UNIVERSITY OF CALIFORNIA SAN DIEGO

Building Distributed Systems with Non-Volatile Main Memories and RDMA Networks

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

in

Computer Science

by

Jian Yang

Committee in charge:

Professor Steven Swanson, Chair
Professor Pamela Cosman
Professor Ryan Kastner
Professor Dean Tullsen
Professor Geoffery Voelker

2019

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

_____

_____

_____

_____

_____

Chair

University of California San Diego

2019

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I am grateful for the support received from many great people during the long journey of pursuing a Ph.D.

I would like to acknowledge my advisor, Professor Steven Swanson, for his kind support as the chair of my committee. His guidance helped me overcome the difficulties and achieve my goals throughout my graduate school time, and enlighted me towards a better career path.

I thank my father, who started me on the path to computer science. I still remember the days playing the 80386 computer in his office. I thank my mother for her giving me unconditional love and belief in me in every choice that I have made in my life. I want to thank my wife Jiawei Gao, for her most reliable support. I could not imagine completing this journey without her.

I would like to extend my gratitude to Professor Pamela Cosman, Professor Ryan Kastner, Professor Dean Tullsen and Professor Geoffery Voelker for their valuable comments and feedback as my committee members. I would also like to take this chance to express my gratitude to my managers and mentors during the summer internships.

I want to acknowledge Prof. Yiying Zhang and Prof. Joseph Izraelevitz for their help and guidance in the papers that we co-authored.

I want to thank other members at NVSL, with whom I had a wonderful time. I give my heartfelt thanks to my colleagues: Morteza Hoseinzadeh, Yanqin Jin, Juno Kim, Jing Li, Robert Liu, Amirsaman Memaripour, Andiry Xu and Lu Zhang.

I also thank my undergraduate research advisor, Professor Haibo Chen, for encouraging me to pursue a Ph.D.

Chapter 1, Chapter 2 and Chapter 3 contain material from“Mojim: A Reliable and Highly-Available Non-Volatile Memory System”, by Yiying Zhang, Jian Yang, Amirsaman Memaripour and Steven Swanson, which appears in the Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2015). The dissertation author is the second investigator and author of this paper. The material in these

chapters is copyright ©2015 by Association for Computing Machinery.

Chapter 1, Chapter 2 and Chapter 4 contain material from "Orion: A Distributed File System for Non-Volatile Main Memories and RDMA-Capable Networks", by Jian Yang, Joseph Izraelevitz and Steven Swanson, which appears in the Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST 2019). The dissertation author was the primary investigator and first author of this paper. The material in these chapters is copyright ©2019 by the USENIX Association.

Chapter 1, Chapter 2 and Chapter 5 contain material from "FileMR: Rethinking RDMA Networking for Scalable Persistent Memory", by Jian Yang, Joseph Izraelevitz and Steven Swanson, which is submitted for publication. The dissertation author was the primary investigator and first author of this paper.

Chapter 2 contains material from "An Empirical Guide to the Behavior and Use of Scalable Persistent Memory," by Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz and Steven Swanson, which is submitted for publication. The dissertation author was the primary investigator and first author of this paper.

VITA

2013	Bachelor of Engineering, Fudan University
2017	Master of Science, University of California, San Diego
2019	Doctor of Philosophy, University of California, San Diego

ABSTRACT OF THE DISSERTATION

Building Distributed Systems with Non-Volatile Main Memories and RDMA Networks

by

Jian Yang

Doctor of Philosophy in Computer Science

University of California San Diego, 2019

Professor Steven Swanson, Chair

High-performance, byte-addressable non-volatile main memories (NVMMs) allow application developers to combine storage and memory into a single layer. These high-performance storage systems would be especially useful in large-scale data center environments where data is distributed and replicated across multiple servers.

Unfortunately, existing approaches of providing remote storage access rest on the assumption that storage is slow, so the cost of the software and protocols is acceptable. Such assumption no longer holds for the fast NVMM. As a result, taking full advantage of NVMMs' potential will require changes in system software and networking protocol.

This thesis focuses on accessing remote NVMM efficiently using remote direct memory

access (RDMA) network. RDMA enables a client to directly access memory on a remote machine without involving its local CPU.

This thesis first presents Mojim, a system that provides replicated, reliable, and highly-available NVMM as an operating system service. Applications can access data in Mojim using normal load and store instructions while controlling when and how updates propagate to replicas using system calls. Our evaluation shows Mojim adds little overhead to the un-replicated system and provides 0.4 to 2.7× the throughput of the un-replicated system.

This thesis then presents Orion, a distributed file system designed from for NVMM and RDMA networks. Traditional distributed file systems are designed for slower hard drives. These slower media incentivizes complex optimizations (e.g., queuing, striping, and batching) around disk accesses. Orion combines file system functions and network operations into a single layer. It provides low latency metadata accesses and outperforms existing distributed file systems by a large margin.

Finally, an NVMM application can map files backed by an NVMM file system into its address space, and accesses them using CPU instructions. In this case, RDMA and NVMM file systems introduce duplication of effort on permissions, naming, and address translation. We introduce two changes to the existing RDMA protocol: the file memory region (FileMR) and range based address translation. By eliminating redundant translations, FileMR minimizes the number of translations done at the NIC, reducing the load on the NIC's translation cache and resulting in application performance improvement by 1.8× - 2.0×.

Chapter 1

Introduction

Over the past decades, the increasing capacity of DRAM and the prevalence of NAND flash-based solid state drive (SSDs) revolutionized the design of data center applications. The emerging non-volatile memories (NVMs) such as spin-transfer torque, phase change, resistive memories [31, 55, 82], and Intel and Micro's 3D XPoint [69] technology will offer a storage class that further dramatic increases the storage system performance. These new types of memory provide performance characteristics comparable to DRAM and vastly faster than either hard drives or SSDs. They blur the line between storage and memory, forcing system designers to rethink how volatile and non-volatile data interact and how to manage non-volatile memories as reliable storage.

The appearance of these non-volatile memories on the processor's memory bus will offer an abrupt and dramatic increase in storage system performance. These non-volatile main memories (NVMM) can communicate directly with a processor's memory controller and offer the opportunity to build low-latency storage systems by providing "byte-addressable" storage that survives power outages.

Taking full advantage of NVMMs' potential will require changes in system software [7]. The need for such changes is especially acute in large-scale data center environments where storage systems require more than simple non-volatility. These environments demand reliability, availability and scalability.

When accessing NVMM remotely, media access performance is no longer the primary determiner of performance. Instead, network performance, software overhead, and data placement all play central roles. Consequently, the traditional ways of accessing remote storage squander NVMM performance — the previously negligible inefficiencies quickly become the dominant source of delay.

This thesis focuses on using Remote Direct Memory Access (RDMA) to build systems that enable low latency remote NVMM accesses by reducing software and networking overhead. RDMA is a networking protocol that allows a node to perform one-sided read/write operations from/to memory on a remote node on pre-registered memory regions. RDMA has become popular in building memory-intensive applications [3,49,50,51,70,75,90,94,103,104] as it offloads most of the networking stack onto hardware and provides close-to-hardware abstractions, exhibiting much better latency compared to heavier protocols like TCP/IP.

This thesis first studies the case of using RDMA to provide reliability and availability to NVMM. Without this reliability and availability, NVMM will only be suitable as a transient data store or as a caching layer—it will not be able to serve as a reliable primary storage medium.

Traditionally, data center environments use replication [14,26,34,101,105] or erasure coding schemes [40,41] to recover from hardware, software, and network failures. These approaches rest on the assumption that storage is slow, so the cost of the network and software protocols required to implement replications is acceptable.

NVMM will change this situation completely, since the networking and software overhead of existing replication mechanisms will squander the low latency that NVMM can provide. The interface with NVMM is also different from traditional storage: applications access NVMM directly with fine-grained memory operations.

We propose *Mojim*, a system that provides replicated, reliable, and highly-available NVMM as an operating system service. Applications can access data in Mojim using normal load and store instructions while controlling when and how updates propagate to replicas using system calls. Mojim allows applications to build data persistence abstractions ranging from simple log-based

systems to complex transactions.

This thesis next studies the case of using NVMM and RDMA to build distributed file systems. Scalable, enterprise NVMM-based systems will demand distributed access to NVMMs.

File systems are a convenient way to access NVMM because it provides naming and isolation while allows user applications use both traditional I/O interface or memory map to access file data. Several NVMM-based file systems have been proposed [21, 27, 29, 111, 112].

For distributed file system on NVMM, the issue of software overhead also applies: disk-based, distributed storage systems provide scalable access to distributed and replicated data, but their large software and network overheads obscure the benefits that NVMM should offer. As such, most distributed file systems have used two-layer designs that divide the network and storage layers into separate modules. Two-layer designs trade efficiency for ease of implementation. Designers can build a user-level daemon that stitches together off-the-shelf networking packages and a local file system into a distributed file system. While expedient, this approach results in duplicated metadata, excessive copying, unnecessary event handling, and places userspace protection barriers on the critical path.

Additionally, the conventional balance between network and storage speed has wider implications as well. If storage is inherently slow, accessing it remotely is not a significant cost, especially on data center networking with high bandwidth. So, running a distributed file system on a dedicated set of nodes rather than co-locating the file system with applications does not hurt the performance. Additionally, techniques such as queuing and striping can effectively improve the overall throughput of a distributed file system. However, they do not provide substantial benefits when applying on NVMM.

We present *Orion*, a distributed file system designed from the ground up for NVMM and RDMA networks. While other distributed systems [65, 90] have integrated NVMMs, Orion is the first distributed file system to systematically optimize for NVMMs throughout its design.

Orion merges the network and storage functions into a single, kernel-resident layer optimized for RDMA and NVMM that handles data, metadata, and network access. This decision allows Orion

to explore new mechanisms to simplify operations and scale performance.

Consequently, the location of stored data is a crucial performance concern for Orion. This concern is an important difference between Orion and traditional block-based designs that generally distinguish between client nodes and a pool of centralized storage nodes [24, 89]. Pooling makes sense for block devices, since access latency is determined by storage, rather than network latency, and a pool of storage nodes simplifies system administration. However, the speed of NVMMs makes a storage pool inefficient, so Orion optimizes for locality. To encourage local accesses, Orion migrates durable data to the client whenever possible and uses a novel delegated allocation scheme to manage free space efficiently.

Finally, This thesis studies the conflicting metadata management between NVMM and RDMA, which causes expensive translation overhead for userspace remote NVMM accesses.

Local NVMM applications can map a file into their address space then access it using normal loads and stores, drastically reducing the latency for access to persistent data. Ideally, we could combine NVMM and RDMA into a unified network-attached persistent memory to perform remote NVMM access without trapping into the operating system. Unfortunately, NVM file systems and the RDMA network protocol were not designed to work together. As a result, there are many redundancies, particularly where the systems overlap in memory. Only RDMA provides network data transfer and only the NVMM file system provides persistent memory metadata, but both systems implement protection, address translation, naming, and allocation across different abstractions: for RDMA, memory regions, and for NVMM file systems, files. Naively using RDMA and NVMM file systems together results in a duplication of effort and inefficient translation layers between their abstractions. These translation layers are expensive, especially since RNICs can only store translations for limited amount of memory while NVM capacity can be extremely large.

Both Mojim and Orion circumvent this issue by letting the operating system perform networking requests. This thesis introduces a new abstraction to the RDMA protocol, called a *file memory region* (FileMR). FileMR combines the best of both RDMA and NVM file systems into a design that can provide fast, network-attached, file-system managed, persistent memory. It

accomplishes this goal by offloading most RDMA-required tasks related to memory management to the NVM file system through the new memory region type; the file system effectively becomes RDMA's control plane.

With the FileMR abstraction, a client establishes an RDMA connection backed by *files*, instead of memory address ranges (*i.e.* an RDMA memory region). RDMA reads and writes are directed to the file through the file system, and addressed by the file offset. The translation between file offset and the physical memory address is routed through the NVMM file system, which stores all its files in persistent memory. Access to the file is mediated via traditional file system protections (e.g. access control lists). To further optimize address translation, we integrate a *range-based translation* system, which uses address ranges (instead of pages) for translation, into the RNIC, reducing the space needed for translation and resolving the abstraction mismatch between RDMA and NVMM file systems.

The rest of the thesis is organized as follows: In Chapter 2, we survey the technological opportunities and challenges that motivate the research efforts in this thesis. We use Intel's Optane DC persistent memory as a case study to understand real capabilities, limitations, and characteristics of NVMMs.

In Chapter 3, we present the design, implementation and evaluation of Mojim. Mojim provides reliable and highly-available NVMM by using a two-tier architecture and efficiently replicates data in NVMM.

Our evaluation shows that, surprisingly, Mojim reduces the average latency of the un-replicated system by 27% to 63%, even when it provides strongly consistent copies of data. Mojim's performance gain is mainly due to inefficiencies in the current instruction sets the un-replicated system uses to enforce data persistence. Mojim provides 0.4 to 2.7$\times$ the throughput of the un-replicated system. We also run several popular applications including a file system [29], the Google Hash Table [36], and MongoDB [72] on Mojim. The MongoDB results are the most striking: Mojim is 3.4 to 4$\times$ faster than the MongoDB replication mechanism and 35 to 741$\times$ faster than

un-replicated MongoDB.

In Chapter 4, we show the design, implementation and evaluation of Orion, a file system for distributed NVMM and RDMA networks. By combining file system functions and network operations into a single layer, Orion provides low latency metadata accesses and allows clients to access their local NVMMs directly while accepting remote accesses. The design of Orion reduces the file system access latency significantly and provides file system properties similar to a local NVMM file system, including atomicity of file system operations on data and metadata.

Our evaluation shows that Orion outperforms existing distributed file systems by a large margin. Relative to local NVMM filesystems, it provides comparable application-level performance when running applications on a single client. For parallel workloads, Orion shows good scalability: performance on an 8-client cluster is between 4.1× and 7.9× higher than running on a single node.

In Chapter 5, we introduce and evaluate two modifications to the existing RDMA protocol: the FileMR and range-based translation, thereby providing an abstraction that combines memory regions and files. It drastically improves the performance of RDMA-accessible NVMMs by eliminating extraneous translations, while conferring other benefits to RDMA including more efficient access permissions and simpler connection management. These extensions minimize the amount of translation done at the NIC, reducing the load on the NIC's translation cache and improving hit rate by 3.8× - 340× and resulting in application performance improvement by 1.8× - 2.0×.

Acknowledgments

This chapter contains material from"Mojim: A Reliable and Highly-Available Non-Volatile Memory System", by Yiying Zhang, Jian Yang, Amirsaman Memaripour and Steven Swanson, which appears in the Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2015). The dissertation author is the second investigator and author of this paper.

This chapter contain material from "Orion: A Distributed File System for Non-Volatile Main Memories and RDMA-Capable Networks", by Jian Yang, Joseph Izraelevitz and Steven Swanson, which appears in the Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST 2019). The dissertation author was the primary investigator and first author of this paper. The material in these chapters is copyright

This chapter contains material from "FileMR: Rethinking RDMA Networking for Scalable Persistent Memory", by Jian Yang, Joseph Izraelevitz and Steven Swanson, which is submitted for publication. The dissertation author was the primary investigator and first author of this paper.

Chapter 2

Motivation and Background

Long-term trends in the performance of data center networking combined with new, extremely fast storage devices are driving dramatic changes in how large-scale systems store, manage, and access persistent data. The unique characteristics of NVMMs give system designers flexibility to use them as memory, as storage, or as both [91], which poses new challenges for system designers and architects.

Previous research on NVMM has focused on how to use these memories in a single machine [7, 20, 21, 29, 71, 106, 110], while most mission-critical data resides in distributed, replicated storage systems (e.g., in data centers). For NVMM to succeed as a first-class storage technology, it must provide the reliability, availability and scalability that these storage systems require [47], while preserving the unique characteristics of the NVMM, such as its DRAM-alike access latency.

This section gives a comprehensive introduction to the background and motivation of this thesis. Section 2.1 describes the characteristics of NVMM. Section 2.2 gives an empirical study of Intel's 3D XPoint DIMM, which is one of the first commercially available NVMM. Section 2.3 introduces the RDMA networking protocol, and Section 2.4 introduces the unique challenges of building distributed NVMM applications.

2.1 Non-volatile Main Memory

NVMM is comprised of non-volatile DIMMs (NVDIMMs) attached to a CPU's memory bus alongside traditional DRAM DIMMs. Battery-backed NVDIMM-N modules are commercially available from multiple vendors [88], and Intel's 3DXPoint memory [69] debuted in early 2019. Other technologies such as spin-torque transfer RAM (STT-RAM) [55], ReRAM [31] are in active research and development.

Attaching next-generation NVMs to the main memory bus provides a raw storage medium that is orders of magnitude faster than modern disks and SSDs. NVMMs appear as contiguous, persistent ranges of physical memory addresses [88] and allow direct access via a load/store interface.

Researchers and companies have developed several file systems designed specifically for NVMM [21, 27, 29, 111, 112]. Other developers have adapted existing file systems to NVMM by adding DAX support [17, 108].

Unlike traditional file systems built for slower block devices, NVMM-aware file systems play a critical role in providing efficient NVMM access — the DRAM-comparable latency of NVMM means software overhead can dominate performance. Instead of using block-based interface, file systems can issue load and store instructions to NVMMs directly. NVMM-aware file systems [17, 108, 111, 112] adapt this ability in two ways:

First, NVMM file systems provide this ability via direct access (or "DAX"), which allows read and write system calls to bypass the page cache.

Second, they support the direct access mmap (*DAX-mmap*) capability. DAX-mmap allows applications to map NVMM files directly into their address spaces and to perform data accesses via simple loads and stores. This scheme allows applications to bypass the kernel and file system for most data accesses, drastically improving performance for file access.

Additionally, the file system also must account for the 8-byte atomicity guarantees that NVMMs provide (compared to sector atomicity for disks). They also must take care to ensure

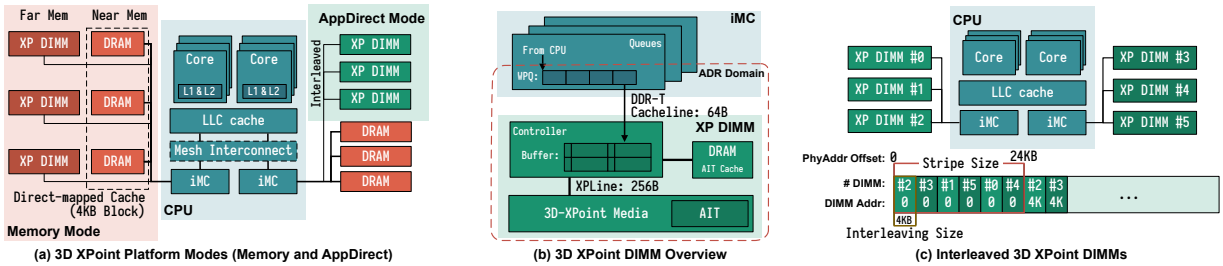

Figure 2.1: **Overview of (a) 3D XPoint platform, (b) 3D XPoint DIMM and (c) how 3D XPoint memories interleave across channels.** 3D XPoint DIMM can work as volatile far memory with DRAM as cache or persistent memory for application accesses.

crash consistency by carefully ordering updates to NVMMs using cache flush and memory barrier instructions.

2.2 Case Study: 3D XPoint DIMM

Intel's Optane DC Persistent Memory Module (which we refer to as 3D XPoint DIMM) is the first scalable, commercially available NVMM. Compared to existing storage devices (including the Optane SSDs) that connect to an external interface such as PCIe, the 3D XPoint DIMM has lower latency, higher read bandwidth, and presents a memory address-based interface instead of a block-based NVMe interface. Compared to DRAM, it has higher density and persistence.

With its arrival, we can start to understand real capabilities, limitations, and characteristics of these memories and starting designing and refining systems to fully leverage them.

2.2.1 Intel's 3D XPoint DIMM

Like traditional DRAM DIMMs, the 3D XPoint DIMM sits on the memory bus, and connects to the processor's integrated memory controller (iMC) (Figure 2.1(a)). Intel's Cascade Lake processors are the first (and only) CPUs to support 3D XPoint DIMM. On this platform, each processor contains one or two processor dies which comprise separate NUMA nodes. Each processor die has two iMCs, and each iMC supports three channels. Therefore, in total, a processor

die can support a total of six 3D XPoint DIMMs across its two iMCs.

To ensure persistence, the iMC sits within the *asynchronous DRAM refresh (ADR)* domain — Intel's ADR feature ensures that CPU stores that reach the ADR domain will survive a power failure (i.e., will be flushed to the NVDIMM within the hold-up time, $< 100\ \mu$s) [87]. The iMC maintains read and write pending queues (RPQs and WPQs) for each of the 3D XPoint DIMMs (Figure 2.1(b)), and the ADR domain includes WPQs. Once data reaches the WPQs, the ADR ensures that the iMC will flush the updates to 3D XPoint media on power failure. The ADR domain does not include the processor caches, so stores are only persistent once they reach the WPQs.

The iMC communicates with the 3D XPoint DIMM using the DDR-T interface in cache-line (64-byte) granularity. This interface shares a mechanical and electrical interface with DDR4 but uses a different protocol that allows for asynchronous command and data timing since 3D XPoint memory access latencies are not deterministic.

Memory accesses to the NVDIMM (Figure 2.1(b)) arrive first at the on-DIMM controller (referred as *XPController* in this thesis), which coordinates access to the 3D XPoint media. Similar to SSDs, the 3D XPoint DIMM performs an internal address translation for wear-leveling and bad-block management, and maintains an *address indirection table* (AIT) for this translation [10].

After address translation, the actual access to storage media occurs. As the 3D XPoint physical media access granularity is 256 bytes (referred as *XPLine* in this thesis), the XPController will translate smaller requests into larger 256-byte accesses, causing write amplification as small stores become read-modify-write operations. The XPController has a small write-combining buffer (referred as *XPBuffer* in this thesis), to merge adjacent writes.

3D XPoint DIMMs can operate in two modes (Figure 2.1(a)): Memory and App Direct.

Memory mode uses 3D XPoint to expand main memory capacity without persistence. It combines a 3D XPoint DIMM with a conventional DRAM DIMM on the same memory channel that serves as a direct-mapped cache for the NVDIMM. The cache block size is 4 KB, and the CPU's memory controller manages the cache transparently. The CPU and operating system simply see the 3D XPoint DIMM as a larger (volatile) portion of main memory.

App Direct mode provides persistence and does not use a DRAM cache. The 3D XPoint DIMM appears as a separate, persistent memory device. The system can install a file system or other management layer on the device to provide allocation, naming, and access to persistent data. 3D XPoint-aware applications and file systems can access the 3D XPoint DIMMs with load and store instructions.

In App Direct mode, programmers can directly modify the 3D XPoint DIMM's contents using store instructions, and those stores will, eventually, become persistent. The cache hierarchy, however, can reorder stores, making recovery after a crash challenging [20, 46, 64, 78, 106].

Intel processors offer programmers a number of options to control store ordering. The instruction set provides `clflush` and `clflushopt` instructions to flush cache lines back to memory, and `clwb` can write back (but not evict) cache lines. Alternately, software can use non-temporal stores (`ntstore`) to bypass the cache hierarchy and write directly to memory. All these instructions are non-blocking, so the program must issue an `sfence` to ensure that a previous cache flush, cache write back, or non-temporal store is complete and persistent.

In both modes, 3D XPoint memory can be (optionally) interleaved across channels and DIMMs (Figure 2.1(c)). On our platform, the only supported interleaving size is 4 KB, which ensures that accesses to a single page fall into a single DIMM. With six DIMMs, an access larger than 24 KB will access all the DIMMs.

2.2.2 Typical Latency

Read and write latencies are key memory technology parameters. We measure read latency by timing the average latency for individual 8-byte load instructions to sequential and random memory addresses. To eliminate caching and queuing effects, we empty the CPU pipeline and issue a memory fence (`mfence`) between measurements (`mfence` serves the purpose of serialization for reading timestamps). For writes, we load the cache line into the cache and then measure the latency of one of two instruction sequences: a 64-bit store, a `clwb`, and an `mfence`; or a non-temporal store

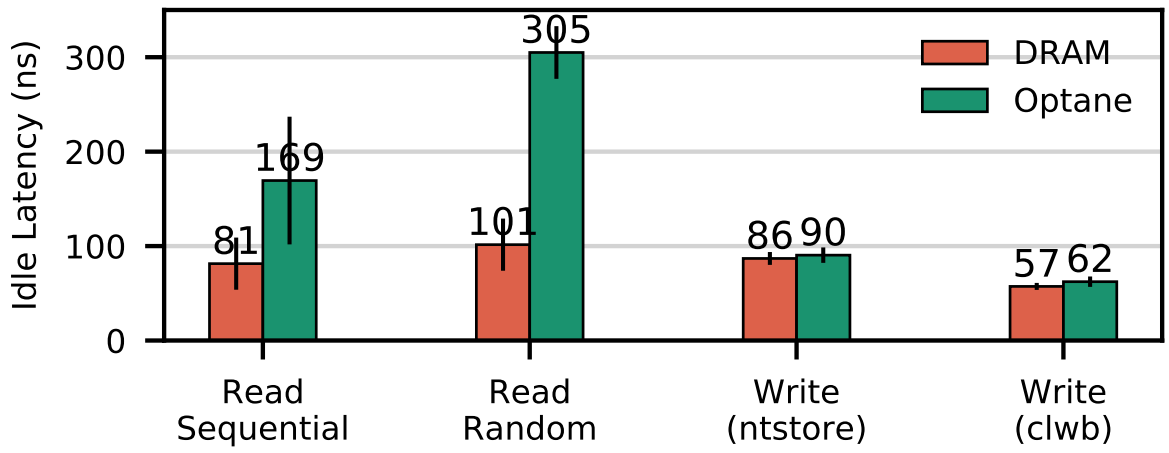

Figure 2.2: **Best-case latency.** An experiment showing random and sequential read latency, as well as write latency using cached write with `clwb` and `ntstore` instructions. Error bars show one standard deviation.

followed by an `mfence`.

These measurements reflect the load and store latency as seen by software rather than those of these underlying memory device. For loads, the latency includes the delay from the on-chip interconnect, iMC, XPController and the actual 3D XPoint media. Our results (Figure 2.2) show the read latency for 3D XPoint is 2×–3× higher than DRAM. We believe most of this difference is due to 3D XPoint having a longer media latency. 3D XPoint memory is also more pattern-dependent than DRAM. The random-vs-sequential gap is 20% for DRAM but 80% for 3D XPoint memory, and we believe this gap is a consequence of the XPBuffer. For stores, the memory store and fence instructions commit once the data reaches the ADR at the iMC. Both DRAM and 3D XPoint memory show a similar latency. Non-temporal stores are more expensive than writes with cache flushes (`clwb`).

In general, the latency variance for 3D XPoint is extremely small, save for an extremely small number of "outliers", which we investigate in the next section. The sequential read latencies for 3D XPoint DIMMs have higher variances, as the first cache line access loads the entire XPLine into XPBuffer, and the following three accesses read data in the buffer.

2.2.3 Bandwidth

Detailed bandwidth measurements are useful to application designers as they provide insight into how a memory technology will impact overall system throughput. We measure 3D XPoint and DRAM bandwidth for random and sequential reads and writes under different levels of concurrency.

Figure 2.3 shows the bandwidth achieved at different thread counts for sequential accesses with 256 B access granularity. We show loads and stores (`Write(ntstore)`), as well as cached writes with flushes (`Write(clwb)`). All experiments use AVX-512 instructions and access the data at 64 B granularity. The left-most graph plots performance for interleaved DRAM accesses, while the center and right-most graphs plot performance for non-interleaved and interleaved 3D XPoint. In the non-interleaved measurements all the accesses go to a single DIMM.

The data shows that DRAM bandwidth is both significantly higher than 3D XPoint and scales predictably (and monotonically) with thread count until it saturates the DRAM's bandwidth and that bandwidth is mostly independent of access size.

The results for 3D XPoint are wildly different. First, for a single DIMM, the maximal read bandwidth is 2.9× of the maximal write bandwidth (6.6 GB/s and 2.3 GB/s respectively), where DRAM has a smaller gap (1.3×) between read and write bandwidth.

Second, with the exception of interleaved reads, 3D XPoint performance is non-monotonic with increasing thread count. For the non-interleaved (i.e., single-DIMM) cases, performance peaks at between one and four threads and then tails off. Interleaving pushes the peak to twelve threads for `store+clwb`.

Third, 3D XPoint bandwidth for random accesses under 256 B is poor. This "knee" corresponds to XPLine size. DRAM bandwidth does not exhibit a similar "knee" at 8 kB (the typical DRAM page size), because the cost of opening a page of DRAM is much lower than accessing a new page of 3D XPoint.

Interleaving (which spreads accesses across all six DIMMs) adds further complexity: Figure 2.3(right) and Figure 2.4(right) measure bandwidth across six interleaved NVDIMMs. Inter-

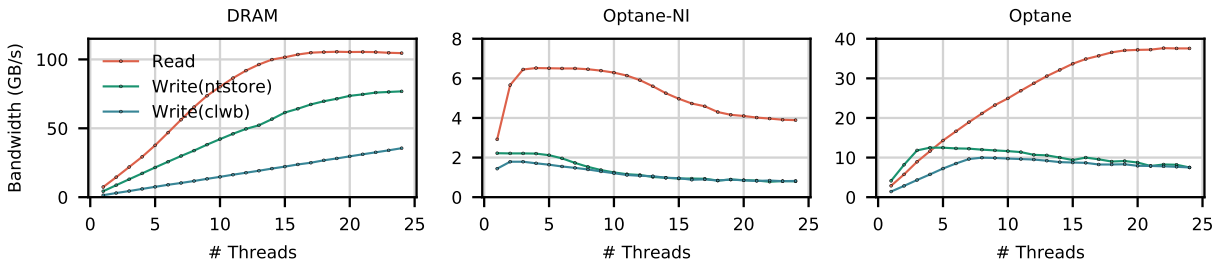

Figure 2.3: **Bandwidth vs. thread count.** An experiment showing maximal bandwidth as thread count increases (from left to right) on local DRAM, non-interleaved and interleaved 3D XPoint memory. All threads use a 256 B access size. (Note the difference in vertical scales).

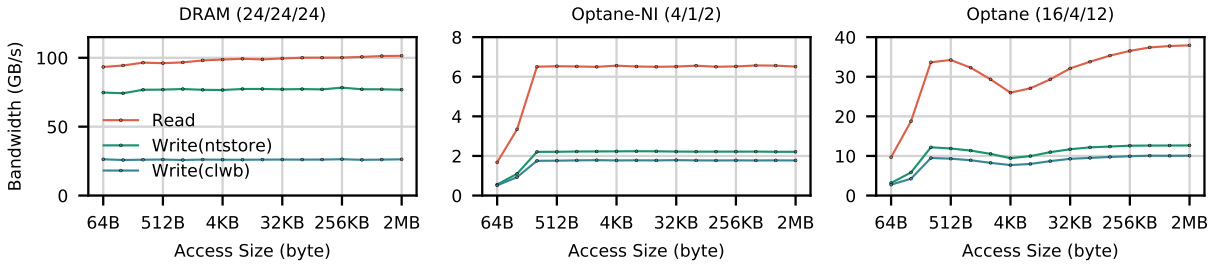

Figure 2.4: **Bandwidth over access size.** An experiment showing maximal bandwidth over different access sizes on (from left to right) local DRAM, interleaved and non-interleaved 3D XPoint memory. Graph titles include the number of threads used in each experiment (`Read/Write(ntstore)/Write(clwb)`).

leaving improves peak read and write bandwidth by 5.8× and 5.6×, respectively. These speedups match the number of DIMMs in the system and highlight the per-DIMM bandwidth limitations of 3D XPoint. The most striking feature of the graph is a dip in performance at 4 KB — this dip is an emergent effect caused by contention at the iMC, and it is maximized when threads perform random accesses close to the interleaving size.

2.2.4 NUMA Effects

NUMA effects for 3D XPoint are much larger than they are for DRAM. The cost is especially steep for accesses that mix load and stores and include multiple threads. Between local and remote 3D XPoint memory, the typical read latency difference is 1.79× (sequential) and 1.20× (random), respectively. For writes, remote 3D XPoint memory's latency is 2.53× (ntstore) and 1.68× higher

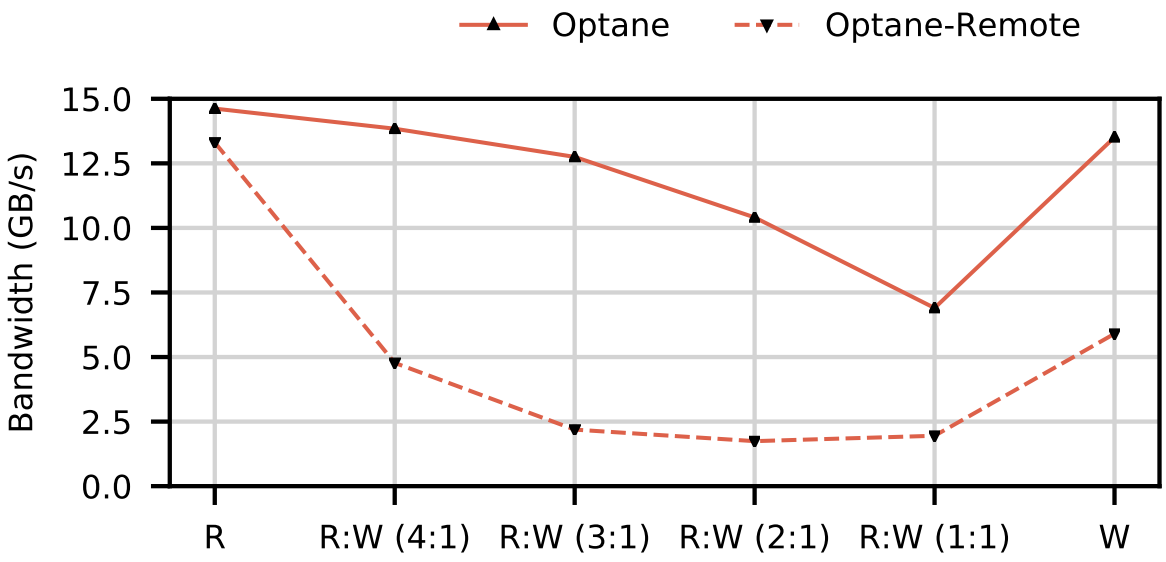

Figure 2.5: **Memory bandwidth on local (Optane) and remote (Optane-Remote) NUMA nodes.** This chart shows memory bandwidth as we vary the mix of accesses for both one and four threads. Pure reads or pure writes perform better on NUMA than mixed workloads and increased thread count generally hurts NUMA performance.

compared to local. For bandwidth, remote 3D XPoint can achieve 59.2% and 61.7% of local read and write bandwidth at optimal thread count (16 for local read, 10 for remote read and 4 for local and remote write).

The performance degradation ratio above is similar to remote DRAM to local DRAM. However, the bandwidth of 3D XPoint memory is drastically degraded when either the thread count increases or the workload is read/write mixed. Based on the results from our systematic sweep, the bandwidth gap between local and remote 3D XPoint memory for the same workload can be over 30×, while the gap between local and remote DRAM is, at max, only 3.3×.

In Figure 2.5, we show how the bandwidth changes for 3D XPoint on both local and remote CPUs by adjusting the read and write ratio. We show the performance of a single thread and four threads, as local 3D XPoint memory performance increases with thread count up to four threads for all the access patterns tested.

Single-threaded bandwidth is similar for local and remote accesses. For multi-threaded

accesses, remote performance drops off much more quickly as store intensity rises, leading much lower performance relative to the local case.

2.2.5 Lessons Learned

As the first commercially available NVMM, Optane DIMMs can achieve DRAM-level performance on characteristics such as access latency and read bandwidth. At the same time, it adds complexity to the storage architecture, causing issues such as scalability and heavier penalties over NUMA.

These observations have implications to remote NVMM accesses as well: Since networking traffic is handled by DMA engines on the host and the NIC, access remote NVMMs does not exhibit issues such as the NUMA penalties and reverse scaling with thread contention. On the other hand, achieving low-latency fine-grained accesses will become a crucial requirement for systems handling remote NVMMs.

2.3 RDMA Networking

RDMA has become a popular networking protocol, especially for distributed applications [3, 49,50,51,70,75,90,94,103,104]. Current RDMA protocols are based on virtual interface architecture (VIA) [30], a user-level memory-mapped communication architecture designed in two decades ago.

RDMA exposes a machine's memory to direct access from the RDMA network interface (RNIC), allowing remote clients to directly access a machine's memory without involving the local CPU.

The RDMA hardware supports a set of operations (called *verbs*). *One-sided* verbs, for instance, "read" and "write", directly access remote memory without requiring anything of the remote CPU, in fact, these verb bypasses the remote CPU entirely. *Two-sided* verbs, in contrast, require both machines to post matching requests, for instance, "send" and "receive", which transfer data from the memory of the sender, from an address chosen by the sender, to the memory of the

receiver, to an address chosen by the receiver.

To establish an RDMA connection, an application registers one or more *memory regions* (*MRs*) that grant the local RNIC access to part of the local address space. The MR functions as both a name and a security domain: To give a client access to a region the local RNIC supplies the MR's virtual address, size and a special 32-bit "rkey". Rkeys are sent with any one-sided verb and allow the receiving RNIC to verify the client has direct access to the region. For two-sided verbs, a send/recv operation requires both the sender and receiver to post matching requests, each attached to some local, pre-registered, memory region, negating the need for rkeys.

2.3.1 Idiosyncrasies of RDMA

As a fast transport layer, RDMA offloads must of its networking stack onto hardware, and certain access patterns of the verbs will have an impact on its performance:

Registration is expensive: RDMA allows RNICs to translate virtual addresses into physical (DMA) addresses for the incoming packets. The cost of registering an MR consists of populating and pinning page table entries and creating a copy of the page table in an RNIC accessible structure, such as a flat table. For NVMM, the registration cost is linear to the size of the NVMM that accepts remote accesses. Both Mojim and Orion use physical addresses in verbs, avoiding this cost.

Inbound verbs are cheaper: Inbound verbs, including recv and incoming one-sided read-/write, incur lower overhead for the target, so a single node can handle many more inbound requests than it can initiate itself [95]. Orion's mechanisms for accessing data and synchronizing metadata across clients both exploit this asymmetry to improve scalability.

RDMA favors short transfers: RNICs implement most of the RDMA protocol in hardware. Compared to transfer protocols like TCP/IP, transfer size is more important to transfer latency for RDMA because sending smaller packets involves fewer PCIe transactions [51]. Also, modern RDMA hardware can inline small messages along with WQE headers, further reducing latency. To exploit these characteristics, Mojim and Orion aggressively minimize the size of the transfers it

makes.

2.3.2 RDMA Persistence

For local NVMM, a store instruction is persistent once data is evicted from CPU last-level cache (via cache flush instructions and memory fences). A mechanism called asynchronous DRAM refresh (ADR) ensures that the write queue on a memory controller is flushed to non-volatile storage in the event of a power failure. There are no similar mechanisms in the RDMA world since ADR does not extend to PCIe devices. Making the task even more difficult, modern NICs are capable of placing data into CPU cache using direct cache access (DCA) [42], conceivably entirely bypassing NVMM.

The current workaround to ensure RDMA write persistency is to disable DCA and issue another RDMA read to the last byte of a pending write [28], forcing the write to complete and write to NVMM. Alternatively, the sender request that the receiving CPU purposefully flush data it received; either embedding the flush request in an extra send verb, or the immediate field of a write verb.

A draft standards working document has proposed adding a `commit` [97] verb to the RDMA protocol to solve the write persistency problem. A `commit` verb lists memory locations that need to be flushed to persistence. When the remote RNIC receives a commit verb, it ensures the all listed locations are persistent before acknowledging completion of the verb.

2.4 Challenges of Accessing Remote NVMM

There are several challenges present in building distributed systems that combines NVMM and RDMA. In this section, we describe these challenges that motivate us building systems in this thesis.

2.4.1 NVMM Availability, Reliability, and Consistency

Although NVMM protects against power failure by making the contents of memory persistent, it does not address the other ways that systems fail, including software, hardware, and networking errors that are common in data centers [32, 73]. Providing availability and reliability in such environments is important to meet client SLAs [101] and application requirements. Strong consistency is also desirable in storage systems, since it makes it easier to reason about system correctness.

Adding redundancy or replication is a common technique for providing reliability and availability [2, 14, 26, 34, 37, 40, 77, 79, 84, 85, 105, 114]. Various protocols exist to provide different consistency levels among redundant copies of data [4, 13, 26, 43, 58, 79]. For traditional storage systems with slow hard disks and SSDs, the performance overhead of replication is small relative to the cost of accessing a hard drive or SSD, even with complex protocols for strong consistency. With NVMMs, however, the networking round trips and software overhead involved in these techniques [13, 43, 58, 105] threaten to outstrip the low-latency benefit of using NVMMs in the first place. Even for systems with weak consistency [26, 79], increasing the rate of reconciliation between inconsistent copies of data can threaten performance [37, 52].

Since NVMM is vastly faster than existing storage technologies, it presents new challenges to data replication. First, NVMM-based systems must deliver high performance to justify their increased cost relative to disks or SSDs. Existing replication mechanisms built for these slower storage media have software and networking performance overhead that would obscure the performance benefits that NVMM could provide.

Second, NVMM is memory, and applications should be able to use it like memory (*i.e.*, via load and store instructions without operating system overheads for most accesses) rather than as a storage device (*i.e.*, via I/O system calls).

2.4.2 Issues with Userspace RDMA Accesses

Both Mojim and Orion provide an interface that is processed in operating system services. Userspace RDMA accesses and NVMM mmapped-DAX accesses share a critical functionality: they allow direct access to memory without involving the kernel. Broadly speaking, we can divide both NVMM file systems and RDMA into a data plane that accesses the memory a control plane that manages the memory exposed to user applications. The data plane is effectively the same for both: it consists of direct loads and stores to memory. The control plane, in contrast, differs drastically between the systems.

For both RDMA and NVMM file systems, the control plane must provide four services for memory management. First, it must provide naming to ensure that the application can find the appropriate region of memory to directly access. Secondly, it must provide access control, to prevent an application from accessing data it should not. Thirdly, it must provide a mechanism to allocate and free resources to expand or shrink the memory available to the application. Finally, it must perform translation between application level names (i.e., virtual addresses, or offsets from a base address or within a file) to physical memory addresses. In practice, this final requirement means that both RDMA and NVMM file systems must work closely with the virtual memory subsystem.

Table 2.1 summarizes the control plane metadata operations for RDMA and NVMM. These memory management functionalities are attached to different abstractions in RDMA and NVMM file systems. For RDMA, we use memory regions and for NVMM file systems we use files.

Naming: Names provide a hardware-independent way to refer to physical memory locations. In RDMA applications, the virtual address of a memory region, along with its "host" machine's location (e.g., IP address or GID) serves as a globally (i.e., across nodes) meaningful name for regions of physical memory. These names are transient, since they become invalid when the application that created them exits, and inflexible since they prevent an RDMA-exposed page from changing its virtual to physical address mapping while accessible. To share a name with a client that wishes to directly access it via reads and writes, the host gives it the metadata of the MR. For

two-sided verbs (i.e., send/receive) naming is ad-hoc: the receiver must use an out-of-band channel to decide where to place the received data.

NVMM-based file systems use filenames to name regions of physical memory on a host. Since files outlive applications, the file system manages names independent of applications and provides more sophisticated management for named memory regions (i.e., hierarchical directories and text-based names). To access a file, clients and applications on the host request access from the file system.

Permissions: Permissions determine what processes have access to what memory. In RDMA, permissions are enforced in two ways. To grant a client direct read/write access to a memory location, the host shares a memory region specific "rkey." The rkey is a 32-bit key that is attached to all one-sided verbs (such as read and write) and is verified by the RNIC to ensure the client has access to the addressed memory region. For every registered region, the RNIC driver maintains the rkey, along with other RDMA metadata that provides isolation and protection in hardware-accessible structures in DRAM.

Permissions are established when the RDMA connection between nodes is created and are granted by the application code establishing the connection. They do not outlive the process or survive a system restart. For two-sided verbs protection is enforced by the receiving application in an ad-hoc manner: The receiver uses an out-of-band channel to decide what permissions the sender has.

Access control for NVMM uses the traditional file system design. Permissions are attached to each file and designated for both individual users and groups. Unlike RDMA memory regions and their rkeys, permissions are a property of the underlying data and survive both process and system restart.

Allocation: RDMA verbs and NVMM files both directly access memory, so allocation and expansion of available memory is an important metadata operation.

For NVMM file systems, the file system maintains a list of free physical pages that can be used to create or extend files. Creation of a file involves martialing the appropriate resources

Table 2.1: **Control plane roles for RDMA and NVMM.** This table shows the features provided by RDMA and NVMM vs. FileMR.

Role	RDMA + File System	FileMR
Naming	Both (Redundant)	FS Managed
Permissions	Both (Redundant)	FS Managed
Allocation	Both (Redundant)	FS Managed
Appending	Not Allowed	FS Managed
Remapping	Not Allowed	FS Managed
Defragmentation	Not Allowed	FS Managed
Translation	Both (Incompatible)	FS Managed
Persistence	FS Only	FS Managed
Networking	RDMA	RDMA
CPU-Bypass	RDMA	RDMA

and linking the new pages into the existing file hierarchy. Similarly, free pages can be linked to or detached from existing files to grow or shrink the file. Changing the size of DAX-mmap'd files is easy as well with calls to `fallocate` and `mremap`.

Creating a new RDMA memory region consists of allocating the required memory resources, pinning their pages, and generating the rkey. Note that the physical address of the memory region out of the programmer's control (it depends, instead, on the implementation of `malloc`), and the page is pinned once the region is registered, leading to a fragmented physical address space.

In addition, changing the mapping of a memory region is expensive. For example, to increase the memory region size, the host server needs to deregister the memory region, reregister a larger region, and send the changes to any interested clients. The `rereg_mr` verb combines deregistration and registration but still carries significant overhead. Alternatively, programmers can add another memory region to the connection or protection domain, but, as memory regions require non-negligible metadata and RDMA does not support multi-region accesses, this solution adds significant complexity.

This fixed size limitation also prohibits common file system operations and optimizations, such as appending to a file, remapping file content, and defragmentation.

Address Translation: RDMA and NVMM file system address translation mechanisms ensure that their direct accesses hit the correct physical page.

As shown in Figure 2.6, RDMA solves the problem of address translation by *pinning* the virtual to physical address, that is, for as long as a memory region is registered, its virtual and physical addresses cannot change. Once this mapping is fixed, the RNIC is capable of handling memory regions registered on virtual address ranges directly: the RNIC translates from virtual addresses to physical addresses for incoming RDMA verbs. To do this translation, the NIC maintains a *memory translation table* (MTT) that holds parts of the system page tables.

The MTT flattens the translation entries for the relevant RDMA accessible pages and can be cached in the RNIC's pin-down cache [102] to accelerate lookups of this mapping. The pin-down cache is critical for getting good performance out of RDMA — the pin-down cache is small (a few megabytes), a miss is expensive, and due to its addressing mechanism, the pin-down cache requires all pages in a region be the same size. As a consequence of these limitations, researchers have done significant work trying to make the most of the cache for addressing large memories [38, 51, 74, 75, 90, 95, 104]. While complex solutions exist, the most common recommendation is to reduce the number of translations needed (e.g., addressing large contiguous memory regions using with either huge pages, or using physical addresses).

The NVMM file system handles address translation in two ways, both different from RDMA. For regular reads and writes, the file system translates file names with offsets to physical addresses; this translation is done in the kernel during the syscall. For memory mapped accesses, `mmap` establishes a virtual to physical address mapping from user space directly to the file's contents in NVMM, loading the mapping into the page table. The file system only interferes on the page fault handling when a translation is missing between the user and physical addresses (i.e. a soft page fault); the file system is bypassed on normal data accesses.

The different translation schemes interfere with each other to create performance problems. If a page is accessible via RDMA, it is pinned to a particular physical address, and furthermore, every page within the region must be the same size. As a consequence, the file system is unable to update the layout of the open file, e.g. to defragment or grow the file. As RDMA impedes defragmentation of files and prohibits mixing page sizes in RDMA accessible memory, memory

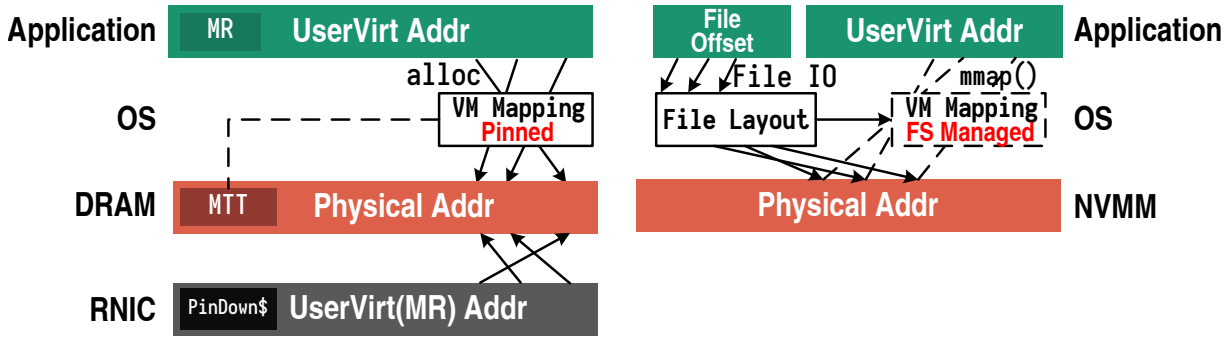

Figure 2.6: **Address translation for RDMA and NVMM..** RDMA (left) uses NIC-side address translation with pinning, while NVMM (right) allows the file system to maintain the layout of a file mapped to user address space.

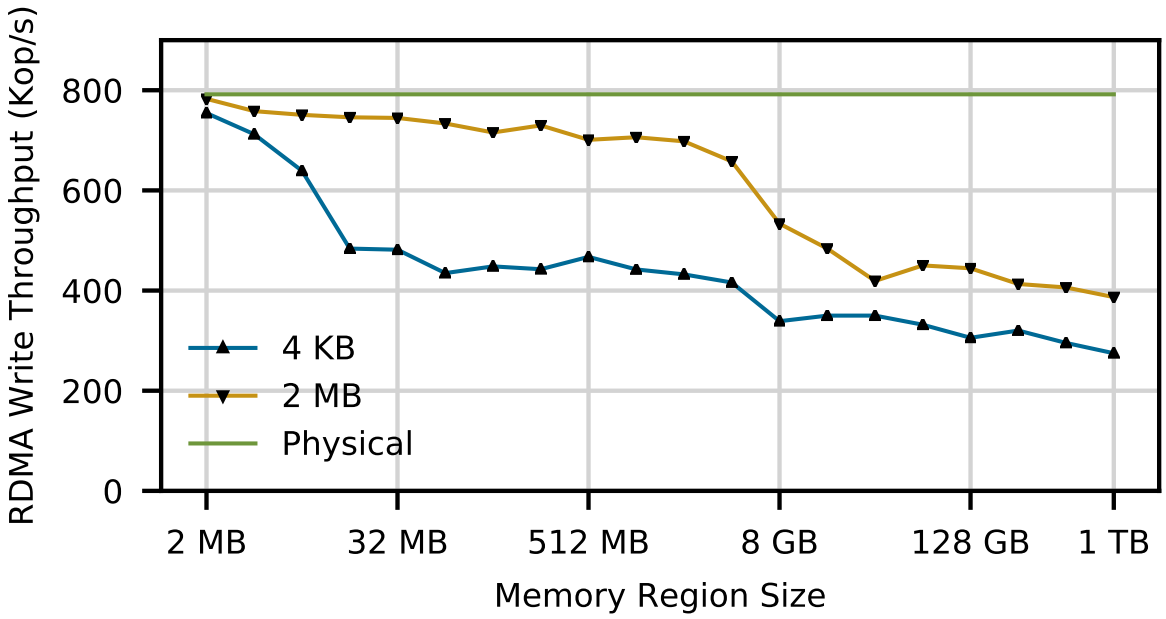

Figure 2.7: **RDMA Write performance over different memory region sizes..** This figure shows the throughput of 8-byte RDMA writes affected by the pin-down cache misses. Data measured on Intel Optane DC Persistent Memory with an Mellanox CX-3 RNIC.

regions backed by files must use many, small pages to address large regions, overwhelming the pin-down cache and decimating RDMA performance.

Figure 2.7 shows the impact of pin-down cache misses on RDMA write throughput. Each work request writes 8 bytes to a random 8-byte aligned offset. When the memory region size is

16 MB, Using 4 kB achieves 61.1% of the baseline (sending physical addresses, no TLB or pin-down cache misses) performance compared to 95.2% when using 2 MB hugepages. When the region size hits 16 GB, even 2 MB pages is not sufficient — achieving only 61.2% performance.

Acknowledgments

This chapter contains material from“Mojim: A Reliable and Highly-Available Non-Volatile Memory System”, by Yiying Zhang, Jian Yang, Amirsaman Memaripour and Steven Swanson, which appears in the Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2015). The dissertation author is the second investigator and author of this paper. The material in these chapters is copyright ©2015 by Association for Computing Machinery.

This chapter contain material from “Orion: A Distributed File System for Non-Volatile Main Memories and RDMA-Capable Networks”, by Jian Yang, Joseph Izraelevitz and Steven Swanson, which appears in the Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST 2019). The dissertation author was the primary investigator and first author of this paper. The material in these chapters is copyright ©2019 by the USENIX Association.

This chapter contains material from “FileMR: Rethinking RDMA Networking for Scalable Persistent Memory”, by Jian Yang, Joseph Izraelevitz and Steven Swanson, which is submitted for publication. The dissertation author was the primary investigator and first author of this paper.

This chapter contains material from “An Empirical Guide to the Behavior and Use of Scalable Persistent Memory,” by Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz and Steven Swanson, which is submitted for publication. The dissertation author was the primary investigator and first author of this paper.

Chapter 3

Mojim: A Reliable and Highly-Available NVMM System

Fast, non-volatile memory technologies such as phase change memory (PCM), spin-transfer torque magnetic memories (STTMs), and the memristor are poised to radically alter the performance landscape for storage systems. They will blur the line between storage and memory, forcing designers to rethink how volatile and non-volatile data interact and how to manage non-volatile memories as reliable storage.

Data center environments demand reliability and availability in the face of hardware, software, and network failures. Without this reliability and availability, NVMM will only be suitable as a transient data store or as a caching layer—it will not be able to serve as a reliable primary storage medium.

We propose *Mojim*, a system that provides replicated, reliable, and highly-available NVMM as an operating system service. Applications can access data in Mojim using normal load and store instructions while controlling when and how updates propagate to replicas using system calls. Mojim allows applications to build data persistence abstractions ranging from simple log-based systems to complex transactions.

Mojim uses a two-tier architecture that allows flexibility in choosing different levels of

reliability, availability, consistency, and monetary cost, while minimizing performance overhead. The primary tier includes one *primary node* and one *mirror node*. Mojim can, depending on the configuration, keep these nodes strongly or weakly consistent. An optional secondary tier provides an additional level of redundancy with one or more *backup node*s that are weakly consistent with the primary tier.

Mojim efficiently replicates fine-grained data from the primary node to the mirror node using an optimized RDMA-based protocol that is simpler than existing replication protocols. The mirror node replicates data to the backup nodes in the background, thus keeping the secondary tier off the performance-critical path. This design offers good performance and two strongly consistent copies of data plus more copies of weakly consistent data. To further improve availability and reliability, Mojim also provides a fast recovery process and atomic semantics that guarantee data integrity.

In building Mojim, we explore the performance and monetary cost impacts of providing availability, reliability, and consistency with NVMM, and we explore trade-offs among replication protocols for NVMM. Interestingly, we find that adding availability, reliability, and consistency does not necessarily impair NVMM performance, as long as the replication protocols and software layers are optimized for NVMM.

We evaluate Mojim using raw DRAM as a proxy for future NVMMs and with an industrial NVMM emulation system. Our evaluation shows that, surprisingly, Mojim reduces the average latency of the un-replicated system by 27% to 63%, even when it provides strongly consistent copies of data. Mojim's performance gain is mainly due to inefficiencies in the current instruction sets the un-replicated system uses to enforce data persistence. Mojim provides 0.4 to 2.7$\times$ the throughput of the un-replicated system. We also run several popular applications including a file system [29], the Google Hash Table [36], and MongoDB [72] on Mojim. The MongoDB results are the most striking: Mojim is 3.4 to 4$\times$ faster than the MongoDB replication mechanism and 35 to 741$\times$ faster than un-replicated MongoDB.

The rest of the chapter is organized as follows. We present Mojim and its implementation in

```
int fd = open("/mnt/mmapfile", O_CREAT|O_RDWR);
// open a file in mounted Mojim region
void *base = mmap(NULL, 40960, PROT_WRITE,
                  MAP_SHARED, fd, 0);
// mmap a 40KB area in the file

unsigned long *access_count_p = base;
// access count of the log
unsigned long *log_size_p = base + sizeof(unsigned long);  // size of the log
int *log = base + 2*sizeof(unsigned long);                 // the log

*access_count_p = *access_count_p + 1;
// memory load and store
msync(access_count_p, sizeof(unsigned long), MS_SYNC);
// call conventional msync

int beautiful_num = 24;
unsigned long curr_log_pos = *log_size_p;
// memory load and store
log[curr_log_pos] = beautiful_num;
*log_size_p = *log_size_p + 1;
struct msync_input { void *address; int length; };
struct msync_input input[2];
input[0].address = &(log[curr_log_pos]);
input[0].length = sizeof(int);
input[1].address = log_size_p;
input[1].length = sizeof(unsigned long);
gmsync(input, 2, MS_MOJIM);
// call gmsync to commit the log append
```

Figure 3.1: **Sample code to use Mojim.** Code snippet that implements a simple log append operation with Mojim.

Sections 3.1 and 3.2. Section 3.3 describes our experience adapting applications to use Mojim. We then present the evaluation results of Mojim in Section 3.4. Finally, conclude in Section 3.5.

3.1 Mojim Design Overview

Mojim provides an easy-to-use, generic layer of replicated NVMM that ensures reliability, availability, and consistency, while sacrificing as little of NVMM's performance as possible. Mojim uses a two-tier architecture and supports several operating modes to let applications tune Mojim's reliability, availability, and consistency to match their particular needs.

This section discusses Mojim's interfaces and architecture and the different modes Mojim provides.

3.1.1 Mojim's Interfaces

Mojim is an operating system service that provides reliable and highly-available NVMM. This section describes Mojim's typical usage scenario and the interface it provides.

To use Mojim, a system configuration file specifies a set of *Mojim regions* on the *primary node* to be replicated, along with a *mirror node* and a list of *backup nodes* where the replicas should reside. The primary node supports reads and writes to the replicated data. The mirror node and backup nodes support reads only. Kernel modules can access these regions and use them to build complex, replicated, memory-based services such as a kernel-level persistent key value store, a persistent disk cache, or a file system. The kernel could also make these services available to applications via a `malloc()`-like interface.

While Mojim can serve as the basis for many memory-based services, deploying an NVMM-aware file system to manage the replicated NVMM region would provide the most flexibility in application usage models. The file system would provide familiar file-system-based mechanisms of allocation and naming as well as conventional file-based access for non-performance-critical applications. The key requirement of the file system is that, for an `mmap()`'d file, it maps the the NVMM pages corresponding to the file directly into the applications' address spaces rather than paging them in and out of the kernel's buffer cache. In our experiments, we use PMFS [29] for this purpose.

With a file system in place, applications can create files in the Mojim-backed file system and map them into their address space using `mmap()`. We call the NVMM area mapped by applications the *data area*. After an `mmap()`, applications can perform direct memory accesses to the data area using load and store instructions on the primary node and load instructions on the mirror node.

Mojim provides a mechanism called a *sync point* that allows applications to control when

and what updates in the data area propagate to the replicas. At each sync point, Mojim atomically replicates all memory regions specified by an application.

Two APIs allow applications to create sync points: *msync* and *gmsync*.

Mojim leverages the existing *msync* system call to specify a sync point that applies to a single, contiguous address range. The semantics of Mojim's *msync* correspond to conventional *msync*, and applications that use *msync* will work correctly without modification under Mojim. Mojim allows an application to specify a fine-grained memory region in the *msync* API and replicates it atomically, while traditional *msync* flushes page-aligned memory regions to persistent storage and does not provide atomicity guarantees [76].

Mojim's *gmsync* adds the ability to specify multiple memory regions for the sync point to replicate, allowing for more flexibility than *msync*.

Mojim provides a mechanism to allow applications to make their data persistent atomically, but it does not provide primitives for synchronization. It would be possible to add synchronization primitives to Mojim, but this would increase the complexity of the system and require selecting a set of synchronization mechanisms to support. A better approach would be to build synchronization mechanisms that leverage Mojim's mechanisms.

Figure 3.1 shows a simple example in C of how to use Mojim to manage an append-only log on Mojim. The program first opens and `mmap()`s a file in a Mojim region. It then updates the access count of the log and makes this value persistent with the conventional *msync* API. Next, it appends a log entry and updates the size of the log. It makes both these data persistent with an *gmsync* call. The atomicity that *gmsync* provides guarantees that the log size is consistent with the log content on the replica nodes.

3.1.2 Architecture

Mojim uses a two-tier architecture. The primary tier contains a primary node and its read-only mirror node; the secondary tier includes one or more backup nodes with weakly consistent,

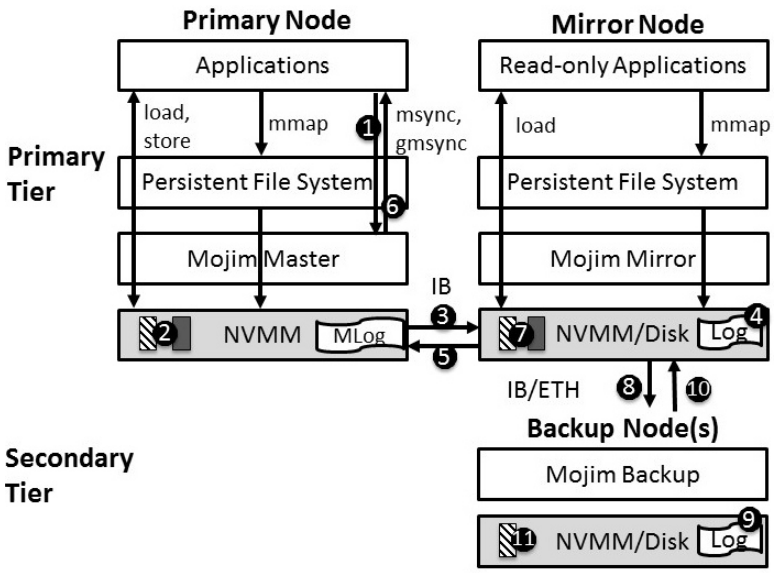

Figure 3.2: **Mojim architecture.** The numbered circles represent different steps in the Mojim replication process. MLog stands for the metadata log.

read-only copies of data. Figure 3.2 depicts the architecture of Mojim.

Mojim's primary tier contains a pair of mirroring nodes: a primary node replicates data to its mirror node at each *sync point* (i.e., call to *msync* or *gmsync*). The application can read and write data on the primary node, but Mojim only allows reads from the mirror node.

The primary tier offers good performance even when guaranteeing strong consistency, since it requires only one networking round trip for each sync point. Existing architectures that allow writes to all replicas (*E-writeall*) [58], or that use one primary and multiple secondary nodes (*E-chain and E-broadcast*) [4, 105], require multiple networking round trips or other performance overhead to guarantee strong consistency.

To further improve performance, we connect the primary node and the mirror node with a high-speed Infiniband link and use an efficient software and networking layer to replicate data between them.

To improve reliability, we place the primary node and the mirror node on different racks, since failure bursts often happen within the same rack [32, 73].

The optional secondary tier includes one or more backup nodes to maintain additional

Table 3.1: **Replication schemes.** Mojim supports a wide range of reliability, availability, consistency, and monetary cost levels (columns 2-5). The reliability column represents the number of node failures that can be tolerated in a system of N nodes. The last three rows compare Mojim to other existing replication schemes.

Scheme	R	A	C	$
S-unreplicated	0	Worst	N/A	Low
M-async	1	Good	Weak	Fair
M-sync	1	Good	Strong	Fair
M-syncdisk	1	OK	Strong	Low
M-syncsec	$N-1$	Best	Strong+Weak	High
M-syncseceth	$N-1$	Good	Strong+Weak	Fair
E-writeall	$N-1$	Best	Strong	High
E-chain	$N-1$	Best	Strong	High
E-broadcast	$N-1$	Best	Strong	High

copies of data. It provides additional reliability and availability, so that failure bursts will not be catastrophic. The mirror node replicates data to the backup nodes in the background. Thus, data in the backup nodes is not strongly consistent with data in the primary tier. By keeping the replication to the secondary tier in the background and off the performance-critical path, Mojim ensures good application performance.

With both tiers in operation and a total of N nodes, Mojim can tolerate $N-1$ node failures. In most environments, one or a few backup nodes are enough to prevent data loss, since failure bursts are more likely to involve only a small number of nodes [32, 73]. Also, in most failure bursts, the nodes do not all fail at the same time; failures are usually separated by a few seconds. A fast recovery can thus prevent data loss even with few copies of replicated data. We discuss recovery optimizations in Section 3.2.3.

3.1.3 Mojim Modes and Replication Protocols

Mojim supports several replication modes and protocols that allow users to choose different levels of performance, reliability, consistency, availability, and monetary cost depending on application requirements.

Table 3.1 summarizes these different modes and their properties, and we discuss them below

using the numbered circles in Figure 3.2 to illustrate the replication process in each mode.

Across all the modes Mojim provides, Mojim achieves most of its performance by adopting a different architecture than most replicated storage systems. Instead of supporting multiple consistent replicas, Mojim only supports strong consistency at a single mirror node. This decision makes our replication protocols much simpler (e.g., there's no need for multi-phase commit or a complex consensus protocol) and, therefore, allows for much higher performance.

Mojim achieves the goal of providing its atomic data persistence interface by ensuring that atomic operations are replicated atomically to the mirror node and the backup nodes, by appending replicated data to logs on the mirror node and the backup nodes.

Un-replicated without Mojim: A single machine without Mojim (*S-unreplicated*) must flush an *msync*'d memory region from the processor's caches to ensure data persistence. S-unreplicated has poor availability and is only as reliable as the NVM devices. Moreover, even if the NVMM is recovered after a crash, data can still be corrupted. For example, if a crash occurs after a pointer is made persistent but before the data it points to becomes persistent, the system will contain corrupted data.

Sync: Mojim's *M-sync* mode guarantees strong consistency between the primary and the mirror node. It provides improved reliability and availability over S-unreplicated, since in the case of a failure the mirror node can take the place of the primary node without losing data.

In M-sync, when an application calls *msync* or *gmsync* (① in Figure 3.2), Mojim pushes data from the primary node to the mirror node via RDMA (③) and writes the data in the mirror node log (④). The primary node waits for the acknowledgment from the mirror node (⑤), and then returns the *msync* or *gmsync* call (⑥). The mirror node later takes a checkpoint to apply the log contents to the data area (⑦). Mojim stores both the mirror node logs and its data area in NVMM for high performance and fast recovery.

In M-sync, Mojim does not flush data from the primary node's caches (②). Modern RDMA devices are cache-coherent, so they will send the most up-to-date data [45,57]. Thus, the mirror node always gets the latest data and pushing data to the mirror node is sufficient to ensure persistence. If

the primary node crashes, the mirror node has the most up-to-date data. If the mirror node crashes, the primary node has all the data, but it may not be persistent, so the primary node immediately flushes its caches to prevent data loss. This means there is a small "window of vulnerability" after a mirror node failure during which a primary node failure could result in data loss. On our system, this window lasts for 450μs, the time required to flush the processor caches.

Surprisingly, our evaluation results show that M-sync offers performance comparable to or better than S-unreplicated because flushing CPU caches is often more expensive than pushing the data over RDMA. The current *clflush* instruction is strongly ordered and cannot utilize the parallelism offered in modern processor architecture. Intel recently announced two instructions that are more efficient than *clflush* and that will be available on future systems [44], which should help resolve this problem.

Sync with cache flush: To close the window of vulnerability mentioned above, Mojim can flush data from the primary node's caches (②) before returning to applications' *msync* or *gmsync* calls (⑥). This mode is called *M-syncflush*, and with M-syncflush, all data can survive simultaneous failures of the primary node and the mirror node.

Async: *M-async* provides weaker consistency between the primary node and the mirror node. M-async ensures that data is persistent on the primary node for each sync point (②) and pushes the data to the mirror node (③), but it does not wait for the mirror node's acknowledgment (⑤) to complete the application's *msync* or *gmsync* call (⑥). Thus, data on the mirror node can be out of date relative to the primary node. M-async must flush the primary node CPU caches at each sync point to ensure that the latest data is persistent.

Sync with slow storage: To reduce the monetary cost of M-sync, Mojim supports a mode that stores the log on the mirror node in NVMM, but stores the mirror node's data area on a hard disk or SSD (*M-syncdisk*). M-syncdisk has a slower recovery process than M-sync, since Mojim needs to read data from hard disk or SSD to NVMM before applications can access them.

Sync with the secondary tier: *M-syncsec* adds the secondary tier to M-sync and increases reliability and availability by adding more copies of data. Mojim replicates data from the mirror

node to the backup node in the background (⑧-⑪). M-syncsec provides two strongly-consistent copies of the data at the primary and mirror nodes and one weakly-consistent data copy at each backup node. The amount of inconsistency between the mirror node and backup nodes is tunable and affects the recovery time. Even though the data at each backup node may be out-of-date, it still represents a consistent snapshot of application data because of the atomic semantics Mojim provides. Our evaluation results show that M-syncsec delivers performance similar to M-sync because replication to the backup nodes takes place in the background.

Sync with low-cost secondary tier: M-syncsec requires fast networks between the mirror node and backup nodes, which increases the monetary cost and networking bandwidth consumption of the system. A lower cost option, *M-syncseceth*, uses Ethernet between the mirror node and backup nodes. M-syncseceth has the worst performance of all the Mojim modes, but it still provides the same reliability, availability, and consistency guarantees as M-syncsec.

3.2 Mojim Implementation

This section describes our implementation of Mojim in the Linux kernel. The core of Mojim comprises an optimized network stack and the replication and recovery code.

3.2.1 Networking

The networking delay of data replication is the most important factor in determining Mojim's overall performance. Mojim uses Infiniband (IB), a high-performance switched network that supports RDMA. RDMA is crucial because it allows the primary node to transfer data directly into the mirror node's NVMM without requiring additional buffering, copying, or cache flushes.

Mojim uses *IB-Verbs*, a set of native IB APIs based on send, receive, and completion queues [67]. IB-Verbs requires the application to post send (receive) requests to send (receive) queues. It uses completion messages in the completion queue to indicate the completion of requests and supports both polling and interrupts to detect completions. IB-Verbs offers native IB performance

and outperforms alternative IB protocols such as IPoIB and RDS (see Section 3.4.2). Existing IB-Verbs implementations are userspace libraries that bypass the kernel. We created a kernel version of IB-Verbs for Mojim.

Mojim uses a thin protocol based on the reliable transportation mode of IB-Verbs. The Mojim protocol directly fetches data from the primary node and writes it to NVMM on the mirror node. For each sync point, the primary node posts a send request on the IB send queue and polls for its completion. The mirror node posts a set of receive requests in advance and polls for the arrival of incoming messages. Our measurements show that polling is more efficient than interrupts.

The protocol does not require explicit acknowledgment messages from the mirror node to the primary node, since we configure the IB link to provide a successful completion notification for the primary node's send request only once the data transfer succeeds. In the event of an error or a timeout, the primary node resends the message to the backup node. After a set number of unsuccessful re-send attempts, Mojim invokes its recovery process.

To sustain high bandwidth, Mojim creates multiple IB connections to handle client requests. For each connection, we assign one thread on the mirror node to poll for incoming messages. On the primary node, we let the application thread perform IB send operations for M-sync and use a background thread to post these operations for M-async.

3.2.2 Replication

We now describe the Mojim replication process and the techniques that we use to enable reliable, atomic, and consistent data replication.

Primary tier replication: At each sync point, the primary node posts IB send requests containing the target memory regions. Mojim ensures that all requests belonging to the same atomic operation are consecutive and on the same IB connection and marks the last request to let the mirror node know the end of an atomic operation. Since Mojim's reliable IB protocol ensures ordering in each IB connection, these requests will appear in the same order on the mirror node. A unique ID

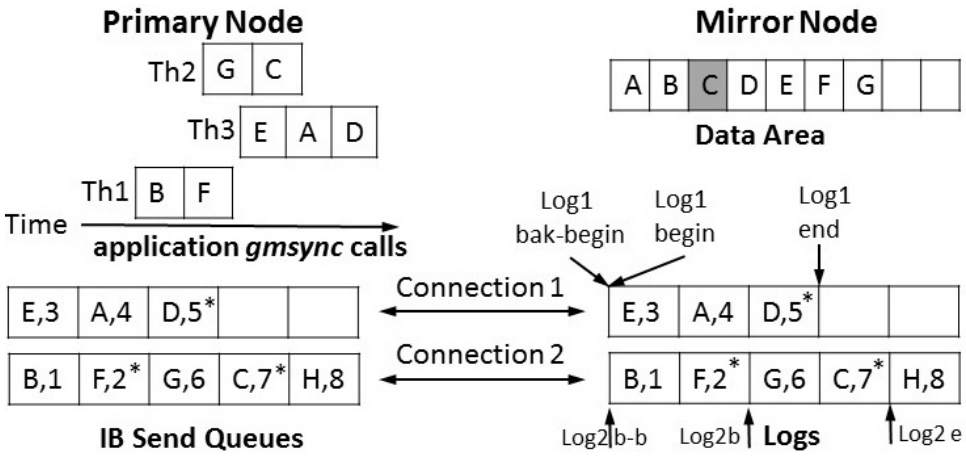

Figure 3.3: **Example of Mojim replication.** An example of Mojim's replication process. Each cell represents a request. The letter in the cell stands for the memory address and the number in the cell represents its unique ID. The upper-left part shows three threads placing three *gmsync* calls. The upper-right part shows the data area on the mirror node. The * represents the end mark of a *gmsync* operation. The gray cell in the mirror node data area represents the data that is recovered after a crash.

on each send request allows the mirror node to keep updates ordered across IB connections. For recovery purposes, the primary node stores the memory addresses of the most recent requests in a *metadata log*.

For each IB connection, the mirror node maintains a circular log and a thread that polls incoming requests. Mojim places the logs in NVMM for good performance and persistence and pre-allocates fixed-size buffers on the logs for RDMA accesses. With pre-allocated memory slots, Mojim only needs one IB roundtrip to replicate data from the primary node to the mirror node. Because the receive buffer size is fixed, we limit the size of each send request on the primary node and break original memory regions into multiple send requests if needed. Since RDMA writes directly to NVMM, there is no need to flush the cache on the mirror node.

After all the data for a sync point has arrived on the log, the mirror node can write them to their permanent locations in the data area. This checkpointing happens periodically after a configurable number of requests (*CHECKPOINT_THRESH*) have been received, as well as when

the system is idle and when a log is full. Mojim maintains global pointers to the beginning and end of each log to indicate data available for checkpointing.

To ensure that read-only applications on the mirror node see a consistent view of their data, Mojim removes the page table entries of the affected memory locations before a checkpointing operation. During the checkpointing, an application reading from those pages will generate a page fault. We changed the page fault handler to wait until Mojim completes the checkpointing and then restore the page table entries and return the application read.

Secondary tier replication: Replication to the secondary backups occurs in the background when there is data on the mirror node's logs. The protocol for replication to the backup node mimics the replication to the mirror node. The mirror node maintains a pointer for each log to indicate the amount of data that has not yet been replicated to the backup node. Mojim uses a threshold (*SECONDARY_TIER_THRESH*) to limit the amount of such un-replicated data on the mirror node and stalls further replication to the mirror node until un-replicated data drops below SECONDARY_TIER_THRESH.

Example: Figure 3.3 illustrates an example of Mojim's data structures and its replication process. In this example, Mojim uses two IB connections and two mirror node logs. Three application threads post three *gmsync* calls to the two IB send queues. To guarantee atomicity, Mojim serializes thread 2's requests after thread 1's requests on the second send queue. Mojim then sends these requests to the mirror node's logs. The mirror node threads poll for the completion of these writes and update the log-end pointers when they have received all requests belonging to one *gmsync* call. The checkpointing service processes the logs from the log-begin pointer to the log-end pointer. The mirror node replicates the log content between the log-bak-begin pointer and the log-end pointer to the backup node.

3.2.3 Recovery

Fast recovery is crucial to providing high availability and preventing data loss in the event of a failure. There are three types of failure scenarios: primary, mirror, and backup node failures. Mojim uses heartbeats to detect failures, but other techniques [19, 60] are possible.

When the primary node fails, the mirror node becomes the new primary node and a backup node becomes the new mirror node. The new primary node first sends the un-replicated data in its logs to the new mirror node and checkpoints its log content to its data area after the failure. For M-syncdisk, the new primary node needs to load data from the data file on disk to the NVMM. After these operations, applications can restart on the new primary node. Until these operations complete, the Mojim contents will be unavailable.

One option for activating a new backup node is to wait for the failed node to come back online. Rebooting the machine is often sufficient and more efficient than constructing a new node [32]. When the crashed primary node restarts, it receives the new data accumulated during its down time from the new primary node. When the failed node cannot reboot fast enough, a human operator or a system monitoring service selects a new backup node based on its available NVMM size, its networking topology, and other criteria [18, 92]. The new node receives a complete copy of the memory region and begins processing updates from the new mirror node.

When the mirror node or the backup node fails, the recovery process is similar. If the mirror node fails, the primary node first flushes its CPU caches. It also uses its metadata log to locate un-replicated data and sends them to the backup node. To restart the mirror node or the backup node, Mojim replays the logs and writes only the completed atomic operation content to the data area, with the help of the atomic operation end mark and unique IDs. In the example in Figure 3.3, the mirror node crashes after Mojim checkpoints *G*. The recovery process will checkpoint *C* and discard *H*. If the failed node cannot restart, a newly chosen node receives replicated data from the primary node as described above.

When both the primary node and the mirror node fail in quick succession, Mojim falls

back to the backup node. Now Mojim needs to reconstruct two new nodes that the administrating node selects. This recovery process is more costly than the recovery of a single node failure. We reduce the risk of this situation by placing nodes on different racks and by setting a small SECONDARY_TIER_THRESH, thus speeding up the recovery process of a single node failure.

3.3 Mojim Applications

We have ported several existing systems to Mojim to illustrate how applications can use Mojim's interface. The applications include the PMFS file system [29], the Google hash table [36], and MongoDB [72].

3.3.1 PMFS

The Persistent Memory File System [29] (PMFS) provides a conventional file-system-like interface to NVMM, allowing applications to allocate space with file creation, limit access to data via file permissions, and name portions of the NVMM using file names. The key difference between PMFS and a conventional file system is that its implementation of `mmap()` maps the physical pages of NVMM into the applications' address spaces rather than moving them back and forth between the file store and the buffer cache.

PMFS ensures persistence using *sfence* and *clflush* instructions. Mojim invokes its replication when PMFS performs its persistence procedure. Mojim's M-sync also removes *clflush* and only performs *sfence* on the primary node. Mojim's change required modifications to just 20 lines of PMFS source code. Applications can use `mmap()` to gain load/store access to a file's contents and then use *fsync*, *msync*, or *gmsync* to manage replication and data consistency.

3.3.2 Google hash table

Google hash table [36] is an open source implementation of sparse and dense hash tables. Our Mojim-enabled version of the hash table stores its data in `mmap()`'d PMFS files and performs *msync* at each insert and delete operation to let Mojim replicate the data. Porting the Google hash table to Mojim requires changes to just 18 lines of code.

3.3.3 MongoDB

MongoDB [72] is a popular NoSql database. Several aspects of MongoDB make it a good comparison point for Mojim. First, MongoDB stores its data in memory-mapped files and performs memory loads and stores for data access—a perfect match for Mojim's NVMM interface. Second, MongoDB supports both single node and replication in a set of nodes in several different modes (called "write concerns") that trade off among performance, reliability, and availability. Mojim provides similar functionality with a more general mechanism.

By default, MongoDB logs data in a journal file and checkpoints the data to the memory-mapped data file in a lazy fashion. With the *JOURNALED* write concern, MongoDB blocks a client call until the updated data is written to the journal file. With the *FSYNC_SAFE* write concern, MongoDB flushes all the dirty pages to the data file after each write operation and blocks the client call until this operation completes.

MongoDB supports data replication across a set of machines. A primary node in a MongoDB replica set serves all write requests and pushes *operation logs* to the secondary nodes. Secondary nodes can serve read requests but may return stale data. The MongoDB write concern *REPLICAS_SAFE* returns the client request after at least two secondary nodes have received the corresponding operation log. The REPLICAS_SAFE write concern does not wait for journal writes or checkpointing on the primary node.

Mojim offers another way to provide reliability and availability to MongoDB. With the help of Mojim's *gmsync* API and its reliability guarantees, we can remove journaling from MongoDB

and still achieve the same consistency level. To guarantee the same atomicity of client requests as available through MongoDB, we modify the storage engine of MongoDB to keep track of all writes to the data file and group the written memory regions belonging to the same client request into a *gmsync* call. In total, this change requires modifying 117 lines of MongoDB.

An alternative way of using Mojim is to run unmodified MongoDB on Mojim by configuring MongoDB to place both its data file and journal file in Mojim's `mmap()`'d data area. When MongoDB commits data to the journal or checkpoints the data to the data file, it performs an *msync* operation, which will trigger Mojim's data replication transparently.

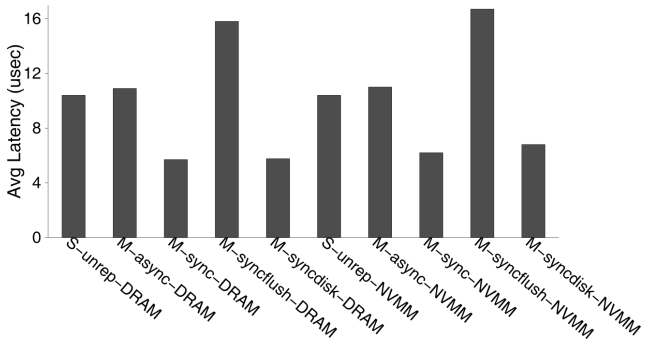

Figure 3.4: **msync latency with DRAM and NVMM.** The average 4 KB *msync* latency with PMEP's DRAM and NVMM modes.

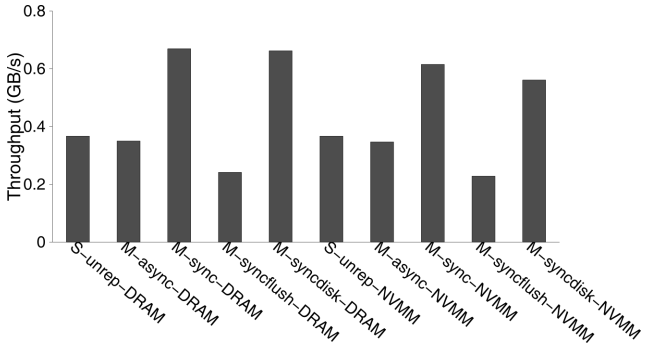

Figure 3.5: **msync throughput with DRAM and NVMM.** The 4 KB *msync* bandwidth with PMEP's DRAM and NVMM modes.

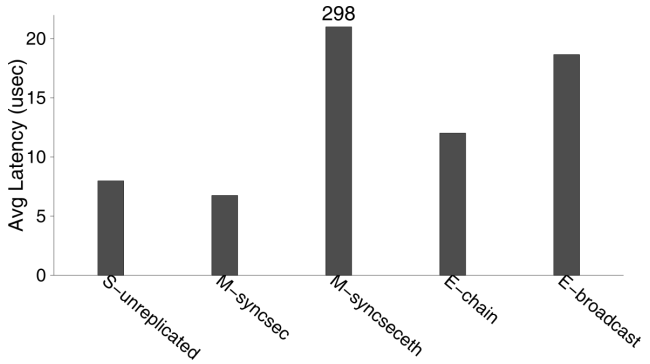

Figure 3.6: **msync latency with DRAM-based machines.** The average 4 KB *msync* latency with S-unreplicated and Mojim two-tier architecture.

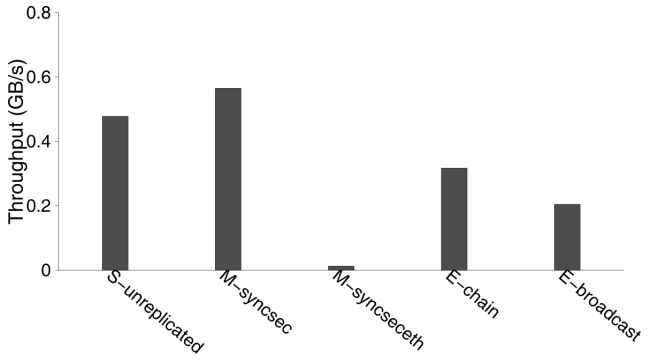

Figure 3.7: **msync throughput with DRAM-based machines.** The 4 KB *msync* throughput with S-unreplicated and Mojim two-tier architecture.

3.4 Evaluation with DRAM

In this section, we study the performance of Mojim under each of the configurations and applications we described in Sections 3.1 and 3.3. Specifically, we first evaluate the performance of different Mojim modes and compare them to existing replication methods. We then evaluate the effects of different application parameters and Mojim configurations, the performance of applications ported to Mojim, and Mojim's recovery costs.

3.4.1 Test Bed Systems

We use two different systems to evaluate Mojim. The first is an industrial NVMM emulation system from Intel called PMEP [29]. PMEP augments an off-the-shelf, dual-socket server platform with special CPU microcode and custom firmware. It partitions the system's DRAM into emulated NVMM and regular DRAM. PMEP emulates NVMM read latency, read and write bandwidth, and data persistence costs. For read latency and read/write bandwidth, PMEP modifies the CPU and the memory controller. The PMEP platform uses write-back CPU caches and does not emulate NVMM write latency. It uses software to emulate the cost of data persistence: the kernel running on PMEP issues *clflush* instructions followed by an *sfence*, and adds a *write barrier* delay to model the cost of ensuring data persistence in NVMM. In our experiments, we emulate NVMM by setting the read latency to 300 *ns*, read and write bandwidth to 5 GB/s and 1.6 GB/s (1/8 of DRAM bandwidth), and the write barrier delay to 1 *ms*, the configuration used in Intel's PMFS project [29].

Each PMEP node has two 2.6 GHz 8-core Intel Xeon processors, 40 MB of aggregate CPU cache, 8 GB of DDR3 DRAM used as normal DRAM, 128 GB of DRAM used as emulated NVMM, and a 7200 RPM 4 TB hard disk. They also have 40 Gbps Mellanox Infiniband NICs and are directly connected to each other via Infiniband without a switch. The platforms run Ubuntu 13.10 and the 3.11.0 Linux kernel.

We have access to only two PMEP machines (located at an Intel facility), so to evaluate Mojim modes that require more than two machines, we use similar machines in our lab that do not include PMEP functionality and use ordinary DRAM as a proxy for NVMM. Each of these machines has two Intel Xeon X5647 processors, 48 GB DRAM, one 40 Gbps Mellanox Infiniband NIC, and a 1000 Mbps Ethernet. A QLogic Infiniband Switch connects these machines' IB links. All machines run the CentOS 6.4 distribution and the 3.11.0 Linux kernel.

In all experiments, unless otherwise specified, we set CHECKPOINT_THRESH (the frequency of checkpointing the mirror node logs) to 1 (after each log write) and SECONDARY_TIER_THRESH (the threshold for sending un-replicated data to the backup nodes) to 40 MB.

3.4.2 Overall Replication Performance

We first compare the microbenchmark performance of Mojim modes that only involve two nodes using the PMEP platforms. To evaluate the impact of NVMM vs. DRAM, we run the same experiments with both PMEP's DRAM mode and its emulated NVMM.

Figures 3.4 and 3.5 present the average latency and throughput of *msync* calls with S-unreplicated, M-async, M-sync, M-syncflush, and M-syncdisk. For each experiment, we perform 10000 random 4 KB *msync* calls in a 4 GB `mmap()`'d file.

Surprisingly, M-sync outperforms S-unreplicated significantly for both DRAM and emulated NVMM (reducing latency by 45% and 40% respectively). Even though M-sync waits for a networking round trip between the primary node and the mirror node, it still outperforms S-unreplicated because it does not need to flush data from processors' caches, while S-unreplicated must flush data on each *msync*. M-async's performance is similar to S-unreplicated, as it also needs to flush primary node's caches. M-syncflush has higher latency than S-unreplicated, since it performs both cache flushes and networking round trips.

Placing the mirror node data on disk adds only 1% to 10% overhead. However, M-syncdisk does not support read applications on the mirror node and adds an overhead in recovery time (see Section 3.4.5).

Comparing DRAM and emulated NVMM, the performance with emulated NVMM for all schemes is close to that with DRAM, indicating that the performance degradation of NVMM over DRAM only has a very small effect over application-level performance.

Next, to augment the PMEP results with more machines and to test Mojim's two-tier architecture, we use three DRAM-based machines in our lab to evaluate the performance of Mojim's two-tier modes and two existing schemes that use a one-primary, multiple-secondary architecture (Table 3.1). One of these existing schemes, *E-chain*, allows writes only at the primary node and propagates data replication from the primary node to the secondary nodes in a serialized order [4, 105]. The other existing scheme, *E-broadcast* [34], is similar to E-chain but broadcasts

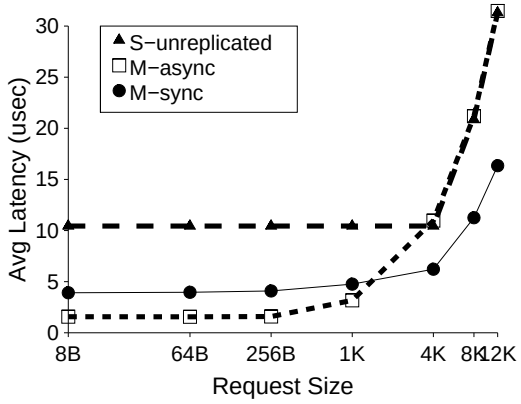

Figure 3.8: **Average *msync* latency with different *msync* sizes on emulated NVMM.** The average latency of *msync* operation on NVMM with request sizes from 8 bytes to 12 KB.

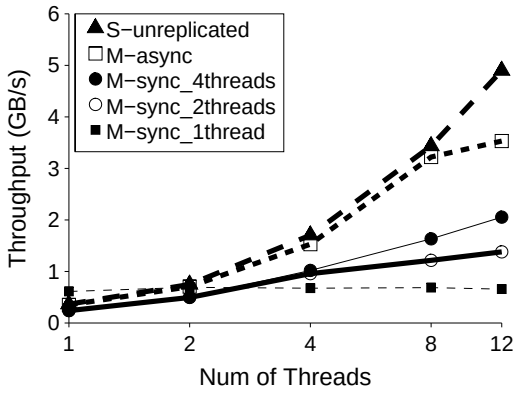

Figure 3.9: **Throughput with different application threads on emulated NVMM.** The *msync* throughput with 1 to 12 threads performing *msync*.

updates to the secondary nodes. E-chain and E-broadcast use one primary node and two secondary nodes interconnected by IB. They use the same IB protocol that we implemented for Mojim.

Figures 3.6 and 3.7 plot the average latency and throughput of using our lab machines to run the experiments shown in Figures 3.4 and 3.5. Compared to S-unreplicated, Mojim with the secondary tier does not degrade performance if a fast network connects the backup node. However, the lower-cost Ethernet configuration degrades performance by 37×, because the mirror node cannot drain its circular log fast enough and has to stall the primary tier replication.

Both E-chain and E-broadcast are slower than Mojim, increasing latency by 1.8× and 2.8× respectively, compared to M-syncsec.

Finally, we compare Mojim with two existing IB kernel protocols, RDS and IPoIB. We find that they both have worse performance than Mojim's networking protocol on IB-Verbs, with 4.9× and 31× slowdown.

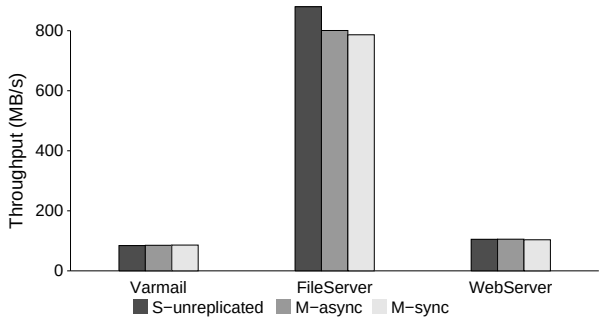

Figure 3.10: **Filebench throughput with emulated NVMM.** The throughput of three Filebench workloads with single machine and no replication, the M-async mode, and the M-sync mode.

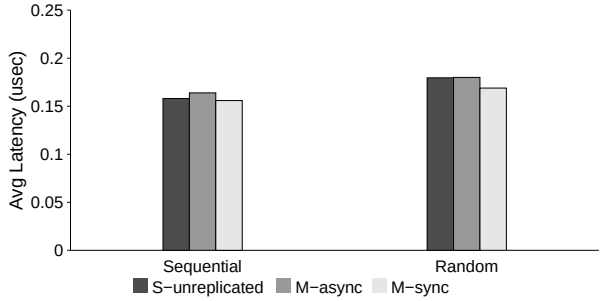

Figure 3.11: **Google hash table average latency with emulated NVMM.** The average latency of sequentially and randomly inserting key-value pairs to the Google dense hash table.

Overall, Mojim delivers performance similar to or better than no replication while adding reliability and availability. Mojim's good performance is due to its efficient replication protocol, its ability to avoid expensive cache flush operations, and its optimized software and networking stacks.

3.4.3 Sensitivity Analysis

Both Mojim's configuration parameters and application-level behavior can affect performance. In this section, we measure their impact on Mojim's performance.

msync Size

The amount of data per *msync* has a strong impact on performance. Figure 3.8 plots the average latency of performing *msync* calls to random memory regions of 8 bytes to 12 KB with S-unreplicated, M-async, and M-sync using PMEP's emulated NVMM.

For smaller request sizes, M-async performs much better than S-unreplicated. S-unreplicated underperforms because of the current way *msync* call are implemented in Linux, with the *msync* call handler checking for the range of the *msync* memory and rounding it to memory pages (4 KB page for the default Linux kernel). With Mojim, we modify the *msync* call handler to allow any memory address range and only flush and replicate the application-specified memory regions.

M-sync does not perform *clflush* (since transferring the data to the mirror node guarantees persistence). As a result, its performance is always better than S-unreplicated and is better than M-async when *msync* size is bigger than 1 KB.

Application Threads and Networking Connections

Application thread count and the number of network connections Mojim uses also impact performance. Figure 3.9 presents the 4 KB *msync* throughput with one to 12 application threads for S-unreplicated, M-async, and M-sync using PMEP's emulated NVMM.

Both M-async and S-unreplicated scale well with the number of application threads, while M-sync with one IB connection (and thus one log) scales poorly. With more connections, M-sync's scaling improves. A tradeoff with increasing networking connections is that Mojim uses more threads to poll for receiving messages, consuming more CPU cycles.

Checkpoint and Secondary Tier Replication Thresholds

We change the two thresholds Mojim uses in its configurations: we change CHECKPOINT_THRESH, the frequency of checkpointing the mirror node logs, from 1 to 10000, and we change SECONDARY_TIER_THRESH, the amount of un-replicated data to the backup node, from

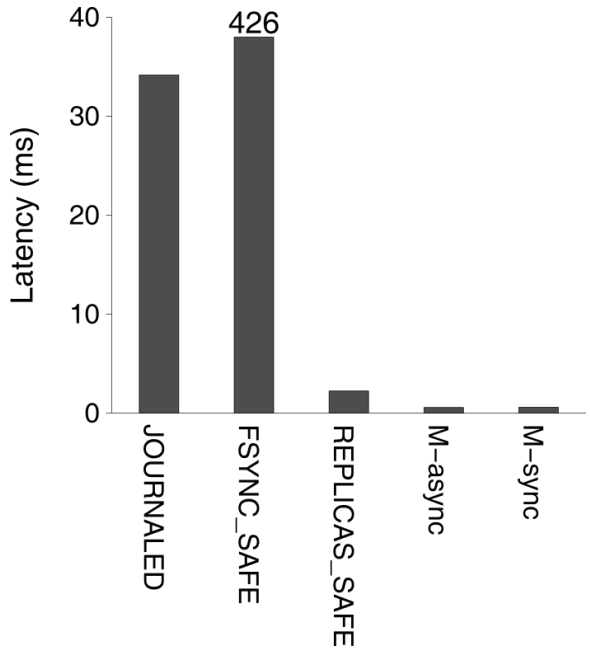

Figure 3.12: **YCSB insert average latency.** Average latency of inserting key-value pairs on emulated NVMM.

40 KB to 400 MB. We find that neither CHECKPOINT_THRESH nor SECONDARY_TIER_THRESH affects the application performance, because both the checkpointing process and the secondary tier replication via IB are fast enough not to block the primary tier replication.

3.4.4 Application Performance

In this section, we present the evaluation results for three applications: a file system, a hash table, and a NoSql database.

PMFS

We use the FileServer, WebServer, and VarMail workloads in the Filebench suite [100] to evaluate different Mojim modes under PMFS using emulated NVMM. Figure 3.10 presents the throughput of the three workloads of Filebench. For WebServer and Varmail, both M-async and M-sync yield performance similar to S-unreplicated. For FileServer, M-async and M-sync have

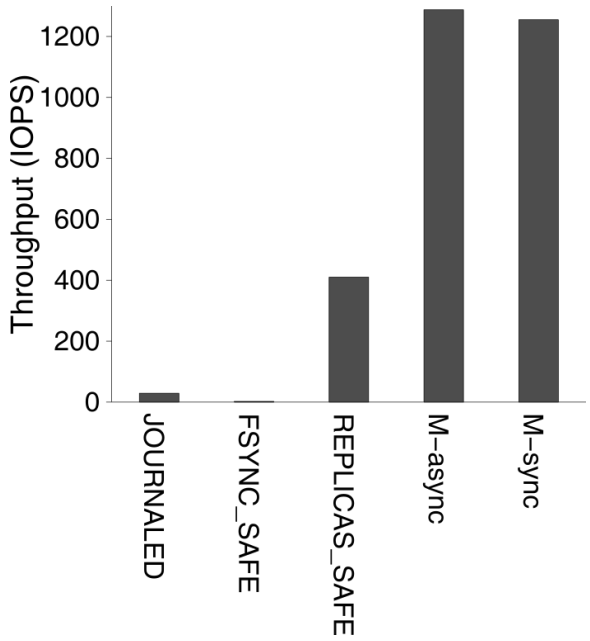

Figure 3.13: **YCSB insert throughput.** Throughput of inserting key-value pairs on emulated NVMM.

slightly worse performance than S-unreplicated.

Google hash table

We perform sequential and random key-value insertion to the Google Dense Hash Table [36]. Each key-value pair contains an integer key and a random integer value. Figure 3.11 plots the average latency of S-unreplicated, M-async, and M-sync with emulated NVMM. For both workloads, all three schemes have similar performance, showing that Mojim has small performance overhead when it comes to hash table operations.

MongoDB

MongoDB is a natural fit for Mojim. We evaluate how MongoDB and Mojim compare using micro- and macro-benchmarks.

Microbenchmark: Our microbenchmark inserts key-value pairs to MongoDB. Each insert

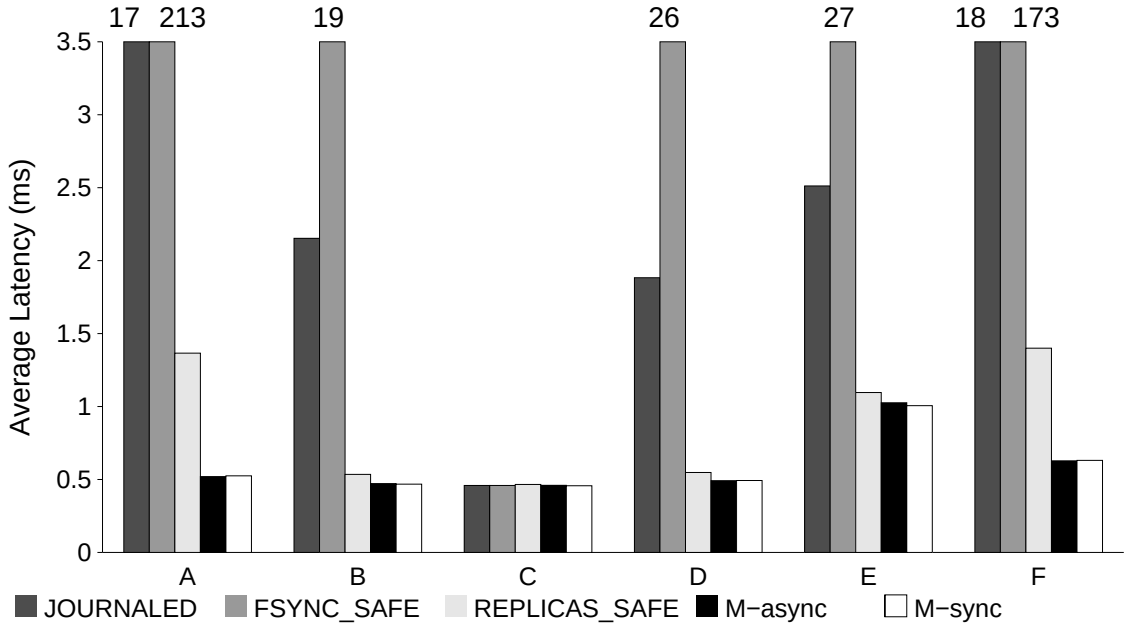

Figure 3.14: **YCSB average latency.** Average latency of YCSB workloads on emulated NVMM.

Table 3.2: **YCSB workload properties.** The percentage of different operations in each YCSB workload.

Workload	Read	Update	Scan	Insert	Read&Update
A	50	50	-	-	-
B	95	5	-	-	-
C	100	-	-	-	-
D	95	-	-	5	-
E	-	-	95	5	-
F	50	-	-	-	50

operation contains 10 key-value pairs, with each pair containing 100 bytes of randomly generated data. Figures 3.12 and 3.13 present the average latency and throughput of key-value pair insertions with PMEP's emulated NVMM. We set the MongoDB replication method to use two replicas (the primary node and the secondary node) and connect these nodes with IB.

MongoDB with Mojim outperforms the MongoDB replication method REPLICAS_SAFE by 3.7 to 3.9×. This performance gain is due to Mojim's efficient replication protocol and networking stack.

Mojim also outperforms the un-replicated JOURNALED MongoDB by 56 to 59× and the un-replicated FSYNC_SAFE by 701 to 741×. JOURNALED flushes journal content for each client

write request. FSYNC_SAFE performs *fsync* of the data file after each write operation to guarantee data reliability without journaling. Both these operations are expensive.

To evaluate Mojim's two-tier architecture with MongoDB, we perform the same set of experiments using three DRAM-based machines in our lab. Similar to the PMEP results, Mojim's M-syncsec outperforms MongoDB's replication method by 3.4 to 4×, the un-replicated JOURNALED MongoDB by 35 to 43×, and the un-replicated FSYNC_SAFE by 238 to 311×, suggesting that Mojim's replication is better than MongoDB replication.

Finally, MongoDB can run unmodified on Mojim by configuring both its journal and data file to be in a `mmap()`'d NVMM region. In this case, its performance is similar to JOURNALED, with a performance overhead of 0.2% to 6%. However, Mojim provides better reliability and availability than the un-replicated MongoDB.

Macrobenchmark: YCSB [22] is a benchmark designed to evaluate key-value store systems. YCSB includes six workloads that imitate web applications' data access models. The workloads contain a combination of read, update, scan, and insert operations. Table 3.2 summarizes the number of these operations in the YCSB workloads. Each workload performs 1000 operations on a database with 1000 1 KB records.

Figure 3.14 presents the latency of MongoDB and Mojim using the six YCSB workloads on emulated NVMM. For most workloads, both M-async and M-sync outperform the un-replicated and replicated MongoDB schemes. The performance improvement is especially high for write-heavy workloads. We also find similar results with three DRAM-based machines.

3.4.5 Recovery

Recovery performance is important because it directly affects availability and may impact reliability, since Mojim is vulnerable to additional node failures during some recovery scenarios. To test the robustness of the system, we stop a Mojim node at random and find that the rest of the system can continue serving client requests correctly. We further measure the recovery time in the

event of a node failure.

We use a typical recovery scenario to illustrate Mojim's recovery performance. When a mirror node fails with M-syncsec, the recovery process requires sending the remaining, un-replicated data to the backup node, flushing the CPU caches on the primary node, and copying all the data areas to the new mirror node. Mojim performs these operations in parallel. We set SECONDARY_TIER_THRESH to 40 MB and use three machines in our lab to perform the recovery performance evaluation.

Mojim takes 450 μs to flush the 26 MB CPU caches on the primary node. Before the primary node flushes all its caches, if it also fails, there will be data loss. The window of vulnerability also depends on how soon the failure can be detected, thus in practice it will be longer than 450 μs [19]. It takes 14 *ms* to send 40 MB of data to the backup node and 1.9 seconds to send a 5 GB data area to the new mirror node. The whole recovery process completes in 1.9 seconds for a 5 GB NVMM. Even for a 1 TB NVMM, the recovery process will only take 6.5 minutes. Notice that the vulnerability window depends on how fast primary node detects a failure and flushes its caches, not on NVMM size.

For M-syncdisk, Mojim also needs to read the data file from the disk to the NVMM before applications can access the data. In this case, recovery takes 17 seconds for a 5 GB data file, a much higher cost in availability than when we use NVMM for the data area.

3.5 Summary

We have described Mojim, a system for providing reliable and highly-available NVMM. Mojim uses a two-tier architecture and efficiently replicates data in NVMM. Our results demonstrate that Mojim can provide replication with small cost, in many cases even outperforming the un-replicated system. In doing so, Mojim paves the way for deploying NVMM in data centers that wish to take advantage of NVMM's enhanced performance but require strong guarantees about data safety.

Acknowledgments

This chapter contains material from“Mojim: A Reliable and Highly-Available Non-Volatile Memory System”, by Yiying Zhang, Jian Yang, Amirsaman Memaripour and Steven Swanson, which appears in the Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2015). The dissertation author is the second investigator and author of this paper.

The author thank Dulloor Subramanya, Jeff Jackson, and the vLab team from Intel Corp. for their help with the PMEP platforms.

Chapter 4

Orion: A Distributed File System for NVMM and RDMA-Capable Networks

In a distributed file system designed for block-based devices, media performance is almost the sole determiner of performance on the data path. The glacial performance of disks (both hard and solid state) compared to the rest of the storage stack incentivizes complex optimizations (e.g., queuing, striping, and batching) around disk accesses. It also saves designers from needing to apply similarly aggressive optimizations to network efficiency, CPU utilization, and locality, while pushing them toward software architectures that are easy to develop and maintain, despite the (generally irrelevant) resulting software overheads.

This chapter, we introduce Orion, a distributed file system designed from the ground up for NVMM and Remote Direct Memory Access (RDMA) networks. While other distributed systems [65, 90] have integrated NVMMs, Orion is the first distributed file system to systematically optimize for NVMMs throughout its design. As a result, Orion diverges from block-based designs in novel ways.

Orion focuses on several areas where traditional distributed file systems fall short when naively adapted to NVMMs. We describe them below.

Use of RDMA

Orion targets systems connected with an RDMA-capable network. It uses RDMA whenever possible to accelerate both metadata and data accesses. Some existing distributed storage systems use RDMA as a fast transport layer for data access [15, 24, 98, 99, 109] but do not integrate it deeply into their design. Other systems [65, 90] adapt RDMA more extensively but provide object storage with customized interfaces that are incompatible with file system features such as unrestricted directories and file extents, symbolic links and file attributes. system structures and interfaces (Octopus).

Orion is the first full-featured file system that integrates RDMA deeply into all aspects of its design. Aggressive use of RDMA means the CPU is not involved in many transfers, lowering CPU load and improving scalability for handling incoming requests. In particular, pairing RDMA with NVMMs allows nodes to directly access remote storage without any target-side software overheads.

Software Overhead

Software overhead in distributed files system has not traditionally been a critical concern. As such, most distributed file systems have used two-layer designs that divide the network and storage layers into separate modules. Two-layer designs trade efficiency for ease of implementation. Designers can build a user-level daemon that stitches together off-the-shelf networking packages and a local file system into a distributed file system. While expedient, this approach results in duplicated metadata, excessive copying, unnecessary event handling, and places user-space protection barriers on the critical path.

Orion merges the network and storage functions into a single, kernel-resident layer optimized for RDMA and NVMM that handles data, metadata, and network access. This decision allows Orion to explore new mechanisms to simplify operations and scale performance.

Table 4.1: **Characteristics of memory and network devices** We measure the fisrt 3 lines on Intel Sandy Bridge-EP platform with a Mellanox ConnectX-4 RNIC and an Intel DC P3600 SSD. NVMM numbers are estimated based on assumptions made in [113]

.

	Read Latency	Bandwidth GB/s	
	512 B	Read	Write
DRAM	80 ns	60	30
NVMM	300 ns	8	2
RDMA NIC	3 μs	5 (40 Gbps)	
NVMe SSD	70 μs	3.2	1.3

Locality

RDMA is fast, but it is still several times slower than local access to NVMMs (Table 4.1). Consequently, the location of stored data is a key performance concern for Orion. This concern is an important difference between Orion and traditional block-based designs that generally distinguish between client nodes and a pool of centralized storage nodes [24, 89]. Pooling makes sense for block devices, since access latency is determined by storage, rather than network latency, and a pool of storage nodes simplifies system administration. However, the speed of NVMMs makes a storage pool inefficient, so Orion optimizes for locality. To encourage local accesses, Orion migrates durable data to the client whenever possible and uses a novel delegated allocation scheme to efficiently manage free space.

Our evaluation shows that Orion outperforms existing distributed file systems by a large margin. Relative to local NVMM filesystems, it provides comparable application-level performance when running applications on a single client. For parallel workloads, Orion shows good scalability: performance on an 8-client cluster is between 4.1$\times$ and 7.9$\times$ higher than running on a single node.

The rest of the chapter is organized as follows. Section 4.1 gives an overview of Orion's architecture. We describe the design decisions we made to implement high-performance metadata access and data access in Sections 4.2 and 4.3 respectively. Section 4.4 evaluates these mechanisms. Finally, We conclude in Section 4.5.

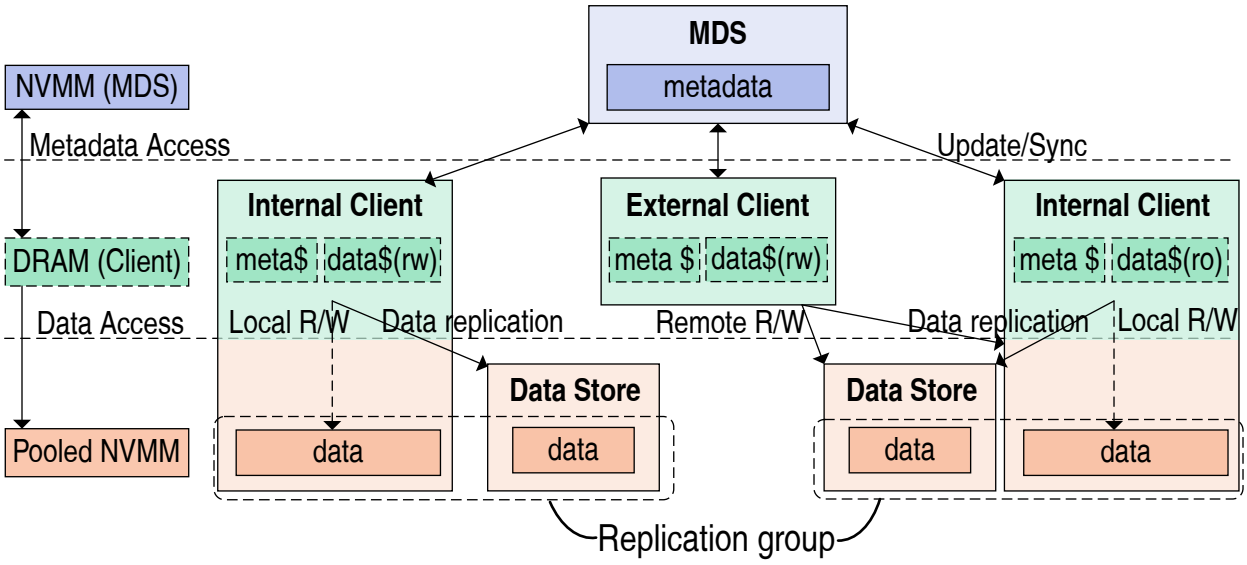

Figure 4.1: **Orion cluster organization.** An Orion cluster consists of a metadata server, clients and data stores.

4.1 Orion Design Overview

Orion is a distributed file system built for the performance characteristics of NVMM and RDMA networking. NVMM's low latency and byte-addressability fundamentally alter the relationship among memory, storage, and network, motivating Orion to use a clean-slate approach to combine the file system and networking into a single layer. Orion achieves the following design goals:

Scalable performance with low software overhead: Scalability and low-latency are essential for Orion to fully exploit the performance of NVMM. Orion achieves this goal by unifying file system functions and network operations and by accessing data structures on NVMM directly through RDMA.

Efficient network usage on metadata updates: Orion caches file system data structures on clients. A client can apply file operations locally and only send the changes to the metadata server over the network.

Metadata and data consistency: Orion uses a log-structured design to maintain file system consistency at low cost. Orion allows read parallelism but serializes updates for file system data structures across the cluster. It relies on atomically updated inode logs to guarantee metadata and data consistency and uses a new coordination scheme called *client arbitration* to resolve conflicts.

DAX support in a distributed file system: DAX-style (direct load/store) access is a key benefit of NVMMs. Orion allows clients to acces in its local NVMM just as it could access a DAX-enabled local NVMM file s

Repeated access become local access: Orion exploits locality by migrating data to where writes occur and making data caching an integral part of the file system design. The log-structured design reduces the cost of maintaining cache coherence.

Reliability and data persistence: Orion supports metadata and data replication for better reliability and availability. The replication protocol also guarantees data persistency.

The remainder of this section provides an overview of the Orion software stack, including its hardware and software organization. The following sections provide details of how Orion manages metadata (Section 4.2) and provides access to data (Section 4.3).

4.1.1 Cluster Organization

An Orion cluster consists of a *metadata server (MDS)*, several *data stores (DSs)* organized in replication groups, and *clients* all connected via an RDMA network. Figure 4.1 shows the architecture of an Orion cluster and illustrates these roles.

The MDS manages metadata. It establishes an RDMA connection to each of the clients. Clients can propagate local changes to the MDS and retrieve updates made by other clients.

Orion allows clients to manage and access a global, shared pool of NVMMs. Data for a file can reside at a single DS or span multiple DSs. A client can access a remote DS using one-sided RDMA and its local NVMMs using load and store instructions.

Internal clients have local NVMM that Orion manages. Internal clients also act as a DSs for

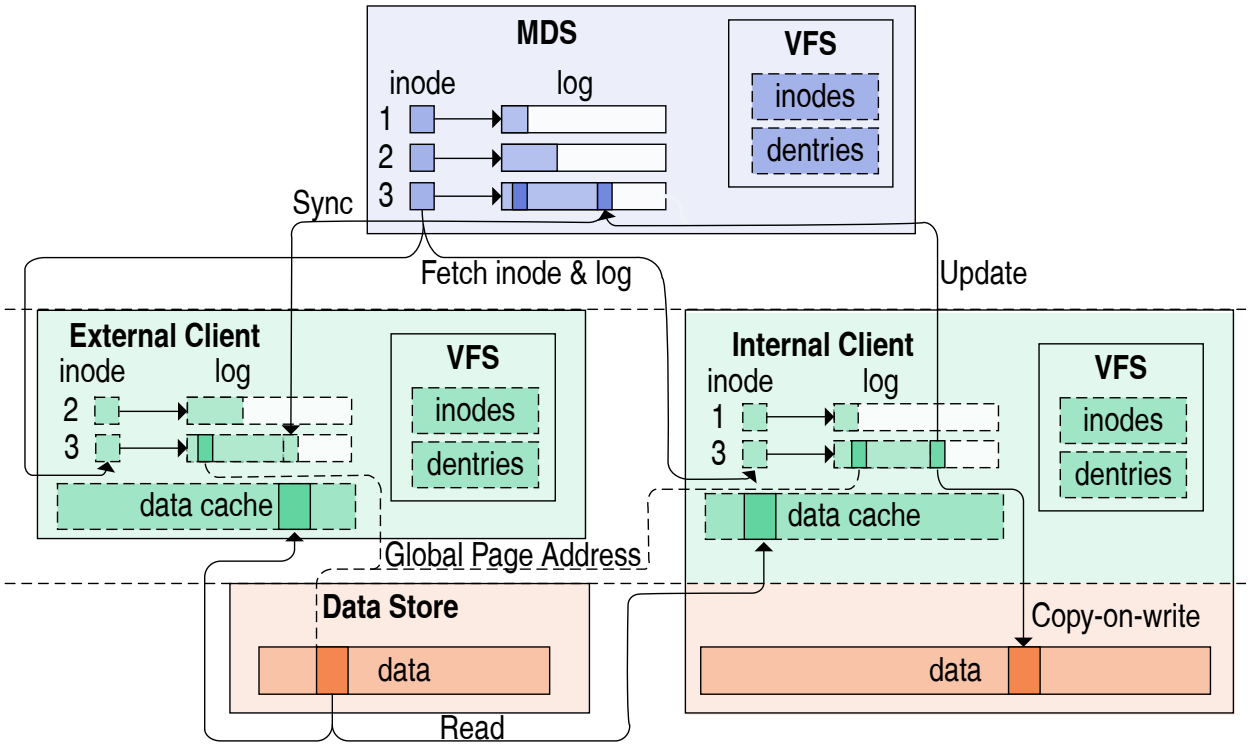

Figure 4.2: **Orion software organization.** Orion exposes as a log-structured file system across MDS and clients. Clients maintain local copies of inode metadata and sync with the MDS, and access data at remote data stores or local NVMM directly.

other clients. *External clients* do not have local NVMM, so they can access data on DSs but cannot store data themselves.

Orion supports replication of both metadata and data. The MDS can run as a high-availability pair consisting of a primary server and a mirror using Mojim-style replication. Mojim provides low latency replication for NVMM by maintaining a single replica and only making updates at the primary.

Orion organizes DSs into replication groups, and the DSs in the group have identical data layouts. Orion uses broadcast replication for data.

4.1.2 Software Organization

Orion's software runs on the clients and the MDS. It exposes a normal POSIX interface and consists of kernel modules that manage file and metadata in NVMM and handle communication between the MDS and clients. Running in the kernel avoids the frequent context switches, copies, and kernel/user crossing that conventional two-layer distributed file systems designs require.

The file system in Orion inherits some design elements from NOVA [111, 112], a log-structured POSIX-compliant local NVMM file system. Orion adopts NOVA's highly-optimized mechanisms for managing file data and metadata in NVMM. Specifically, Orion's local file system layout, inode log data structure, and radix trees for indexing file data in DRAM are inherited from NOVA, with necessary changes to make metadata accessible and meaningful across nodes. Figure 5.7 shows the overall software organization of the Orion file system.

An Orion inode contains pointers to the head and tail of a metadata log stored in a linked list of NVMM pages. A log's entries record all modifications to the file and hold pointers to the file's data blocks. Orion uses the log to build virtual file system (VFS) inodes in DRAM along with indices that map file offsets to data blocks. The MDS contains the metadata structures of the whole file system including authoritative inodes and their logs. Each client maintains a local copy of each inode and its logs for the files it has opened.

Copying the logs to the clients simplifies and accelerates metadata management. A client can recover all metadata of a file by walking through the log. Also, clients can apply a log entry locally in response to a file system request and then propagate it to the MDS. A client can also tell whether an inode is up-to-date by comparing the local and remote log tail. An up-to-date log should be equivalent on both the client and the MDS, and this invariant is the basis for our metadata coherency protocol. Because MDS inode log entries are immutable except during garbage collection and logs are append-only, logs are amenable to direct copying via RDMA reads (see Section 4.2).

Orion distributes data across DSs (including the internal clients) and replicates the data within replication groups. To locate data among these nodes, Orion uses *global page addresses*

(GPAs) to identify pages. Clients use a GPA to locate both the replication group and data for a page. For data reads, clients can read from any node within a replication group using the global address. For data updates, Orion performs a copy-on-write on the data block and appends a log entry reflecting the change in metadata (e.g.write offset, size, and the address to the new data block). For internal clients, the copy-on-write migrates the block into the local NVMM if space is available.

An Orion client also maintains a client-side data cache. The cache, combined with the copy-on-write mechanism, lets Orion exploit and enhance data locality. Rather than relying on the operating system's generic page cache, Orion manages DRAM as a customized cache that allows it to access cached pages using GPAs without a layer of indirection. This also simplifies cache coherence.

4.2 Metadata Management

Since metadata updates are often on an application's critical path, a distributed file system must handle metadata requests quickly. Orion's MDS manages all metadata updates and holds the authoritative, persistent copy of metadata. Clients cache metadata locally as they access and update files, and they must propagate changes to both the MDS and other clients to maintain coherence.

Below, we describe how Orion's metadata system meets both these performance and correctness goals using a combination of communication mechanisms, latency optimizations, and a novel arbitration scheme to avoid locking.

4.2.1 Metadata Communication

The MDS orchestrates metadata communication in Orion, and all authoritative metadata updates occur there. Clients do not exchange metadata. Instead, an Orion client communicates with the MDS to fetch file metadata, commit changes and apply changes committed by other clients.

Clients communicate with the MDS using three methods depending on the complexity of the operation they need to perform: (1) direct *RDMA reads*, (2) speculative and highly-optimized

log commits, and (3) acknowledged *remote procedure calls* (RPCs).

These three methods span a range of options from simple/lightweight (direct RDMA reads) to complex/heavyweight (RPC). We use RDMA reads from the MDS whenever possible because they do not require CPU intervention, maximizing MDS scalability.

Below, we describe each of these mechanisms in detail followed by an example. Then, we describe several additional optimizations Orion applies to make metadata updates more efficient.

RDMA reads: Clients use one-sided RDMA reads to pull metadata from the MDS when needed, for instance, on file open. Orion uses wide pointers that contain a pointer to the client's local copy of the metadata as well as a GPA that points to the same data on the MDS. A client can walk through its local log by following the local pointers, or fetch the log pages from the MDS using the GPAs.

The clients can access the inode and log for a file using RDMA reads since NVMM is byte addressable. These accesses bypass the MDS CPU, which improves scalability.

Log commits: Clients use log commits to update metadata for a file. The client first performs file operations locally by appending a log entry to the local copy of the inode log. Then it forwards the entry to the MDS and waits for completion.

Log commits use RDMA sends. Log entries usually fit in two cache lines, so the RDMA NIC can send them as inlined messages, further reducing latencies. Once it receives the acknowledgment for the send, the client updates its local log tail, completing the operation. Orion allows multiple clients to commit log entries of a single inode without distributed locking using a mechanism called *client arbitration* that can resolve inconsistencies between inode logs on the clients (Section 4.2.3).

Remote procedure calls: Orion uses synchronous remote procedure calls (RPCs) for metadata accesses that involve multiple inodes as well as operations that affect other clients (e.g. a file write with `O_APPEND` flag).

Orion RPCs use a send verb and an RDMA write. An RPC message contains an opcode along with metadata updates and/or log entries that the MDS needs to apply atomically. The MDS performs the procedure call and responds via one-sided RDMA write or message send depending

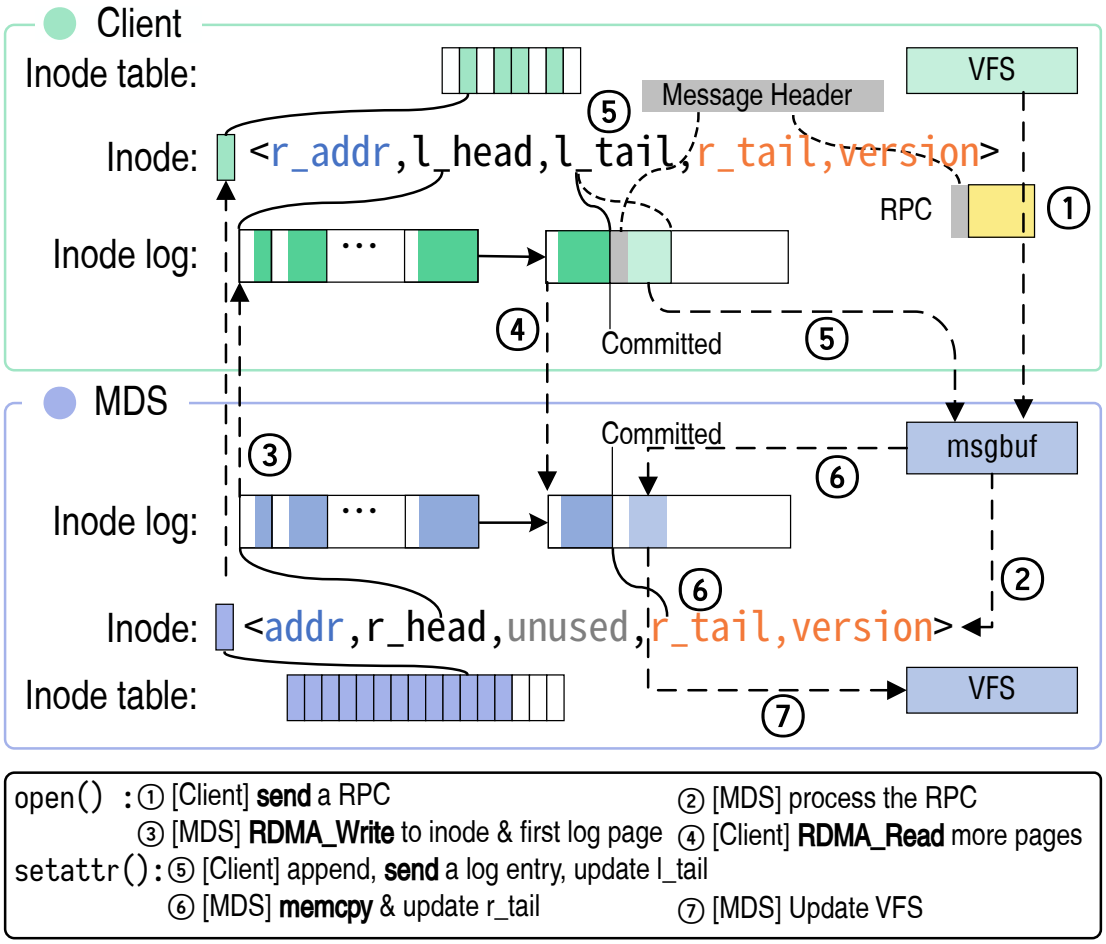

Figure 4.3: **Orion metadata communication.** Orion maintains metadata structures such as inode logs on both MDS and clients. A client commit file system updates through Log Commits and RPCs.

on the opcode. The client blocks until the response arrives.

Example: Figure 5.8 illustrates metadata communication. For `open()` (an RPC-based metadata update), the client allocates space for the inode and log, and issues an RPC ①. The MDS handles the RPC ② and responds by writing the inode along with the first log page using RDMA ③. The client uses RDMA to read more pages if needed and builds VFS data structures ④.

For a `setattr()` request (a log commit based metadata update), the client creates a local entry with the update and issues a log commit ⑤. It then updates its local tail pointer atomically

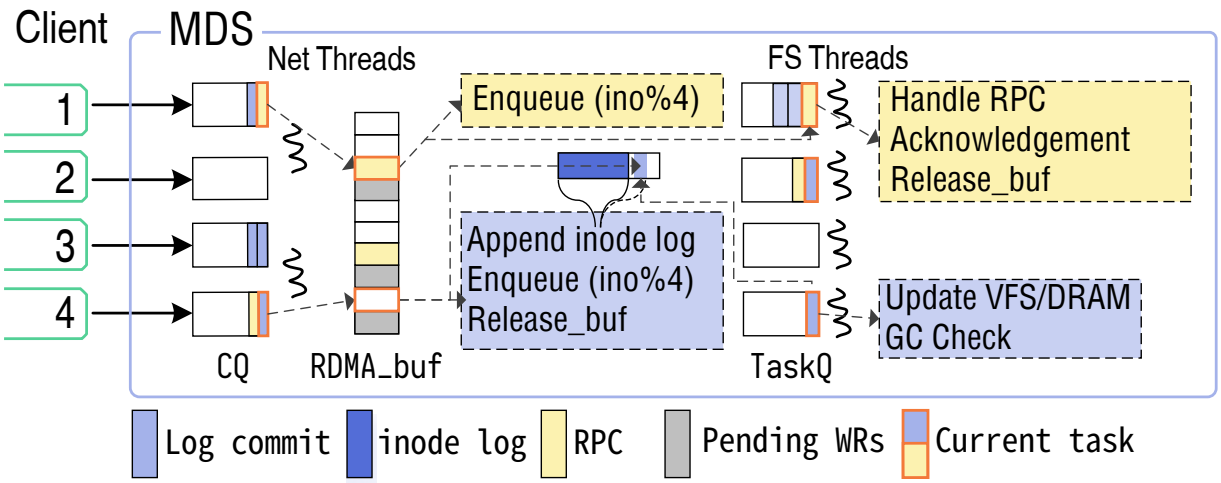

Figure 4.4: **MDS request handling.** The MDS handles client requests in two stages: First, networking threads handle RDMA completion queue entries (CQEs) and dispatch them to file system threads. Next, file system threads handle RPCs and update the VFS.

after it has sent the log commit. Upon receiving the log entry, the MDS appends the log entry, updates the log tail ⑥, and updates the corresponding data structure in VFS ⑦.

RDMA Optimizations: Orion avoids data copying within a node whenever possible. Both client-initiated RDMA reads and MDS-initiated RDMA writes (e.g.in response to an RPC) target client file system data structures directly. Additionally, log entries in Orion contain extra space (shown as *message headers* in Figure 5.8) to accommodate headers used for networking. Aside from the DMA that the RNIC performs, the client copies metadata at most once (to avoid concurrent updates to the same inode) during a file operation.

Orion also uses *relative pointers* in file system data structures to leverage the linear addressing in kernel memory management. NVMM on a node appears as contiguous memory regions in both kernel virtual and physical address spaces. Orion can create either type of address by adding the relative pointer to the appropriate base address. Relative pointers are also meaningful across power failures.

4.2.2 Minimizing Commit Latency

The latency of request handling, especially for log commits, is critical for the I/O performance of the whole cluster. Orion uses dedicated threads to handle per-client receive queues as well as file system updates. Figure 4.4 shows the MDS request handling process.

For each client, the MDS registers a small (256 KB) portion of NVMM as a communication buffer. The MDS handles incoming requests in two stages: A *network thread* polls the RDMA completion queues (CQs) for work requests on pre-posted RDMA buffers and dispatches the requests to *file system threads*. As an optimization, the MDS prioritizes log commits by allowing network threads to append log entries directly. Then, a file system thread handles the requests by updating file system structures in DRAM for a log commit or serving the requests for an RPC. Each file system thread maintains a FIFO containing pointers to updated log entries or RDMA buffers holding RPC requests.

For a log commit, a network thread reads the inode number, appends the entry by issuing non-temporal moves and then atomically updates the tail pointer. At this point, other clients can read the committed entry and apply it to their local copy of the inode log. The network thread then releases the recv buffer by posting a recv verb, allowing its reuse. Finally, it dispatches the task for updating in-DRAM data structures to a file system thread based on the inode number.

For RPCs, the network thread dispatches the request directly to a file system thread. Each thread processes requests to a subset of inodes to ensure better locality and less contention for locks. The file system threads use lightweight journals for RPCs involving inodes that belong to multiple file system threads.

File system threads perform garbage collection (GC) when the number of "dead" entries in a log becomes too large. Orion rebuilds the inode log by copying live entries to new log pages. It then updates the log pointers and increases the version number. Orion makes this update atomic by packing the version number and tail pointer into 64 bits. The thread frees stale log pages after a delay, allowing ongoing RDMA reads to complete. Currently we set the maximal size of file writes

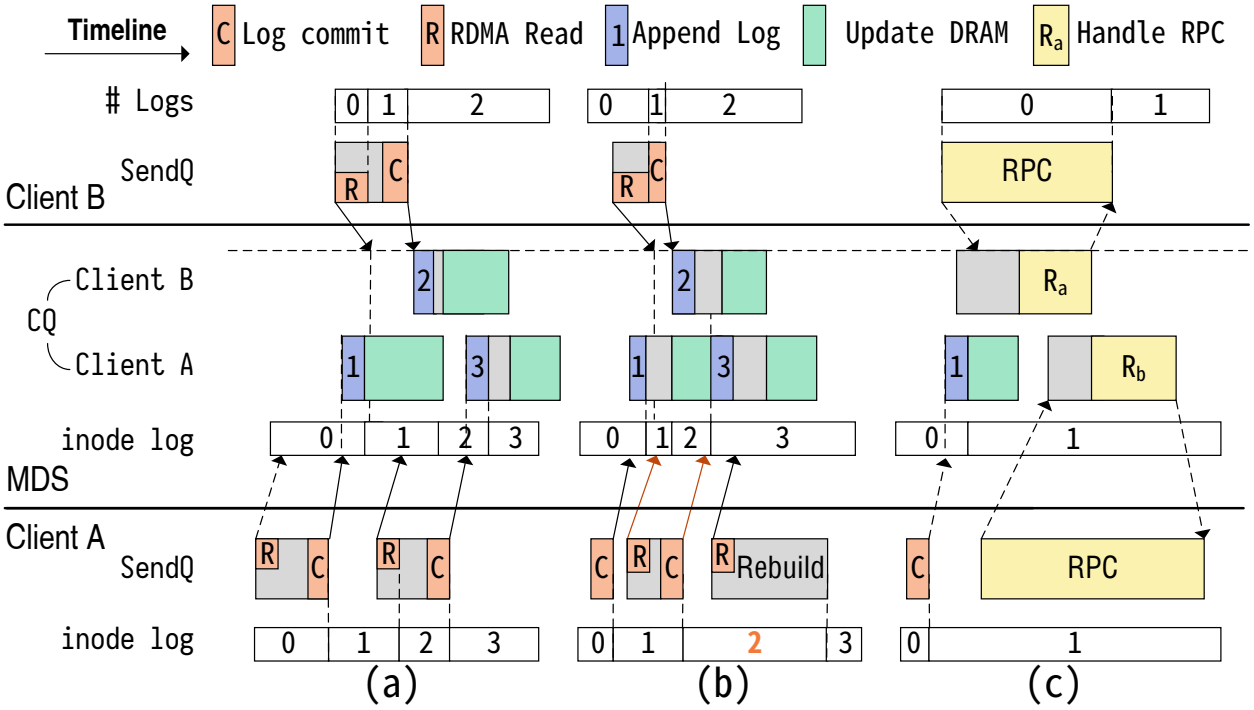

Figure 4.5: **Metadata consistency in Orion.** The inode log on Client A is consistent after (a) updating the log entry committed by another client using RDMA reads, (c) issuing an RPC, and (b) rebuilding the log on conflicts.

in a log entry to be 512 MB.

4.2.3 Client Arbitration

Orion allows multiple clients to commit log entries to a single inode at the same time using a mechanism called *client arbitration* rather than distributed locking. Client arbitration builds on the following observations:

1. Handling an inbound RDMA read is much cheaper than sending an outbound write. In our experiments, a single host can serve over 15 M inbound reads per second but only 1.9 M outbound writes per second.

2. For the MDS, CPU time is precious. Having the MDS initiate messages to maintain consistency will reduce Orion performance significantly.

3. Log append operations are lightweight: each one takes around just 500 CPU cycles.

A client commits a log entry by issuing a send verb and polling for its completion. The MDS appends log commits based on arrival order and updates log tails atomically. A client can determine whether a local inode is up-to-date by comparing the log length of its local copy of the log and the authoritative copy at the MDS. Clients can check the length of an inode's log by retrieving its tail pointer with an RDMA read.

The client issues these reads in the background when handling an I/O request. If another client has modified the log, the client detects the mismatch and fetches the new log entries using additional RDMA reads and retries.

If the MDS has committed multiple log entries in a different order due to concurrent accesses, the client blocks the current request and finds the last log entry that is in sync with the MDS, it then fetches all following log entries from the MDS, rebuilds its in-DRAM structures, and re-executes the user request.

Figure 4.5 shows the three different cases of concurrent accesses to a single inode. In (a), the client A can append the log entry #2 from client B by extending its inode log. In (b), the client A misses the log entry #2 committed by client B, so it will rebuild the inode log on the next request. In (c), the MDS will execute concurrent RPCs to the same inode sequentially, and the client will see the updated log tail in the RPC acknowledgment.

A rebuild occurs when all of the following occur at the same time: (1) two or more clients access the same file at the same time and one of the accesses is log commit, (2) one client issues two log commits consecutively, and (3) the MDS accepts the log commit from another client after the client RDMA reads the inode tail but before the MDS accepts the second log commit.

In our experience this situation happens very rarely, because the "window of vulnerability" – the time required to perform a log append on the MDS – is short. That said, Orion lets applications identify files that are likely targets of intensive sharing via an `ioctl`. Orion uses RPCs for all updates to these inodes in order to avoid rebuilds.

4.3 Data Management

Orion pools NVMM spread across internal clients and data stores. A client can allocate and access data either locally (if the data are local) or remotely via one-sided RDMA. Clients use local caches and migration during copy-on-write operations to reduce the number of remote accesses.

4.3.1 Delegated Allocation

To avoid allocating data on the critical path, Orion uses a distributed, two-stage memory allocation scheme.

The MDS keeps a bitmap of all the pages Orion manages. Clients request large chunks of storage space from the MDS via an RPC. The client can then autonomously allocate space within those chunks. This design frees the MDS from managing fine-grain data blocks, and allows clients to allocate pages with low overhead.

The MDS allocates internal clients chunks of its local NVMM when possible since local writes are faster. As a result, most of their writes go to local NVMM.

4.3.2 Data Access

To read file data, a client either communicates with the DS using one-sided RDMA or accesses its local NVMM via DAX (if it is an internal client and the data is local). Remote reads use one-sided RDMA reads to retrieve existing file data and place it in local DRAM pages that serve as a cache for future reads.

Remote writes can also be one-sided because allocation occurs at the client. Once the transfer is complete, the client issues a log commit to the MDS.

Figure 4.6 demonstrates Orion's data access mechanisms. A client can request a block chunk from the MDS via an RPC ①. When the client opens a file, it builds a radix tree for fast lookup from file offsets to log entries ②. When handling a `read()` request, the client reads from the DS

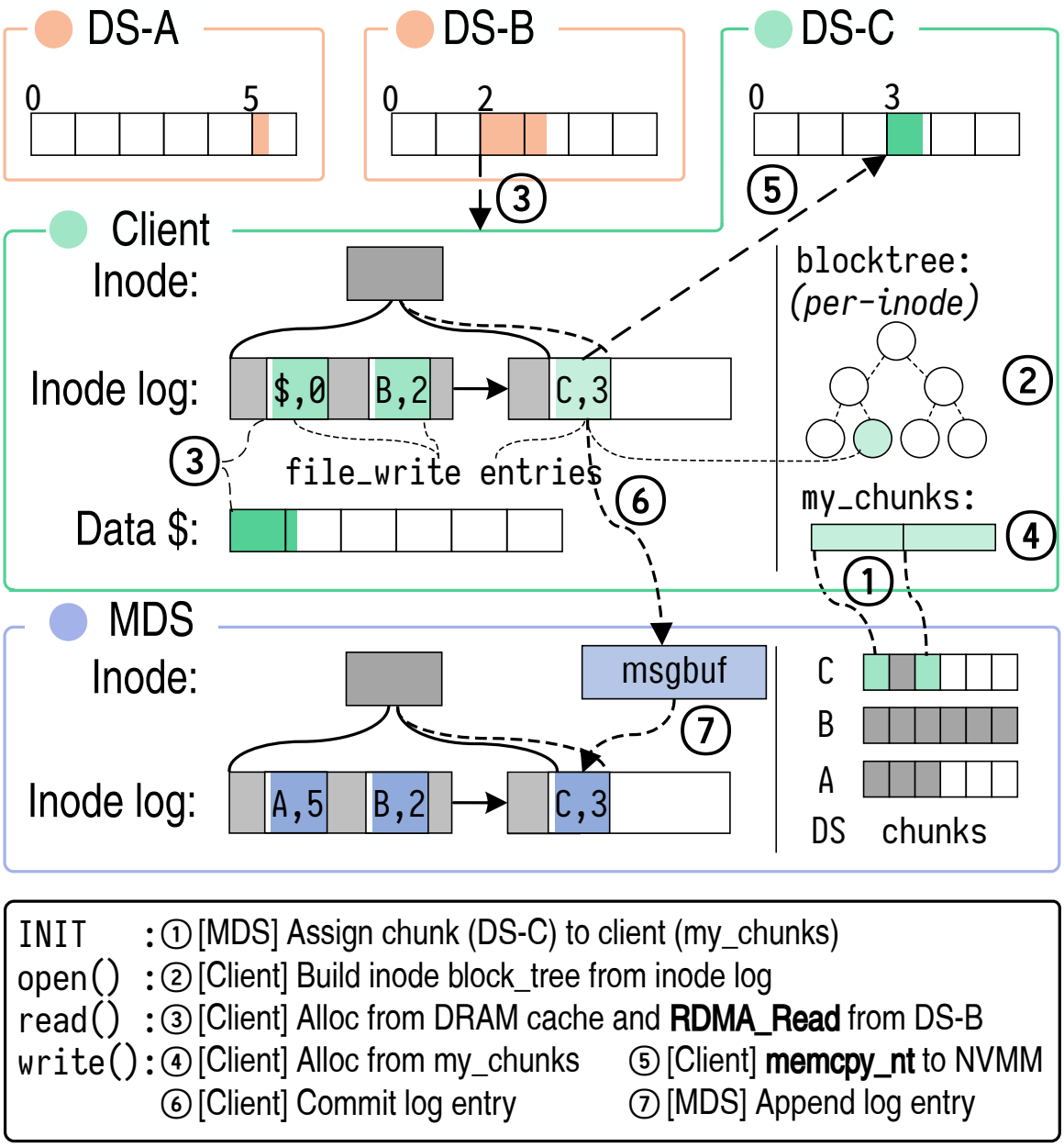

Figure 4.6: **Orion data communication.** Orion allows clients manage and access data independently.

(DS-B) to its local DRAM and update the corresponding log entry ③. For a `write()` request, it allocates from its local chunk ④ and issues `memcpy_nt()` and `sfence` to ensure that the data reaches its local NVMM (DS-C) ⑤. Then a log entry containing information such as the GPA and size is committed to the MDS ⑥. Finally, the MDS appends the log entry ⑦.

4.3.3 Data Persistence

Orion always ensures that metadata is consistent, but, like many file systems, it can relax the consistency requirement on data based on user preferences and the availability of replication.

The essence of Orion's data consistency guarantee is the extent to which the MDS delays the log commit for a file update. For a weak consistency guarantee, an external client can forward a speculative log commit to the MDS before its remote file update has completed at a DS. This consistency level is comparable to the write-back mode in ext4 and can result in corrupted data pages but maintains metadata integrity. For strong data consistency that is comparable to NOVA and the data journaling mode in ext4, Orion can delay the log commit until after the file update is persistent at multiple DSs in the replication group.

Achieving strong consistency over RDMA is hard because RDMA hardware does not provide a standard mechanism to force writes into remote NVMM. For strongly consistent data updates, our algorithm is as follows.

A client that wishes to make a consistent file update uses copy-on-write to allocate new pages on all nodes in the appropriate replica group, then uses RDMA writes to update the pages. In parallel, the client issues a speculative log commit to the MDS for the update.

DSs within the replica group detect the RDMA writes to new pages using an RDMA trick: when clients use RDMA writes on the new pages, they include the page's global address as an *immediate value* that travels to the target in the RDMA packet header. This value appears in the target NIC's completion queue, so the DS can detect modifications to its pages. For each updated page, the DS forces the page into NVMM and sends an acknowledgment via a small RDMA

write to the MDS, which processes the client's log commit once it reads a sufficient number of acknowledgments in its DRAM.

4.3.4 Fault Tolerance

The high performance and density of NVMM makes the cost of rebuilding a node much higher than recovering it. Consequently, Orion makes its best effort to recover the node after detecting an error. If the node can recover (e.g. after a power failure and most software bugs), it can rejoin the Orion cluster and recover to a consistent state quickly. For NVMM media errors, module failures, or data-corrupting bugs, Orion rebuilds the node using the data and metadata from other replicas. It uses relative pointers and global page addresses to ensure metadata in NVMM remain meaningful across power failures.

In the metadata subsystem, for MDS failures, Orion builds a Mojim-like high-availability pair consisting of a primary MDS and a mirror. All metadata updates flow to the primary MDS, which propagates the changes to the mirror. When the primary fails, the mirror takes over and journals all the incoming requests while the primary recovers.

In the data subsystem, for DS failures, the DS journals the immediate values of incoming RDMA write requests in a circular buffer. A failed DS can recover by obtaining the pages committed during its downtime from a peer DS in the same replication group. When there are failed nodes in a replication group, the rest of the nodes work in the strong data consistency mode introduced in Section 4.3.3 to ensure successful recovery in the event of further failures.

4.4 Evaluation

In this section, we evaluate the performance of Orion by comparing it to existing distributed file systems as well as local file systems. We answer the following questions:

- How does Orion's one-layer design affect its performance compared to existing two-layer distributed file systems?

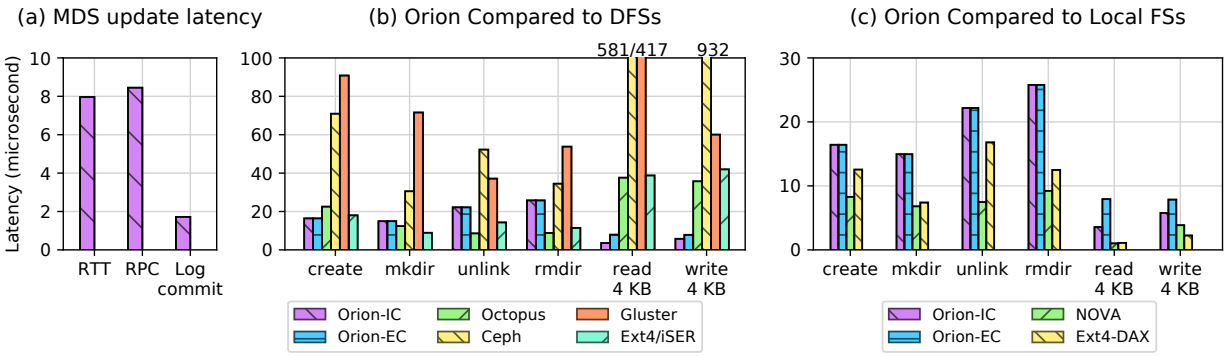

Figure 4.7: **Average latency of Orion metadata and data operations.** Orion is built on low-latency communication primitive (a). These lead to basic file operation latencies that are better than existing remote-access storage system (b) and within a small factor of local NVMM file systems (c).

- How much overhead does managing distributed data and metadata add compared to running a local NVMM file system?
- How does configuring Orion for different levels of reliability affect performance?
- How scalable is Orion's MDS?

We describe the experimental setup and then evaluate Orion with micro- and macrobenchmarks. Then we measure the impact of data replication and the ability to scale over parallel workloads.

4.4.1 Experimental Setup

We run Orion on a cluster with 10 nodes configured to emulate persistent memory with DRAM. Each node has two quad-core Intel Xeon (Westmere-EP) CPUs with 48 GB of DRAM, with 32 GB configured as an emulated `pmem` device. Each node has an RDMA NIC (Mellanox ConnectX-2 40 Gbps HCA) running in Infiniband mode and connects to an Infiniband switch (QLogic 12300). We disabled the Direct Cache Access feature on DSs. To demonstrate the impact to co-located applications, we use a dedicated core for issuing and handling RDMA requests on each client.

We build our Orion prototype on the Linux 4.10 kernel with the RDMA verb kernel modules from Mellanox OFED [66]. The file system in Orion reuses code from NOVA but adds ∼8K lines of code to support distributed functionalities and data structures. The networking module in Orion is built from scratch and comprises another ∼8K lines of code.

We compare Orion with three distributed file systems Ceph [107], Gluster [25], and Octopus [65] running on the same RDMA network. We also compare Orion to ext4 mounted on a remote iSCSI target hosting a ramdisk using iSCSI Extension over RDMA (iSER) [16] (denoted by Ext4/iSER), which provides the client with private access to a remote block device. Finally, we compare our system with two local DAX file systems: NOVA [111, 112] and ext4 in DAX mode (ext4-DAX).

4.4.2 Microbenchmarks

We begin by measuring the networking latency of log commits and RPCs. Figure 4.7(a) shows the latency of a log commit and an RPC compared to the network round trip time (RTT) using two sends verbs. Our evaluation platform has a network round trip time of 7.96 μs. The latency of issuing an Orion RPC request and obtaining the response is 8.5 μs. Log commits have much lower latency since the client waits until receiving the acknowledgment of an RDMA send work request, which takes less than half of the network round trip time: they complete in less than 2 μs.

Figure 4.7(b) shows the metadata operation latency on Orion and other distributed file systems. We evaluated basic file system metadata operations such as `create`, `mkdir`, `unlink`, `rmdir` as well as reading and writing random 4 KB data using FIO [6]. Latencies for Ceph and Gluster are between 34% and 443% higher than Orion.

Octopus performs better than Orion on `mkdir`, `unlink` and `rmdir`, because Octopus uses a simplified file system model: it maintains all files and directories in a per-server hash table indexed by their full path names and it assigns a fixed number of file extents and directory entries to each file and directory. This simplification means it cannot handle large files or directories.

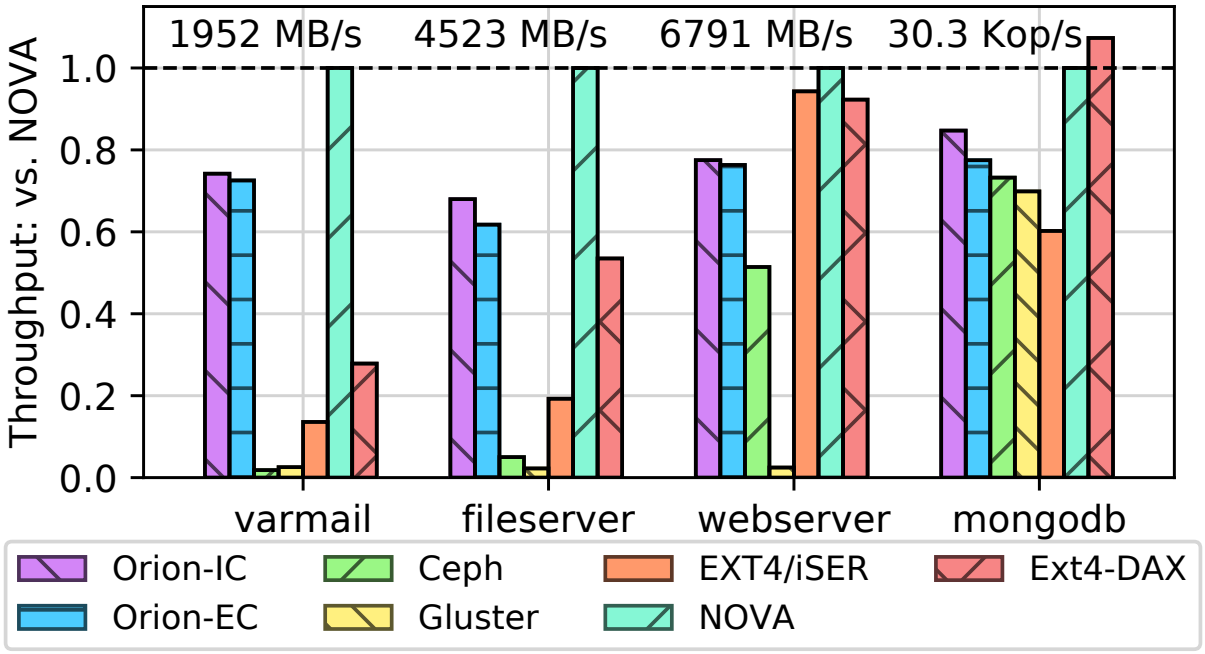

Figure 4.8: **Application performance on Orion.** The graph is normalized to NOVA, and the annotations give NOVA's performance. For write-intensive workloads, Orion outperforms Ceph and Gluster by a wide margin.

Ext4/iSER outperforms Orion on some metadata operations because it considers metadata updates complete once they enter the block queue. In contrast, NVMM-aware systems (such as Orion or Octopus) report the full latency for persistent metadata updates. The 4 KB read and write measurements in the figure give a better measure of I/O latency – Orion outperforms Ext4/iSER configuration by between 4.9× and 10.9×.

For file reads and writes, Orion has the lowest latency among all the distributed file systems we tested. For internal clients (Orion-IC), Orion's 4 KB read latency is 3.6 μs and 4 KB write latency of 5.8 μs. For external clients (Orion-EC), the write latency is 7.9 μs and read latency is similar to internal clients because of client-side caching. For cache misses, read latency is 7.9 μs.

We compare Orion to NOVA and Ext4-DAX in Figure 4.7(c). For metadata operations, Orion sends an RPC to the MDS on the critical path, increasing latency by between 98% to 196% compared to NOVA and between 31% and 106% compared to Ext4-DAX. If we deduct the networking round

Table 4.2: **Application workload characteristics** This table includes the configurations for three filebench workloads and the properties of YCSB-A.

Workload	# Threads	# Files	Avg. File Size	R/W Size	Append Size
varmail	8	30 K	16 KB	1 MB	16 KB
fileserver	8	10 K	128 KB	1 MB	16 KB
webserver	8	50 K	64 KB	1 MB	8 KB
mongodb	12	YCSB-A, RecordCount=1M, OpCount=10M			

trip latency, Orion increases software overheads by 41%.

4.4.3 Macrobenchmarks

We use three Filebench [100] workloads (varmail, fileserver and webserver) as well as MongoDB [72] running YCSB's [22] Workload A (50% read/50% update) to evaluate Orion. Table 4.2 describes the workload characteristics. We could not run these workloads on Octopus because it limits the directory entries and the number of file extents, and it ran out of memory when we increased those limits to meet the workloads' requirements.

Figure 4.8 shows the performance of Orion internal and external clients along with other file systems. For filebench workloads, Orion outperforms Gluster and Ceph by a large margin (up to 40×). We observe that the high synchronization cost in Ceph and Gluster makes them only suitable for workloads with high queue depths, which are less likely on NVMM because media access latency is low. For MongoDB, Orion outperforms other distributed file systems by a smaller margin because of the less intensive I/O activities.

Although Ext4/iSER does not support sharing, file system synchronization (e.g. `fsync()`) is expensive because it flushes the block queue over RDMA. Orion outperforms Ext4/iSER in most workloads, especially for those that require frequent synchronization, such as varmail (with 4.5× higher throughput). For webserver, a read-intensive workload, Ext4/iSER performs better than local Ext4-DAX and Orion because it uses the buffer cache to hold most of the data and does not flush writes to storage.

Orion achieves an average of 73% of NOVA's throughput. It also outperforms Ext4-DAX on

metadata and I/O intensive workloads such as varmail and filebench. For Webserver, a read-intensive workload, Orion is slower because it needs to communicate with the MDS.

The performance gap between external clients and internal clients is small in our experiments, especially for write requests. This is because our hardware does not support the optimized cache flush instructions that Intel plans to add in the near future [86]. Internal clients persist local writes using `clflush` or non-temporal memory copy with fences; both of which are expensive.

4.4.4 Metadata and Data Replication

Figure 4.9 shows the performance impact of metadata and data replication. We compare the performance of a single internal client (IC), a single external client (EC), an internal client with one and two replicas (IC+1R, +2R), and an internal client with two replicas and MDS replication (+2R+M). For a 4 KB write, it takes an internal client 12.1 μs to complete with our strongest reliability scheme (+2R+M), which is 2.1$\times$ longer than internal client and 1.5$\times$ longer than an external client. For filebench workloads, overall performance decreases by between 2.3% and 15.4%.

4.4.5 MDS Scalability

We measure MDS performance scalability by stressing it with different types of requests: client initiated inbound RDMA reads, log commits, and RPCs. Figure 4.10 measures throughput for the MDS handling concurrent requests from different numbers of clients. For inbound RDMA reads (a), each client posts RDMA reads for an 8-byte field, simulating reading the log tail pointers of inodes. In (b) the client sends 64-byte log commits spread across 10,000 inodes. In (c) the clients send 64-byte RPCs and the MDS responds with 32-byte acknowledgments. Each RPC targets one of the 10,000 inodes. Finally, in (d) we use FIO to perform 4 KB random writes from each client to private file.

Inbound RDMA reads have the best performance and scale well: with eight clients, the MDS

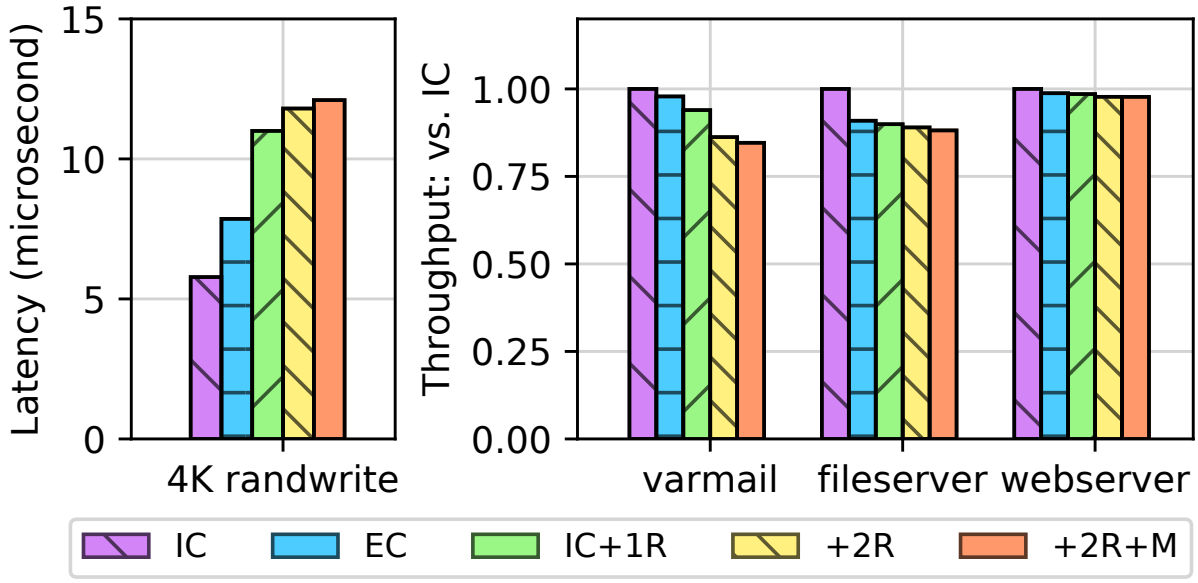

Figure 4.9: **Orion data replication performance.** Updating a remote replica adds significantly to random write latency, but the impact on overall benchmark performance is small.

performs 13.8 M RDMA reads per second – 7.2× the single-client performance. For log commits, peak throughput is 2.5 M operations per second with eight clients – 4.1× the performance for a single client. Log commit scalability is lower because the MDS must perform the log append in software. The MDS can perform 772 K RPCs per second with seven clients (6.2× more than a single). Adding an eighth does not improve performance due to contention among threads polling CQEs and threads handling RPCs. The FIO write test shows good scalability – 7.9× improvement with eight threads. Orion matches NOVA performance with two clients and out-performs NOVA by 4.1× on eight clients.

Orion uses a single MDS with a read-only mirror to avoid the overhead of synchronizing metadata updates across multiple nodes. However, using a single MDS raises scalability concerns. In this section, we run an MDS paired with 8 internal clients to evaluate the system under heavy metadata traffic.

Orion is expected to have good scaling under these conditions. Similar to other RDMA based studies, Orion is suitable to be deployed on networks with high bisectional bandwidth and predictable

Figure 4.10: **Orion metadata scalability for MDS metadata operations and FIO 4K randwrite.** Orion exhibits good scalability with rising node counts for inbound 8 B RDMA reads (a), 64 B log commits (b), RPCs (c), and random writes (d).

end-to-end latency, such as rack-scale computers [23, 59]. In these scenarios, the single MDS design is not a bottleneck in terms of NVMM storage, CPU utilization, or networking utilization. Orion metadata consumes less than 3% space compared to actual file data in our experiments. Additionally, metadata communication is written in tight routines running on dedicated cores, where most of the messages fit within two cache lines. Previous works [8, 62] show similar designs can achieve high throughput with a single server.

In contrast, several existing distributed file systems [12, 25, 35, 107] target data-center scale applications, and use mechanisms designed for these conditions. In general, Orion's design is orthogonal to the mechanisms used in these systems, such as client side hashing [25] and partitioning [107], which could be integrated into Orion as future work. On the other hand, we expect there may be other scalability issues such as RDMA connection management and RNIC resource contention that need to be addressed to allow further scaling for Orion. We leave this exploration as future work.

4.5 Summary

This chapter describes Orion, a file system for distributed NVMM and RDMA networks. By combining file system functions and network operations into a single layer, Orion provides low latency metadata accesses and allows clients to access their local NVMMs directly while accepting

remote accesses. Our evaluation shows that Orion outperforms existing NVMM file systems by a wide margin, and it scales well over multiple clients on parallel workloads.

Acknowledgments

This chapter contain material from "Orion: A Distributed File System for Non-Volatile Main Memories and RDMA-Capable Networks", by Jian Yang, Joseph Izraelevitz and Steven Swanson, which appears in the Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST 2019). The dissertation author was the primary investigator and first author of this paper.

Chapter 5

FileMR: Rethinking Userspace RDMA Networking for Scalable Persistent Memory

We have introduced Mojim and Orion, two systems that combine NVMM and RDMA into a unified network-attached persistent memory. Both of them work as an operating system service that indirect user requests into the operating system service.

We need such indirection because NVM file systems and the RDMA network protocol were not designed to work together. As a result, when an userspace application maps NVMM into its address space and accepts remote RDMA accesses directly, there are many redundancies, particularly where the systems overlap in memory. Only RDMA provides network data transfer and only the NVMM file system provides persistent memory metadata, but both systems implement protection, address translation, naming, and allocation across different abstractions: for RDMA, memory regions, and for NVMM file systems, files. Naively using RDMA and NVMM file systems together results in a duplication of effort and inefficient translation layers between their abstractions. These translation layers are expensive, especially since RNICs can only store translations for limited amount of memory while NVM capacity can be extremely large.

In this chapter, we present a new abstraction, called a *file memory region* (FileMR), that combines the best of both RDMA and NVM file systems into a design that can provide fast, network-

attached, file-system managed, persistent memory. It accomplishes this goal by offloading most RDMA-required tasks related to memory management to the NVM file system through the new memory region type; the file system effectively becomes RDMA's control plane.

With the FileMR abstraction, a client establishes an RDMA connection backed by *files*, instead of memory address ranges (i.e. an RDMA memory region). RDMA reads and writes are directed to the file through the file system, and addressed by the file offset. The translation between file offset and physical memory address is routed through the NVMM file system, which stores all its files in persistent memory. Access to the file is mediated via traditional file system protections (e.g. access control lists). To further optimize address translation, we integrate a *range-based translation* system, which uses address ranges (instead of pages) for translation, into the RNIC, reducing the space needed for translation and resolving the abstraction mismatch between RDMA and NVMM file systems.

Our FileMR design with range-based translation provides a way to seamlessly combine RDMA and NVMM. Compared to simplying layering traditional RDMA memory regions on top of NVMM, FileMR provides the following benefits:

- It minimizes the amount of translation done at the NIC, reducing the load on the NIC's translation cache and improving hit rate by 3.8× - 340×.
- It simplifies memory protection by using existing file access control lists instead of RDMA's ad-hoc memory keys.
- It simplifies connection management by using persistent file names instead of ephemeral memory region IDs.
- It allows network-accessible memory to be moved or expanded without revoking permissions or closing a connection, giving the file system the ability to defragment and append to files.

The rest of this chapter is organized as follows. Section 5.1 describes the design of the FileMR. Section 5.2 describes our proposed changes to RDMA stack and RNICs, and Section 5.3

introduce two case studies. Section 5.4 provides experimental results. Section 5.5 discusses the applicability of the FileMR on real hardware and Section 5.6 concludes.

5.1 FileMR Overview

The FileMR extends the existing RDMA protocol to provide remote access to NVMM files. The FileMR is a new type of memory region that is also an NVMM file. The FileMR provides an efficient and coordinated memory management layer across the userspace application, the file system, and the RDMA networking stack.

The FileMR requires minimal changes to existing RDMA networking stack and does not rely on any specific design of the file systems. The FileMR can coexist with conventional RDMA memory regions, ensuring backward-compatibility.

As shown in Table 2.1, the FileMR system resolves the conflicting systems of RDMA and NVMM files that cause unnecessary restrictions and performance degradation through several innovations.

Merged control plane: With an RDMA FileMR, a client uses a *file offset* to address memory, instead of a virtual or physical address. The FileMR also leverages the naming, addressing, and permissions of the file system to streamline RDMA access.

Range-based address translation: The FileMR leverages the file system's efficient, extent-based layout description mechanism to reduce the amount of state the NIC must hold. As files are already organized in continuous extents, we extend this addressing mechanism to the RNIC, allowing the RNIC's pin-down cache to use a space efficient translation scheme to address large amounts of RDMA accessible memory.

The rest of this section continues as follows. We begin be describing the FileMR abstraction and range-based address translation. Then, we describe the system architecture required to support our new abstraction.

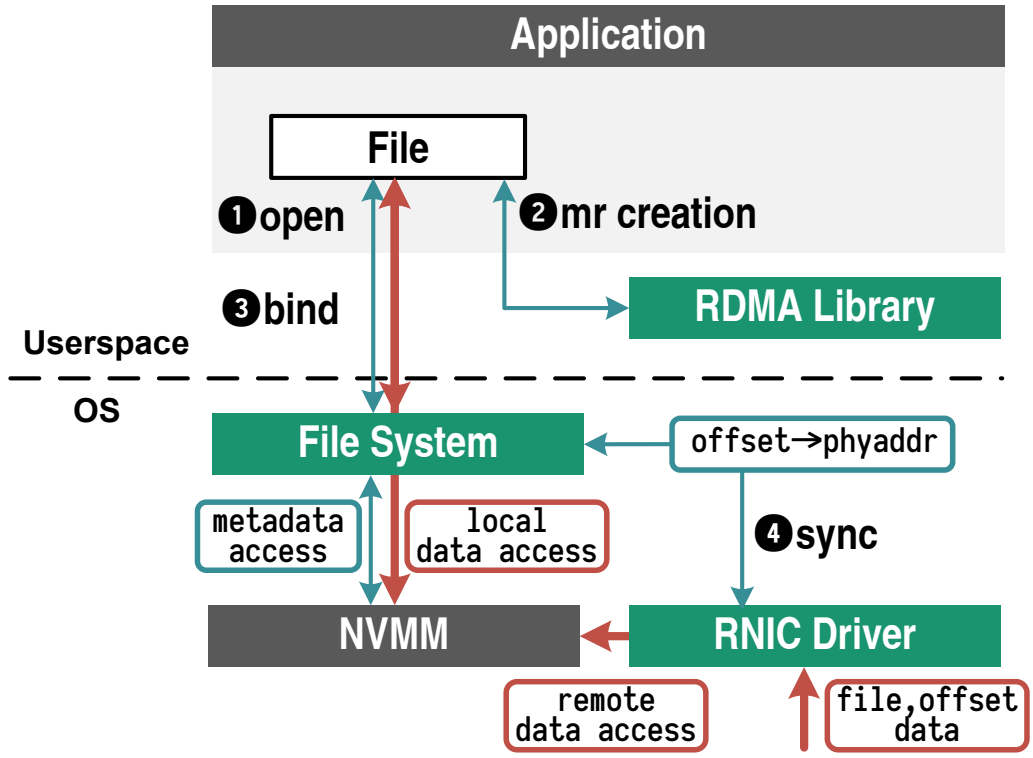

Figure 5.1: **FileMR: Control path and data path..** The user application communicates with the RDMA libraries and file system in control path (green), and access local and remote NVMM directly in data path (red).

5.1.1 FileMR

Our new abstraction, the FileMR, is an RDMA memory region that is also an NVMM file. This allows the RDMA and NVMM control planes to interoperate smoothly. RDMA accesses to the FileMR are addressed by file offset, and the file system manages the underlying file's access permissions, naming, and allocation as it would any file. NVMM files are always backed by physical pages managed by the file system, so, when using a FileMR, the RDMA subsystem can simply reuse the translation, permission, and naming information already available in the file system metadata for the appropriate checks and addressing.

Figure 5.1 shows an overview of metadata and data access with FileMR. For metadata, before creating a FileMR, the application opens the backing file with the appropriate permissions

(step ❶). Next, the application creates the FileMR (step ❷) and *binds* (step ❸) the region to a file, which completes the region's initialization. Binding the FileMR to the file produces a *filekey*, analogous to an rkey, that remote clients can use to access the FileMR. Once the FileMR is created and bound to a backing file, the file system will keep the file's addressing information in sync with the RNIC (step ❹).

For data access to a remote FileMR and its backing NVMM file, applications use the FileMR (with the filekey to prove its permissions) and a *file offset*. The RNIC translates between file offsets and physical addresses using translation information provided by the file system. In addition to one-sided read and write verbs to the FileMR, we introduce a new one-sided `append` verb that grows the region. When sending the append verb, the client does not include the remote address, and the server handles it like the an one-sided write with address equal to current size of FileMR. It then updates the FileMR size and and notifies the file system. As an optimization, to prevent faulting on every append message, the file system can pre-allocate translation entries beyond the size of a file. Even while the backing file is opened and accessible via a FileMR, local applications can continue to access it using normal file system calls or mmaped addresses — any change to the file metadata will be propagated to the RNIC.

5.1.2 Range-based Address Translation

NVMM file systems try to store file data in large, linear extents in NVMM. FileMR uses *range-based address translation* within the MTT and pin-down cache through a *RangeMTT* and *range pin-down* cache, respectively. This change is a significant departure from traditional RDMA page-based addressing. Unlike page-based translations, which translate a virtual to physical address using sets of fixed size pages, range-based translation (explored used previously in CPU-side translation [9, 33, 53, 80]) maps a variably sized virtual address range to physical address. Range-based address translation is useful when addressing large linear memory regions (which NVMM file systems strive to create) and neatly leverages the preexisting extent-based file organization.

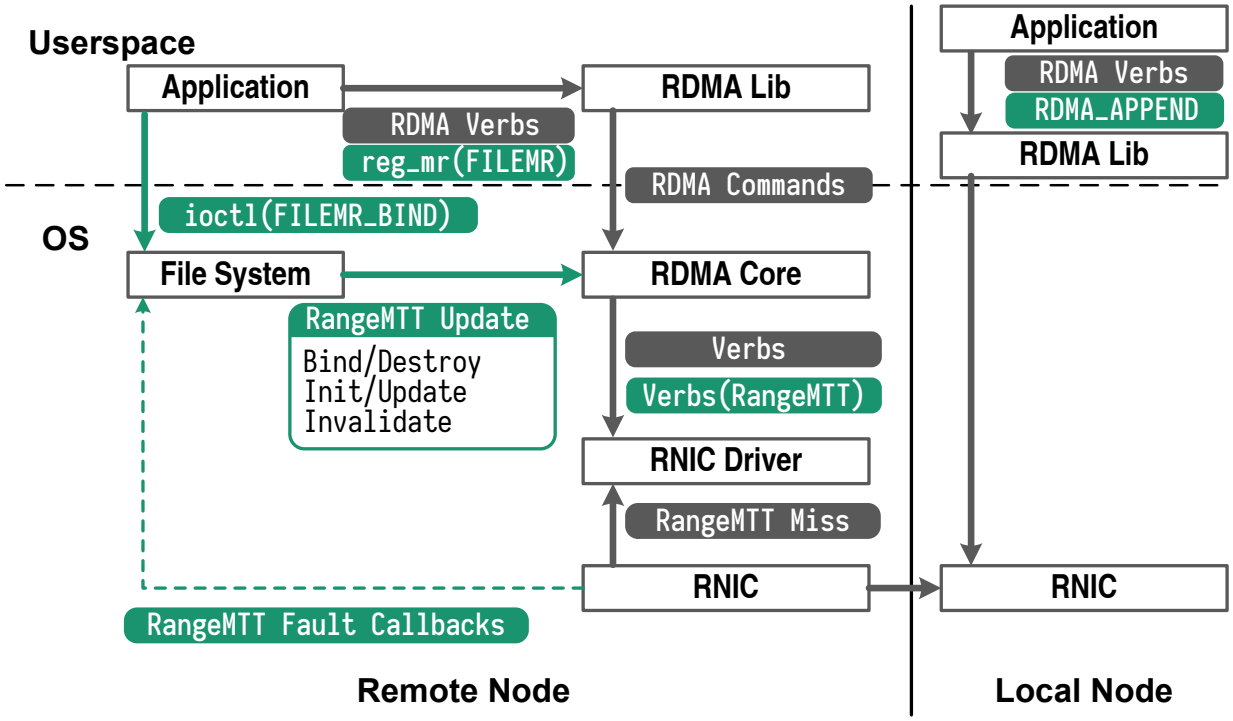

Figure 5.2: **Overview of FileMR components..** Implementing FileMR requires changes in file system, RDMA stack and hardware (in green).

For the FileMR, range-based address translation has two major benefits: both the space required to store the mapping and the time to register a mapping does not scales with the number of variable-sized extents rather than with the number of fixed size pages Registering a page in the MTT and pin-down cache takes about 5 μs, and this process cannot be parallelized. As a result, a single core can only register memory at 770 MB/s with 4 kB pages. For NVMMs on the order of terabytes, the result registration time will be unacceptably long.

5.1.3 Design Overview

The implementation of the FileMR RDMA extension requires coordination and changes across several system components: the file system, the RNIC, the core RDMA stack, and the application. Figure 5.2 shows the vanilla RDMA stack (in grey) along with the necessary changes to adopt FileMR (in green).

To support the FileMR abstraction, the file system is required to implement the `bind()` function to associate a FileMR and a file, and, when necessary, notify the RDMA stack (and eventually the RNIC's RangeMTT and pin-down cache) when bound file's metadata changes via callbacks (see Table 5.1). These callbacks allow the RNIC to maintain the correct range-based mappings to physical addresses for incoming RDMA requests. Note that the concept of a file system is loosely defined for the FileMR, so long as it invokes the correct callbacks to the RNIC. FileMR can be integrated with a kernel file system, a userspace file system, or an userspace NVMM library that accesses raw NVMM (also known as device-DAX) and provides naming.

Optionally, the file system can also register a set of callback functions triggered when RNIC cannot find a translation for an incoming address. This process is similar to on-demand paging [61, 63] and is required to support our new append verb, which both modifies the file layout and writes to it.

Supporting the FileMR abstraction also requires changes to the RNIC hardware. With our proposed RangeMTT, RNIC hardware and drivers would need to adopt range-based addressing within both the MTT and pin-down cache. Hardware range-based addressing schemes [9, 39, 53, 80] can be used to implement range-based address lookup. In our experiments we simulate these changes using a software RNIC (see Section 5.2).

The FileMR also adds incremental, backwards compatible changes to the RDMA interface itself (see Table 5.2). It adds a new access flag for memory region creation to identify the creation of a FileMR. After its creation, the FileMR is marked as a being in an unprovisioned state — the subsequent `bind()` call into the file system will allocate the FileMR's translation entries in the RangeMTT (via the `cm_bind` callback from the file system). The `bind()` method can be implemented with an `ioctl()` (for kernel-level file systems) or a library call (for user-level file systems). The FileMR also adds the new one-sided RDMA `append` verb. Converting existing applications to use FileMRs is easy as the application need only change its region creation code.

Table 5.1: **File system to RNIC callbacks for FileMRs.** These callbacks are used by the file system to notify the RDMA stack and RNIC that file layouts (and consequently address mappings) have changed.

API	Description
`cm_bind()`	To notify RNIC of new bound file
`cm_init()`	To initialize RangeMTT entries
`cm_update()`	To update a RangeMTT entry
`cm_invalidate()`	To invalidate a RangeMTT entry
`cm_destroy()`	To destroy a file binding

Table 5.2: **New/changed RDMA methods.** These methods in the RDMA interface are new or have new flags under the FileMR system.

API	Description
`bind()`	Binds an opened file to FileMR
`ibv_reg_mr()`	Creates a FileMR with `FILEMR` flag
`ibv_post_send()`	Posts append w/ `APPEND` flag (`uverb`)
`ib_post_send()`	Posts append w/ `APPEND` flag (`kverb`)

5.2 FileMR Implementation

We implemented the FileMR and RangeMTT for both the kernel space and userspace RDMA stack in Linux, and our implementations support the callbacks described in Table 5.1 and the changed methods in Table 5.2. The kernel implementation is based on Linux version 4.18, and userspace implementation is based on rdma-core (userspace) packages shipped with Ubuntu 18.04. Table 5.3 summarizes our implementation of FileMR with RangeMTT.

For our FileMR implementation on the NIC side, our implementation is based on a software-based RNIC: Software RDMA over Converged Ethernet (Soft-RoCE) [5, 54]. Soft-RoCE is a software RNIC built on top of ethernet's layer 2 and layer 3. It fully implements the ROCEv2 specification. Future research could work to build a FileMR compatible RNIC in real hardware.

To implement our RangeMTT, we followed the design introduced in Redundant Memory Mappings [53]: each FileMR points to a b-tree that stores offsets and lengths, and we use the offsets as index. All RangeMTT entries are page-aligned addresses, since OS can only manage virtual memory in page granularity.

Unlike page-aligned RangeMTT, FileMR supports arbitrary sizes and allows sub-page

Table 5.3: **Summary of FileMR implementation.** This table shows the components modified to implement FileMR. The first column indicates the change is in kernel space (K) or user space (U).

	Item	Description	LOC
FileMR Implementation on RDMA stack			
K	`ibcore`	Range-based TLB	367
K	`ibverbs`	Kernel verbs	293
U	`libibverbs`	Userspace verbs	53
FileMR support for RNIC			
K	`rxe`	Soft-RoCE: device driver	752
U	`librxe`	Soft-RoCE: userspace driver	142
FileMR support for file system			
K	`nova`	a NVMM-aware file system	455
Applications adapting FileMR			
U	`novad`	allowing remote NVMM access	285
U	`libpmemlog`	NVMM log library	198

files/objects. Each RangeMTT entry consists a page address, a length field and necessary bits. These entries are non-overlapping and can have gaps for sparse files.

To support the append verb, the FileMR allows translation entries beyond its size. The append is one-sided but does not specify remote server addresses in the WR. On the server side, the RNIC always attempts to DMA to the current size of the FileMR and increase its size on success. When the translation is missing, the server can raise a IO page fault when IOMMU is available and a file system routine will be called to fulfill the faulty entries. Alternatively, if such support is unavailable, the server signals the client via a message similar to a receiver not ready (RNR) error.

Like most hardware-based RNICs, Soft-RoCE manages the MTT entries as a flat array of 64-bit physical addresses, with lookup compleixty of $O(1)$. For FileMR with a range pin-down cache miss, the entry lookup will traverse the registered data structures with higher time complexity ($O(log(n))$).

Soft-RoCE does not have a pin-down cache since the mappings are in DRAM. To emulate the pin-down cache, we built a 4096-entry 4-way associative cache to emulate the traditional pin-down cache, and 4096-entry, 4-way associative range pin-down cache for FileMR. Each range translation entry consists of a 32 bit page address and a 32-bit length, which allows a maximal FileMR size of 16 TB (4 kB pages) or 8 PB (2 MB pages).

We adapted two applications to use FileMR. For a kernel file system, our implementation is based on NOVA [111], a full-fledged kernel space NVMM-aware file system with good performance. We added 455 lines of code to implement codebase. We also adapted the FileMR to libpmemlog, part of pmdk [81], a user-level library that manages local persistent objects, to build a remotely accessible persistent log. It took 198 lines of code to implement.

5.3 Case Studies

In this section we demonstrate the utility of our design with our two case studies. In Section 5.3.1, we demonstrate how to use FileMR APIs to enable remote file accesses with consistent addressing for local and remote NVMM. In Section 5.3.2, we extend libpmemlog [81], a logging library designed for local persistent memory into a remotely accessible log, demonstrating how FileMR can be applied to userspace libraries.

5.3.1 Remote File Access in NOVA

In this section, we demonstrate an example usage of our FileMR by extending a local NVMM file system (NOVA [111]). By combining the NVMM file system, RDMA, and our new FileMR abstraction, we can support fast remote file accesses that entirely bypass the kernel.

NOVA is a log-structured POSIX-compliant local NVMM file system. In NOVA, each file is organized as a persistent log of variably sized extents, where the extents reside on persistent memory. The file data is allocated and maintained by the file system through per-cpu free lists.

To handle metadata operations on the remote file system, we designed a lightweight user-level daemon `novad`. The daemon opens the file to establish an FileMR. It also receives any metadata updates (e.g. directory creation) from remote applications communicated through a traditional RDMA memory region and two-sided send/receive verbs and applies them the local file system. NOVA manages file metadata updates and file layout in NVMM.

On the client side, an application opens the file remotely by communicating with `novad` and

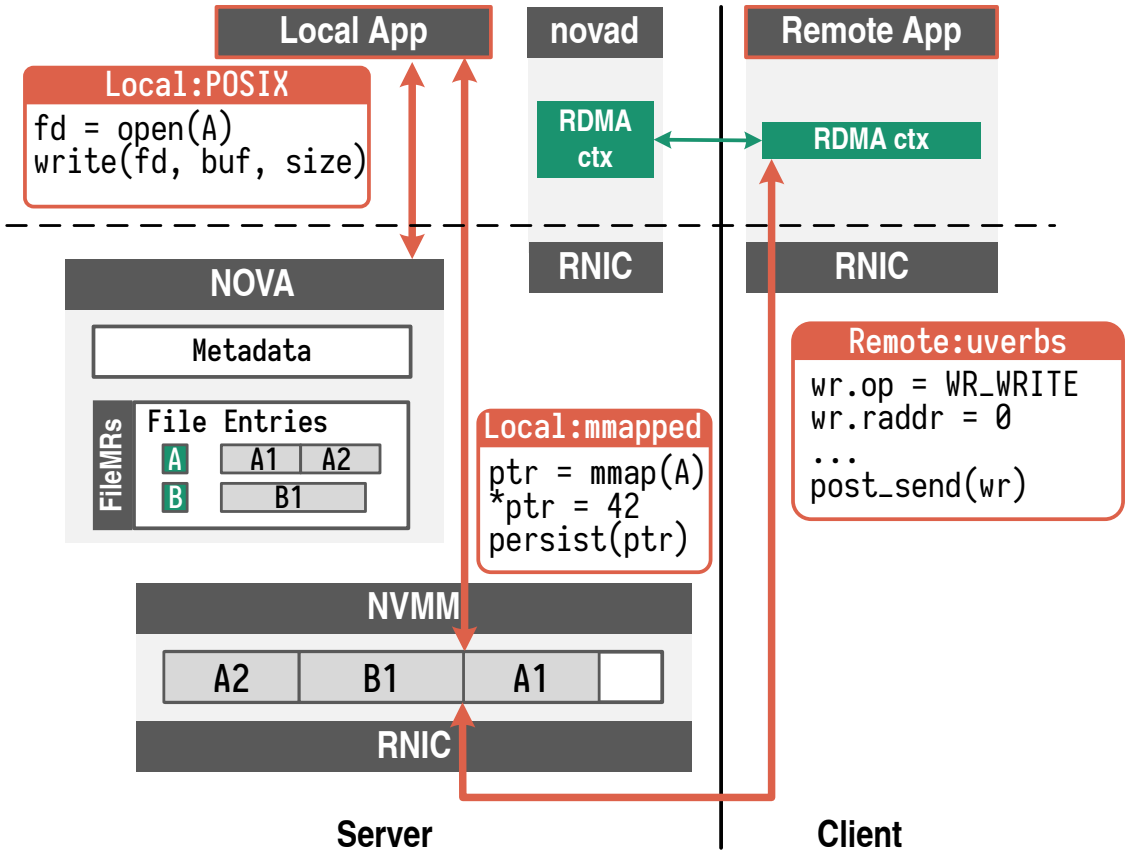

Figure 5.3: **Enabling remote NOVA accesses using FileMR..** Using FileMR, remote file accesses share a similar interface over RDMA as local NVMM accesses.

receiving the filekeys. It can then send one-sided RDMA verbs to directly access remote NVMM. At the same time, applications running locally can still access the file with traditional POSIX I/O interface, or map the file to its address space and issue loads and stores instructions.

Our combined system can also easily handle data replication. By using several FileMRs, we can simply duplicate a verb (with the same or different `filekes` depending on the file system implementation) and send to multiple hosts, without considering the physical address of the files (so long as their names are equivalent).

5.3.2 Remote NVMM Log with libpmemlog

The FileMR abstraction only requires that the backing "file system" to appropriately implement the `bind()` method, RNIC callbacks, and have access to raw NVMM. For instance, a FileMR can be created by an application having access to the raw NVMM device. In this section, we leverage this flexibility build a remote NVMM log based on `libpmemlog`.

We modify the allocator of `libpmemlog` to use the necessary FileMR callbacks — that is, whenever memory is allocated or freed for the log the RNIC's RangeMTT is updated. The client appends to the log with the new `append` verb. On the server side, when the FileMR size is within the mapped RangeMTT, the RNIC can perform the translation while bypassing the server application. If not, a range fault occurs and the library expands the region by allocating and mapping additional memory.

5.4 Evaluation

In this section, we evaluate the performance of the FileMR. First, we measure control plane metrics such as registration cost, memory utilization of the FileMR, as well as the efficiency of RangeMTT. Then we evaluate application-level data plane performance from our two case studies and compare FileMR-based applications with existing systems.

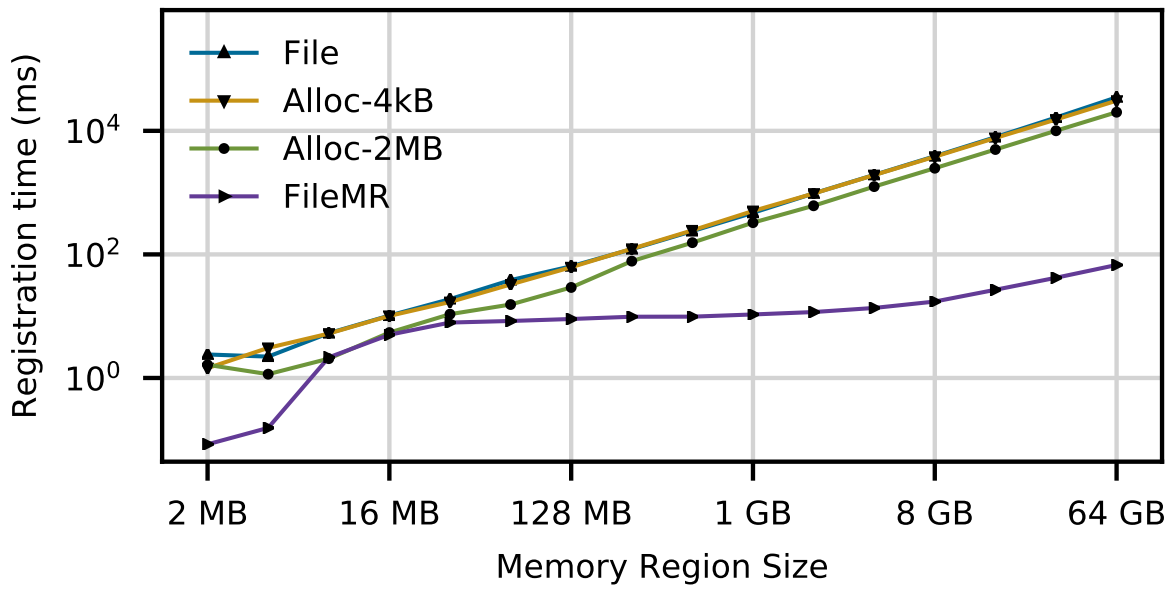

Figure 5.4: **FileMR registration time..** This figure shows the time consumed to register a fixed size memory region.

5.4.1 Experimental Setup

We run our FileMR on servers configured to emulate persistent memory with DRAM. Each node has two Intel Xeon (Broadwell) CPUs with 10 cores and 256 GB of DRAM, with 64 GB configured as an emulated NVMM devices. We setup Soft-RoCE on an Intel X710 10GbE NIC connected to a switch.

5.4.2 Registration Overhead

Allocated Regions

We measure the time consumed in memory region registration using FileMRs versus conventional user-level memory regions backed by NOVA with 4 kB pages and anonymous buffers with 4 kB and 2 MB pages. This experiment demonstrates the use case when an application allocates and maps a file directly, without updating its metadata. For FileMR, we also include the time generating

Table 5.4: **Workload Characteristics.** Description of workloads to evaluate registration cost of FileMR and pin-down cache hit rate.

Workload	#Th	# Files	Avg. Size	Description
Fileserver	20	7980	6.82 MB	File I/Os
Varmail	20	4511	11.3 kB	Random I/Os
Redis	12	2	561 MB	Write + Append
SQLite	4	1	109 MB	Write + Sync

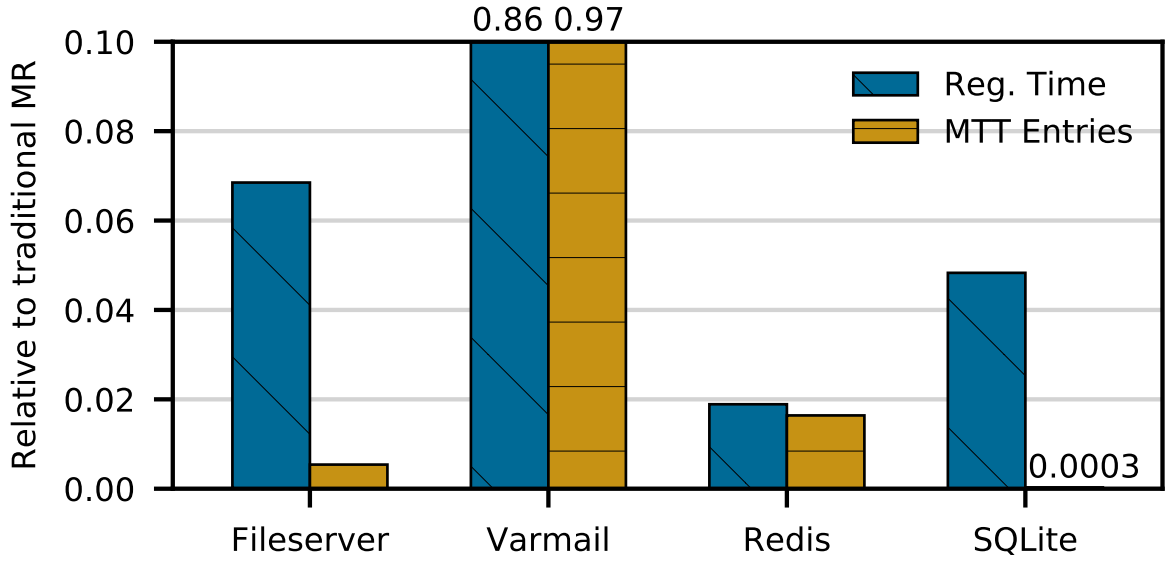

Figure 5.5: **FileMR on fragmented files..** Compared to traditional MRs, FileMR saves registration cost and RNIC translation entries.

range entries from NOVA logs, which happens when an application opens the file for the first time.

As shown in Figure 5.4, registering a large size memory region consumes a non-trival amount of time. It takes over 30 seconds to register a 64 GB persistent (**File**) and volatile (**Alloc-4K**) memory region with 4 kB pages. Using hugepages (**Alloc-2M**) reduces the registration cost to 20 seconds, while it only takes 67 ms for FileMR (three orders of magnitude lower). The FileMR registration time increases modestly as the region size grows mainly due to the internal fragmentation of the file system allocator.

For small files, NOVA only creates one or two extents for the file, while conventional MRs still interacts with the virtual memory routines of the OS, causing overhead.

Fragmented Files

Another common usage for NVMM file systems is to use POSIX I/O for control plane operations, and POSIX or mmapped I/O for data plane operations. As a result, the file can often be fragmented. To evaluate the fragmented case, we first warmed up the file system using four I/O intensive workloads: varmail and fileserver workloads in filebench [100], Redis [83] and SQLite [93] (using MobiBench [48]). Once the file system warm and fragmented, we created memory regions over all files in NVMM. Table 5.4 summarizes the workloads.

As shown in Figure 5.5, running FileMR over the fragmented file still shows dramatic improvement on region registration time and memory consumption for MTT entries. Fileserver demonstrates the case with many files, where FileMR only creates 0.5% of the entries of traditional memory regions, and requires only 6.8% of the registration time. For a metadata-heavy workload (Varmail), FileMR only reduces the number of entries by 3% (due to the heavy internal fragmentation and small file size – 11.3 kB), but it still saves 20% on registration time. Redis is a key value store that persists an append-only file on the I/O path, and flush the database asynchronously — little internal fragmentation means that it requires 2% of the space and time of traditional memory regions. Similarly, SQLite also uses logging, resulting in little fragmentation and drastic space and time savings.

5.4.3 Translation Cache Effectiveness

The performance degradation of RDMA over large NVMM is mainly caused by the pin-down cache misses (Figure 2.7). Since Soft-RoCE encapsulates RDMA messages in UDP and accesses all RDMA state in DRAM, we cannot measure the effectiveness of the cache through end-to-end performance.

Instead, we measure the cache hit ratio of our emulated pin-down cache and range pin-down cache for FileMR. We collect the trace of POSIX I/O system calls for workloads described in Table 5.4, and replay them through one-sided RDMA verbs to a remote host.

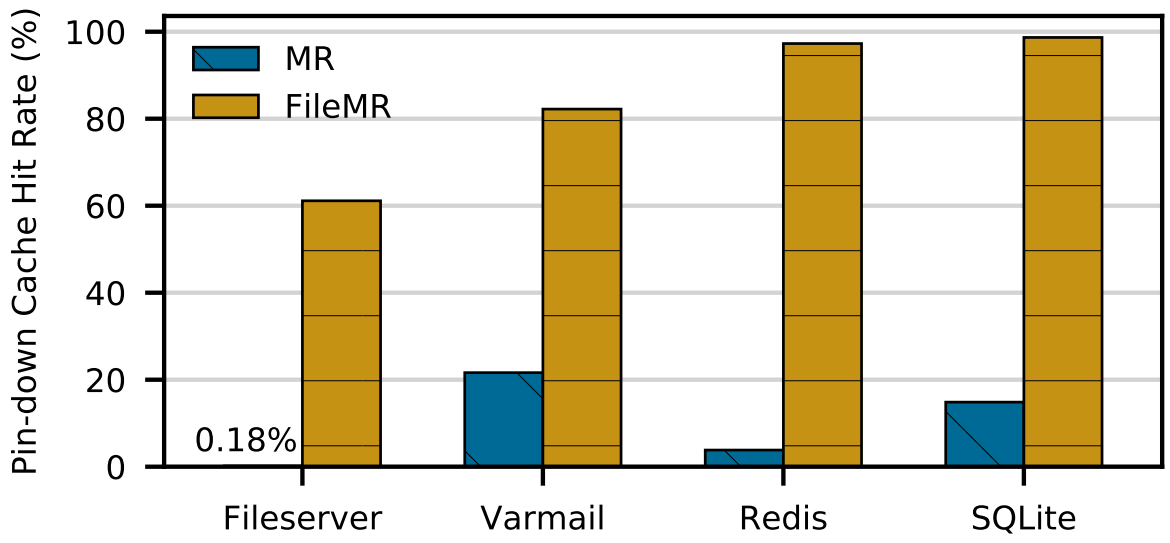

Figure 5.6: **Translation cache effectiveness..** FileMR signifacntly increases the effectiveness of the pin-down cache.

Figure 5.6 shows the evaluation result. Our emulated range-based pin-down cache is significantly more efficient (3.8× - 340×) than the page-based pin-down cache for fragmented files. For large allocated files with a few entries, the range-based pind-down cache shows near 100% hit rate (not shown in figure).

5.4.4 Accessing Remote Files

To evaluate the data path performance, we let a client access files on a remote server running `novad` (introduced in Section 5.3.1). The client issues random 1 kB writes using RDMA write verbs, and we measure the latency between the client application issuing the verb and the remote RNIC DMAs to the target memory address (`memcopy` for Soft-RoCE).

We compare FileMR both with mmapped local accesses and other distributed systems that provide distributed storage access. We implemented datapath-only versions of Mojim-Emu, LITE-Emu [104] and Orion-Emu for Soft-RoCE. All these systems avoid translation overhead by sending physical addresses on the wire.

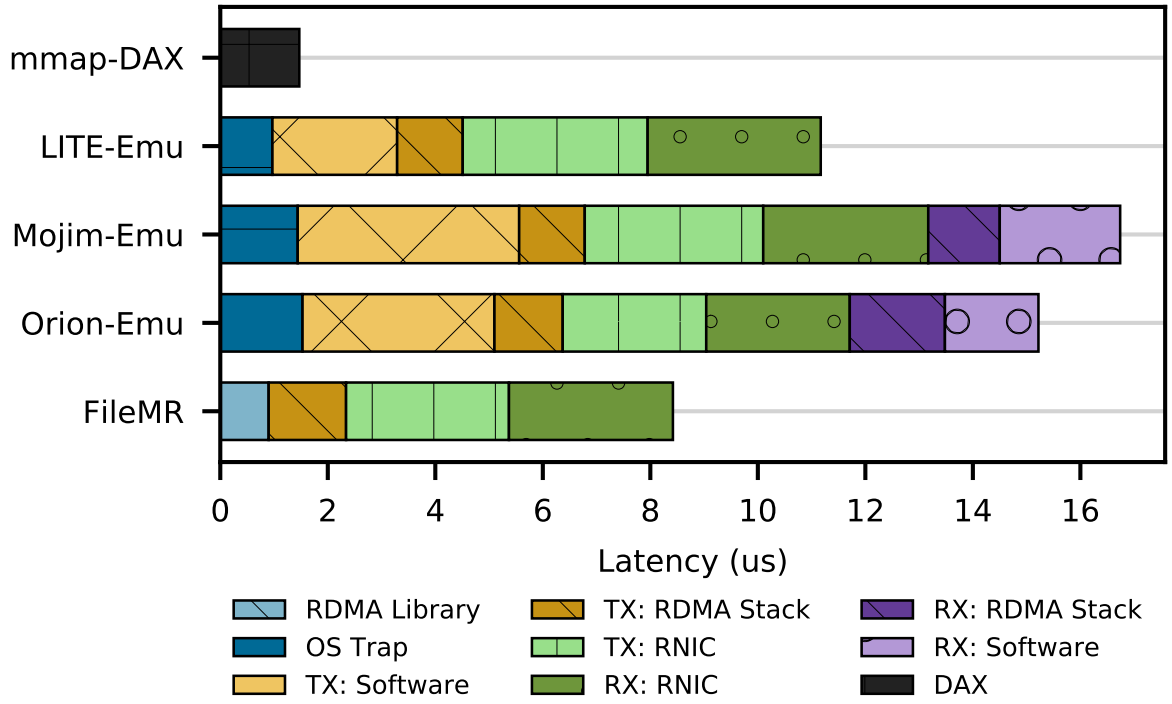

Figure 5.7: **Latency breakdown of accessing remote file..** FileMR can access remote file location without indirection on data path.

In Figure 5.7, we show the latency breakdown of these systems, omitting the latency of UDP packet encapsulation and delivery for RDMA-based systems, which dominates the end-to-end latency. It only takes 1.5 μs to store and persist 4 kB data to local NVMM, FileMR has lower latency than other systems because it eliminates the need for any indirection layer (`msync()` syscall for Mojim, shared memory write for LITE, and POSIX write for Orion).

5.4.5 Accessing Remote NVMM logs

Finally, we evaluate our introduction of the new append verb using our implementation of remote log implementation, introduced in Section 5.3.2. We compare to a baseline `libpmemlog` on using local NVMM (bypassing the network), as well as with logging within the HERD RPC RDMA library [50, 51].

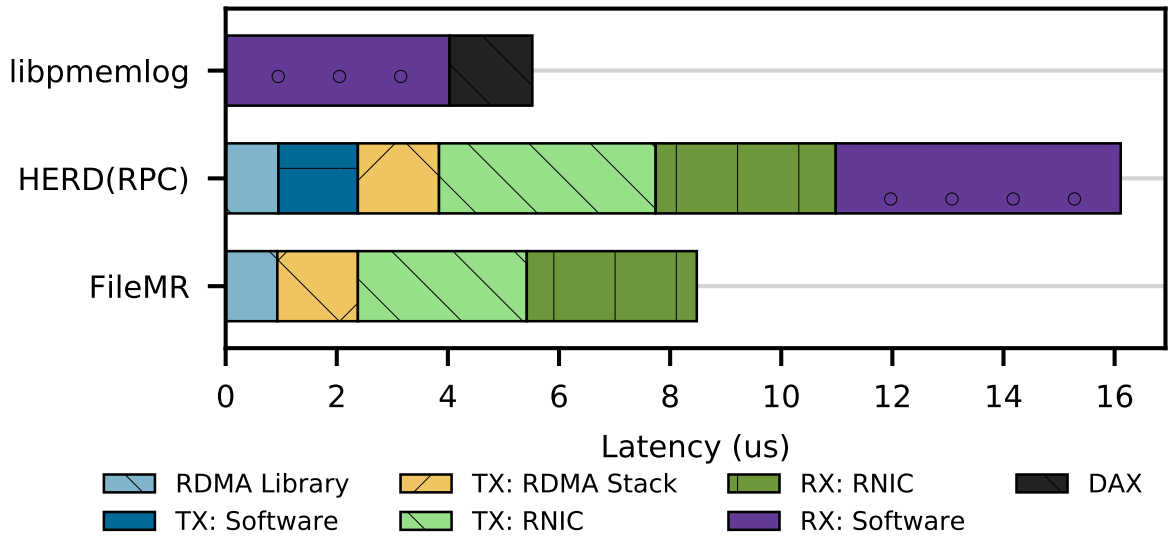

Figure 5.8: **Latency breakdown of accessing remote log..** With the append verb, Remote logging with FileMR achieves similar performance to local ones.

Figure 5.8 shows the latency breakdown of creating a 64 Byte log entry. It takes 5.5 μs to log locally with `libpmemlog`. FileMR adds 53% overhead for remote vs. local logging, while the HERD RPC-based solution adds 192% overhead.

5.5 Discussion

The current FileMR implementation relies on software-based RDMA protocols. In this section, we discuss the potential benefits and challenges of applying FileMR on hardware and other deeper changes to the RDMA protocol. We consider them to be the future work.

5.5.1 Data Persistence

For local NVMM, a store instruction is persistent once data is evicted from CPU last-level cache (via cache flush instructions and memory fences). A mechanism called asynchronous DRAM refresh (ADR) ensures that the write queue on a memory controller is flushed to non-volatile storage

in the event of a power failure. There are no similar mechanisms in the RDMA world since ADR does not extend to PCIe devices. Making the task even more difficult, modern NICs are capable of placing data into CPU cache using direct cache access (DCA) [42], conceivably entirely bypassing NVMM.

The current workaround to ensure RDMA write persistency is to disable DCA and issue another RDMA read to the last byte of a pending write [28], forcing the write to complete and write to NVMM. Alternatively, the sender request that the receiving CPU purposefully flush data it received; either embedding the flush request in an extra send verb, or the immediate field of a write verb.

A draft standards working document has proposed adding a `commit` [97] verb to the RDMA protocol to solve the write persistency problem. A `commit` verb lists memory locations that need to be flushed to persistence. When the remote RNIC receives a commit verb, it ensures the all listed locations are persistent before acknowledging completion of the verb.

With the introduction of FileMR, implementing data persistence is simplified since there is no longer need to track modified locations at the client: the RNIC already maintains information about FileMRs. A commit verb can simply request that all updates to a file before persistent. In this way, a commit verb becomes analogous to an `fsync` syscall to a local file. Even better, since the commit needs little state, a commit flag can be embedded to the latest write verb, reducing communication overhead.

5.5.2 Connection Management

Orion and several other NVMM-based storage systems [65, 90, 104] store data across nodes, or use a model similar to distributed shared memory. This model requires establishing N^2 connections for N servers with NVMM. For user-level applications, the reliable connection transport enforces the protection domain within the scope of a process. Thus a cluster with N servers running p processes will establish $N^2 * p^2$ connections.

Existing works [1, 1, 96] reduces this complexity by sharing queue pairs [96], multiplexing connections [56] or dynamically allocating connections [1] to reduce the RDMA states. These optimizations work well for MPI-based applications, but it is challangeing to implement them for NVMM applications, especially for application with fine-grained access control. In particular, a file system supports complex access control schemes, which may disallow sharing and multiplexing.

With the FileMR, the file permission is checked at the bind step, and so each server only requires a single connection to handle *all* file system requests, drastically reducing the amount of state required to store on the RNIC.

5.5.3 Page Fault on NIC

Some ethernet and RNICs support page fault or on-demand paging [61, 63] (ODP). When using ODP, instead of pinning memory pages, the IOMMU marks the page as not present in I/O virtual addresses. The RNIC will raise an interrupt to operating system when attemping DMA to a page that is not present. The I/O page fault handler then fills the entry with the mapping.

With ODP, a page fault is very expensive. In our experiment, it takes 475 μs to fulfill an I/O page fault and complete a 8-byte RDMA write on an Mellanox CX-4 RNIC. In contrast, it only takes 1.4 μs to complete when the mapping is cached in the RNIC. In general, ODP requires extensive prefetching to mitigate the expensive page fault.

The design of FileMR is orthogonal to ODP, though it leverages it for the `append` verb. Fortunately, the file system is the ideally situated to provide better locality by prefetching ranges based on the file access pattern.

5.5.4 Multicast

The existing RDMA protocol only supports multicast with restrictions such as using unreliable datagram and two-sided verbs. For NVMM, data replication is essential for reliability and availability. Mojim, Orion and other existing RDMA-aware systems on distributed NVMM [11, 68, 90]

replicate NVMM data in multiple stages or create verbs and send to multiple hosts. As a result, when an application replicating data to N replicas, either the end-to-end latency will increase by $N\times$, or the bandwidth will reduce to $1/N$.

Our current implementation of FileMR still requires a connection, yet with proper network support this requirement can disappear. The RDMA payload on wire is mostly stateless (except for file key) and can be multicasted by the current network infrastructure, allowing a single RDMA verb to modify multiple copies of the same file. With a proper FileMR subscription mechanism, hardware-based multicast support can simplify data replication over RDMA.

5.6 Summary

The conflicting systems on metadata management between NVMM and RDMA causes expensive translation overhead and prevents the file system from changing its layout. This chapter introduces two modifications to the existing RDMA protocol: the FileMR and range-based translation, thereby providing an abstraction that combines memory regions and files. It drastically improves the performance of RDMA-accessible NVMMs by eliminating extraneous translations, while conferring other benefits to RDMA including more efficient access permissions and simpler connection management.

Acknowledgments

This chapter contains material from "FileMR: Rethinking RDMA Networking for Scalable Persistent Memory", by Jian Yang, Joseph Izraelevitz and Steven Swanson, which is submitted for publication. The dissertation author was the primary investigator and first author of this paper.

Bibliography

[1] Dynamically Connected Transport. https://www.openfabrics.org/images/eventpresos/workshops2014/DevWorkshop/presos/Monday/pdf/05_DC_Verbs.pdf.

[2] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[3] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novaković, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 775–787, Boston, MA, 2018.

[4] Peter A. Alsberg and John D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering (ICSE '76)*, San Francisco, California, October 1976.

[5] InfiniBand Trade Association. SOFT-RoCE RDMA Transport in a Software Implementation, 2015.

[6] Jens Axboe. Flexible I/O Tester, 2017. https://github.com/axboe/fio.

[7] Katelin Bailey, Luis Ceze, Steven D. Gribble, and Henry M. Levy. Operating system implications of fast, cheap, non-volatile memory. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systemsi (HotOS '13)*, Napa, California, May 2011.

[8] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. Corfu: a shared log design for flash clusters. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, pages 1–1, Berkeley, CA, USA, 2012. USENIX Association.

[9] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient virtual memory for big memory servers. In *Proceedings of the 40th Annual*

International Symposium on Computer Architecture, ISCA '13, pages 237–248, New York, NY, USA, 2013. ACM.

[10] Brain Beeler. Intel optane dc persistent memory module (pmm). 2019. Accessed 04/18/2019.

[11] Jonathan Behrens, Sagar Jha, Ken Birman, and Edward Tremel. Rdmc: A reliable rdma multicast for large objects. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 71–82. IEEE, 2018.

[12] Peter J Braam. The Lustre storage architecture, 2004.

[13] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, Washington, November 2006.

[14] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Cascais, Portugal, October 2011.

[15] Brent Callaghan, Theresa Lingutla-Raj, Alex Chiu, Peter Staubach, and Omer Asad. NFS over RDMA. In *Proceedings of the ACM SIGCOMM workshop on Network-I/O convergence: experience, lessons, implications*, pages 196–208. ACM, 2003.

[16] Mallikarjun Chadalapaka, Hemal Shah, Uri Elzur, Patricia Thaler, and Michael Ko. A Study of iSCSI Extensions for RDMA (iSER). In *Proceedings of the ACM SIGCOMM Workshop on Network-I/O Convergence: Experience, Lessons, Implications*, NICELI '03, pages 209–219. ACM, 2003.

[17] Dave Chinner. xfs: updates for 4.2-rc1, 2015. `http://oss.sgi.com/archives/xfs/2015-06/msg00478.html`.

[18] Mosharaf Chowdhury, Srikanth Kandula, and Ion Stoica. Leveraging endpoint flexibility in data-intensive clusters. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM (SIGCOMM '13)*, Hong Kong, China, August 2013.

[19] Byung-Gon Chun, Frank Dabek, Andreas Haeberlen, Emil Sit, Hakim Weatherspoon, M. Frans Kaashoek, John Kubiatowicz, and Robert Morris. Efficient replica maintenance for distributed storage systems. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI '06)*, San Jose, California, May 2006.

[20] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference*

on Architectural Support for Programming Languages and Operating Systems, ASPLOS '11, pages 105–118, New York, NY, USA, 2011. ACM.

[21] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 133–146, New York, NY, USA, 2009. ACM.

[22] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[23] Paolo Costa, Hitesh Ballani, Kaveh Razavi, and Ian Kash. R2c2: A network stack for rack-scale computers. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 551–564. ACM, 2015.

[24] Patrice Couvert. High speed IO processor for NVMe over fabric (NVMeoF). *Flash Memory Summit*, 2016.

[25] Alex Davies and Alessandro Orsaria. Scale out with glusterfs. *Linux Journal*, 2013(235):1, 2013.

[26] Guiseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swami Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, Stevenson, Washington, October 2007.

[27] Mingkai Dong and Haibo Chen. Soft Updates Made Simple and Fast on Non-volatile Memory. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 719–731, Santa Clara, CA, 2017. USENIX Association.

[28] Chet Douglas. RDMA with PMEM, Software mechanisms for enabling access to remote persistent memory. `http://www.snia.org/sites/default/files/SDC15_presentations/persistant_mem/ChetDouglas_RDMA_with_PM.pdf`. Accessed 2019-01-05.

[29] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, New York, NY, USA, 2014. ACM.

[30] Dave Dunning, Greg Regnier, Gary McAlpine, Don Cameron, Bill Shubert, Frank Berry, Anne Marie Merritt, Ed Gronke, and Chris Dodd. The virtual interface architecture. *IEEE micro*, (2):66–76, 1998.

[31] R. Fackenthal, M. Kitagawa, W. Otsuka, K. Prall, D. Mills, K. Tsutsui, J. Javanifard, K. Tedrow, T. Tsushima, Y. Shibahara, and G. Hush. A 16Gb ReRAM with 200MB/s write and 1GB/s read in 27nm technology. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*, pages 338–339, Feb 2014.

[32] Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in Globally Distributed Storage Systems. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, Canada, December 2010.

[33] J. Gandhi, V. Karakostas, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Ünsal. Range translations for fast virtual memory. *IEEE Micro*, 36(3):118–126, May 2016.

[34] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, New York, October 2003.

[35] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43. ACM, 2003.

[36] Google Inc. Google Sparse Hash. `http://goog-sparsehash.sourceforge.net`.

[37] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data (SIGMOD '96)*, New York, New York, June 1996.

[38] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 202–215. ACM, 2016.

[39] Swapnil Haria, Mark D Hill, and Michael M Swift. Devirtualizing memory in heterogeneous systems. In *ACM SIGPLAN Notices*, volume 53, pages 637–650. ACM, 2018.

[40] Lisa Hellerstein, Garth A. Gibson, Richard M. Karp, Randy H. Katz, and David A. Patterson. Coding Techniques for Handling Failures in Large Disk Arrays. *Algorithmica*, 12(2):182–208, August 1994.

[41] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in windows azure storage. In *Proceedings of the USENIX Annual Technical Conference (USENIX '12)*, Boston, Massachusetts, June 2012.

[42] Ram Huggahalli, Ravi Iyer, and Scott Tetrick. Direct cache access for high bandwidth network i/o. In *Computer Architecture, 2005. ISCA'05. Proceedings. 32nd International Symposium on*, pages 50–59. IEEE, 2005.

[43] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '10)*, Boston, Massachusetts, June 2010.

[44] Intel. Add Support for New Persistent Memory Instructions. `http://www.lwn.net/Articles/619851`.

[45] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual. `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf`.

[46] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 427–442, New York, NY, USA, 2016. ACM.

[47] James Pinkerton. The Future of Computing: The Convergence of Memory and Storage through Non-Volatile Memory (NVM). Storage Industry Summit, San Jose, California, Jan 2014.

[48] Sooman Jeong, Kisung Lee, Jungwoo Hwang, Seongjin Lee, and Youjip Won. AndroStep: Android Storage Performance Analysis Tool. In *Software Engineering (Workshops)*, volume 13, pages 327–340, 2013.

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

[50] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using RDMA efficiently for key-value services. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 295–306. ACM, 2014.

[51] Anuj Kalia Michael Kaminsky and David G Andersen. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference*, page 437, 2016.

[52] Brent ByungHoon Kang, Robert Wilensky, and John Kubiatowicz. The hash history approach for reconciling mutual inconsistency. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS '03)*, Providence, Rhode Island, May 2003.

[53] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. Redundant memory mappings for fast access to large memories. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 66–78, New York, NY, USA, 2015. ACM.

[54] Gurkirat Kaur and Manju Bala. Rdma over converged ethernet: A review. *International Journal of Advances in Engineering & Technology*, 6(4):1890, 2013.

[55] Takayuki Kawahara. Scalable Spin-Transfer Torque RAM Technology for Normally-Off Computing. *Design & Test of Computers, IEEE*, 28(1):52–63, Jan 2011.

[56] Matthew J Koop, Jaidev K Sridhar, and Dhabaleswar K Panda. Scalable mpi design over infiniband using extended reliable connection. In *2008 IEEE International Conference on Cluster Computing*, pages 203–212. IEEE, 2008.

[57] Amit Kumar and Ram Huggahalli. Impact of cache coherence protocols on the processing of network traffic. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '07)*, Chicago, Illinois, Dec 2007.

[58] Leslie Lamport. Paxos Made Simple. *ACM SIGACT News*, 32(4):18–25, November 2001.

[59] Sergey Legtchenko, Nicholas Chen, Daniel Cletheroe, Antony IT Rowstron, Hugh Williams, and Xiaohan Zhao. Xfabric: A reconfigurable in-rack network for rack-scale computers. In *NSDI*, volume 16, pages 15–29, 2016.

[60] Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos K. Aguilera, and Michael Walfish. Detecting Failures in Distributed Systems with the Falcon Spy Network. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Cascais, Portugal, October 2011.

[61] Ilya Lesokhin, Haggai Eran, Shachar Raindel, Guy Shapiro, Sagi Grimberg, Liran Liss, Muli Ben-Yehuda, Nadav Amit, and Dan Tsafrir. Page fault support for network controllers. *ACM SIGOPS Operating Systems Review*, 51(2):449–466, 2017.

[62] Sheng Li, Hyeontaek Lim, Victor W Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G Andersen, O Seongil, Sukhan Lee, and Pradeep Dubey. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 476–488. ACM, 2015.

[63] Liran Liss. On demand paging for user-level networking, 2013.

[64] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L. Scott, Sam H. Noh, and Changhee Jung. iDO: Compiler-directed failure atomicity for nonvolatile memory. In *51st IEEE/ACM International Symposium on Microarchitecture*, MICRO '18, October 2018.

[65] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: an RDMA-enabled Distributed Persistent Memory File System. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 773–785, Santa Clara, CA, 2017. USENIX Association.

[66] Mellanox. Mellanox OFED for Linux User Manual, 2017.

[67] Mellanox Technologies. Rdma aware networks programming user manual. `http://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf`.

[68] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. Atomic In-place Updates for Non-volatile Main Memories with Kamino-Tx. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 499–512, New York, NY, USA, 2017. ACM.

[69] Micron. 3D XPoint Technology, 2017. `http://www.micron.com/products/advanced-solutions/3d-xpoint-technology`.

[70] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *USENIX Annual Technical Conference*, pages 103–114, 2013.

[71] Jeffrey C. Mogul, Eduardo Argollo, Mehul Shah, and Paolo Faraboschi. Operating system support for nvm+dram hybrid main memory. In *The Twelfth Workshop on Hot Topics in Operating Systems (HotOS XII)*, Monte Verita, Switzerland, May 2009.

[72] MongoDB, Inc. MongoDB, 2017. `https://www.mongodb.com`.

[73] Suman Nath, Haifeng Yu, Philip B. Gibbons, and Srinivasan Seshan. Subtleties in tolerating correlated failures in wide-area storage systems. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI '06)*, San Jose, California, May 2006.

[74] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, and Marcos Aguilera. Storm: A fast transactional dataplane for remote data structures. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, SYSTOR '19, pages 97–108, New York, NY, USA, 2019. ACM.

[75] Tayo Oguntebi, Sungpack Hong, Jared Casper, Nathan Bronson, Christos Kozyrakis, and Kunle Olukotun. FARM: A prototyping environment for tightly-coupled, heterogeneous architectures. In *Proceedings of the 2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '10, pages 221–228, Washington, DC, USA, 2010. IEEE Computer Society.

[76] Stan Park, Terence Kelly, and Kai Shen. Failure-atomic msync (): a simple and efficient mechanism for preserving the integrity of durable data. In *Proceedings of the EuroSys Conference (EuroSys '13)*, Prague, Czech Republic, April 2013.

[77] David Patterson, Garth Gibson, and Randy Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD Conference on the Management of Data (SIGMOD '88)*, Chicago, Illinois, June 1988.

[78] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecture*, ISCA '14, pages 265–276, Piscataway, NJ, USA, 2014. IEEE Press.

[79] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, Saint-Malo, France, October 1997.

[80] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. Colt: Coalesced large-reach TLBs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 258–269. IEEE Computer Society, 2012.

[81] pmem.io. Persistent Memory Development Kit, 2017. `http://pmem.io/pmdk`.

[82] S. Raoux, G.W. Burr, M.J. Breitwisch, C.T. Rettner, Y.C. Chen, R.M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H. L Lung, and C.H. Lam. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(4.5):465–479, July 2008.

[83] redislabs. Redis, 2017. `https://redis.io`.

[84] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Pond: The oceanstore prototype. In *Proceedings of the 2nd USENIX Symposium on File and Storage Technologies (FAST '03)*, San Francisco, California, April 2003.

[85] Antony Rowstron and Peter Druschel. Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, October 2001.

[86] Andy Rudoff. Processor Support for NVM Programming. `http://www.snia.org/sites/default/files/AndyRudoff_Processor_Support_NVM.pdf`. Accessed 2019-01-05.

[87] Andy Rudoff. Deprecating the pcommit instruction. 2016. Accessed 04/18/2019.

[88] Arthur Sainio. NVDIMM: Changes are Here So What's Next? In *In-Memory Computing Summit 2016*, 2016.

[89] Julian Satran, Kalman Meth, C Sapuntzakis, M Chadalapaka, and E Zeidner. Internet small computer systems interface (iSCSI), 2004.

[90] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 323–337. ACM, 2017.

[91] SNIA. Nvm programming model. `https://www.snia.org/tech_activities/standards/curr_standards/npm`.

[92] David Spence, Jon Crowcroft, Steven Hand, and Tim Harris. Location based placement of whole distributed systems. In *Proceedings of the 2005 ACM Conference on Emerging Network Experiment and Technology (CoNEXT '05)*, Toulouse, France, October 2005.

[93] SQLite. SQLite, 2017. `https://www.sqlite.org`.

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

[95] Maomeng Su, Mingxing Zhang, Kang Chen, Zhenyu Guo, and Yongwei Wu. RFP: When RPC is Faster Than Server-Bypass with RDMA. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 1–15. ACM, 2017.

[96] Sayantan Sur, Lei Chai, Hyun-Wook Jin, and Dhabaleswar K Panda. Shared receive queue based scalable mpi design for infiniband clusters. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pages 10–pp. IEEE, 2006.

[97] Talpey and Pinkerton. RDMA Durable Write Commit. `https://tools.ietf.org/html/draft-talpey-rdma-commit-00`. Accessed 2019-01-05.

[98] Haodong Tang, Jian Zhang, and Fred Zhang. Accelerating Ceph with RDMA and NVMeoF. In *14th Annual OpenFabrics Alliance (OFA) Workshop*, 2018.

[99] Wittawat Tantisiriroj, Seung Woo Son, Swapnil Patil, Samuel J Lang, Garth Gibson, and Robert B Ross. On the duality of data-intensive file system design: reconciling HDFS and PVFS. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 67. ACM, 2011.

[100] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX; login*, 41, 2016.

[101] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-Based Service Level Agreements for Cloud Storage. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Farmington, Pennsylvania, November 2013.

[102] Hiroshi Tezuka, Francis O'Carroll, Atsushi Hori, and Yutaka Ishikawa. Pin-down cache: A virtual memory management technique for zero-copy communication. In *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International... and Symposium on Parallel and Distributed Processing 1998*, pages 308–314. IEEE, 1998.

[103] Animesh Trivedi, Patrick Stuedi, Bernard Metzler, Clemens Lutz, Martin Schmatz, and Thomas R Gross. Rstore: A direct-access DRAM-based data store. In *Distributed Computing Systems (ICDCS), 2015 IEEE 35th International Conference on*, pages 674–685. IEEE, 2015.

[104] Shin-Yeh Tsai and Yiying Zhang. LITE: Kernel RDMA support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 306–324. ACM, 2017.

[105] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, California, December 2004.

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

[107] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

[108] Matthew Wilcox. Add support for NV-DIMMs to ext4, 2014. `https://lwn.net/Articles/613384/`.

[109] Jiesheng Wu, Pete Wyckoff, and Dhabaleswar Pandac. PVFS over InfiniBand: Design and performance evaluation. In *2003 International Conference on Parallel Processing, 2003. Proceedings.*, pages 125–132. IEEE, 2003.

[110] Xiaojian Wu and A.L.N. Reddy. Scmfs: A file system for storage class memory. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*, Nov 2011.

[111] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, February 2016. USENIX Association.

[112] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System. In *26th Symposium on Operating Systems Principles (SOSP '17)*, pages 478–496, 2017.

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

[114] Ming Zhong, Kai Shen, and Joel Seiferas. Replication degree customization for high availability. In *Proceedings of the EuroSys Conference (EuroSys '08)*, Glasgow, Scotland UK, March 2008.