PMDK's allocator only supports redo logging. Restricting the programmer to the specific characteristics of their implementation. On the other hand, Snapshot provides programmer the ability to independently choose the memory allocator and logging technique, suiting the specific needs of the workload.

2.3.5 Putting it all Together

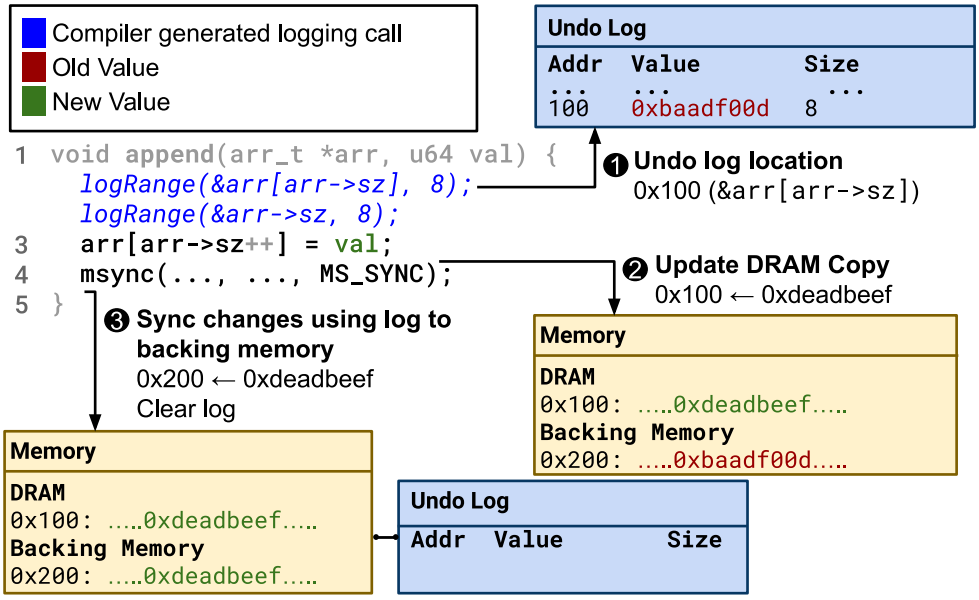


Figure 2.4: Instrumented binary calls `libsnapshot.so`'s logging function for every store. Changes are atomically durable on `msync()`.

To see how snapshot works in practice, consider an array `append()` function that takes a value and inserts it into the next available slot in the array. Figure 2.4 shows the implementation of this function along with the memory and log states as the program executes. In the example, the instrumented program automatically undo-logs the updated location (i.e., call to `logRange`). For brevity, we only show updates to the array element, not the array size.

When the program starts executing ❶, the instrumentation calls the logging function with the address of updated locations (`&arr[arr->sz]` and `&arr->sz`), and update sizes. The function logs the address by creating a new entry in the thread- local undo log.

Next, ❷ the program continues and updates memory locations. Since the program directly interacts only with the DRAM, the value in the DRAM is updated, but the value in the backing memory

(e.g., PM) is unchanged. Finally, the application calls `msync()` to update the backing memory. On this call, Snapshot iterates over the log to find all locations that have been updated and uses them to copy updates from DRAM to PM. After updating PM with the values from DRAM, Snapshot drops the log by marking it as invalid. Once `msync()` returns, any failure would reflect the persistent state of the most recent `msync()`.

2.3.6 Correctness Check

We test our compiler pass and resulting binary to ensure correctness and crash consistency. We test for crash consistency bugs by injecting a crash into the program before it commits a transaction when Snapshot has copied all the changes to the backing store but has not invalidated the log. On a restart, Snapshot should recover and let the application continue its normal execution. This is only possible if the compiler pass correctly annotates all the store instructions and the logging function logs them. We ran these tests for multiple configurations and inputs and found that the compiler pass and the runtime recovered the application each time.

2.4 Results

To understand Snapshot's performance, instrumentation overhead, and the impact of various optimizations, we evaluate several microbenchmarks, persistent memory applications, including Kyoto Cabinet, and crash-consistency solutions. Further, to get an estimate of Snapshot's performance on future hardware, we evaluate Snapshot against PMDK on a CXL-based emulated memory-semantic SSD.

2.4.1 Failure Atomic `msync()` Implementations

Three implementations of failure atomic `msync()` are possible candidates for comparison with Snapshot. The original implementation, FAMS, is by Park et al. [81]. The other implementations,

`famus_snap` [60] and AdvFS [95] use reflinks and file cloning, respectively, to create shallow copies of the backing file on `msync()`.

FAMS by Park et al. [81] is not open-sourced. We use POSIX `msync()` with data journalling enabled to approximate its performance. FAMS works by reconfiguring the `Ext4`'s data journal to not write back to the backing media until the application calls `msync()`. Since FAMS uses data journalling to implement failure atomicity, their implementation performs similarly to `msync()` on `Ext4` mounted with the option `data=journal`, as shown by Park et al. [81].

Implementation of failure atomic `msync()` by Verma et al. on AdvFS [95] is not open-sourced, however, Kelly's `famus_snap` [60] is open-sourced and can be evaluated. `famus_snap` uses reflinks to create a snapshot of the memory-mapped file on a call to `msync()`. `famus_snap`, however, is much slower than the POSIX `msync()` due to slow underlying `ioctl(FICLONE)` calls. In our evaluation, we found that `famus_snap` is between 4.57× to 338.57× slower than `msync()` for the first and 500$^{\text{th}}$ calls, respectively. Since `famus_snap` is orders of magnitude slower than `msync()`, we do not evaluate it further.

Table 2.1: Evaluated configurations.

Config	Description	Dirty data tracking	Crash consistent	Working memory
PMDK	Intel's PMDK-based implementation.	Programmer (byte)	✓	PM
Snapshot-NV	Snapshot, tracking using undo-log.	Auto., (byte)	✓	DRAM
Snapshot	Snapshot, tracking using a volatile list (Section IV-C).	Auto., (byte)	✓	DRAM
msync() 4 KiB	Page cache mapped, 4 KiB pages.	Auto., OS (4KiB)	✗	DRAM
msync() 2 MiB	Page cache mapped, 2 MiB pages.	Auto., OS (2MiB)	✗	DRAM
msync() data journal	Page cache, `ext4(data=journal)`, 4 KiB Pages.	Auto., OS (4 KiB)	✗	DRAM

Table 2.2: System configuration.

Component	Configuration	Details
CPU	2 × Intel 6230	40 HW threads
DRAM	192 GiB	DDR4
Optane	100 series	128×12 = 1.5 TiB, AppDirect Mode
OS & Kernel	Ubuntu 20.04.3	Linux 6.0.0
Build system	LLVM/Clang	13.0.1
Block Device	Intel Optane SSD DC P4800X	For emulation

2.4.2 Configuration

Table 2.1 lists the six different configurations we use to compare the performance of Snapshot. With PMDK, the workloads are implemented using its software transactional memory implementation. The Snapshot-NV and Snapshot implementations are similar, with the difference in how they track dirty data in DRAM. The Snapshot-NV implementation uses the undo log to flush the dirty data on a call to `msync()`. In comparison, the Snapshot implementation uses a separate, volatile list to flush the dirty data (Section IV-C). All implementations of Snapshot otherwise have this optimization enabled. Table 2.2 lists the configuration used for all experiments in the results section.

2.4.3 Evaluating Snapshot on CXL-based Memory Semantic SSDs

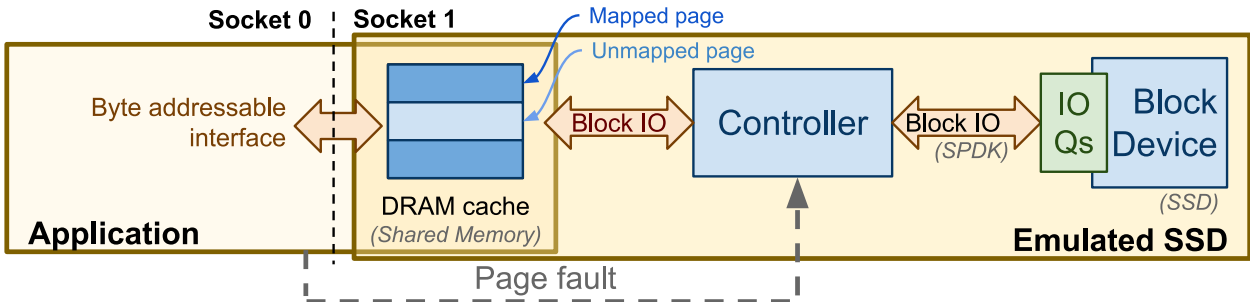


Figure 2.5: Speedup of NT-stores over `clwb` instructions for PM writes. NT-stores always outperform write+`clwb`.

CXL-attached memory semantic SSDs [29] are CXL-based devices with a large DRAM cache backed by a block device. These devices appear as memory devices to the host processor and support byte-addressable accesses.

To understand how Snapshot would perform on CXL-attached memory semantic SSDs, we created a NUMA-based evaluation platform. In our emulation, we implement the DRAM cache using shared memory and service cache miss from a real SSD in userspace using SPDK [105]. The target application and the emulated SSD are pinned to different sockets to emulate a CXL link (similar to Maruf et al. [74]). Figure 2.5 shows the architecture of our emulated memory semantic SSD.

In our implementation, the shared memory used to emulate the DRAM cache has only a limited number of pages (128 MiB) mapped. On access to an unmapped page by the application, the emulated SSD finds a cold page to evict using Intel PEBS [3] and reads and maps the data from the SSD.

Our implementation has a 14.3 µs random access latency at a 91.8% DRAM cache miss rate and a 2.4 µs latency at a 16.3% DRAM cache miss rate. While these latencies might be high compared to Intel

Optane DC-PMM, they represent a slower, byte-addressable media and are close to low-latency flash, e.g., Samsung Z-NAND [20].

2.4.4 Microbenchmarks

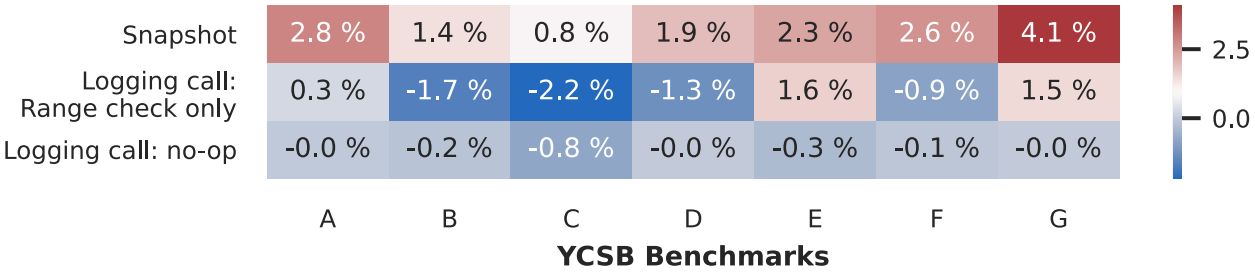


Figure 2.6: Speedup of NT-stores over `clwb` instructions for PM writes. NT-stores always outperform write+`clwb`.

Next, we use microbenchmarks to study how Snapshot's store instrumentation affects performance. To understand the instrumentation overhead of Snapshot, we run it with and without instrumentation and logging enabled. Figure 2.6 shows the performance of different variants of Snapshot relative to the "No Instrumentation" variant running the YCSB workload. The "Logging call: no-op" variant returns from the logging call without performing any checks or logging. The "Logging call: range check only" measures the execution where the logging call only performs the range check but does not log any data. Finally, the "No instrumentation" variant is compiled without Snapshot's compiler pass and thus has no function call overhead. Among these, only Snapshot logs the modifications and is crash-consistent. In all other variants, a call to `msync()` is a no-op.

The results show that even with the compiler's limited information about a store instruction, the overhead from the instrumentation is negligible since stores are relatively few compared to other instructions.

Store Instrumentation Statistics. Across the workloads, Snapshot instrumented 10.8% of store instructions on average. Out of all the store instructions, Snapshot skipped 84.6% because they were stack locations, and 4.54% because they were aliased to stack locations. In case a location is not a stack location, or aliased to one, Snapshot errs on the side of caution and instruments it. During runtime, the instrumentation checks the store's destination to ensure it is to a persistent location.

2.4.5 Persistent Memory Applications and Data-Structures

We evaluate several applications to show that Snapshot consistently outperforms PMDK across various workloads and POSIX `msync()` on write-heavy workloads when running on Intel Optane DC-PMM. Workloads include a linked list and a b-tree implementation from Intel's PMDK, PMDK's KV-Store using the YCSB workload, and Kyoto Cabinet.

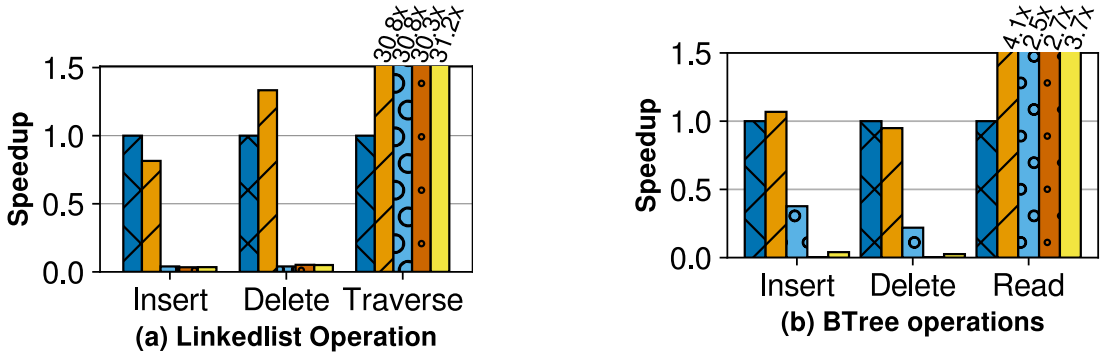


Figure 2.7: Performance comparison of PMDK, Snapshot, and msync() on Intel Optane DC-PMM. Higher is better.

Linked List. Figure 2.7a shows the performance of a linked list data structure implemented using PMDK, traditional `msync()` with 4 KiB and 2 MiB page size, `msync()` with data journal, and Snapshot. Insert inserts a new node to the tail of the list, Delete removes a node from the head, and Traverse visits every node and sums up the values. PMDK and Snapshot are crash-consistent, while the `msync()` implementations are not. Each operation is repeated 1 million times. Since Snapshot runs the application entirely on DRAM and performs userspace synchronization with the backing media on a call to `msync()`, it significantly outperforms PMDK on Traverse workload while being competitive in Insert and Delete. On every call to `msync()`, the traditional filesystem implementation needs to perform an expensive context switch and TLB shootdown, slowing it considerably compared to PMDK and Snapshot.

B-Tree. We compare Snapshot against PMDK using a b-tree data structure of order 8 with 8-byte keys and values on Intel Optane DC-PMM. We use three workloads: (1) Insert workload generates 1 million random 8-byte keys and values. (2) Read workload traverses the tree in depth-first order. Finally, (3) the delete workload deletes all the keys in the insertion order. Figure 2.7b shows that Snapshot performs similarly to PMDK for the insert and delete workloads while outperforming all `msync()` implementations by at least 2.8×. For the read workload, Snapshot and `msync()` achieve significant speedup (4.1×) over PMDK.

Table 2.3: Description of the YCSB workload.

		Description
Workload	A	Read: 50%, Update: 50%
	B	Read: 95%, Update: 5%
	C	Read: 100%
	D	Insert & Read latest, delete old
	E	Read-modify-write
	F	Short range scans
	G	Update: 100%

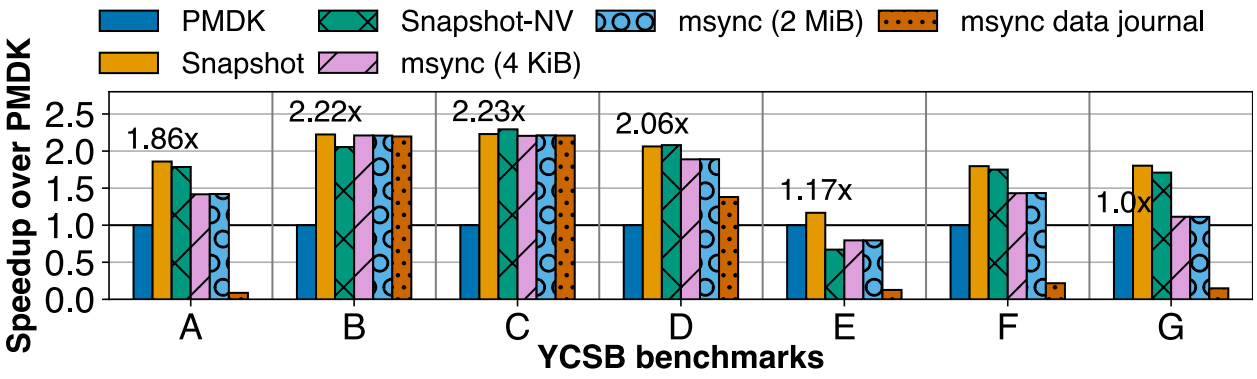


Figure 2.8: KV-store speedup over PMDK with YCSB workload. Higher is better. The non-volatile variant uses the undo log to identify modified locations. Workload description in Table 2.3.

KV-Store. Next, we compare the performance of Snapshot on Optane DC-PMM using a key-value store implemented using a hash table where each bucket is a vector. For evaluation, we use the YCSB workloads A-F and an additional write-only workload, G. Each workload performs 5 million operations on a database with 5 million key-value pairs each.

Figure 2.8 shows the performance of the KV-store against PMDK using different Snapshot configurations described in Table 2.2. For Snapshot, we present the results using volatile and non-volatile lists

for finding modified cachelines on `msync()`. Overall, Snapshot shows between 1.2× and 2.2× performance improvement over PMDK.

Against `msync()`, Snapshot shows a significant performance improvement, especially when compared to `msync()` with data journal enabled. Snapshot does this while providing an automatic crash-consistency guarantee. Moreover, several Snapshot optimizations, including using a separate volatile list for tracking dirty data in the memory (Section IV-C) provide considerable improvement to Snapshot's performance.

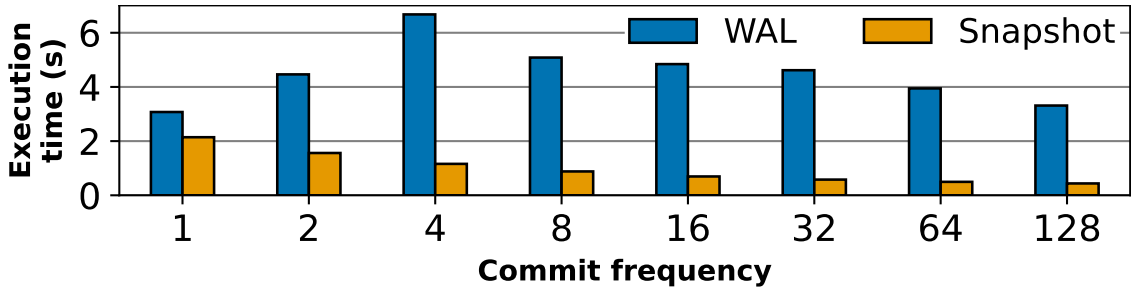


Figure 2.9: Performance comparison of commit frequency for writes in Kyoto Cabinet on Intel Optane DC-PMM. Average of six runs. Lower is better.

Kyoto Cabinet. Figure 2.9 shows the performance comparison between Kyoto Cabinet's built-in WAL+`msync()` based crash- consistency mechanism and Snapshot with a varying number of updates per transaction. Overall, Snapshot outperforms Kyoto's transaction implementation by 1.4× to 8.0×.

For crash consistency, Kyoto Cabinet combines WAL and `msync()`. For the Snapshot version, we disable Kyoto Cabinet's WAL implementation. This version, when compiled with Snapshot's compiler, is automatically crash-consistent.

2.4.6 CXL-Based Memory Semantic SSDs

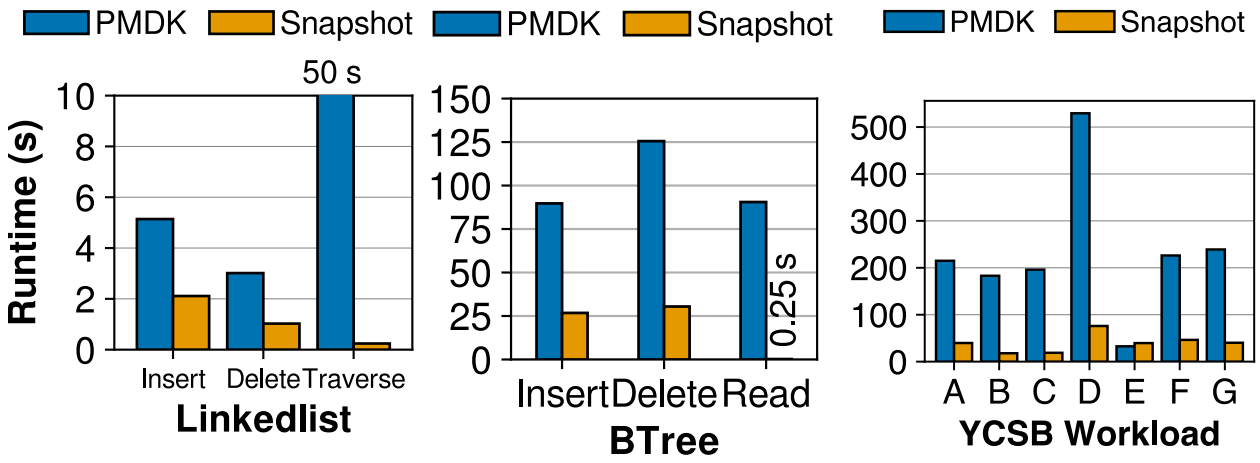


Figure 2.10: Linked list, b-tree and KV-store on Emulated Memory Semantic SSD (2.4.3). Lower is better.

We evaluate Linked list, B-tree, and KV-Store on our emulated memory semantic SSD and observe that Snapshot significantly outperforms PMDK for linked list and b-tree (Figure 10). For linked list, Snapshot outperforms PMDK by 1.7×, 3.2×, and 171.0× for insert, delete, and read, respectively. For b-tree, Snapshot outperforms PMDK by 3.4×, 4.1×, and 364.5× for insert, delete, and read workloads, respectively.

For the KV-Store benchmark, Snapshot outperforms PMDK by up to 10.9× for all but the 'E' workload, where PMDK is 1.23× faster. As our emulation is software-based, it does not support the `msync()` system call, so we did not evaluate Kyoto Cabinet.

2.4.7 Programming effort

Implementing Snapshot did not require changing `boost.interprocess` as any changes to the memory allocator's state are automatically persisted with the `msync()` call by the workloads. For Kyoto

Cabinet, we changed 11 lines of code, including disabling its built-in crash-consistency mechanism, demonstrating the utility of Snapshot's simple programming interface for achieving crash consistency.

2.5 Related Work

Many prior works have proposed techniques to simplify the persistent memory programming model. Romulus [27] uses a twin-copy design, storing both copies on PM for fast persistence. Romulus uses a redo log to synchronize the active copy with the backing copy on a transaction commit. Pisces [39] is a similar PM transaction implementation that uses dual version concurrency control (DVCC), where one of the versions is the application data on persistent memory, and the other is the redo log. Pisces improves performance by using a three-stage commit protocol where the stages where the data is durable, visible, and propagated are decoupled. Pronto [76] introduces the idea of asynchronous semantic logging (ASL). Calls to the data structures' public functions in Pronto are recorded along with their arguments in a persistent semantic log. The semantic log is asynchronously applied to the PM copy using a background thread. Like Snapshot, in Pronto, the application runs on DRAM and gains performance improvement by faster access latency of DRAM.

Libnvmmio [21] provides an `msync()`-based interface, but it does so by intercepting filesystem IO calls, e.g., read() and write(). Thus, unlike Snapshot, Libnvmmio does not support the memory-mapped file interface. Similarly, DudeTM [65], while uses a memory-mapped interface and stages working copy in the DRAM, requires the programmer to use a PMDK-like transactional interface, increasing programming effort.

Automated solutions like a compiler pass to simplify the programming effort include Atlas [15], which adds crash- consistency to existing lock-based programs by using the outermost lock/unlock oper-

ations. Synchronization Free Regions (SFR) [35] extend this idea and provide failure-atomicity between every synchronization primitive and system call.

Similarly, some works use language support to automatically add persistence to applications written for volatile memory. Breeze [75] uses compiler instrumentation for logging updates to PM but requires the programming to explicitly wrap code regions in transactions and annotate PM objects. NVTraverse [31] and Mirror [32] convert lock-free data structures persistent using programmer annotation and providing special compare and swap operations.

Memory allocators are important in achieving crash consistency. To resume after a crash, persistent memory allocators need to save their metadata state along with the allocated data. Several works in the past have proposed PM allocators. Romulus [27] supports porting any sequential memory allocator designed for volatile memory allocation to PM by wrapping all the persistent types in a special class that interposes store accesses. This is similar to Snapshot's compiler-based instrumentation, but in contrast to Romulus, Snapshot requires no programmer effort to use a volatile memory allocator for PM.

LRMalloc [13] is a PM allocator that persists only information that is needed to reconstruct the allocator state after a crash. Metall Allocator [50] provides a coarse crash-consistency mechanism by using the underlying DAX filesystem's copy-on-write mechanism. However, Metall only guarantees persistency when the Metall allocator's destructor is called, making it impractical for applications that need higher frequency persistency (e.g., databases). Kelly et al. [59] present a persistent memory allocator (PMA) to provide a persistent heap for conventional volatile programs and create a persistent variant of gawk, pm-gawk [61].

2.6 Summary

Snapshot provides a userspace implementation of failure atomic `msync()` (FAMS) that overcomes its performance limitation. Snapshot's sub-page granularity dirty data tracking based crash consistency outperforms both per-page tracking of `msync()` and manual annotation-based transactions of PMDK across several workloads.

Snapshot alleviates limitations of FAMS, enabling applications to take advantage of faster, byte-addressable storage devices. Moreover, Snapshot, unlike FAMS, completely avoids any system calls for crash consistency or manual annotation and transactional semantics required by PMDK.

2.7 Acknowledgement

This chapter contains material from "Snapshot: Fast, Userspace Crash Consistency for CXL and PM Using `msync`," by Suyash Mahar, Mingyao Shen, Terence Kelly, and Steven Swanson, which appeared in the IEEE 41st International Conference on Computer Design (ICCD 2023). The dissertation author is the primary investigator and the first author of this paper.

Chapter 3

Puddles: Application-Independent Recovery and Location-Independent Data for Persistent Memory

Existing PM programming systems are built on a patchwork of modifications to the memory-mapped file interface and thus make several compromises in how persistent data is accessed. These systems use custom pointer formats, handle logging through ad-hoc mechanisms, and implement recovery using diverse but incompatible logging and transactional semantics.

In this paper, we show that the design of existing PM libraries results in PM programming models that severely limit programming flexibility and introduce additional unnatural constraints and performance problems.

For example, opening multiple copies of a pool that resides at a fixed address would result in address conflicts. For another, using non-native (*i.e.*, "smart" or "fat") pointers avoids the need for fixed addresses but adds performance overhead to common-case accesses, makes persistent data unreadable by non–PM-aware code, leaves software tools (*e.g.*, debuggers) unable to interpret that data, and locks-in the PM data to a particular PM library. Further, current implementations of fat-pointers do not allow

multiple copies of PM data to be mapped simultaneously unless the PM library first translates all pointers in one of the copies. Finally, enforcing crash consistency in the application requires that after a crash, 1) the application is still available, 2) the application still has write permissions for the data (even if the application only wants to read it), and 3) the system knows which application was running at the time of the crash–none of which are true in general.

Today's PM programming libraries thus leave the critical issue of data integrity in the hands of the programmer and system administrators rather than robustly ensuring those properties at the system level. Further, existing PM programming libraries restrict basic operations like opening cloned copies of PM data simultaneously, reading PM data without write access, or using legacy pointer-based tools to access PM data. A storage system with these characteristics represents a step back in safety and data integrity compared to the state-of-the-art persistent storage systems – namely filesystems.

To solve these problems, we propose a new persistent memory programming library, *Puddles*. Puddles solve these problems while preserving the speed and flexibility that the existing PM programming interface provides. Puddles provide the following properties:

1. *Application-independent crash-recovery*: PM recovery after a crash in Puddles completes before *any* application accesses the data. Recovery succeeds even if the application writing data at the time of the crash is absent after restart or no longer has the write permissions.
2. *Native pointers for PM data*: Puddles use native pointers and, thus, allow code written with other PM libraries or non-PM-aware code (*e.g.*, compilers and debuggers) to read and reason about it. Pointers are a fundamental and universal tool for in-memory data structure construction.

3. *Relocatability*: Puddles can transparently relocate data to avoid any address conflicts and thereby enable sharing and relocation of PM data between machines.

Puddles is the first PM programming system that provides application-independent recovery on a crash and supports both native pointers and relocatability while providing a traditional transactional interface. Designing Puddles, however, is challenging as native pointers, relocatability, and mappable PM data are properties that are at odds with each other. For example, native-pointers have traditionally prevented relocatable PM data, and non-mappable data like JSON does not support pointers.

To resolve these conflicts, the Puddles system divides PM pools into *puddles*. Each puddle is a small, modular region of persistent memory (several MiBs) that the Puddles library can map into an application's address space. Puddles provide non-PM-aware applications access to PM data by allowing programs to use native pointers. To support sharing puddles between processes and shipping puddles between machines, puddles are relocatable—they can be mapped to arbitrary virtual addresses to resolve address conflicts. To support relocation, puddles are structured so that all pointers are easy to find and translate while, dividing pools into puddles allows translation to occur incrementally and on demand. The Puddles library works in tandem with a privileged system service that allocates, manages, and protects the puddles.

To ensure that puddles are always consistent after a crash, puddle programs register log regions with the system service and store the logs in those regions in a format the service can safely apply after a crash. After a crash, the system applies logs before *any* application can access the PM data. Puddles' flexible log format can accommodate a wide range of logging styles (undo, redo, and hybrid). While applications

can access individual puddles, Puddles supports composing them into seamless collections that resemble traditional PM pools, allowing applications to allocate data structures that span multiple puddles.

We compare Puddles against PMDK and other PM programming libraries using several workloads. Puddles implementation is always as fast as and up to 1.34× faster than PMDK across the YCSB workloads. Puddles' use of native virtual pointers allows them to significantly outperform PMDK in pointer-chasing benchmarks. Against Romulus, a state-of-the-art persistent memory programming library that uses DRAM+PMEM, Puddles, a PMEM-only programming library is between 36% slower to being equally fast across the YCSB workloads.

3.1 Limitations of Current PM Systems

Table 3.1: Puddles vs. recent PM programming libraries.

System	Transactional Support	Native Pointers	Application Independent Recovery	Object Relocatability	Region Relocatability	Cross-pool Transaction
PMDK [49]	✓	✗	✗	✗	✓	✗
TwizzlerOS [12]	✓	✗	✗	✓	✓	✗
Mnemosyne [97]	✓	✓	✗	✗	✗	✓
NV-Heaps [25]	✓	✗	✗	✗	✓	✗
Corundum [44]	✓	✗	✗	✗	✓	✗
Atlas [15]	✓	✓	✗	✗	✓	✗
Clobber-NVM [104]	✓	✓	✗	✗	✓	✗
Puddles	✓	✓	✓	✓	✓	✓

Current persistent memory programs suffer from a host of problems that limit their usability, reliability, and flexibility in ways that would be unthinkable for more mature data storage systems (e.g., file systems, object stores, or databases). In particular, they rely on the program running at the time of the crash for recovery, use proprietary pointers that lock data into a single application or library, and place limits on the combination of pools (i.e., files) an application can have open at one time.

A novel file system with similar properties would garner little notice as a serious storage mechanism, and we should hold PM systems to a similar standard.

To understand the limitations and fragmented feature space of PM libraries, Table 3.1 compares several PM programming libraries across multiple axes. Puddles is the only PM programming library that supports features like application-independent recovery, object relocatability (moving individual objects in a process's address space), region relocatability (moving groups of objects), and the ability to modify any global PM data in a transaction, features that users expect from a mature storage system.

The rest of the section examines problems that are endemic to existing PM programming solutions. First, we look closely at these problems that plague current PM programming solutions and then understand how they hold back PM applications.

3.1.1 PM Crash Recovery is Brittle and Unreliable

When an application crashes, current PM programming libraries require the user to restart the application that was running at crash time to make the data consistent. This design decision breaks the common understanding of data recovery.

For example, if a PDF editor crashes while editing a PDF file stored in a conventional file system, the user can reopen the file with a different PDF editor and continue their work. With current PM

programming libraries, this is not possible. The user must re-run the same program again, or the data may be inconsistent.

This problem may seem benign, but this crash-consistency model relies on several assumptions that do not hold in general—like the availability of the original writer application and need for write access after a crash. The net result is an ad hoc approach to ensuring data consistency that is far removed from what state-of-the-art file systems provide.

Indeed, ensuring recovery may not be possible at all in some circumstances.

For example, the user might lose write access to the data if their credentials have expired, preventing them from opening the file to perform recovery. Alternatively, the original application may no longer be available either because the licenses have expired, OS and PM library updates have changed the transactional semantics, or if the file is restored from a backup on another system or the physical storage media is moved to a new system. If any of these assumptions fail, recovery will be impossible, and the data will be left in an inconsistent state.

PMDK, the most widely used PM library, illustrates how a lack of permissions can prevent recovery. In PMDK, recovery is triggered only after the application restarts and reads the same PM data; otherwise, the data is inconsistent. When the inconsistent data is eventually read, PMDK looks for any incomplete transactions to recover the PM data to a consistent state. PMDK thus needs both read and write permissions to the data before the application can read it.

3.1.2 PM Pointers are Restrictive and Inflexible

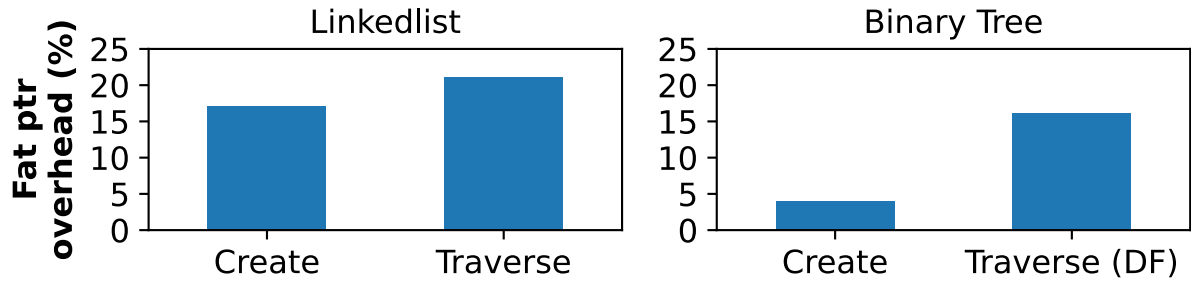


Figure 3.1: Linkedlist and binary tree creation and traversal microbenchmarks, showing overhead of fat pointers vs. native pointers. Single-threaded workload. Linked list's length: 2^{16}, and tree height: 16

Persistent memory enables pointer-rich persistent data, but existing PM systems offer programmers two non-optimal choices: (a) use fat-pointers (base+offset) or self-relative pointers and add overhead to pointer dereference, or (b) use native pointers and abandon relocatability.

Because fat pointers need to be translated to the native format on every dereference, they suffer from a significant performance overhead. Further, the large size of these pointers (in most cases, 128 bits) results in a worse cache locality. Figure 3.1 shows the overhead of fat pointers over native pointers when creating and traversing a linked list and a binary tree. Fat pointers show up to 16% runtime overhead and result in an 18% higher L1 cache miss rate for the binary search tree microbenchmark.

Finally, using a non-native format for pointers makes them opaque and uninterpretable to existing tools like compilers and debuggers.

3.1.3 PM Data is Hard to Relocate and Clone

Regardless of the pointer format choice, PM data is hard to relocate. Consequently, with existing PM systems, users cannot create copies of PM data and open them simultaneously, as the copies would

either map to the same address (with native pointers), or have the same UUID (with fat pointers). Likewise, while some pointer schemes (e.g., self-relative [25]) allow for relocation, they require relocating the entire pool at once and do not support pointers between pools.

When using native pointers, cloned PM data contains conflicting pointers, and the library has no way of rewriting them as the application does not know where the pointers are. A similar problem exists with fat pointers: the application would need to rewrite the base address of each pointer which is impossible in current PM programming systems.

For example, the most widely used PM library, PMDK [49], identifies each "pool" of PM with a UUID and embeds that UUID in its fat pointers. This design requires a specialized tool to copy pools because the copy needs a new UUID and all the pointers it contains need updating. PMDK thus prevents users from opening multiple copies of a pool by checking if the UUID of the pool was already registered when it was first opened. Further, the design also disallows pointers between pools.

With persistent memory becoming more ubiquitous with the emergence of CXL-based memory semantic SSDs [29] and ReRAM-based SoCs [47], beyond just Intel's Optane, the challenges of current persistent memory programming remain present.

3.2 Overview

The *Puddles* library is a new persistent memory library to access PM data that supports application-independent recovery, and implements cheap, transparent relocatability, all while supporting native pointers. To provide these features, Puddles implement system-supported logging and recovery, a shared, machine-local PM address space for PM data, and transparent pointer rewriting to resolve address space

conflicts. In Puddles, every application that needs to access its data does so by mapping a puddle in its virtual address space.

3.2.1 Pools and Puddles

Pools in the Puddle system are named collections of persistent memory regions (i.e., puddles) that allow programmers to allocate and deallocate objects. Pools automatically acquire new memory for object allocation and logging and free any unused memory to the system.

Pools are made of puddles that are mappable units of persistent memory in the Puddle system. While smaller than a pool, puddles can span multiple system pages to accommodate large data structures. The size of a puddle does not change, but pools can grow and shrink with the addition or removal of puddles. Finally, `Libpuddles` supports sharing of pools across machines in its in-memory representation enabling sharing PM data with no serialization.

3.2.2 Puddles Implementation

The Puddle system consists of three major system components (Figure 3.2) that work together to provide application support for mapping and managing puddles.

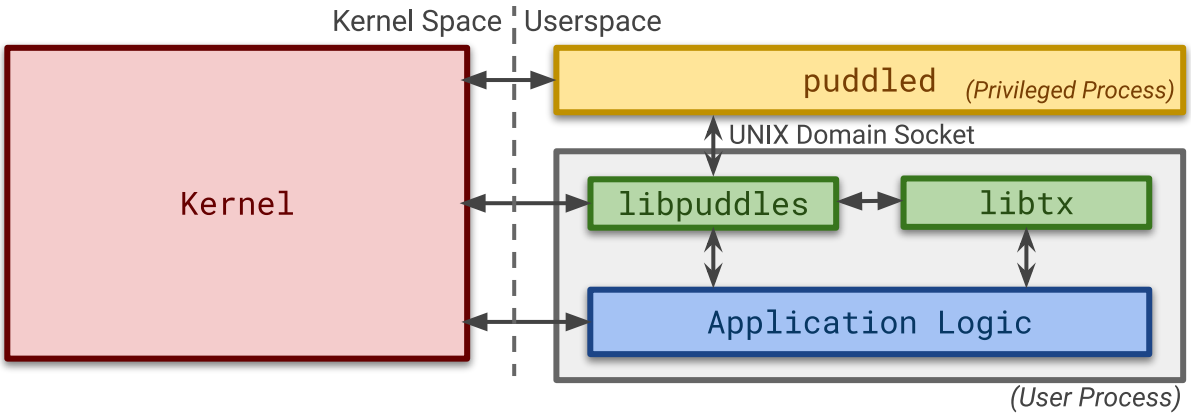


Figure 3.2: The Puddles system includes `Puddled` for system-supported persistence, `Libpuddles`, and `Libtx` for a simple programming interface on top of `Puddled`'s primitives.

1. *Puddled* is the privileged daemon process that manages access to all the puddles in a machine. Puddled implements access control and provides APIs for system-supported recovery and relocating persistent memory data.
2. *Libpuddles* talks to Puddled and provides functions to allocate and manage puddles and pools.
3. *Libtx* is a library that builds on Libpuddles to provide failure-atomic transactions that resemble the familiar PMDK transactions.

Together, Libpuddles and Libtx provide a PMDK-like interface allowing the application to open pools, allocate objects, and execute transactions without managing or caring about individual puddles.

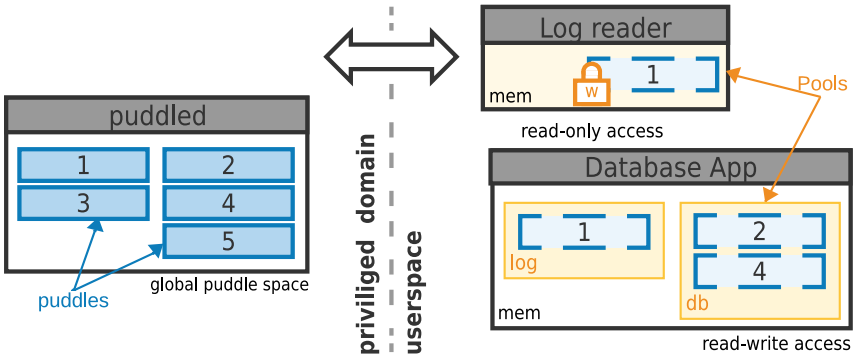


Figure 3.3: Puddles system overview. Each application talks to the Puddles daemon (Puddled) to access the puddles in the system. Applications might map the same puddle with different permission.

Figure 3.3 shows an example database application that demonstrates the benefits of Puddles' approach where the database and logs are partitioned into pools. The application manages a PM database and writes event logs using the *Database app*. A separate *Log reader* process has read-only access to the event logs. Since both the database and the event logs are part of the same global persistent space of a machine, the application can write to both the database and the event log in the same transaction. The

application can also have pointers between the event log and the database, and the Puddles system would make sure that they work in any application with permission to access the data.

3.2.3 Application Independent Recovery.

In Puddles, the application specifies how to recover from a failure, and the system is responsible for recovering the data after a crash. Applications use Puddles' logging interface to register logging regions with `Puddled`. The logging interface is expressive enough to encode undo, redo, and hybrid logging schemes.

In Puddles, which component applies the logs depends on the context: during normal execution, the application applies the logs (if needed), but after a crash, the system applies them on the application's behalf. In the common case, the only additional overhead for the application is the one-time cost of registering the logging region. This interface adds negligible logging overhead relative to PMDK or other PM libraries.

3.2.4 The Puddle Address Space

`Puddled` maintains a machine-wide shared persistent memory space that all puddles in a system are part of. At any time, an application only has parts of the puddle address space mapped into its virtual address space. A single persistent memory space in a machine allows Puddles to support cross-pool pointers and cross-pool transactions.

Applications allocate and request access to puddles from `Puddled`, which grants them the ability to map the puddle into their virtual address space.

The puddle address space is divided into virtual memory pages where the puddles are allocated as contiguous pages. This global PM range only contains the application's persistent data; other parts of

the application's address space, like the text, execution stack, and volatile heaps are still managed using the OS-allocated memory regions. In our implementation of Puddles, we reserve 1 TiB of address space as the global puddle space at a fixed virtual address, disabling Linux's ASLR for the address range. This range is implementation-dependent and is limited only by the virtual memory layout.

3.2.5 Native, Relocatable, and Discoverable Pointers

Puddles contain normal (*i.e.*, neither smart nor fat) pointers to themselves or other puddles. This ensures that normal (non-PM-aware) code can dereference the pointers and read data stored in puddles.

To ensure pointers are meaningful, each puddle must have a current (although not fixed) address that is unique in the machine.

This requirement raises the possibility of address conflicts: If an external puddle (e.g., transferred from another machine) needs to be mapped, its current address may conflict with another pre-existing puddle. In this case, `Libpuddles` will rewrite the pointers when mapping the new puddle into the application's address space. To be able to rewrite pointers, `Libpuddles` stores the type information with allocated objects, allowing it to quickly locate all pointers to support on-demand, incremental relocation (see 3.3.2).

3.2.6 Puddles Programming Interface

To allocate objects, a pool provides a `malloc()`/`free()`-style memory management interface. Allocations made through this API might reside in any puddle in the pool. A Pool's `malloc()` API takes as input the object's type in addition to its size.

```
TX_BEGIN(pool) {
  /*Allocate new node*/
  node_t *new_node
    = pool->malloc<node_t>();
  new_node->data = data;
  new_node->next = nullptr;

  /*Link the new node*/
  TX_ADD(tail);
  tail->next = new_node;
  ...
} TX_END;
```

(a) Puddles

```
TX_BEGIN(pool) {
  /*Allocate new object*/
  TOID(struct node_t) *new_node
    = TX_NEW(node_t);
  D_RW(new_node)->data = data;
  D_RW(new_node)->next = TOID_NULL;

  /*Link the new node*/
  TX_ADD(tail);
  D_RW(tail)->next = new_node;
  ...
} TX_END;
```

(b) PMDK

Figure 3.4: List append example using (a) Puddles, which uses virtual pointers, and (b) using PMDK, which uses base+offset pointers.

Transactions in Puddles are similar to traditional PM transactions (e.g., PMDK-like `TX_BEGIN`... `TX_END`, that mark the start and end of a transaction). `Libpuddles` does not directly manage concurrency or IO in transactions. Instead, like PMDK, it relies on the programmer to use mutexes to implement concurrent transactions and avoid non-transaction-safe IO.

Figure 3.4 is an example of a list append function written using both Puddles and PMDK. The code snippet allocates a new node on persistent memory and appends it to a linked list. Puddles' transactions are thread-local, but unlike PMDK, they support writing to any arbitrary PM data and are not limited to a single pool.

Finally, while Puddles has a C-like API and is implemented using C++, similar to PMDK, Puddles could be extended to support other managed languages like Java.

3.3 System Architecture

Next, we discuss the details of how Puddles provides a flexible logging interface to enable system-supported recovery, handles recovery in case of a failure, and supports relocating PM data within the

virtual address space to provide location independent data. Finally, we complete the discussion with details on various puddle system components.

3.3.1 Crash Consistency

Puddles implement centralized crash consistency by providing system support to guarantee that PM data is consistent before any program accesses the data.

To guarantee the consistency of PM data after a crash, the system needs to be able to replay the application's crash-consistency logs.

To support this, `Libpuddles` communicates the location and format of its logs to the puddle daemon before accessing any data. Further, the logging format (a) needs to be able to support a variety of logging methods, (b) should be safe to apply independently of the application after a crash, and (c) should not add significant runtime overhead.

To solve these challenges, Puddles implement a flexible, system-wide, and low-overhead logging format.

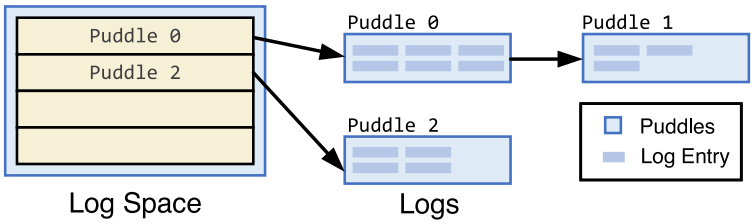


Figure 3.5: Application registers a logspace with the system. A logspace space lists all puddles that the application uses to log data for crash consistency.

Managing logs using log puddles and log spaces. Puddles organize logs using a directory, called a *log space*, that tracks all the active crash-consistency logs. To simplify the implementation,

`Libpuddles` stores both the log space and the logs in designated global puddles not assiciated with any pools. As shown in Figure 3.5, the *log space puddle* is a list of *log space entries*, each identifying a *log puddle* that the application is using to store a log. For instance, an application might have one log puddle per thread to support concurrent transactions. Each of these log puddles would have its own entry in the log space. Once registered, the application can update its log space or modify the logs without notifying the daemon.

Logs in the puddle system can span multiple puddles, enabling them to be arbitrarily long. Figure 3.5 shows an instance of this, where the first log in the log space spans two puddles (Puddle 0 and 1).

Flexible logging format. Applications use a wide range of logging mechanisms (undo [15, 34], redo [97], and hybrid [25, 44, 49]), and Puddles must be flexible enough to support as many as possible. To allow this, Puddles' logging format is expressive enough to cover a wide range of logging schemes and structured enough for `Puddled` to apply them safely after a crash. To achieve flexibility, `Libpuddles` allows the application to write undo- and redo-log entries to a `Puddled` registered log. When the application calls transaction commit, `Libpuddles` processes log entries to be able to recover from a crash.

Puddles ensure that replaying a log can only modify data that the application that created the log could have modified. To accomplish, this libpuddle maintains a persistent record of which puddles an application has access to, and uses this record to check permissions during recovery.

A log in Puddles is a sequence of log entries and includes the metadata to control their recovery behavior. To provide a flexible logging interface, each log entry in Puddles contains the virtual address, checksum, flags field, log data, and the data size. Puddles use a combination of *sequence number* (one for each log entry) and a *sequence range* (one for each log) to control the recovery behavior. For every

log, the log entries that have their sequence number within the log's sequence range are valid, allowing `Libpuddles`(or the application) to selectively (and atomically) enable and disable specific types of log entries.

To implement a variety of logging techniques, Puddles' logging interface allows the application to (1) mark log entries to be of different types (e.g., an undo or redo entry). (2) Disable log entries by their type so `Puddled` will skip them during recovery. (3) Specify recovery order (e.g., recover undo-log entries in reverse order). (4) And, verify that the log entry is complete and uncorrupted.

Figure 3.6 illustrates Puddles' log and log-entry layout. The "`Sequence Range`" in the log and the "`Seq`" field in log-entry control recovery behavior by specifying which entries will be used during recovery. The "`order`" field specifies the order in which log entries will be applied (forward for redo logging, backward for undo logging). "`Next log Ptr`" and "`Last Log Entry Ptr`" track log entry allocation. And, the checksum, like in PMDK, allows the recovery code to identify and skip any entry that only partially persisted because of a crash. The log's metadata includes a pointer to find the next free log entry, a pointer to the current tail entry, and the maximum size of the log.

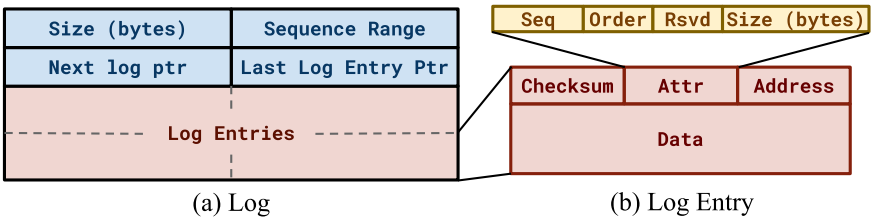
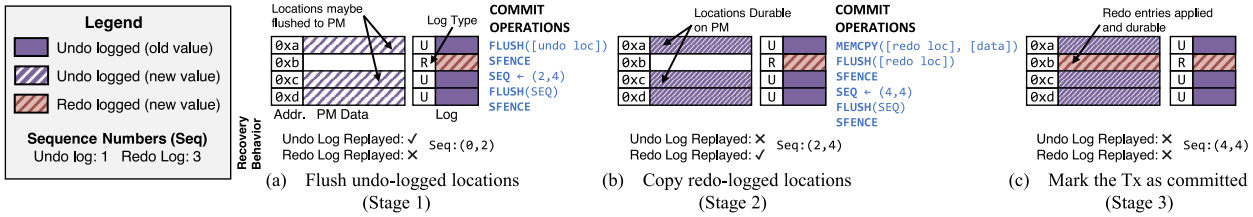


Figure 3.6: Puddles' log-entry and log format.

Finally, to keep transaction costs low, every thread caches a reference to the log puddle used on the first transaction of that thread and reuses it for future transactions. This prevents `Libpuddles` from allocating

a new puddle and adding it to the log space on every transaction. Once the transaction commits, the log is dropped and is ignored by the `Puddled`.

Example hybrid logging implementation. To illustrate the flexibility of Puddle's log format, we will demonstrate how it can implement a hybrid (undo+ redo) logging scheme. Hybrid logging enables low programming complexity for application programmers that use undo logging while allowing libraries to implement their internals using faster but more complex redo logging. For example, PMDK uses hybrid logging to improve performance of allocation/free requests in transactions[8]. While we implement hybrid logging, the programmer can enable support for undo- or redo-only logging by creating only those entries in the log.



Transaction commit. The application (perhaps via a library like `Libtx`) commits a transaction in three stages and without communicating with puddled. Some logging schemes may not need all three stages. For instance, a purely redo-based logging scheme could skip the undo stage. This is shown in Figure 3.7 with the `sfence` and `clwb` ordering and the sequence numbers used for delineating the stages (elaborated on later in this section). The first two stages work on the undo and redo logs, respectively, and the final stage marks the log as invalid. These three stages are:

1. *Stage 1, Flush undo logged locations* (Figure 3.7 a). `Libpuddles` goes through the undo log entries and makes the corresponding locations durable on the PM.

2. *Stage 2, Apply the redo log* (Figure 3.7 b). Once all the undo-logged locations are flushed to PM, `Libpuddles` starts copying new data from the redo logs. Redo logged locations were unchanged before the commit, so, `Libpuddles` copies the new data from the log entry to the corresponding memory location.
3. *Stage 3, TX complete* (Figure 3.7 c). The transaction is complete, and all changes are durable. The log is marked as invalid.

Recovery. After a crash on an unclean shutdown, `Puddled` applies any valid logs it finds in PM. The Puddles recovery process depends on during which stage of a transaction commit the application crashed:

1. *Recovery from crash on or before Stage 1 (Rollback).* First, `Puddled` applies all valid undo-log entries in reverse order.

   At "Stage 1" in Figure 3.7, the undo log entries are still valid, but some locations covered by the undo entries might have been modified before the crash. Thus, on recovery, `Puddled` rolls back the transaction by replaying the undo log in reverse. Which entries to apply and in what order is clear from the sequence range, sequence numbers, and recovery order fields in the log entries.
2. *Recovery from crash on Stage 2 (Roll forward).* If the application crashed during stage 2, `Puddled` applies the redo-log entries from the log.

   In the example, all the undo-logged locations are durable, and `Libpuddles` might have applied some of the redo log entries. On a crash, the recovery would simply roll the transaction forward by resuming the redo log replay.

3. *Recovery from crash on Stage 3 (TX complete).* The TX was marked complete, and all changes are durable.

   No recovery is needed; any logs will be dropped.

After a crash, the daemon compares each log entry's sequence number with the log's sequence range to identify the active stage before the crash. In the hybrid logging example, the application can assign sequence number 1 to the undo log entries and 3 to the redo log entries (Figure 3.7). This assignment allows the `Libpuddles` to mark stages 1, 2, and 3 by setting the log's sequence range to $(0, 2)$ to only replay the undo logs, $(2, 4)$ to only replay the redo logs, and $(4, 4)$ to replay neither.

In Puddles, regardless of whether an entry is an undo or redo log entry, to apply an active log entry, the daemon needs to only copy the entry's content to the corresponding memory location. For example, in undo logging, the entry contains the old data, copying its contents would "undo" the memory location. Similarly, copying contents of a redo log entry would apply the entry, resulting in a "redo" operation.

Logging interface example. Although the application can directly write to the logs most programmers will use PMDK-like transactions provided by `Libtx` to atomically update PM data and create the logs. To undo and redo log data within a transaction, the programmer uses `TX_ADD()` and `TX_REDO_SET()`, respectively. Once the transaction commits, all changes are made durable.

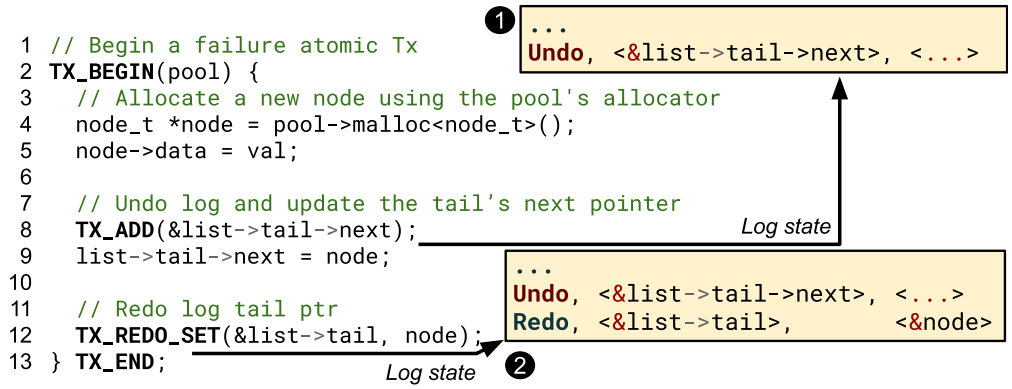


Figure 3.8: Linked List using Puddles' programming interface along with the log's state after various operations.

Figure 3.8 shows an example of a simple linked list implementation to understand the programmer's view of the puddle logging interface. The linked list implementation uses the puddle allocator to allocate a new node (line 4). This new node is automatically undo-logged by the allocator. Next, when the execution of line 8 completes, the log now contains a new undo log entry for the next field of the current tail (). Next, the application redo logs the update to the list's tail pointer (line 12). Being redo logged by the application, this update is performed only on the log; the actual write location will be updated on the transaction commit. Since the application uses hybrid logging, after line 12, the log now contains both undo and redo log entries ().

Once the execution reaches the `TX_END`, `Libpuddles` executes the three stages described in Figure 3.7 to commit the transaction and make the changes durable.

Logging design choices. An alternative (and superficially attractive) option to keeping a single log for all puddles would be to keep per-puddle logs since this would make puddles more self-contained.

Per-puddle logs, however, would have several problems. First, concurrent transactions on a puddle would require multiple logs per puddle, taking up additional space and adding significant complexity in

managing and coordinating these logs. Second, transactions that span puddles (the common case in large data structures) would require a more expensive multi-phase commit protocol.

For logging, Puddles' interface is limited to conventional per-location recovery and does not support implementations that re-execute or resume execution[51, 67, 104], semantic log operations[76], or shadow logging in DRAM and flushing it to PM[14, 66, 107]. These systems use custom logging techniques that require complex recovery conditions that make it difficult to provide a unified interface.

In addition to persistent memory locations, Puddles logs can contain log entries for volatile memory locations that the applications apply on abort to keep volatile and persistent memories consistent with each other. During recovery after a crash, `Puddled` ignores these logs as the destination address does not belong to the global puddles space and the volatile state is lost.

3.3.2 Location Independence

Puddles' ability to relocate PM data within the virtual address space allows it to support location independence and movability with native pointers. Thus, unlike existing PM programming solutions, Puddles allows applications to relocate PM data between pools or create copies of PM data without reallocating and rebuilding the contained data structures in a new PM pool.

In the common case where the assigned address of PM data does not conflict with any existing puddles, `Libpuddles` can simply map the puddle to the application's address space. However, if the puddle's address (3.2.5) is already occupied, Puddles support moving data in the global persistent address space. The ability to move data on conflict is essential to support shipping PM data between machines.

Pointer translation in `Libpuddles` works by incrementally rewriting pointers in puddles. `Libpuddles` maps a puddle on demand and maintains a "*frontier*" of puddles that are unmapped but have a reserved and available location in the global persistent address space.

Frontier puddles 1) are not yet mapped but their eventual location in the global persistent address space is reserved, and 2) are the target of a pointer in a mapped puddle.

Specifically, when an application dereferences a pointer to a frontier puddle, it causes a page fault that `Libpuddles` intercepts. In response, `Libpuddles` maps it to the puddle's assigned virtual address or, on a conflict, to an unreserved range. Next, `Libpuddles` iterates through all the pointers in the puddle and checks if the pointer's destination address is already reserved. If the address is reserved, `Puddled` assigns the puddle pointed by the pointer a new address. This effectively relocates the target puddle in the Puddles' global address space, even if the puddle has not yet been opened or mapped to this location. To accelerate finding pointers in a puddle's internal heap, puddles use allocator metadata to locate internal heap objects (3.3.5).

Once all the pointers in a puddle are rewritten, `Libpuddles` makes it available to the application to access. At this point, only the puddle requested by the application is mapped. If the application dereferences any pointer that points to an unmapped puddle, it generates another page fault.

By marking puddles that have been assigned a new address and have not been mapped, Puddles create a cascading effect of *on-demand* pointer rewrite where the pointers are only rewritten when the data is mapped. Further, since all puddles in a machine are part of the same virtual memory address range, `Libpuddles` can transparently catch access to any unmapped data that is part of this range and map it to the application's address space.

Finally, `Puddled` persistently tracks puddles that were part of a frontier, including puddles that are not yet mapped. In case the machine crashes with some puddles unmapped, the next time one of the puddles from a frontier is mapped, the relocation process resumes.

Pointer maps. For the puddle system to rewrite pointers, it needs to know their location. Puddles solve this problem by requiring the application to register *pointer maps* with `Puddled` for each persistent type used by the application. These pointer maps are simply a list, where each element contains the offset of a pointer within the object and the type of the pointer.

To allow Puddles to rewrite pointers, every allocation in Puddles is associated with a type ID, stored as a 64-bit identifier in the allocator's metadata along with the allocated object. Every `class` or `struct` with a unique name corresponds to a unique type in Puddles. Further, since allocations of a type share their layout, Puddles only need one pointer map per type. To ensure each unique class's name results in a unique type, Puddles rely on C++'s `typeid()` operator, just like PMDK [22]. `typeids` are generated using the Itanium ABI used by gcc and clang, which results in consistent `typeid` across at least gcc v8-12 and clang v7-12.

The overhead of registering pointer maps with `Puddled` is negligible since the number of unique objects an application uses is typically much greater than the number of unique types it uses.

Similar to the centralization of logs discussed in 3.3.1, we centralize the pointer maps in `Puddled` to simplify puddle metadata management. `Puddled` stores the pointer maps in a simple persistent memory hashmap along with its other metadata. While pointer maps could be stored in each puddle for the types in the puddle, doing so would require dynamic memory management of the puddle's metadata. Since the overhead of storing the pointer map information with `Puddled` is low, and it is easy for `Puddled` to export

its pointer maps along with exported puddles, we found the complexity of storing pointer maps in puddles rendered it not worth pursuing.

Relocation on import. Puddles allow sharing of PM data by "exporting" part of the global persistent space. Once exported, PM data retains its in-memory representation, allowing Puddles to "import" it back into the same address space as a copy, or into a different machine.

When importing data, the application asks `Libpuddles` to map a pool into its address space. Pools are a collection of puddles with a designated root puddle, the puddle that holds the pool's root object. Puddles support relocation on import by first mapping the root puddle of the pool. Once `Libpuddles` maps the root puddle, it can begin its pointer rewrite operation and relocate any conflicting data.

Location independence in Puddles extends to support movability by allowing the application to export the underlying data in its in-memory form. Exporting pools in Puddles does not require any serialization and exports the raw in-memory data structures. Once exported, the PM data can be re-imported into the machine's global PM space with no user intervention.

Referential Integrity. While the referential integrity of exported data is a concern, applications are expected to only export self-contained pools. In Puddles, this can be accomplished by limiting inter-pool links or using programming language support to prevent inter-pool pointers.

3.3.3 Puddle Layout

A puddle has two parts, a header, and a heap. Every puddle in the global puddle PM space has a 128-bit universally unique identifier (UUID). The header stores the puddle's metadata information like the puddle's UUID, its size, and allocation metadata. The heap is managed by the `Libpuddles`' allocator

and contains all allocated objects and their associated type IDs. We have configured Puddles to have 4 KiB of header space for every 2 MiB of heap space (0.2% overhead).

3.3.4 Pools

The Puddles system provides a convenient *pool* abstraction on top of puddles to create data-structures that span puddles. Programmers use a pool's `malloc()`-like API to avoid needing to manually manage objects between puddles.

Internally, `Puddled` and `Libpuddles` identify a pool as a collection of puddles and a designated "root" puddle. The root puddle of a pool is the puddle that contains the root of the data structure contained in that pool.

After `Puddled` verifies that the application has access permission to a pool, the library receives the pool's root puddle and maps the puddle to its virtual memory address space. `Libpuddles` then maps the puddle lazily using the on-demand mapping mechanism described in 3.3.2.

Segmenting the persistent memory address space into small puddles to provide the pool interface enables Puddles to relocate, share, and recover individual objects with fine granularity, resulting in low performance and space overhead. For example, puddles limit the cost of pointer rewrite when importing large PM data, limiting the overhead to a few puddle at a time.

3.3.5 Object Allocator

While each puddle can be independently used to allocate objects, applications typically use pools to allocate objects. Using a pool makes it easier for the application to package and send its data structures to a different address space. Further, to track the allocation's type, pool's `malloc()` API takes as input the type of the object in addition to its size.

Since the object allocator always allocates the first object at a fixed offset (root offset) in the puddle, when the application asks `Libpuddles` for the root object, `Libpuddles` can return its address using a simple base and offset calculation.

Object allocations in puddles are handled in the userspace by `Libpuddles`, similar to PMDK. Puddles use a two-level allocator where per-type slab allocators manage small allocations (< 256 B). Large allocations are allocated from a per-puddle buddy allocator. Two-level allocator hierarchy allows Puddles to perform fast allocations of both large and small sizes.

3.3.6 Access Control

In the puddle system, applications must not access puddles that they do not have permission to, while allowing `Puddled` to manage all puddles in a machine. To achieve this, `Puddled` stores each puddle in a separate file on the PM file system. These files are exclusively owned by `Puddled`, and no other process can access them. For applications to access a puddle, `Puddled` maintains a separate, application-facing, UNIX-like permission model.

When an application requests access to a puddle over the UNIX-domain socket, puddled verifies the caller's access using its group ID and user ID. If approved, puddled returns a file descriptor for the requested puddle using the `sendmsg(2)` system call. This file descriptor serves as a capability, letting the application access the underlying puddle without any direct access to the underlying file. Upon receiving the file descriptor, `Libpuddles` maps the puddle to the application's address space and closes the file descriptor. As applications must communicate with `Puddled` to request access to puddles, `Puddled` starts before any other process in the system and controls access to PM data.

While sending file descriptors simplifies puddle management and mapping, an application can still forward them to other processes. However, this limitation is inherent to the UNIX design, i.e., the same vulnerability applies to files, and thus, we assume a similar adversary model. Had we implemented `Puddled` inside the kernel rather than as a privileged daemon, we could have allowed `Puddled` to directly update the application's page tables to map the puddle. This would eliminate the need for sending the file descriptor through the domain socket. However, we decided to leave `Puddled` in user space because it makes it much easier for users to adopt Puddles. Instead of needing to install a custom-built kernel, users of Puddles only need to run `Puddled` and link to our libraries. Finally, managing puddles as files on a DAX-mapped file system has the added benefit of not needing a puddle-scale allocator.

Recovery. `Puddled` extends the puddle access control to recovery and prevents a process from using recovery logs to modify unwritable addresses. During log replay, `Puddled` recreates the mapping for the crashed process by mapping all puddles in the machine-local persistent address space. Recreating the puddle mapping limits `Puddled`'s recovery to locations that the process had write permission to before the crash. If `Puddled` identifies an invalid log, instead of dropping the log, it will be marked as invalid and will not be replayed as the PM data is possibly in a corrupted state. While this may result in denial of service by a malicious application, the effect would be limited to the data accessible by the application.

Let us explore a scenario where a potentially malicious application logs data and then frees the corresponding puddle. In this context, Puddles ensures data integrity by only allowing access to applications with proper permissions. Two scenarios can arise: (1) a new application acquires the freed puddle but allows the original application access to the puddle, potentially risking data corruption during recovery. However, the malicious application already had access to the data before the crash, and thus the security

guarantees are unaffected. (2) If the puddle is unallocated during recovery, or if another application acquires the puddle but does not permit the original application access to its data, the recovery after a crash will fail as the malicious application no longer has access to the puddle, preserving data integrity.

3.4 Results

Table 3.2: System Configuration

Component	Details	Component	Details
CPU & HW Thr.	Intel Xeon 6230 & 20	Linux Kernel	v5.4.0-89
DRAM / PM	93 GiB / 6×128 GiB	Build system	gcc 10.3.0

Puddles perform as fast or faster than PMDK and are competitive with state-of-the-art PM libraries across all workloads while providing system-supported recovery, simplified global PM space, and relocatability. We evaluate Puddles using a BTree, a KV Store using the YCSB benchmark suite, a Linked List, and several microbenchmarks.

Table 3.2 lists the system configuration. For all experiments, we use Optane DC-PMM in App Direct Mode. All workloads use undo-logging for both the application and allocator data logging except for the linked list workload.

3.4.1 Microbenchmarks

This section presents three different measurements to provide insights into how Puddles perform: (a) performance of Puddles' API primitives to compare and contrast them with PMDK, (b) average latency and frequency of `Puddled` operations, and (c), the time taken by different parts of Puddles' relocatability interface.

Table 3.3: Mean latency of Puddles and PMDK primitives.

Operation	Puddles	PMDK
`TX NOP`	11.0 ns	142 ns
`TX_ADD` (8B/4kB)	0.04/1.1 s	0.3/2.2 s
`malloc` (8B/4kB)	0.1/6.8 s	0.4/0.4 s
`malloc+free` (8B/4kB)	5.6/6.0 s	2.0/3.0 s

API primitives. Measurements in Table 3.3 show that across most API operations, Puddles outperform PMDK. To measure the overhead of starting and committing a transaction, we measure the latency of executing an empty transaction – `TX NOP`. Since Puddles' transactions are thread-local and do not allocate a log at the beginning of a transaction, they are extremely lightweight. For an empty transaction, Puddles' overhead only includes a single function call to execute an empty function.

For undo-logging operations (`TX_ADD`), Puddles have latencies similar to PMDK. However, we observe slower allocations (`malloc()`) and de-allocations (`free()`) for Puddles. The performance difference is an artifact of the implementation. For example, Puddles uses undo logging while PMDK uses redo logging for the allocator.

Daemon primitives. Since the Puddles system offers application-independent recovery, it needs to talk to `Puddled` to allocate puddles and perform other housekeeping operations. The daemon communicates with the application using a UNIX domain socket. On average, a round-trip message (no-op) between the daemon and the application takes 46.9 s. Most daemon operations take in the order of a few hundred microseconds to complete.

During execution, the function `RegLogSpace` is called once to register a puddle as the log space and takes on average 134.0 s. `GetNewPuddle` and `GetExistPuddle` are called every time the application needs a puddle. Internally, `Puddled` manages each puddle as a file and returns a file descriptor for puddle requests. Allocating a new file slows down `GetNewPuddle`, and it takes considerably longer (1705.0 s) than calls to `GetExistPuddle` (125.3 s). Even though the call to `GetNewPuddle` is relatively expensive, `Libpuddles` mitigates their overhead by caching a few puddles when the application starts. Caching puddles in the application avoids calls to the daemon when the application runs out of space in a puddle. As we will see with the workload performance, even with relatively expensive daemon calls, Puddles outperform PMDK.

Finally, in addition to the runtime overheads, recovery from a crashed transaction takes 110.1 s in Puddles.

Relocatability primitives. On a request to export a pool, `Puddled` creates copies of the puddles and the associated metadata (e.g., pointer maps). Data export cost, therefore, scales linearly with the size of the PM data and includes a constant overhead per puddle. Exporting a pool takes 0.3 s for 16 B and 0.5 s for 16 MiB of PM data in our implementation. Importing data, on the other hand, is nearly free, as it only includes registering the imported puddles with the daemon (1.5 ms for both 16 B and 16 MiB). After import, if the imported data conflicts with an existing range, the puddle system automatically rewrites all the pointers in the mapped puddle. During pointer rewrite, every pointer in the pool must be visited, so runtime scales linearly with the number of pointers in the pool. Rewriting pointers takes 0.2 ms for 20 pointers, 1.6 ms for 2000 pointers, and 0.5s for 2 million pointers.

Correctness Check. To ensure the correctness of Puddles' logging implementation, we inject crashes into Puddles' runtime and run system-supported recovery. We do this for undo and redo logging and find that Puddles recover application data to a consistent and correct state every time.

3.4.2 Workload evaluation

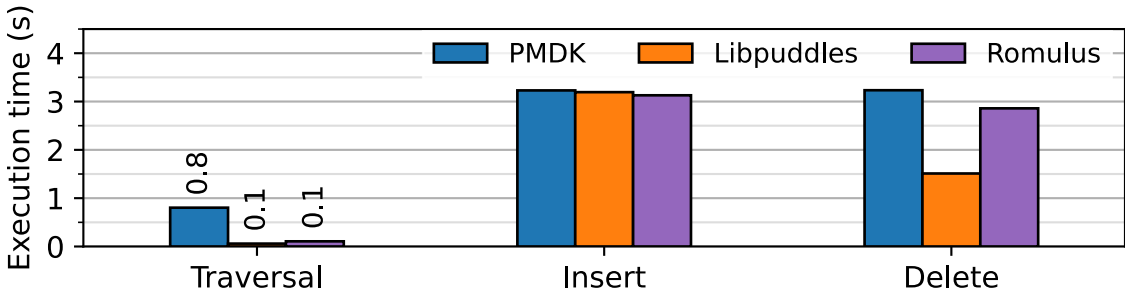


Figure 3.9: Puddles' performance against PMDK and Romulus for singly linked list (lower is better). Native pointers offer a significant performance advantage for Puddles.

To evaluate Puddles' performance, this section includes results for several workloads implemented with Puddles, PMDK, Romulus, go-pmem, and Atlas. Further, to understand the overhead of fat-pointers in PMDK, we used stack samples from PMDK workloads and find that the overhead of fat-pointers ranges from 8.5% for btree, which has multiple pointer dependencies, to 0.76% for the KV-store benchmark that uses fewer pointers per request by making extensive use of hash map and vectors. Finally, across workloads, the daemon primitives result in an additional overhead of about 0.2 ms. This overhead is primarily from registering the first log puddle during the transaction of the benchmark.

Linked List. We compare Puddles' implementation of a singly linked list against PMDK and Romulus. Figure 3.9 compares the performance of three different operations (each performed 10 million times): (a) Insert a new tail node, (b) delete the tail node, and (c) sum up the value of each node. For the

insert, PMDK and Puddles perform similarly and Romulus performs slightly worse, but delete and sum in Puddles outperform PMDK by a significant margin. This performance gap is from the native pointers' lower performance overhead and better cache locality in Puddles. In addition to Puddles' undo logging implementation presented here, we evaluated a hybrid log implementation using undo logging for the allocator and redo logging for the application data and found the performance to be similar to the undo-logging only version, that is, within 5%.

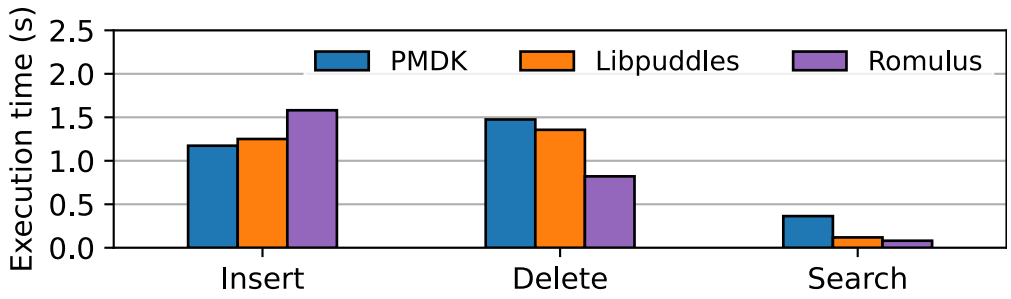


Figure 3.10: Performance of Puddles, PMDK, and Romulus's implementation of an order 8 Btree (lower is better).

B-Tree. Figure 3.10 shows the performance of an identical order 8 B-Tree implementation in PMDK, Puddles, and Romulus. Both the keys and the values are 8 bytes. Similar to the Linked List benchmark, Puddles perform as fast as or better than PMDK across the three operations while being competitive with Romulus. In summary, Puddles' native-pointer results in a much faster (3.1×) performance over PMDK for search operations.

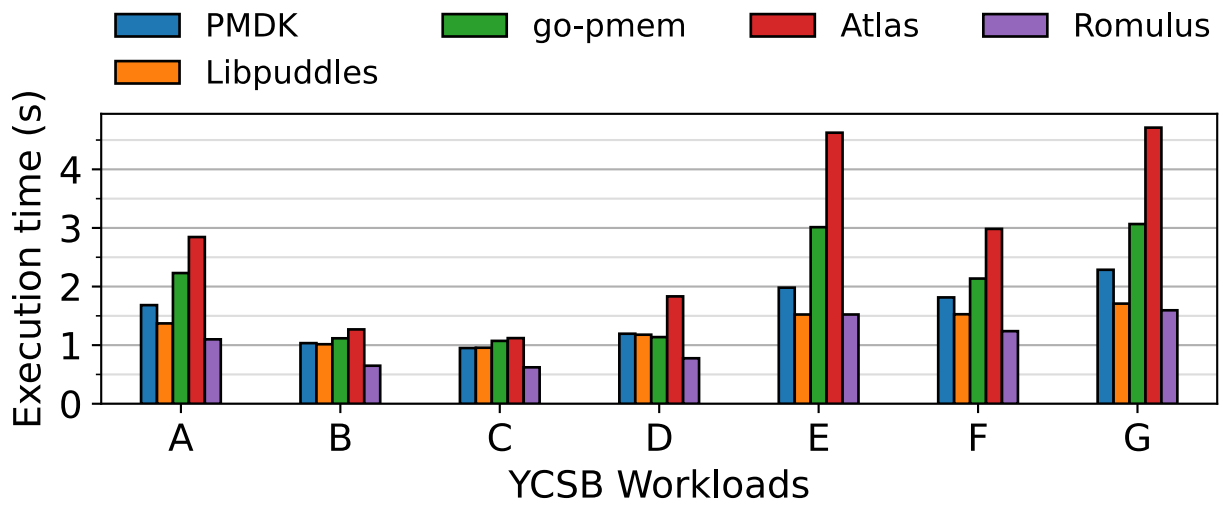


Figure 3.11: KV Store implementation using different PM programming libraries, evaluated using YCSB workloads. Workload D and E use latest and uniform distribution, respectively, while all other workloads use zipfian distribution [26].

KV-Store. To evaluate Puddles' performance in databases, we evaluate PMDK's Key-Value store using Puddles(using undo logging), PMDK, Atlas [15], go-pmem [34], and Romulus [27]. Figure 3.11 shows the performance across these libraries using the YCSB [26] benchmark. For each workload, we run a 1 million keys load workload followed by a run workload with 1 million operations. Across the workloads, Puddles are at least as fast and up to 1.34× faster than PMDK. Against Romulus, Puddles is between 36% slower to being equally fast across the YCSB workloads. Romulus's performance improvement is from its use of DRAM for storing crash-consistency logs. While Puddles' implementation is slower than Romulus, Puddles' relocation and native pointer support is compatible with in-DRAM logs and could be used to improve its performance.

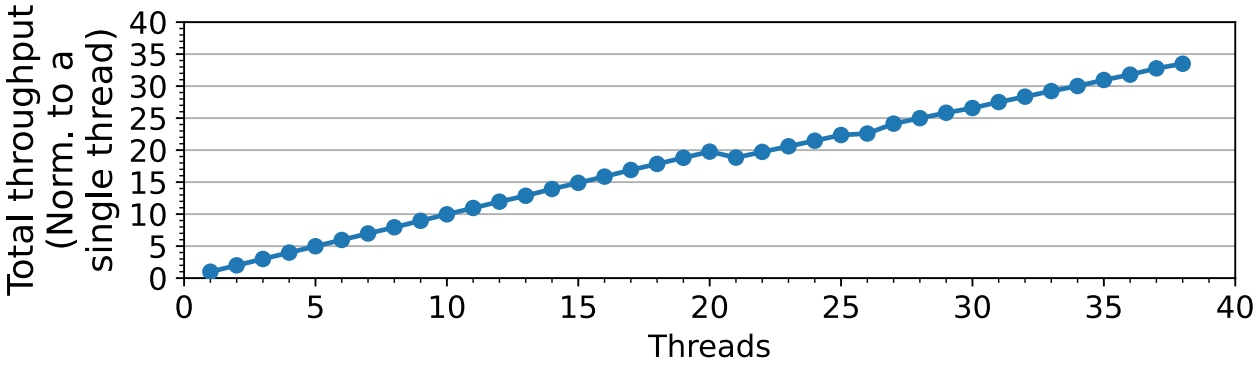


Figure 3.12: Multithreaded workload that processes $1/n^{th}$ of the array per thread.

Multithreaded scaling. To study the multithreaded scalability of Puddles, we used an embarrassingly parallel workload that computes Euler's identity for a floating-point array with a million elements. Figure 3.12 shows the normalized time taken by the workload with the increasing thread count scales linearly and is not limited by Puddles' implementation. In the benchmark, each worker thread works on a small part of the array at a time using a transaction. The workload's throughput scales linearly with the number of threads until it uses all the physical CPUs (20); increasing the number of threads further still results in performance gains, albeit smaller. Puddles' asynchronous logging interface, along with thread-local transactions, allows it to have fast and scalable transactions.

3.4.3 Relocation: Sensor Network Data Aggregation

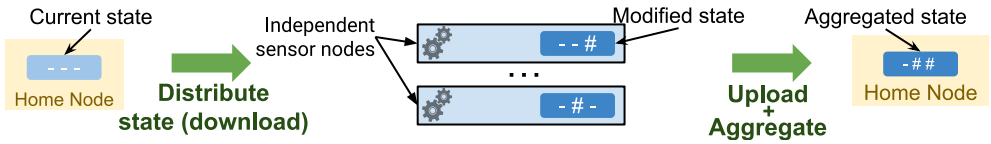


Figure 3.13: **Data Aggregation Workload.** Independent sensor nodes modify copies of pointer-rich data-structures and a home node aggregates the copies into a single copy.

Puddles' ability to relocate data allows it to merge copies of PM data without performing expensive reallocations or serialization/de-serializations. In contrast, applications using traditional PM libraries cannot clone and open multiple copies of PM data because they contain embedded UUIDs or virtual memory pointers.

To demonstrate the ability to relocate PM data across machines, we model a sensor network data-aggregation workload that combines several copies of PM data structures together. Figure 3.13 shows the processing pipeline for this workload. A home node copies a PM-data structure to multiple independent sensor nodes that have their own puddle space. The independent nodes modify these copies and upload the result back to the home node which aggregates the states into a single data structure. Each node modifies the state data using Puddles' transactions and can crash during writes. To model independent nodes with isolated persistent address spaces, we run the nodes in isolated docker containers.

Puddles' ability to resolve address space conflicts in PM data and support for aggregating data allow the nodes to export their state as a portable format to the file system. The home node aggregates the states by reopening the data from each node, and Puddles seamlessly rewrite all the pointers to make the data available for access. PMDK, on the other hand, does not support reading multiple copies of the same data within a single process. For the home node to aggregate the state, it needs to open each copy sequentially and reallocate the data into a larger pool.

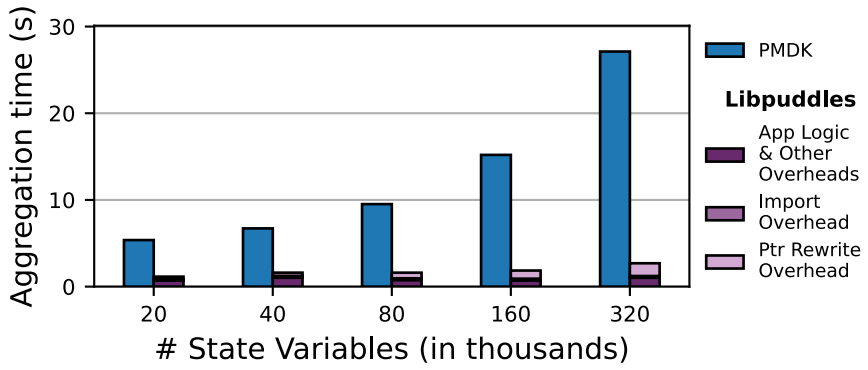


Figure 3.14: Total time taken by PMDK and Puddles to aggregate PM data from 200 sensor nodes.

Figure 3.14 shows the total time spent and Puddles' break down while aggregating states from 200 nodes with 100 to 1600 state variables each. Since PMDK needs to reallocate all the data, it is between 4.7× to 10.1× slower than Puddles. For Puddles, the aggregation has a constant import overhead of 0.2 s, while pointer rewrite overhead scales with the number of elements and increasingly dominates the execution.

3.5 Related Work

Prior persistent memory works have used a variety PM pointer formats; PM Libraries often use non-standard pointer formats that require translation to use [12, 25, 44, 49] or do not allow the programmer to reference data across PM regions, e.g., pools [25, 44, 49], limiting PM programming flexibility. Some persistent memory programming libraries like Pronto [76] simplify PM programming by semantically recording updates like linked list inserts instead of individual memory writes. Unlike Puddles, Pronto and Romulus [27] use DRAM for the working copy of the application data.

Researchers have previously proposed having a global unified virtual memory space that all applications allocate from [11, 12, 56, 82]. TwizzlerOS [12] is one such system for persistent memory that proposes a global persistent object space similar to Puddles. Puddles differ from TwizzlerOS in three major

ways: (a) recovery in TwizzlerOS, like PMDK, relies on the application, (b) unlike native virtual pointers in Puddles, TwizzlerOS uses redirection tables and index-based pointers that can have up to 2 levels of indirections, and (c) finally, unlike Puddles, TwizzlerOS does not support exporting data structure out of its global object space. However, Puddles' recovery and relocation support are orthogonal to TwizzlerOS and can be implemented as an extension to TwizzlerOS. While TwizzlerOS offers a new PM model, the open-source version does not support crash-consistent allocations, making meaningful comparison impossible. And thus, we do not evaluate TwizzlerOS against Puddles.

Similar to TwizzlerOS, several previous OS works have looked into using a single per-node unified address space. Opal [17], Pilot [82], and SingularityOS [46] all provide a single address space for all the processes in a system. While OS like Opal support single, unified address space with the ability to address persistent data, they still suffer from the same limitations that today's PM solutions do.

Opal, for example, offers a global persistent address space, yet it lacks consistency or location independence. Data in Opal is inconsistent until a PM-aware application with write permissions reads it. Puddles, on the other hand, guarantee system-supported recovery with no additional cost other than the one-time setup overhead. Further, since Opal has no information about the pointers embedded in the data, like PMDK, it requires expensive serialization/deserialization to replicate data structures within the address space. No support for pointer translation also means that Opal cannot relocate data structures on an address conflict when importing data from a different address space.

GVMS [41] also introduces the idea of a singular global address space, but for all the application data and shares it across multiple cluster nodes to provide shared memory semantics. In contrast, Puddles

provide a unified address space only for PM while still using traditional address spaces for isolation and security.

Hosking *et al.* [45] present an object store model for SmallTalk that maps objects missing from the process address space on a page fault similar to Puddles, but relies on SmallTalk's runtime indirection for checking and rewriting pointers. Moreover, their solution does not allow storing native pointers in storage, requiring translating pointers every time persistent data is loaded. In contrast, Puddles does not depend on a specific runtime for identifying pointers, and provides application independent recovery and location independence.

Wilson and Kakkad et al. [54, 103] propose pointer translation at page fault time similar to Puddles, however, their solution suffers from several problems. One of the major limitations is no support for objects that span multiple pages as each page can be relocated independently, breaking offset-based access into the object. Puddles solve this problem by translating pointers at puddle granularity, allowing objects to span pages. Further, their solution does not support locating pointers in persistent data, and unlike Puddles' pointer maps, they leave it as future work. Finally, unlike direct-access (DAX) support in Puddles, their solution requires mapping data to the page cache as the data is stored in a non-native pointer format.

3.6 Conclusion

Current PM programming solutions' choices introduce several limitations that make PM programming brittle and inflexible. They fail to recover PM data to a consistent state if the original application writing the PM data is no longer available or if the user no longer has write permission to the data. Existing

PM systems also non-optimally choose among pointer choices that result in unrelocatable PM data and, in some cases, performance overhead.

We solve these problems by providing a new PM programming library–Puddles that supports application-independent crash recovery and location-independent persistent data. To support this, Puddles register logs with the trusted daemon that manages and allocate persistent memory and automatically replays logs after a crash. The puddle system has a single global PM address space that every application shares and allocates from. A global address space and PM data relocation support allows the use of native, unadorned pointers.

Puddles' native virtual pointers provide a significant performance improvement over PMDK's fat pointers. Moreover, Puddles support the ability to relocate PM data seamlessly and faster than traditional solutions.

3.7 Acknowledgement

This chapter contains material from "Puddles: Application-Independent Recovery and Location-Independent Data for Persistent Memory," by Suyash Mahar, Mingyao Shen, T. J. Smith, Joseph Izraelevitz, and Steven Swanson, which appeared in the proceedings of the nineteenth European Conference on Computer Systems (EuroSys 2024). The dissertation author is the primary investigator and the first author of this paper.

Chapter 4

RPCool: Fast Shared Memory RPC For Containers and CXL

Communication within the datacenter needs to be fast, efficient, and secure against unauthorized access, rogue actors, and buggy programs. Remote procedure calls (RPCs)[36, 90] are a popular way of communicating between independent applications and make up a significant portion of datacenter communication, particularly among microservices[9, 72]. However, RPCs require substantial resources to serve today's datacenter communication needs. For instance, Google reports[87] that in the tail (e.g., P99), requests spend over 25% of their time in the RPC stack. One of the significant sources of RPC latency is their need to serialize/deserialize and compress/decompress data before/after transmission.

Shared memory offers an exciting alternative by enabling multiple containers on the same host to share a region of memory without any explicit copies behind the scenes. While applications can send RPCs by serializing and copying data to the shared memory region, an attractive alternative is to share pointers to the original data, significantly lowering their CPU usage.

However, accesses to shared memory raise several safety concerns. Shared memory eliminates the traditional isolation of the sender from the receiver that serialized networking provides. For example, the sender could concurrently modify shared data structures while the receiver processes them, leading to

unsynchronized memory sharing between mutually distrustful applications. This lack of synchronization can result in a range of potentially dire consequences.

Additionally, if applications share pointer-rich data structures over shared memory, they need to take special care in making sure the pointers are not invalid or dangling.

To solve these issues, we propose MemRPC, a shared memory-based RPC library that exposes the benefits of shared-memory communication while addressing the pitfalls described above. Using MemRPC, clients and servers can directly exchange pointer-rich data structures residing in coherent shared memory. MemRPC is the first RPC framework to provide fast, efficient, and scalable shared memory RPCs while addressing the security and scalability concerns of shared memory communication.

MemRPC provides the following features:

1. *High-performance, low-latency RPCs.* MemRPC uses shared memory to provide faster RPCs than existing frameworks.
2. *Preventing sender-receiver concurrent access.* MemRPC prevents the sender from modifying in-flight data by restricting the sender's access to RPC arguments while the receiver is processing them.
3. *Lightweight checks for invalid and wild pointers.* MemRPC provides a lightweight sandbox to prevent dereferencing invalid or wild pointers while processing RPC arguments in shared memory.
4. *API compatibility.* MemRPC provides eRPC[55]()-like API when sending RPCs with traditional serialized data.

Using MemRPC, applications can construct pointer-rich data structures with a `malloc()`/`free()`-like API and share them as RPC arguments. Clients can choose whether to share the RPC arguments with other clients or keep them private to the server and the client. Moreover, clients can access the data without

deserializing it, traverse pointers within the argument if they are present, and only access the parts of the arguments they need.

In existing systems, MemRPC can provide fast communication between co-located containerized services, but Compute Express Link 3.0 (CXL 3.0) will extend its reach across servers. CXL 3.0 will provide multi-host shared memory, offering an exciting alternative by providing hardware cache coherency among multiple compute nodes. However, CXL memory coherence will likely be limited to rack-scale systems[77]. MemRPC must provide a reasonable backup plan if CXL is not available. And, multi-node shared memory for communication results in challenges with availability and memory management, for example, memory leaks involving multiple hosts. To prevent data loss or memory leaks, MemRPC must notify applications of shared memory failures, and limit shared memory consumption.

MemRPC addresses CXL's limited scalability by implementing a RDMA-based distributed shared memory (DSM) fallback. To coordinate memory management and decide between CXL and RDMA-based communication, MemRPC includes a global orchestrator.

We compare MemRPC against several other RPC frameworks built for RDMA, TCP, and CXL-based shared memory. MemRPC achieves the lowest round-trip time and highest throughput across them for no-op RPCs. We showcase MemRPC's ability to share complex data structures using a JSON-like document store and compared against eRPC[55], gRPC[36], and ZhangRPC[108]. Our results for MemRPC running over CXL 2.0 show a 8.3× speedup for building the database and a 6.7× speedup for search operations compared to the fastest RPC frameworks. In the DeathStarBench social network microservices benchmark, MemRPC improves Thrift RPC's maximum throughput by 10.0%. However we find that the

benchmark's performance is primarily constrained by the need to update various databases on the critical path.

The rest of the chapter is structured as follows: 4.1 presents the overview of RPCs and their limitations. 4.2 describes and evaluates MemRPC in its most simple form, shared memory buffers across containers. 4.3 introduces MemRPC's powerful pointer-rich RPC interface and 4.4 extends MemRPC to use CXL-based multi-node shared memory. Finally, we discuss works related to MemRPC in 4.5 and conclude in 4.6.

4.1 RPCs in Today's World

Modern RPC frameworks grew from the need to make function calls across process and machine boundaries, making programming distributed systems easier [10]. These RPCs provide an illusion of function calls while relying on a layer cake of underlying technologies that results in lost performance [87]. For example, RPC frameworks waste a significant number of CPU cycles on serializing and deserializing RPC arguments to send them over traditional networking interfaces [57].

Existing RPC libraries ignore potential performance savings from leveraging shared memory when it is available. To better understand the attendant challenges, let us examine the structure and limitations of modern RPC systems. Then, we will follow with a discussion of how shared memory can alleviate these problems.

RPCs provide an interface similar to a local procedure call: the sender makes a function call to a function exported by the framework [36, 90]. The RPC framework (or the application) serializes the arguments and sends them over the network to the receiver. At the receiver, the RPC framework deserializes the arguments and calls the appropriate function.

While RPC frameworks provide a familiar abstraction for invoking a remote operation, the underlying technology used results in several limitations.

First, to enable communication over transports like TCP/IP, RPC frameworks serialize and deserialize RPC arguments and return values. This adds significant overhead to sending complex objects (e.g., the lists and maps that make up a JSON-like object in memory).

Second, most RPC frameworks do not support sharing pointer-rich data structures due to different address space layouts between the sender and the receiver. Applications can circumvent this by using "smart pointers" [108] or "swizzling" [102] pointers, but both of these add additional overheads.

Third, the underlying communication layer limits today's RPC frameworks' performance. For example, the two common RPC frameworks, gRPC [36] and ThriftRPC [90] rely on HTTP and TCP, respectively. Some RPC frameworks like eRPC [55] exploit the low latency and high throughput of RDMA to achieve better performance but are still limited by the underlying RDMA network.

4.2 RPCool

RPCool is a zero-copy framework designed for fast and efficient RPC-based communication between hosts connected via shared memory. The shared memory can be on a single server shared between two containerized services or on two different servers that share memory via CXL 3.0. RPCool's underlying mechanism supports pointer-rich RPCs, allowing applications to allocate complex, pointer-rich data structures directly in the shared memory while maintaining safety and isolation. RPCool libraries can also emulate conventional RPC semantics using these mechanisms.

This section describes and evaluates how RPCool enables processes on the same host to send RPC over local shared memory. Later, 4.3 discusses how an application can take advantage of RPCool's

powerful pointer-rich RPC interface. Then, 4.4 extends RPCool to support CXL-based shared memory, enabling RPCool's to take advantage of multi-host shared memory while outperforming traditional RPCs.

To send an RPC with RPCool, an application first requests a shared memory buffer, then constructs or, in legacy applications, serializes its data in the buffer, and finally sends the RPC to the receiver. RPCool makes the buffer available to the receiver over shared-memory, requiring zero-copy between sender and receiver. This interface is similar to traditional RDMA-based RPCs like eRPC[55], where the sender obtains an RDMA buffer, which the RPC library then copies to the receiver.

While shared-memory RPCs enable low-overhead, efficient communication, a naive implementation would sacrifice the isolation that traditional TCP and RDMA based communication provides. For example, consider the eRPC model, after the sender sends an RPC, it can no longer modify the arguments of the RPC as the library creates a new, unshared copy of the buffer for the receiver. By contrast, with shared-memory RPCs, both the sender and the receiver have access to the same shared memory region.

To address this, RPCool needs to prevent concurrent access to shared data. RPCool should let applications take exclusive access of shared memory data to prevent malicious (or buggy) applications from concurrently modifying it. Shared memory RPCs also face the challenge of communication between servers. Ideally, RPCool should not restrict applications to a single host. RPCool achieves this by providing eRPC-like API, enabling RPCool to use eRPC as its backend when shared memory is not available.

In this section, we look at an overview of RPCool's design and how RPCool addresses these challenges.

4.2.1 RPCool Architecture

RPCool uses shared memory to safely communicate between processes. The framework consists of userspace components, a trusted daemon, and modifications to the OS kernel.

In userspace, the RPCool library, `librpcool`, provides APIs for connecting to a specific process using RPCool, sending/receiving RPCs, and managing shared memory objects. `librpcool` relies on RPCool's support in the kernel which it communicates with via the daemon. The daemon and the kernel provide RPCool's security guarantees and map the shared memory regions into the process's address space.

RPCool's architecture includes channels and connections to provide TCP-like communication primitives, manage shared memory, and support for mutual exclusion using *sealing*.

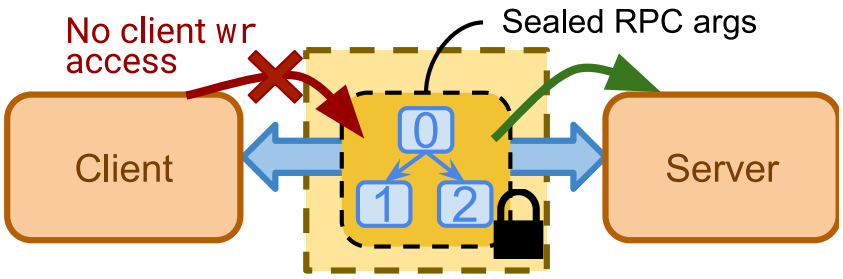


Figure 4.1: Sealing overview.

4.2.2 Channels and Connections

Channels and connections are the basic units for establishing communication between two processes in RPCool. Creating a channel in RPCool is akin to opening a port in traditional TCP-based communication.

Once a channel is open, clients can connect to it and receive a *connection* object that provides access to the connection's shared-memory heap. Every channel in RPCool is identified by a unique, hierarchical name.

Connection heaps hold the connection's RPC buffers, RPC queues to send and receive RPCs, and other metadata. Channels in RPCool automatically use either shared memory or fall back to eRPC, depending on whether the server and client can share memory with each other.

4.2.3 Shared Memory Management

The RPCool daemon tracks applications and shared memory regions so RPCool can cleanup after applications crash and garbage collect orphaned heaps.

To limit the amount of shared memory a process can amass, the daemon enforces a configurable shared-memory quota. The quota limits the amount of heap memory a process has access to at any time, forcing applications to return unused heaps to the kernel.

4.2.4 Shared Memory Safety Issues

When using shared memory to share data structures, there is a risk that a sender might concurrently modify an RPC's arguments while the receiver is processing them. In an untrusted environment, a malicious sender could exploit this to extract sensitive information from the receiver or crash it. While servers usually validate received data, they must also ensure that the sender cannot modify the shared data once it has been validated.

RPCool prevents the sender from modifying RPC arguments while the receiver processes them by revoking write access to the arguments for the sender, thus *sealing* the RPC (Figure 4.1). When an RPC is sealed, the sender cannot modify the arguments until the receiver responds to the RPC.

RPCool's safety mechanisms are designed to limit the effects of a compromised application/microservice. In RPCool's threat mitigation, we assume that the daemon, the kernel, and the hardware are trusted. If a malicious actor compromises an application, RPCool should not allow the malicious actor to access unauthorized memory regions, crash other applications, or extract sensitive information from those applications. These protections are based on the assumption that application validates the data it received before processing it. We assume the receiver of the RPC validates the its arguments as it would with a conventional RPC system.

4.2.5 Sealing RPC Data to Prevent Concurrent Accesses

In scenarios where the receiver does not trust the sender, there are two attractive options to ensure that the senders cannot modify an RPC buffer while an RPC is in flight: First, the application can copy the RPC buffer, which works well for small objects, but for large and complex objects, it is expensive. For these cases, RPCool provides a faster alternative—sealing the RPC buffer. Seals in RPCool apply to the buffer of an in-flight RPC and prevent the sender from modifying them. The sender uses the new `seal()` system call to seal the RPC and relinquish write access to the buffer when required by the receiver. `librpcool` on the receiver can then verify that the region is sealed by communicating with the sender's kernel over shared memory. If not, `librpcool` would return the RPC with an error.

When the receiver has processed the RPC, it marks the RPC as complete. The sender then calls the `release()` system call, and its kernel verifies that the RPC is complete before releasing the seal.

4.2.6 Example RPCool Program

```
1 // Server: Return a string
2 void process(Conn conn) {
3  auto buf = conn->sendbuf;
4  strcpy(buf, "Hello!");
5 }
6 RPC rpc;
7 rpc.open("mychannel");
8 rpc.add(100, &process);
9 auto conn = rpc.accept();
10 conn->listen();
```

(a) Server

```
1 // Client: Make an RPC
2 RPC rpc;
3 rpc.open("mychannel");
4 auto conn = rpc.connect();
5
6 conn->call(100);
7
8 /* prints "Hello!" */
9 std::cout << conn->recvbuf
10           << std::endl;
```

(b) Client

Figure 4.2: A simple ping-pong server using RPCool. The application requests a buffer and shares data using it. `sendbuf` and `recvbuf` point to the same address. Error handling omitted for brevity.

Figure 4.2 shows the source code for an RPCool-based server and client that communicate over an RPCool channel, `mychannel`. RPCool's interface resembles that of eRPC's, but unlike eRPC, RPCool's RPC buffers are zero-copy. First, the server registers `process()` function (Line 8) that responds to the client's requests with a simple string. Once the function is registered, the server listens for any incoming connections (Line 10).

Similarly, once the client has connected to the server (Line 4), calls the server's function (Line 6). Once the server responds to the request, the client prints the result (Line 9–10).

4.2.7 System Details

RPCool's implementation presents several challenges including how to implement sealing efficiently and how to ensure a sender cannot unseal an in-flight RPC. This section details RPCool's system daemon and how RPCool implements sealing.

4.2.7.1 The Daemon and The Kernel

In RPCool, each server runs a trusted daemon that is responsible for handling all connection and channel-related requests.

The daemon is the only entity in RPCool that makes system calls to map or unmap a connection's heap into a process's address space. Consequently, all application must communicate with the daemon to open and close connections or channels. Although applications are permitted to make `seal()` and `release()` calls, they are not allowed to call `mprotect()` on the connection's heap pages. RPCool's kernel enforces this restriction by allowing only the `release()` system call for the address range corresponding to RPCool's heaps, while blocking all other system calls related to page permissions. This prevents applications from bypassing kernel checks for sealed pages.

4.2.7.2 Sealing Heaps

RPCool's seal implementation prevents the sender from concurrently modifying an RPC buffer and enables the receiver to verify the seal before processing an RPC. This section describes how RPCool efficiently implements these features.

Seal implementation. RPCool lets the sender enable sealing on a per-call basis and specify the memory region associated with the request. When a sender requests to seal an RPC, `librpcool` calls a purpose-built `seal()` system call. In response, the kernel makes the corresponding pages read-only for the sender and writes a seal descriptor to a sender-read-only region in the shared memory. The receiver proceeds after it checks whether the region is sealed by reading the descriptor.

Once an RPC is processed, the sender calls the new `release()` system call and the kernel checks to ensure the RPC is complete and breaks the seal. The descriptors are implemented as a circular buffer,

mapped as read-only for the sender but with read-write access for the receiver. These asymmetric permissions allow only the receiver to mark the descriptor as complete and the sender's kernel to verify that the RPC is completed before releasing the seal.

Further, as an application can have several seal descriptors active at a given point in time, the sender also includes an index into the descriptor buffer along with RPC's arguments.

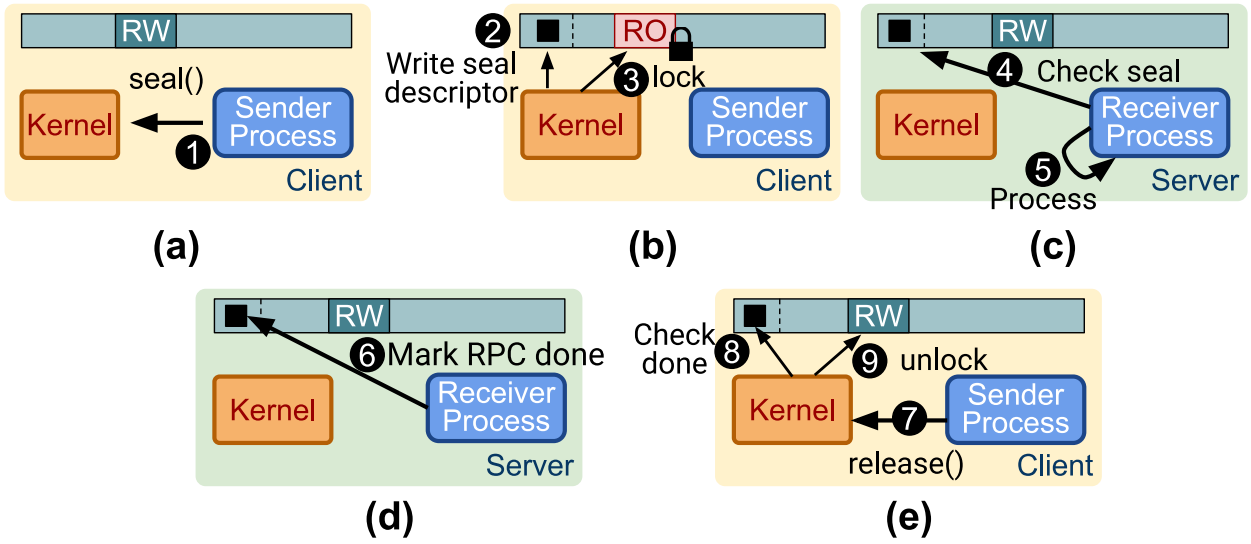


Figure 4.3: **Sealing mechanism overview**. The sender sends a sealed RPC, and the receiver process checks the seal and processes it. Once processed, the receiver marks the RPC as completed, and the sender releases the seal.

Example. Figure 4.3 illustrates the sealing mechanism. Before sending the RPC, the sender calls the `seal()` system call with the region of the memory to seal. Next, the sender's kernel writes the seal descriptor , followed by locking the corresponding range of pages by marking them as read-only in the sender's address space .

Once sealed, the RPC is sent to the receiver. If the receiver is expecting a sealed RPC, it uses `rpc_call::isSealed()` to read and verify the seal descriptor , and processes the RPC if the seal is valid. After processing the request , the receiver marks the RPC as complete in the descriptor and returns the call. Next, when the sender receives the response, it asks its kernel to release the seal . The kernel verifies that the RPC is complete and releases the region by changing the permissions to read-write for the range of pages associated with the RPC .

Optimizing sealing. Repeatedly invoking `seal()` and `release()` incurs significant performance overhead as they manipulate the page table permission bits and evict TLB entries [5]. To mitigate this, RPCool supports batching `release()` calls for multiple RPC buffers. Batching releases amortize the overhead across an entire batch, resulting in fewer TLB shootdowns. To use batched release, applications requests a buffer with batching enabled, copy in or construct the RPC payload in the buffer, and send a sealed RPC.

Upon the RPC's returns, if the application does not immediately need to modify the RPC buffer, it can opt to release the seal in a batch. Batched releases work best when the application does not need to modify the sealed buffer until the batch is processed. However, if needed, the application can invoke `release()` and release the seal on the RPC buffer. In RPCool, each application independently configures the batch release threshold, with a threshold of 1024 achieving a good balance between performance and resource consumption.

4.2.7.3 Adaptive Busy Waiting

RPCool uses busy waiting to monitor for new RPCs and their completion notifications. However, multiple threads busy waiting for RPCs can lead to excessive CPU utilization. To address this issue, RPCool

introduces a brief sleep interval between busy-waiting iterations and can also offload the busy-waiting to a dedicated thread. Specifically, in RPCool, each thread skips sleeping between iterations if the CPU load is less than 20%, sleeps for 5 s when the load is between 20%–30%, and offloads busy-waiting to its dedicated thread if the CPU load exceeds 30%. We observe that this achieves a good balance between CPU usage and performance.

4.2.8 Evaluation

Next, we will look at how RPCool's performance compares against other RPC frameworks using microbenchmarks and two real-world workloads, Memcached and MongoDB. Further, the section evaluates how RPCool with its security features stacks up against traditional RPC frameworks.

Evaluation configuration. All experiments in this work were performed on machines with Dual Intel Xeon Silver 4416+ and 256 GiB of memory. RDMA-based experiments use two servers with Mellanox CX-5 NICs. For the TCP experiments, we use the NIC in Ethernet mode, enabling TCP traffic over the RDMA NICs (IPoIB [23]). Unless stated otherwise, all experiments are run on the v6.3.13 of the Linux kernel with adaptive sleep between busy-wait iterations (4.2.7.3).

All experiments are running on the local shared memory with sealing enabled. RPCool-relaxed refers to RPCool with sealing disabled.

Table 4.1: No-op Latency and throughput of MemRPC (CXL, CXL-relaxed, and RDMA), RDMA-based eRPC, failure-resilient CXL-based ZhangRPC, and gRPC. MemRPC-relaxed is MemRPC with sealing and sandboxing (Section 4.3) disabled.

Framework	**RPCool**	**RPCool Relaxed**	**RPCool (RDMA)**	**eRPC [55]**	**ZhangRPC [108]**	**gRPC [36]**
No-op Latency	1.5 µs	0.5 µs	17.1 µs	3.3 µs	9.4 µs	4.7 ms
Throughput (K req/s)	654.1	1748.1	58.6	303.0	105.3	0.21
Transport	CXL	CXL	RDMA	RDMA	CXL	TCP

Table 4.2: Comparison of various MemRPC operations, repeated 2 million times. Data in column RDMA and about sandboxes will be discussed in Section 4.3. Data in column CXL will be discussed in Section 4.4. (1k = 1024)

	Operation	Mean Latency			Description
		Local DDR	CXL	RDMA	
RPCool Ops	No-op MemRPC-relaxed RPC	0.5 µs	0.5 µs	17.1 µs	RTT for MemRPC no-op RPC.
	No-op Sealed RPC (1 page)	1.3 µs	1.4 µs	—	RTT for MemRPC with seal and no sandbox.
	No-op Sealed+Sandboxed RPC (1 page)	1.5 µs	1.5 µs	—	RTT for MemRPC with seal and a cached sandbox.
	Create Channel	18.7 ms			Channel creation latency
	Destroy Channel	26.5 ms			Channel destruction latency
	Connect Channel	0.3 s			Latency to connect to an existing channel
Sandbox Ops	Cached Sandbox Enter+Exit (1 page)	89.0 ns			Enter+exit a sandbox with a single SHM page
	Cached Sandbox Enter+Exit (1k page)	73.0 ns			Enter+exit a sandbox with 1024 SHM pages
	Cached 8 Sandbox Enter+Exit (1 page)	78.0 ns			Enter+exit 8 sandboxes, no prot. key reassignment
	Uncached 32 Sandbox Enter+Exit (1 page)	0.6 µs			Enter+exit 32 sandboxes, needs reassigning prot. keys
Seal/Release, & `memcpy()`	Seal+standard release, no RPC (1 page)	0.77 ms			Seal and release a single SHM page (no RPC)
	Seal+standard release, no RPC (1k page)	0.92 ms			Seal and release 1024 SHM pages (no RPC)
	Seal+batch release, no RPC (1 page)	0.47 ms			Seal and release in batch a single SHM page (no RPC)
	Seal+batch release, no RPC (1k page)	2.72 ms			Seal and release in batch 1024 SHM pages (no RPC)
	node-node `memcpy()` (1 page)	0.98 µs	1.75 µs	2.31 µs	`memcpy()` latency for node to node copy (1 page)
	node-node `memcpy()` (1k page)	311.9 µs	758.0 µs	368.6 µs	`memcpy()` latency for node to node copy (1024 pages)

No-op round trip latency and throughput. Table 4.1 compares RPCool, RPCool-relaxed variants against several RPC frameworks and shows that RPCool significantly outperforms all other RPC frameworks by a wide margin. Unlike RPCool, ZhangRPC attaches an 8-byte header to every CXL object and uses fat pointers for references. ZhangRPC creates and uses these CXL objects for metadata associated with an RPC, slowing down no-op RPCs and operations like constructing a tree data structure as it require creating a CXL object and a fat pointer per tree node. Further, in ZhangRPC, assigning a node as a child requires the programmer to call a special `link_reference()` API, adding overhead on the critical path.

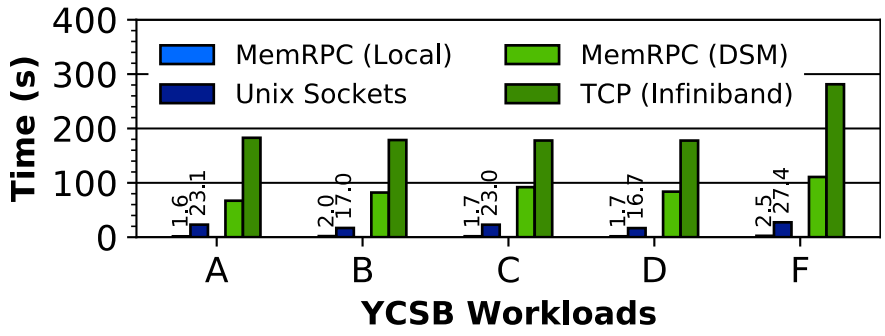


Figure 4.4: Memcached running the YCSB benchmark.

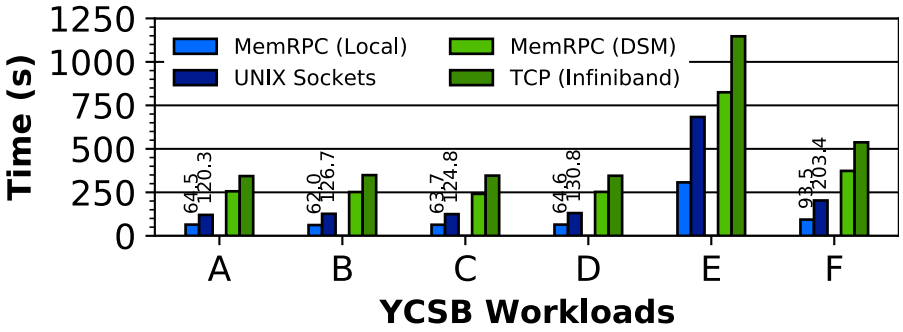


Figure 4.5: MongoDB running the YCSB benchmark.

RPCool operation latencies. Next, we look at the latency of RPCool's features in Table 4.2. RPCool takes only 0.5 s in relaxed mode, and 1.3 s when using the seal operation.

Using Table 4.2, we also look at the cost of the `seal()` and `release()` system calls. Overall, when using standard `seal()`+`release()`, RPCool takes 0.77 s, however, using batched `release()`, this drops to 0.47 s as the cost of changing the page table permission is amortized across multiple `seal()` calls.

Memcached. Figure 4.4 shows the execution time of memcached running the YCSB benchmark [26] using Zipfian distribution. RPCool outperforms UNIX domain sockets by at least 8.55×. As memcached transfers small amounts of data, it uses `memcpy()` instead of sealing for isolation.

For each YCSB workload, we load Memcached with 100k keys and run 1 million operations. Since Memcached is a key-value store, it does not support SCAN operations and thus, it cannot run YCSB's E workload [24].

MongoDB. Figure 4.5 compares the execution time of MongoDB using RPCool vs its built-in UNIX domain socket-based communication. Across the workloads, RPCool's local shared-memory implementation outperforms UNIX domain sockets.

Like Memcached, we evaluate MongoDB with 100k keys and 1 million operations for each YCSB workload and do not implement sealing as MongoDB internally copies the non-pointer-rich data it receives from the client.

4.3 Pointer-Rich RPCs

RPCool supports allocating complex, pointer-rich data structures directly in the shared memory and sharing pointers to them. This unlocks a whole class of use cases for RPCool as applications no longer need to serialize complex data structures in order to share them. In this section, we extend RPCool's

interface by exposing portions of its internal mechanism to the application. This improves performance by allowing end-to-end zero-copy RPCs.

Although sharing complex pointer-rich data structures enables powerful new use cases like shared memory databases where the server shares only a pointer to the requested data, RPCool needs to address several challenges: (a) RPCool should ensure that pointer-rich data structures are valid when shared over RPCs, i.e., pointers do not need to be translated, (b) RPCool should extend the mutual exclusion guarantees between the sender and receiver to any data structures shared by the RPC, and (c) RPCool should enable applications to use native pointers without making them vulnerable to wild or invalid pointers.

4.3.1 Enabling pointer-rich data structures

To address the challenges of sharing pointer-rich data structures, RPCool extends RPC buffers to support native pointers that are valid across servers. This section details how RPCool supports globally valid pointers within RPC buffers while also providing support for connection-specific shared memory heaps.

Globally valid pointers. To be able to share pointer-rich data, RPCool needs to ensure that the shared memory heaps for each connection is mapped to a unique address across servers in a cluster. RPCool ensures this by using a fixed address for each heap across all machines that are under the control of an orchestrator. When a heap is created, the orchestrator assigns it a globally (in the cluster) unique address where the heap will be mapped in a process's address space. Giving each heap a unique address space ensures that a client or server in cluster can safely map it into its address space.

Augmenting RPC buffers for pointer-rich data structures. So far, RPCool provided traditional network-like isolation by sealing the RPC buffer. However, providing similar isolation for pointer-rich data is more challenging as pointer-rich data could be a collection of non-contiguous memory regions making it harder to seal just the object being shared. A naive way of ensuring this isolation would be to let applications to allocate data structures anywhere on the heap, and seal the channel's entire shared memory heap. However, this would make the entire heap, including much unrelated data, unavailable during an RPC.

To address this, RPCool supports allocating complex pointer-rich data structures directly in the RPC buffer where applications can allocate objects and link them together. RPC buffers in RPCool are contiguous sets of pages that hold self-contained data structures. Applications create complex objects in RPC buffers by constructing them directly in the RPC buffer. The sender can thus send an RPC with arguments limited to an RPC buffer, sealing only the data needed for the RPC.

RPCool provides a thread-safe memory allocator to allocate/free objects from the shared memory heaps and RPC buffers. Additionally, RPCool provides several STL-like containers such as `memrpc::vector`, `memrpc::string`, etc. These containers enable programmers to use a familiar STL-like interface for allocating objects but do not preclude custom pointer-rich data structures, e.g., trees or linked lists. The allocator and containers are based on Boost.Interprocess [2]. RPCool provides custom data structures since the C++ standard template library makes no guarantees about accessing data structures from multiple processes or from where the memory for internal need is allocated.

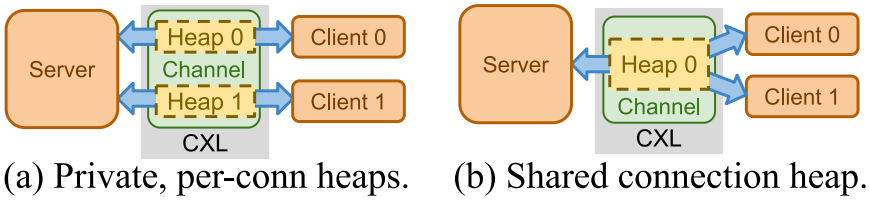


Figure 4.6: Private and public connections in RPCool.

Shared memory heaps. Each connection in RPCool is associated with a shared memory heap, enabling applications to allocate and share RPC data. RPC buffers for a connection are allocated from the connection's heap. Figure 4.6 a–b shows how a single server can serve multiple clients by using independent heaps that are private to each connection (Figure 4.6 a) or by using a single shared heap across multiple connections (Figure 4.6 b). Connections start with a statically sized heap and can allocate additional heaps if they need more space.

4.3.2 Preventing Unsafe Pointer Accesses using Sandboxes

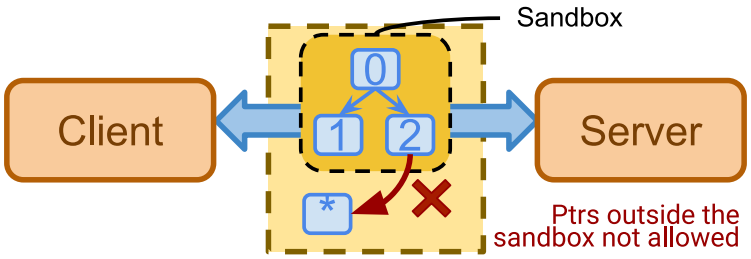


Figure 4.7: Sandboxing overview.

Sharing complex data structures brings the potential for wild or invalid pointers. To ensure safety, RPCool protects applications from these dangers.

When processing an RPC, the receiver might dereference pointers, that point to an invalid memory location and crash the application, or alternatively, they could point to the receiver's private memory, potentially leaking sensitive information. For example, a malicious sender could exploit this by creating a linked list with its tail node pointing to a secret key within the server, thereby extracting the key from a server that computes some aggregate information about the elements in the list.

RPCool includes support for sandboxes, which prevents invalid or wild pointers from causing invalid (or privacy-violating) memory access as the receiver processes the RPC's arguments (Figure 4.7). Sandboxing and sealing are orthogonal and can be applied (or not) to individual RPCs.

Using sandboxes, applications can validate pointers in received RPC data, while sealing ensures that the validated data cannot be modified by the sender while receiver processes it.

When processing a sandboxed RPC, a process enters the sandbox, loses access to its private memory, and has access to only its shared memory heap and a set of programmer-specified variables. If the process tries to access memory outside the sandbox, it receives a signal that the process handles and uses to respond to the RPC.

To minimize the cost of sandboxing incoming RPCs, RPCool relies on Intel's Memory Protection Keys (MPK) [93], avoiding the expensive `mprotect()` system calls. 4.3.5.2 explains the details of how RPCool's sandboxes work.

We considered using non-standard pointers that enable runtime bound checks, but such pointers would limit compatibility with legacy software, compilers, and debuggers and would have significant performance overheads [73].

4.3.3 RDMA Fallback

Shared memory is not always available. While falling back to eRPC is viable for the RPC interface in 4.2, eRPC cannot handle complex data structures. For these, RPCool provides an optimized RDMA-based software coherence system as a fall back.

The system is a minimal two-node RDMA-based shared memory, avoiding the expensive synchronization of multi-node distributed shared memory (DSM) implementations like ArgoDSM [58].

Whenever a node writes to a page, it gets exclusive access to the page by unmapping it from all other nodes that have access to it. After the node has updated the page, it can send an RPC to the other compute node, which can then access the page at which RPCool moves the page to the receiver.

4.3.4 Example RPCool Program

```
// Server: Return a string
void process(Conn conn) {
 auto ret =
  conn->new_<string>(
                 "Hello!");
 printf("ptr: %p\n", ret);
 // Prints: 0xdeadbeef
 return ret;
}
RPC rpc;
rpc.open("mychannel");
rpc.add(100, &process);
auto conn = rpc.accept();
conn->listen();
```

(a) Server

```
// Client: Make an RPC
RPC rpc;
rpc.open("mychannel");
auto conn = rpc.connect();

auto s = conn->call(100);

printf("%p %s\n", s,
       s->c_str());
// Prints: 0xdeadbeef Hello!
```

(b) Client

Figure 4.8: A simple ping-pong server using RPCool. The application requests a buffer and shares pointer-rich data using it. Error checking omitted for brevity.

Figure 4.8 extends the RPCool's previous example (Figure 4.2) to return a pointer to a string instead of copying it into the RPC buffer. Similar to the previous example, once the client has connected to the server (Line 4), it calls the server's function (Line 6). Once the server responds to the request with a pointer to the result, the client prints the result and the pointer of the object received (Line 8–9). As RPCool maps objects at the same address across servers, both the server (line 6) and the client (line 8) will print the same address.

While RPCool enables applications to share complex pointer-rich data structures without serialization, like gRPC and ThriftRPC, RPCool requires the use of custom types for data structures like vectors and strings. For example, to use an array in gRPC, the programmer would use Protobuf schema to declare an array `foo`, compile and link the interfaces with their application, and use methods like `add_foo()` and `foo_size()` to add a new element and check the size of the array, respectively.

4.3.5 System Details

Integrating pointer-rich RPCs into an application requires support for RPC buffers, sandboxes, and the ability for applications to dynamically allocate and share pointer-rich data over shared memory. This section details how RPCool achieves efficient sandboxing while providing an STL-like interface for object management.

4.3.5.1 Extending RPC Buffers

Applications can allocate new objects in the RPC buffer using the buffer's memory management API or by copying in existing object data.

To create an RPC buffer, the programmer requests a buffer of the desired size from the connection's heap using the `Connection::create_buf(size)` API. RPCool allocates the requested amount of memory

from the connection's heap and initializes the buffer's memory allocator. The programmer can then allocate or free objects within the buffer's boundary.

An application can destroy RPC buffers to free the associated memory or reset it to reuse the buffer. Once destroyed or reset, all objects allocated within the buffer are lost.

4.3.5.2 Sandboxes

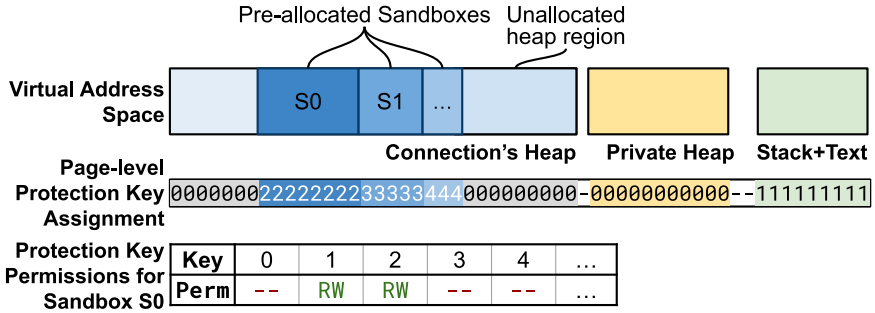


Figure 4.9: Preallocated sandboxes, their key assignment, and key permissions in RPCool.

RPCool enables applications to sandbox an RPC by restricting the processing thread's access to any memory outside of an RPC's arguments. This prevents the applications from accidentally dereferencing pointers to private memory. To be useful, RPCool's sandboxes must have low performance overhead, should allow dynamic memory allocations despite restricting access to the process's private memory, and permit selective access to private variables.

Low overhead sandboxes using Intel MPK. RPCool uses Intel's Memory Protection Keys (MPK) [80] to restrict access to an application's private memory when in a sandbox, avoiding the much more expensive `mprotect()` system call. To use MPK, a process assigns protection keys to its pages and then sets permissions using the per-cpu `PKRU` register. In MPK, keys are assigned to pages at the process-

level, while permissions are set at the thread level. Since MPK permissions are per-thread, they enable support for multiple in-flight RPCs simultaneously. Current Intel processors have 16 keys available.

Once a thread enters a sandbox, it uses Intel MPK to drop access to the process's private memory and any part of the connection's heap except for the sandboxed region. The receiver starts and ends sandboxed execution using the `SB_BEGIN(start_addr, size_bytes)` and `SB_END` APIs. The receiver starts the sandbox with the same address and size as the RPC buffer used for the RPC. However, RPCool also supports sandboxing an arbitrary range of pages within the connection's heap as required by an RPC.

To use Intel's MPK-based permission control, RPCool assigns a key to each region that needs independent access control, as shown in Figure 4.9. RPCool uses one key each for the application's private memory, unsandboxed shared memory regions, and every sandbox. Once a key is assigned to a set of pages, RPCool updates the per-thread `PKRU` register entry to update their permissions.

When an application enters a sandbox, RPCool drops access for all keys except for the one assigned to the sandbox. If the sandboxed thread accesses any memory outside the sandbox, the kernel generates a `SIGSEGV` that the process can choose to propagate to the sender as an error.

Dynamic allocations in sandboxes. As the sandboxed thread no longer has access to the process's private memory, the thread cannot allocate objects in it. However, the application may need to allocate memory from `libc` using `malloc()`/`free()` or invoke a library from within the sandbox that allocates private memory internally.

To address this, RPCool redirects sandboxed `libc` `malloc()`/`free()` calls to a temporary heap instead of the process's private heap. After the sandbox exits, data in this temporary heap is lost. However, redirecting memory allocations works only for libraries and other APIs that free their memory before

returning and do not maintain any state across calls. To safely use stateful APIs over pointer-rich data, an application can validate the pointers in a sandbox before calling the stateful API outside the sandbox.

Accessing data outside the sandbox. When in a sandbox, an application cannot access the connection's private heap, however, in some cases applications might require access to certain private variables to avoid entering and exiting the sandbox multiple times to service an RPC call. To address this, RPCool supports copying programmer-specified private variables into the sandbox's temporary heap. To copy a private variable, the programmer specifies a list of variables in addition to the region to sandbox when starting a sandbox: `SB_BEGIN(region, var0, var1...)`.

To export data generated by a thread inside a sandbox, application can allocate it directly in the buffer for the RPC, and retain access to it after exiting the sandbox.

Optimizing sandboxes. Although changing permissions using Intel MPK takes tens of nanoseconds, assigning keys to pages has similar overheads as the `mprotect()` system call [80]. To avoid assigning keys to on-demand sandboxes, RPCool reserves up to 14 pre-allocated or *cached* sandboxes of varying sizes with pre-assigned keys. This is limited by the number of protection keys available. RPCool reserves 2 keys for the private heap and unsandboxed regions, respectively. To service a request for an uncached sandbox region, RPCool waits for an existing sandbox to end, if needed, and reuses its key. This enables RPCool to dynamically create sandboxes without being limited to 14 pre-allocated sandboxes, albeit at the cost of reassigning protection keys.

4.3.5.3 RDMA Fallback

RPCool includes support for automatic RDMA fallback for pointer-rich communication that spans shared-memory domains. While applications could use traditional RPC frameworks like ThriftRPC or

gRPC to bridge the gap, this leads to additional programming overhead as the programmer needs to pick the API depending on where the target service is running. Moreover, RPCool cannot transparently fall back to an existing RPC system because none of them support sending pointer-based data structures.

RPCool addresses these limitations by implementing a simple RDMA-based shared memory mechanism that is optimized for RPCool's pattern of memory sharing. Where either a server or a client has exclusive access to a shared memory page. When a server attempts to access the data on a page using `load`/`store` instructions, the instruction succeeds if the server has exclusive ownership of the page. If not, the server triggers a page fault, fetches the page from the client, and re-executes the instructions once mapped. Once fetched, the page is marked as unavailable on the client, and it would need to request the page back from the server in order to access the page.

Programming interface. RPCool over RDMA supports communication only between one server and one client. Consequently, RPCool also does not support simultaneous access to a heap over both CXL and RDMA. While RPCool over RDMA only supports two-node communication, all other programmer-facing interfaces are identical to RPCool's CXL implementation, e.g., allocating and accessing shared objects.

This limitation exists because when a process wants exclusive access to a page shared over RDMA, RPCool must unmap the corresponding page from all other processes across the datacenter that have access to it, which adds significant performance overheads and system complexity.

To address this limitation, RPCool includes support for deep-copying pointer-rich data structures between connection heaps using the `conn.copy_from(ptr)` API. `copy_from()` automatically traverses a

linked data structure using Boost.PFR [4] and deep copies to the connection's heap, allowing applications to interoperate between connections of different types without significant programming overhead.

Sealing and sandboxing with RDMA fallback. Sealing and sandboxing for RDMA-based shared memory pages works similarly to RPCool's shared-memory implementation.

When a sender sends a sealed RPC, the corresponding pages are marked as read-only in its address space, preventing any modifications by the sender while the RPC is in-flight. Further, to process an incoming RPC over RDMA fallback, the application can create a sandbox over the RPC's arguments in the same manner as it would for processing an RPC over CXL-based shared memory.

4.3.6 Evaluation

To understand the advantages of integrating RPC with the application, we will look at two applications, CoolDB, a shared memory document store that allows pointer-rich data and a social network website benchmark. Unless noted otherwise, RPCool results are with sealing and sandboxing turned on, while RPCool-relaxed does not turn on sealing or sandboxing. All workloads were evaluated on local shared memory.

RPCool operation latencies. To understand sandbox latency and its impact on RPCool's performance, we measured the latency of a no-op RPC with sealing and sandboxing at 1.5 s compared to 0.5 s with no sealing or sandboxing. When cached, sandboxes (i.e., sandboxes with pre-assigned protection key) have very low enter+exit latency at 78.0 ns. This latency increases to 0.6 s when the sandbox is not cached and RPCool needs to reassign protection keys and set up the sandbox's heap.

Finally, using Table 4.2, we look at the latency of `memcpy()` to compare it against the cost of sealing+sandboxing, which includes sealing a page, starting a sandbox over it, and finally releasing it. This

is because applications can copy RPC arguments to prevent concurrent accesses from the sender without using sealing+sandboxing. We observe that on local DRAM, for more than a page, sealing+sandboxing is faster than `memcpy()` (0.77 s vs 0.98 s). This suggests that for data smaller a page, applications should use `memcpy()`, while for data larger than a page, applications should use sealing+sandboxing.

Memcached and MongoDB. To understand how RPCool's DSM performs, we evaluated Memcached (Figure 4.4) and MongoDB (Figure 4.5) against TCP over Infiniband.

For Memcached and MongoDB, RPCool's DSM implementation outperforms TCP over Infiniband by at least 1.93× and 1.34×, respectively.

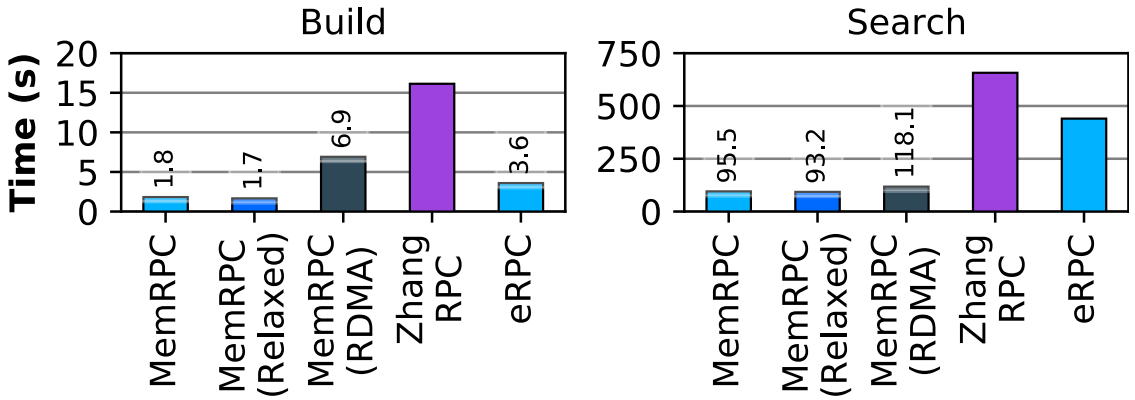


Figure 4.10: RPCool's performance running CoolDB over CXL to showcase worst case performance.

CoolDB. CoolDB is a custom-built JSON document store. Clients store objects in CoolDB by allocating them in the shared memory and passing their references to the database along with a key. CoolDB then takes ownership of the object and associates the object with the key. The clients can read or write to this object by sending CoolDB a read request with the corresponding key. In return, it receives pointer to the in-memory data structure that holds the data.

To evaluate CoolDB, we first populate it with 100k JSON documents using the NoBench load generator [18] (labeled "build" in the figures) and then issue 1000 JSON search queries to the database (labeled "search" in the figures).

Figure 4.10 shows the total runtime of the two operations for the three versions of RPCool (RPCool, RPCool-relaxed, and RPCool-RDMA), ZhangRPC, and eRPC. Overall, RPCool outperforms all other RPC frameworks when running over CXL, including Zhang RPC. However, it slows down considerably when running over RDMA during the build phase, as the shared memory needs to copy multiple pages back and forth. Moreover, as RPCool does not need to serialize the dataset or the queries, it considerably outperforms eRPC for the search operation.

While accessing objects stored in CXL memory has additional latency, CoolDB is a replacement for the use case where applications or microservices use dedicated machines as database, e.g., a dedicated server running MongoDB or Memcached. In such cases, the network access latency would eclipse the additional access latency of CXL shared memory.

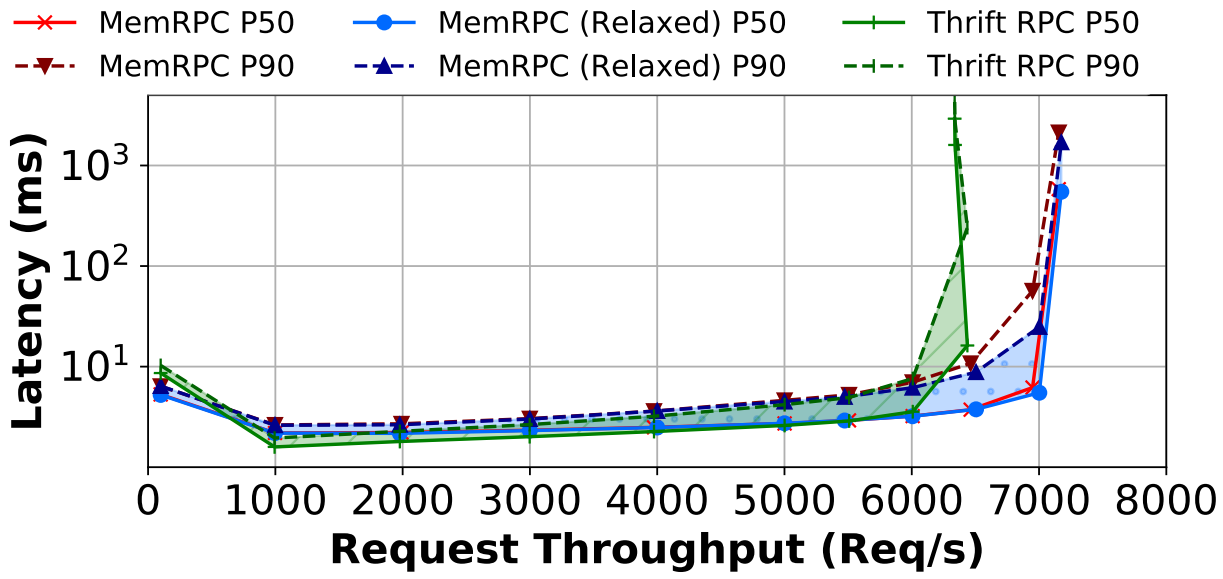


Figure 4.11: DeathStarBench SocialNetwork Benchmark P50 and P90 latencies using ThriftRPC and RPCool (on CXL).

DeathStarBench's Social Network. We evaluate RPCool using the Social Network benchmark from DeathStarBench [33], which models a social networking website. In our evaluation, we replace all ThriftRPC calls among microservices with RPCool on our CXL platform to showcase worst-case performance. However, as DeathStarBench spawns multiple new threads for each request, it contends for the kernel page table lock with RPCool's `seal()` and `release()` calls. To address the issue, we modify the benchmark to use a thread pool instead of creating new threads for each request in both the ThriftRPC and RPCool versions. Additionally, we modified MongoDB to use RPCool. We run DeathStarBench's benchmark that creates user posts under a range of offered loads and measure the median and P90 latency, as shown in Figure 4.11. The experiment is run for 30 seconds for each data point. The results demonstrate

that RPCool (both secure and relaxed versions) and ThriftRPC show similar performance, with RPCool's peak throughput surpassing that of ThriftRPC by 10.0%.

To understand why RPCool performs comparably to Thrift RPC, we looked at where a request spends its time using DeathStarBench's built-in tracing. We found that, on average, about 66% of a request's critical path latency is spent in databases and Nginx, suggesting that DeathStarBench's performance is largely bound by database updates and Nginx.

4.4 RPCool over CXL-Based Shared Memory

CXL promises to deliver multi-node coherent shared memory. RPCool will be able to work on these systems, but a shared memory RPC system over CXL presents additional challenges.

4.4.1 Compute Express Link

CXL 3.0 enables multiple hosts to communicate using fast, byte-addressable, cache-coherent shared memory. CXL-connected hosts will be able to map the same region of shared memory in their address space [38], where updates using `load`/`store` instructions from one host are visible to all other hosts without explicit communication.

To better understand how an RPC framework can exploit CXL's features, we need to first look into how CXL is expected to be deployed. In this work, we consider the scenario where up to 32 servers, with independent OSs are connected to a single pool of shared memory using CXL. Given the challenges of implementing large-scale coherent memory, we assume that CXL memory sharing will not scale far beyond a single rack. We also expect CXL to coexist with conventional networking (TCP and RDMA). Processes within a rack can communicate over the CXL-based shared memory, avoiding expensive network-based communication but can also communicate over RDMA to overcome CXL's limited range.

4.4.2 Challenges

RPCool supports CXL 3.0-based shared memory to enable processes on different hosts to communicate using RPCs. While CXL-based shared memory resembles host-local shared memory, RPCool needs to address the additional challenges of shared memory coordination and failure handling. As shared memory regions are now accessible across hosts and OS domains, RPCool must prevent distributed memory leaks and automatically reclaim memory after failures.

4.4.3 Orchestrator

As RPCool stores RPC queues and buffers in the shared memory, it needs to track the status of each participant to ensure the if a process crashes, other participants are notified in a timely manner. And if everyone accessing the shared memory region crashes, RPCool cleans up any orphaned resources, avoiding memory leaks. Further, RPCool also needs to ensure that no process can consume all the shared memory resources, ensuring fairness.

When the shared memory was limited to a single host, the RPCool daemon was responsible for garbage collection and allocation fairness. However, with multi-host shared memory, RPCool has a global orchestrator that is shared across all nodes participating in RPCool's network.

To address the challenges associated with multi-host shared memory, RPCool's orchestrator uses managed leases on shared memory and imposes shared memory quotas across CXL-connected hosts.

4.4.4 Handling Failures in RPCool

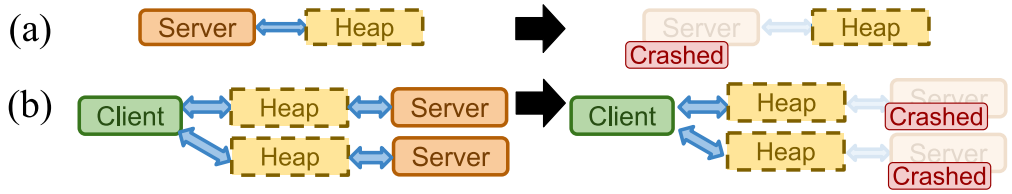


Figure 4.12: **Two possible failure scenarios in RPCool.** (a) Server crash results in an orphaned heap. (b) Client left with heaps after multiple servers crash.

RPCool must be able to deal with the two major shared-memory failure scenarios: (a) if a server process that is not talking to any client dies, the heaps associated with it are leaked, as no process manages them anymore (Figure 4.12 a) and (b) when a client application that connects to multiple servers; if one of these servers fails, the client might not free the associated heaps and retain a significant amount of shared memory (Figure 4.12 b), consuming shared resources.

To address these challenges, RPCool uses leases and quotas. Every time a process maps a heap as part of a connection, it receives a lease from the orchestrator. Applications using shared memory heaps periodically renew their leases. When a process fails, the lease expires, and the orchestrator can notify other participants and clean up any orphaned heaps. Upon a failure notification, an application can either continue using the heap to access previously allocated objects or release it if it is no longer needed, freeing up resources.

However, to implement leases and quotas, RPCool must satisfy three important requirements:

The first is process failure notifications. When any of the communicating processes fail, other processes should be notified of the failure. This notification ensures that clients can perform appropriate housekeeping measures to clean up any partial states associated with a failed server.

Second, in the case of a total failure where multiple processes crash, but the memory node is alive, the system must reclaim memory to prevent memory leaks. Third, RPCool needs to handle scenarios where if one or more servers that a client is communicating with crash, the client could continue using the associated heaps, resulting in the client potentially using up a large portion or all of the shared memory.

Leases. RPCool notifies applications if the server they are communicating with fails and garbage collects orphaned heaps. RPCool achieves this by requiring a lease every time an application maps a connection's heap. Orchestrator uses these leases to track which processes have failed and can notify other applications sharing the memory regions. RPCool creates a lease for each heap, and `librpcool` periodically and automatically renews the lease while the application is running and using the memory.

If the server for a channel fails, the lease expires and the orchestrator notifies all clients connected to the channel of the failure. The clients can continue to access the heap memory but can no longer use it for communication. They can also close the channel. When the last process accessing the heap closes the connection, the orchestrator reclaims the heap.

Quotas. RPCool supports shared memory quotas to limit applications from mapping a large amount of shared memory into their address space. RPCool's orchestrator enforces this configurable quota at the process level. A heap mapped into multiple processes counts against all of their quotas. If mapping a new heap to a process's address space would exceed its quota, the process would need to close enough existing channels to map the new heap.

4.4.5 RPCool's Performance on CXL

In this section, we will look at the performance of RPCool's CXL based multi-host RPCs and application performance using real CXL 2.0 memory expander. As CXL 3.0 devices are not commercially available, we use a CXL 2.0 ASIC-based memory expander with random access latency of 253.4 ns (compared to 114.9 ns for local DRAM) to map all connection heaps. All other configurations are identical to 4.2.8.

Running workloads with RPCool on CXL memory, we observe that workloads that use CXL shared memory solely for communication (e.g., Memcached and MongoDB) do not suffer from the additional access latency of the memory expander. Memcached, MongoDB, and DeathStarBench show a maximum performance reduction of 5.2%, 3.0%, and 1.1%, respectively. The results confirm that the orchestrator needed for enabling CXL-based RPC does not affect RPCool's performance.

However, workloads that store their entire working set on CXL observe higher slowdown. For CoolDB, this reduction is 1.9% and 89.9% for building and searching the database, respectively. Despite this, CoolDB on CXL is still 8.3× and 6.7× faster than eRPC for building and searching the database, respectively.

4.5 Related Work

Some prior works have proposed using RPCs over distributed shared memory. Similar to RPCool, Wang et al. [99] describe RPCs with references to objects over distributed shared memory. However, since they focus on data-intensive applications, they propose immutable RPC arguments and return values and require trust among the applications. Some works also optimize which application unit uses RPCs; Nu [83] breaks down web applications into proclets that share the same address space among multiple hosts and

uses optimized RPCs for communication among them. When proclets are placed on the same machine, they make local function calls, and traditional RPCs otherwise. However, in both cases, proclets need to copy the arguments to the receiver and require mutual trust. Lu et al. [71] improve the performance of serverless functions by implementing `rmap()`, allowing serverless functions to map remote memory, thus avoiding serialization. However, `rmap()` requires mutual trust between the sender and the receiver.

Several other works have looked into using shared memory container communication. Shimmy [62] implement message passing among containers with support for fallback to RDMA. Hobson et al. [42] offer an interface similar to RPCool which support for passing complex pointer rich data structures over shared memory, however, they do not address the security concerns of shared memory communication. PipeDevice [92] offloads inter-container communication to the custom hardware, significantly accelerating it.

Numerous prior studies have explored optimizing the performance of RPC frameworks using RDMA, but they all require serialization and compression, adding performance overheads. HatRPC [64] uses code hints to optimize Thrift RPC and enables RDMA verbs-based communication, while DaRPC [91] implements an optimized RDMA-based custom RPC framework. Kalia et al. [55] propose a highly efficient RDMA-based RPC framework called eRPC that outperforms traditional TCP-based RPCs in latency and throughput. Chen et al. [19] avoid the overhead of sidecars used in RPC deployment by implementing serialization and sidecar policies as a system service. Sidecars are proxy processes that run alongside the main application for policy enforcement, logging, etc., without modifying the application.

Zhang et al. [108] present a memory management system for CXL-based shared memory. Their implementation provides failure resilience against memory leaks without significant performance over-

heads. In addition to failure resiliency, Zhang et al. also propose CXL-based shared memory RPCs, which we refer to as Zhang RPC. However, Zhang RPC performs significantly slower compared to RPCool (Table 4.1), does not scale beyond a rack, and requires mutual trust among applications. Another CXL-based RPC framework, DmRPC [106] supports RPCs over CXL, however, it requires serialization and mutual trust among processes.

Some works have combined CXL-based shared memory with other communication protocols. CXL over Ethernet [98] uses a host-attached CXL FPGA to transmit CXL.mem requests over Ethernet, enabling host-transparent Ethernet-based remote memory. Rcmp [100] overcomes the limited scalability of CXL-based shared memory by extending it using RDMA. However, similar to `rmap()`, it requires applications to mutually trust each other.

Simpson et al. [89] explore the security challenges of deploying RDMA in the datacenter. The challenges listed in their work, e.g., unauditable writes and concurrency problems, are shared by RPCool and other RDMA-based systems alike. Chang et al. [16] discuss the performance overhead of untrusted senders, as the receiver would need to validate the received pointers and data types. Similar to RPCool, for single-machine communication, Chang et al. propose zero-copy RPCs by directly reading the sender's buffer in trusted environments. Schmidt et al. [86] propose a shared memory read-mostly RPC design where the clients have unrestricted read access to a server's data over shared memory but make protected and expensive RPCs to update it. Further, since the clients cannot hold locks in the shared memory, they implement a multi-version concurrency control to allow updates to the data while clients are reading them. Schmidt et al.'s solution is orthogonal to RPCool and can be combined with it by ensuring read-only permissions for channels in clients and exporting separate secure channels for updates. ERIM [94] uses

MPK to isolate sensitive data and to restrict arbitrary code from accessing protected regions. However, unlike RPCool which confines accesses to a shared memory region while processing an RPC, ERIM uses MPK for protecting sensitive data from malicious components. Finally, the new `mseal()` system call in Linux introduces functionality similar to RPCool's `seal()` system call, but makes the mapping permanent and read-only, making it unsuitable for RPCs.

Several prior works, including FaRM [28], RAMCloud [79], Carbink [109], Hydra [63], and AIFM [84] enable distributed shared memory and support varying levels of failure resiliency. However, they require application support for reads and writes and often use non-standard pointers, breaking compatibility with legacy code and adding programming overhead. In contrast, RPCool supports the same `load`/`store` semantics for CXL- and RDMA-based shared memory. Further, while RPCool's RDMA fallback does not implement erasure coding, its design does not preclude such features.

4.6 Conclusion

This work presents RPCool, a fast, scalable, and secure shared memory RPC framework for the cross-container as well as CXL-enabled world of rack-scale coherent shared memory. While shared memory RPCs are fast, they are vulnerable to invalid/wild pointers and the sender concurrently modifying data with the receiver.

RPCool addresses these challenges by preventing the sender from modifying in-flight data using seals, processing shared data in a low-overhead sandbox to avoid invalid or wild pointers, and automatically falling back to RDMA for scaling beyond a rack. Overall, RPCool either performs comparably or outperforms traditional RPC techniques.

4.7 Acknowledgement

This chapter contains material from “MemRPC: Fast Shared Memory RPC For Containers and CXL,” by Suyash Mahar, Ehsan Hajyjasini, Seungjin Lee, Zifeng Zhang, Mingyao Shen, and Steven Swanson, which is under review. The dissertation author is the primary investigator and the first author of this paper.

Chapter 5

Conclusion

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et aut officiis debitis aut rerum necessitatibus saepe eveniet, ut et voluptates repudiandae sint et molestiae non recusandae. Itaque earum rerum defuturum, quas natura non depravata desiderat. Et quem ad me accedis, saluto: 'chaere,' inquam, 'Tite!' lictores, turma omnis chorusque: 'chaere, Tite!' hinc hostis mi Albucius, hinc inimicus. Sed iure Mucius. Ego autem mirari satis non queo unde hoc sit tam insolens domesticarum rerum fastidium. Non est omnino hic docendi locus; sed ita prorsus existimo, neque eum Torquatum, qui hoc primus cognomen invenerit, aut torquem illum hosti detraxisse, ut aliquam ex eo est consecutus? – Laudem et caritatem, quae sunt vitae sine metu degendae praesidia firmissima. – Filium morte multavit. – Si sine causa, nollem me ab eo delectari, quod ista Platonis, Aristoteli, Theophrasti orationis ornamenta neglexerit. Nam illud quidem physici, credere aliquid esse minimum, quod profecto numquam putavisset, si a Polyaeno, familiari suo, geometrica discere maluisset quam illum etiam ipsum.

Bibliography

[1] 2020. Compute Express Link (CXL) Specification. Revision 2.0.

[2] 2022. Boost.Interprocess. In *Boost 1.79.0 Documentation*. Retrieved from https://www.boost.org/doc/libs/1_42_0/doc/html/interprocess.html

[3] 2022. Intel® 64 and IA-32 Architectures Software Developer's Manual. *Volume 3: System programming Guide* (2022).

[4] 2023. 26. Boost.PFR 2.2. In *Boost 1.84.0 Documentation*. Retrieved from https://www.boost.org/doc/libs/1_84_0/doc/html/boost_pfr.html

[5] Nadav Amit, Amy Tai, and Michael Wei. 2020. Don't shoot down TLB shootdowns!. In *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020. 1–14.

[6] PostgreSQL Authors. 2022. PostgreSQL. Retrieved from https://www.postgresql.org/

[7] SQLite Authors. SQLite, Write-ahead logging. Retrieved from https://www.sqlite.org/wal.html

[8] Piotr Balcer. PMDK Pull Request #2716: obj: introduce hybrid transactions. Retrieved from https://github.com/pmem/pmdk/pull/2716

[9] Luiz André Barroso, Jeffrey Dean, and Urs Holzle. 2003. Web search for a planet: The Google cluster architecture. *IEEE micro* 23, 2 (2003), 22–28.

[10] Andrew D Birrell and Bruce Jay Nelson. 1984. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)* 2, 1 (1984), 39–59.

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

[12] Daniel Bittman, Peter Alvaro, Pankaj Mehra, Darrell DE Long, and Ethan L Miller. 2020. Twizzler: a Data-Centric OS for Non-Volatile Memory. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020. 65–80.

[13] Wentao Cai, Haosen Wen, H Alan Beadle, Chris Kjellqvist, Mohammad Hedayati, and Michael L Scott. 2020. Understanding and optimizing persistent memory allocation. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management*, 2020. 60–73.

[14] Daniel Castro, Paolo Romano, and João Barreto. 2018. Hardware Transactional Memory Meets Memory Persistency. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018. 368–377.

[15] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*, 2014. ACM, Portland, Oregon, USA, 433–452. https://doi.org/10.1145/2660193.2660224

[16] Chi-Chao Chang, Grzegorz Czajkowski, Chris Hawblitzel, Deyu Hu, and Thorsten von Eicken. 1998. Security versus performance tradeoffs in RPC implementations for safe language systems. In *Proceedings of the 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications (EW 8)*, 1998. Association for Computing Machinery, Sintra, Portugal, 158–161. https://doi.org/10.1145/319195.319219

[17] Jeff Chase, Hank Levy, Miche Baker-Harvey, and Eld Lazowska. 1992. Opal: a single address space system for 64-bit architecture address space. In *[1992] Proceedings Third Workshop on Workstation Operating Systems*, 1992. 80–85.

[18] Craig Chasseur, Yinan Li, and Jignesh M. Patel. 2013. Enabling JSON Document Stores in Relational Systems. In *International Workshop on the Web and Databases*, June 2013. Retrieved from https://pages.cs.wisc.edu/~jignesh/publ/argo-short.pdf

[19] Jingrong Chen, Yongji Wu, Shihan Lin, Yechen Xu, Xinhao Kong, Thomas Anderson, Matthew Lentz, Xiaowei Yang, and Danyang Zhuo. 2023. Remote procedure call as a managed system service. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI '23)*, 2023. 141–159.

[20] Wooseong Cheong, Chanho Yoon, Seonghoon Woo, Kyuwook Han, Daehyun Kim, Chulseung Lee, Youra Choi, Shine Kim, Dongku Kang, Geunyeong Yu, and others. 2018. A flash memory controller for 15 μs ultra-low-latency SSD using high-speed 3D NAND flash with 3 μs read time. In *2018 IEEE International Solid-State Circuits Conference-(ISSCC)*, 2018. 338–340.

[21] Jungsik Choi, Jaewan Hong, Youngjin Kwon, and Hwansoo Han. 2020. Libnvmmio: Reconstructing software IO path with failure-atomic memory-mapped interface. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, 2020. 1–16.

[22] Igor Chorążewicz. 2018. libpmemobj-cpp Source Code. Retrieved from https://github.com/pmem/libpmemobj-cpp/blob/380becd7170e639af05d122b076ac1c418504ae6/include/libpmemobj++/detail/common.hpp#L174-L177

[23] Jerry Chu and Vivek Kashyap. 2006. Transmission of IP over InfiniBand (IPoIB). https://doi.org/10.17487/RFC4391

[24] cimballihw. 2016. memcached SCAN always fail #668. Retrieved from https://github.com/brianfrankcooper/YCSB/issues/668

[25] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*, 2011. ACM, Newport Beach, California, USA, 105–118. https://doi.org/10.1145/1950365.1950380

[26] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*, 2010. ACM, Indianapolis, Indiana, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[27] Andreia Correia, Pascal Felber, and Pedro Ramalhete. 2018. Romulus: Efficient algorithms for persistent transactional memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, 2018.

[28] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*, April 2014. USENIX Association, Seattle, WA, 401–414. Retrieved from https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi/'c

[29] Samsung Electronics. Samsung Electronics Unveils Far-Reaching, Next-Generation Memory Solutions at Flash Memory Summit 2022. Retrieved from https://news.samsung.com/global/

samsung-electronics-unveils-far-reaching-next-generation-memory-solutions-at-flash-memory-summit-2022

[30] FAL Labs. 2010. Kyoto Cabinet: a straightforward implementation of DBM. Retrieved from http://fallabs.com/kyotocabinet/

[31] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch, and Erez Petrank. 2020. NVTraverse: In NVRAM Data Structures, the Destination is More Important than the Journey. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*, 2020. London, UK, 377–392.

[32] Michal Friedman, Erez Petrank, and Pedro Ramalhete. 2021. Mirror: Making Lock-Free Data Structures Persistent. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 1218–1232.

[33] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, and others. 2019. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019. 3–18.

[34] Jerrin Shaji George, Mohit Verma, Rajesh Venkatasubramanian, and Pratap Subrahmanyam. 2020. go-pmem: Native support for programming persistent memory in go. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020. 859–872.

[35] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. 2018. Persistency for Synchronization-Free Regions. *SIGPLAN Not.* 53, 4 (June 2018), 46–61. https://doi.org/10.1145/3296979.3192367

[36] Google Inc. 2021. gRPC. Retrieved February 21, 2023 from https://grpc.io/

[37] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. 2021. Jaaru: Efficiently model checking persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021. 415–428.

[38] John Groves. 2023. Shared CXL 3 memory: what will be required?. Retrieved from https://lpc.events/event/17/contributions/1455/

[39] Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. Pisces: A Scalable and Efficient Persistent Transactional Memory. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, July 2019. USENIX Association, Renton, WA, 913–928. Retrieved from http://www.usenix.org/conference/atc19/presentation/gu

[40] Part Guide. 2011. Intel® 64 and IA-32 architectures software developer's manual. *Volume 3B: System programming Guide, Part* 2, 11 (2011).

[41] Gernot Heiser, Kevin Elphinstone, Stephen Russell, and Jerry Vochteloo. 1993. Mungi: A distributed single address-space operating system. In *Proceedings of the 17th Australasian Computer Science Conference*, 1993. 271–280.

[42] Tanner Hobson, Orcun Yildiz, Bogdan Nicolae, Jian Huang, and Tom Peterka. 2021. Shared-Memory Communication for Containerized Workflows. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2021. 123–132. https://doi.org/10.1109/CCGrid51090.2021.00022

[43] ARM Holdings. 2019. ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.

[44] Morteza Hoseinzadeh and Steven Swanson. 2021. Corundum: Statically-Enforced Persistent Memory Safety. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021. Association for Computing Machinery, Virtual, USA, 429–442. https://doi.org/10.1145/3445814.3446710

[45] Antony L. Hosking and J. Eliot B. Moss. 1993. Object Fault Handling for Persistent Programming Languages: A Performance Evaluation. In *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '93)*, 1993. Association for Computing Machinery, Washington, D.C., USA, 288–303. https://doi.org/10.1145/165854.165907

[46] Galen C. Hunt and James R. Larus. 2007. Singularity: Rethinking the Software Stack. *SIGOPS Oper. Syst. Rev.* 41, 2 (April 2007), 37–49. https://doi.org/10.1145/1243418.1243424

[47] Crossbar Inc. ReThink Embedded Memory with ReRAM. Retrieved from https://www.crossbar-inc.com/products/high-performance-memory/

[48] Intel. 2017. Intel Optane Memory. Retrieved from http://www.intel.com/content/www/us/en/architecture-and-technology/optane-memory.html

[49] Intel. 2017. Persistent Memory Development Kit (PMDK). Retrieved from http://pmem.io/pmdk

[50] Keita Iwabuchi, Lance Lebanoff, Maya Gokhale, and Roger Pearce. 2019. Metall: A Persistent Memory Allocator Enabling Graph Processing. In *2019 IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, 2019. 39–44. https://doi.org/10.1109/IA349570.2019.00012

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

[52] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *CoRR* (2019).

[53] Anant Jhingran and Pratap Khedkar. 1992. Analysis of recovery in a database system using a write-ahead log protocol. *ACM Sigmod Record* 21, 2 (1992), 175–184.

[54] Sheetal V Kakkad and Paul R Wilson. 1999. Address Translation Strategies in the Texas Persistent Store. In *COOTS*, 1999. 99–114.

[55] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*, 2019. 1–16.

[56] Laxmikant V Kalé, Milind Bhandarkar, Milind Bh, and Terry Wilmarth. 1997. Design and implementation of parallel Java with global object space. (1997).

[57] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*, 2015. Association for Computing Machinery, Portland, Oregon, 158–169. https://doi.org/10.1145/2749469.2750392

[58] Stefanos Kaxiras, David Klaftenegger, Magnus Norgren, Alberto Ros, and Konstantinos Sagonas. 2015. Turning centralized coherence and distributed critical-section execution on their head: A

new approach for scalable distributed shared memory. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, 2015. 3–14.

[59] Terence Kelly, Zi Fan Tan, Jianan Li, and Haris Volos. 2022. Persistent Memory Allocation. *ACM Queue magazine 2022* (2022).

[60] Terence Kelly. 2019. Good Old-Fashioned Persistent Memory. *;login: USENIX Mag.* 44, 4 (2019).

[61] Terence Kelly. 2022. Persistent-Memory gawk User Manual. pm-gawk version 2022.08Aug.03.1659520468.

[62] Manan Khasgiwale, Vasu Sharma, Shivakant Mishra, Biljith Thadichi, Jaiber John, and Rahul Khanna. 2023. Shimmy: Accelerating Inter-Container Communication for the IoT Edge. In *GLOBECOM 2023-2023 IEEE Global Communications Conference*, 2023. 4461–4466.

[63] Youngmoon Lee, Hasan Al Maruf, Mosharaf Chowdhury, Asaf Cidon, and Kang G Shin. 2022. Hydra: Resilient and highly available remote memory. In *20th USENIX Conference on File and Storage Technologies (FAST '22)*, 2022. 181–198.

[64] Tianxi Li, Haiyang Shi, and Xiaoyi Lu. 2021. HatRPC: hint-accelerated Thrift RPC over RDMA. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21)*, 2021. Association for Computing Machinery, St. Louis, Missouri. https://doi.org/10.1145/3458817.3476191

[65] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. DudeTM: Building durable transactions with decoupling for persistent memory. *ACM SIGPLAN Notices* 52, 4 (2017).

[66] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. DudeTM: Building durable transactions with decoupling for persistent memory. *ACM SIGPLAN Notices* 52, 4 (2017), 329–343.

[67] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L Scott, Sam H Noh, and Changhee Jung. 2018. iDO: Compiler-directed failure atomicity for nonvolatile memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018. 258–270.

[68] Sihang Liu, Suyash Mahar, Baishakhi Ray, and Samira Khan. 2021. PMFuzz: test case generation for persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021. 487–502.

[69] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. 2020. Cross-failure bug detection in persistent memory programs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020. 1187–1202.

[70] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. 2019. PMTest: A Fast and Flexible Testing Framework for Persistent Memory Programs. In *the Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[71] Fangming Lu, Xingda Wei, Zhuobin Huang, Rong Chen, Minyu Wu, and Haibo Chen. 2024. Serialization/Deserialization-free State Transfer in Serverless Workflows. In *Proceedings of the Nineteenth European Conference on Computer Systems*, 2024. 132–147.

[72] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. 2021. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*, 2021. 412–426.

[73] Suyash Mahar, Mingyao Shen, TJ Smith, Joseph Izraelevitz, and Steven Swanson. 2024. Puddles: Application-Independent Recovery and Location-Independent Data for Persistent Memory. In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys '24)*, 2024. Association for Computing Machinery, Athens, Greece, 575–589. https://doi.org/10.1145/3627703.3629555

[74] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2022. TPP: Transparent Page Placement for CXL-Enabled Tiered Memory. *arXiv preprint arXiv:2206.02878* (2022).

[75] A. Memaripour and S. Swanson. 2018. Breeze: User-Level Access to Non-Volatile Main Memories for Legacy Software. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, 2018. 413–422.

[76] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. 2020. Pronto: Easy and Fast Persistence for Volatile Data Structures. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*, 2020. Association for Computing Machinery, Lausanne, Switzerland, 789–806. https://doi.org/10.1145/3373376.3378456

[77] James Morra. 2023. CXL Switch SoC Unlocks More Memory for AI. Retrieved from https://www.electronicdesign.com/technologies/embedded/article/21272132/electronic-design-cxl-switch-soc-unlocks-more-memory-for-ai

[78] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. 2020. AGAMOTTO: How Persistent is your Persistent Memory Application?. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020. 1047–1064.

[79] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, and others. 2015. The RAMCloud storage system. *ACM Transactions on Computer Systems (TOCS)* 33, 3 (2015), 1–55.

[80] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. 2019. libmpk: Software abstraction for Intel memory protection keys (Intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC '19)*, 2019. 241–254.

[81] Stan Park, Terence Kelly, and Kai Shen. 2013. Failure-Atomic Msync(): A Simple and Efficient Mechanism for Preserving the Integrity of Durable Data. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*, 2013. Association for Computing Machinery, Prague, Czech Republic, 225–238.

[82] David D Redell, Yogen K Dalal, Thomas R Horsley, Hugh C Lauer, William C Lynch, Paul R McJones, Hal G Murray, and Stephen C Purcell. 1980. Pilot: An operating system for a personal computer. *Communications of the ACM* 23, 2 (1980), 81–92.

[83] Zhenyuan Ruan, Seo Jin Park, Marcos K Aguilera, Adam Belay, and Malte Schwarzkopf. 2023. Nu: Achieving Microsecond-Scale Resource Fungibility with Logical Processes. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI '23)*, 2023. 1409–1427.

[84] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. 2020. AIFM: High-Performance,Application-Integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, 2020. 315–332.

[85] Samsung Semiconductor. 2024. CXL Memory Module - Box (CMM-B). Retrieved from https://semiconductor.samsung.com/us/news-events/tech-blog/cxl-memory-module-box-cmm-b/

[86] Rene W Schmidt, Henry M Levy, and Jeffrey S Chase. 1996. Using shared memory for read-mostly RPC services. In *Proceedings of HICSS-29: 29th Hawaii International Conference on System Sciences*, 1996. 141–149.

[87] Korakit Seemakhupt, Brent E Stephens, Samira Khan, Sihang Liu, Hassan Wassel, Soheil Hassas Yeganeh, Alex C Snoeren, Arvind Krishnamurthy, David E Culler, and Henry M Levy. 2023. A Cloud-Scale Characterization of Remote Procedure Calls. In *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023. 498–514.

[88] Kapil Sethi. 2022. Expanding the Limits of Memory Bandwidth and Density: Samsung's CXL Memory Expander. Retrieved from https://semiconductor.samsung.com/news-events/tech-blog/expanding-the-limits-of-memory-bandwidth-and-density-samsungs-cxl-dram-memory-expander/

[89] Anna Kornfeld Simpson, Adriana Szekeres, Jacob Nelson, and Irene Zhang. 2020. Securing RDMA for High-Performance Datacenter Storage Systems. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '20)*, July 2020. USENIX Association. Retrieved from https://www.usenix.org/conference/hotcloud20/presentation/kornfeld-simpson

[90] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. 2007. Thrift: Scalable cross-language services implementation. *Facebook white paper* 5, 8 (2007), 127.

[91] Patrick Stuedi, Animesh Trivedi, Bernard Metzler, and Jonas Pfefferle. 2014. DaRPC: Data center RPC. In *Proceedings of the ACM Symposium on Cloud Computing*, 2014. 1–13.

[92] Qiang Su, Chuanwen Wang, Zhixiong Niu, Ran Shu, Peng Cheng, Yongqiang Xiong, Dongsu Han, Chun Jason Xue, and Hong Xu. 2022. PipeDevice: a hardware-software co-design approach to intra-host container communication. In *Proceedings of the 18th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '22)*, 2022. Association for Computing Machinery, Roma, Italy, 126–139. https://doi.org/10.1145/3555050.3569118

[93] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. 2020. Intra-unikernel isolation with Intel memory protection keys. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2020. 143–156.

[94] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*, 2019. 1221–1238.

[95] Rajat Verma, Anton Ajay Mendez, Stan Park, Sandya Mannarswamy, Terence Kelly, and Charles B. Morrey. 2015. Failure-Atomic Updates of Application Data in a Linux File System. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*, 2015. USENIX Association, Santa Clara, CA.

[96] Viking Technologies. DDR4 NVDIMM. Retrieved from https://www.vikingtechnology.com/non-volatile-memory/ddr4-nvdimm/

[97] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *ASPLOS '11: Proceeding of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011. ACM, Newport Beach, California, USA, .

[98] Chenjiu Wang, Ke He, Ruiqi Fan, Xiaonan Wang, Wei Wang, and Qinfen Hao. 2023. CXL over Ethernet: A novel FPGA-based memory disaggregation design in data centers. In *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2023. 75–82.

[99] Stephanie Wang, Benjamin Hindman, and Ion Stoica. 2021. In reference to RPC: it's time to add distributed memory. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '21)*, 2021. Association for Computing Machinery, Ann Arbor, Michigan, 191–198. https://doi.org/10.1145/3458336.3465302

[100] Zhonghua Wang, Yixing Guo, Kai Lu, Jiguang Wan, Daohui Wang, Ting Yao, and Huatao Wu. 2024. Rcmp: Reconstructing RDMA-Based Memory Disaggregation via CXL. *ACM Transactions on Architecture and Code Optimization* 21, 1 (2024), 1–26.

[101] Zixuan Wang, Mohammadkazem Taram, Daniel Moghimi, Steven Swanson, Dean Tullsen, and Jishen Zhao. NVLeak: Off-Chip Side-Channel Attacks via Non-Volatile Memory Systems. In *USENIX Security Symposium*,.

[102] Paul R Wilson and Sheetal V Kakkad. 1992. Pointer swizzling at page fault time: Efficiently and compatibly supporting huge address spaces on standard hardware. In *Proceedings of the 1992 International Workshop on Object Orientation in Operating Systems*, 1992. 364–377.

[103] Paul R Wilson. 1991. Pointer swizzling at page fault time: Efficiently supporting huge address spaces on standard hardware. *ACM SIGARCH Computer Architecture News* 19, 4 (1991), 6–13.

[104] Yi Xu, Joseph Izraelevitz, and Steven Swanson. 2021. Clobber-NVM: Log Less, Re-execute More. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021. 346–359.

[105] Ziye Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E. Paul. 2017. SPDK: A development kit to build high performance storage applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2017. 154–161.

[106] Jie Zhang, Xuzheng Chen, Yin Zhang, and Zeke Wang. 2024. DmRPC: Disaggregated Memory-aware Datacenter RPC for Data-intensive Applications. In *40th IEEE International Conference on Data Engineering (ICDE)*, May 13, 2024. IEEE, Utrecht, Netherlands.

[107] Lu Zhang and Steven Swanson. 2019. Pangolin: A Fault-Tolerant Persistent Memory Programming Library. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, 2019. 897–912. Retrieved from https://www.usenix.org/conference/atc19/presentation/zhang-lu

[108] Mingxing Zhang, Teng Ma, Jinqi Hua, Zheng Liu, Kang Chen, Ning Ding, Fan Du, Jinlei Jiang, Tao Ma, and Yongwei Wu. 2023. Partial Failure Resilient Memory Management System for (CXL-based) Distributed Shared Memory. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP'23)*, 2023. Association for Computing Machinery, Koblenz, Germany, 658–674. https://doi.org/10.1145/3600006.3613135

[109] Yang Zhou, Hassan MG Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E Culler, Henry M Levy, and others. 2022. Carbink: Fault-Tolerant Far Memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22)*, 2022. 55–71.