Designing Scalable Storage Systems for Non-Volatile Memory

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Shashank Gugnani, B.E.

Graduate Program in Department of Computer Science and Engineering

The Ohio State University

2020

Dissertation Committee:

Xiaoyi Lu, Advisor

Feng Qin

Yang Wang

Mike Bond

Abstract

Data is growing at an exponential rate. As a consequence, cloud microservices and scientific applications are becoming more complex and computationally demanding. This requires the design of new large-scale storage systems that can handle this ever-increasing load and provide a robust and responsive service platform. New memory technologies such as PCM, STT-RAM, and 3D-XPoint offer persistence with unprecedented performance. Devices are available in the form of NVMe SSDs and NVDIMMs (or PMEM), both of which can be accessed quickly between nodes using RDMA networks. These devices are being increasingly adopted in datacenters and HPC clusters to accelerate data storage and analytics. These technologies have the potential to change the fundamental design principles of storage systems. Unfortunately, existing data storage systems that have been designed around the performance characteristics of archaic hardware such as spinning disks and Ethernet adapters are unable to make efficient use of the technology. This is because these modern storage devices have very low latency and the high software overhead of traditional designs prevents full utilization of available bandwidth. Moreover, prior work has been unable to hide the weak and complicated persistence properties of PMEM while fully exploiting its byte-addressability and performance. In this thesis, we rethink the assumptions and design paradigms of traditional storage systems and propose new algorithms that would have been considered impractical on previous generation hardware. To solve the new challenges with storage system design, we propose Navi Store, a generic

storage sub-system with a focus on improving the performance of three classes of applications – online analytics, scientific HPC, and cloud microservices. Navi Store's design is based on cross-layer holistic thinking and includes four key components.

Specifically, we design Arcadia, a generic replicated log on PMEM to make it easier to use. Arcadia hides the idiosyncratic behavior of PMEM while preserving required guarantees of atomicity, integrity, concurrency, and monotonicity, allowing programmers to fully realize the benefits of PMEM. Using this log design, we propose DIPPER, a new decoupled model which can be used to build fast crash-consistent persistent data structures on PMEM. We show that this model can be used to design a quiescent-free storage system with low tail latency. Such a system is particularly useful for cloud microservices where latency service level objectives are desirable and inconsistent user experience directly results in loss of revenue. We further propose microfs, an abstraction for designing distributed ephemeral storage with synchronization-free control and data planes to reduce software overhead. Based on this abstraction, we design a system to optimize checkpoint IO on supercomputers using NVMe-over-Fabrics and show how it can improve the performance and progress rates of scientific HPC applications. Finally, we design hardware-based arbitration schemes for application-oblivious QoS that utilize cross stream arbitration algorithms implemented in hardware for providing latency and throughput service level objectives to applications in a multi-tenant cloud setting. Our research has shown that the proposed designs are better than state-of-the-art in several aspects including performance, scalability, fault tolerance, and quality of service. The main contribution of our work is new models, algorithms, data structures, and abstractions which enable both programmers and end-users to efficiently utilize new hardware technologies.

Dedicated to my parents for their constant support and love

Acknowledgments

This work was made possible through the love and support of several people who stood by me, through the many years of my doctoral program and all through my life leading to it. I would like to take this opportunity to thank them all.

My advisor – Dr. Xiaoyi Lu, for his guidance and support throughout my doctoral program. I have been able to grow, both personally and professionally, through my association with him. He has not only been a great mentor and advisor, but also a good friend. I admire his drive, commitment, and the energy he puts into his work. His enthusiasm and 'never give up' attitude will always be an inspiration.

My family – my parents, Samir and Arti Gugnani, for giving me unconditional love, freedom, and support. They have always believed in me and encouraged me to pursue my goals, no matter how hard they may be.

My friends – I am very happy to have met and become friends with Jahanzeb Hashmi, Mohammadreza (Mamzi) Bayatpour, Rajarshi Biswas, Haseeb Javed, Moniba Keymanesh, Bita Kamizi, and Shahriar Hooshmand. It is because of these meaningful friendships that I kept my sanity and made lasting memories during my doctoral program that I will cherish forever.

My colleagues – Finally, I would like to thank all my colleagues at the Network-Based Computing Lab, the Parallel and Distributed Systems Lab, and IBM Research, who have helped me through my graduate studies – Sourav Chakraborty, Ammar Ahmad Awan, Dipti

Shankar, Ching-Hsiang Chu, Jeff Smith, Jie Zhang, Mark Arnold, Haiyang Shi, Tianxi Li, Arjun Kashyap, Yujie Hui, Scott Guthridge, Ted Anderson, Frank Schmuck, and Deepavali Bhagwat. This work would remain incomplete without their support and contribution.

This thesis is supported in part by NSF grants #CCF-1822987, #IIS-1447804, #ACI-1664137, #IIS-1636846, and #CNS-1513120.

Vita

2015–present ..... Graduate Research Associate, The Ohio State University, U.S.A

2015–present ..... Ph.D., Computer Science, The Ohio State University, U.S.A

2019–2020 ..... Research Intern, IBM Research, U.S.A

2011–2015 ..... B.E., Computer Science, BITS-Pilani, India

Publications

S. Gugnani and X. Lu, "DStore: A Fast, Tailless, and Quiescent-Free Object Store for PMEM", under review

S. Gugnani, S. Guthridge, T. Anderson, F. Schmuck, D. Bhagwat, and X. Lu, "Taming Evil Persistent Memory with a Replicated Log", under review

S. Gugnani, T. Li, and X. Lu, "NVMe-CR: A Scalable Ephemeral Storage Runtime for Checkpoint/Restart with NVMe-over-Fabrics". *IEEE International Parallel and Distributed Processing Symposium (IPDPS'21)*

S. Gugnani, A. Kashyap, and X. Lu, "Understanding the Idiosyncrasies of Real Persistent Memory". *International Conference on Very Large Data Bases (VLDB'21)*

T. Li, D. Shankar, **S. Gugnani**, and X. Lu, "RDMP-KV: Designing Remote Direct Memory Persistence based Key-Value Stores with PMEM". *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'20)*

A. Kashyap, **S. Gugnani**, and X. Lu, "Impact of Commodity Networks on Storage Disaggregation with NVMe-oF". *International Symposium on Benchmarking, Measuring, and Optimizing (Bench'20)*

S. Gugnani, X. Lu, and D. K. Panda, "Analyzing, Modeling, and Provisioning QoS for NVMe SSDs". *IEEE/ACM International Conference on Utility and Cloud Computing (UCC'18)*

S. Gugnani, X. Lu, H. Qi, L. Zha, and D. K. Panda, "Characterizing and Accelerating Indexing Techniques on Distributed Ordered Tables". *IEEE International Conference on Big Data (BigData'17)*

S. Gugnani, X. Lu, and D. K. Panda, "Swift-X: Accelerating OpenStack Swift with RDMA for Building an Efficient HPC Cloud". *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'17)*

S. Gugnani, X. Lu, and D. K. Panda, "Designing Virtualization-aware and Automatic Topology Detection Schemes for Accelerating Hadoop on SR-IOV-enabled Clouds". *IEEE International Conference on Cloud Computing Technology and Science (CloudCom'16)*

X. Lu, D. Shankar, **S. Gugnani**, H. Subramoni, and D. K. Panda, "Impact of HPC Cloud Networking Technologies on Accelerating Hadoop RPC and HBase". *IEEE International Conference on Cloud Computing Technology and Science (CloudCom'16)*

X. Lu, D. Shankar, **S. Gugnani**, and D. K. Panda, "High-Performance Design of Apache Spark with RDMA and its Benefits on Various Workloads". *IEEE International Conference on Big Data (BigData'16)*

S. Gugnani, X. Lu, and D. K. Panda, "Performance Characterization of Hadoop Workloads on SR-IOV-enabled Virtualized InfiniBand Clusters". *IEEE/ACM International Conference on Big Data Computing, Applications and Technologies (BDCAT'16)*

Fields of Study

Major Field: Computer Science and Engineering

Table of Contents

List of Tables

Table **Page**

List of Figures

Figure **Page**

Chapter 1: Introduction

Modern datacenters, high-performance computing (HPC) clusters, and public clouds enable researchers and users from diverse domains to solve complex problems. Scientific domain-specific applications such as fluid dynamics, structural mechanics, discrete event simulation, and 3D-modeling need to be run at an extremely large scale to solve practical problem sizes. Web 2.0 [136] and emerging Web 3.0 applications, such as hosted services, web applications, and social networking need to run at even larger scales to satisfy global demands. Both kinds of applications place extreme stress on the data storage and analytics components of storage system middleware. This requires the design of new large-scale systems that can handle this ever-increasing load and provide a robust and responsive service platform. Fast storage and analysis of data, in particular, is a must for achieving adequate response times.

New non-volatile memory technologies such as phase change memory (PCM) [101], resistive RAM (ReRAM) [10], spin-transfer torque RAM (STT-RAM) [165], and 3D-XPoint [61] offer unprecedented performance with persistence. These technologies are available in the form of non-volatile memory express (NVMe [131]) solid state drives (SSDs) accessible via peripheral component interconnect express (PCIe) or NVDIMMs accessible through CPU loads and stores, commonly known as persistent memory (PMEM). Further, high-performance remote direct memory access (RDMA) based networks allow

fast remote access to these storage devices. Newly introduced NVMe SSDs based on flash and 3D-XPoint memory [118] offer unprecedented performance and concurrency. For example, new Intel SSDs offer write bandwidth up to 3GB/s [91], an order of magnitude faster than serial AT attachment (SATA) SSDs. These devices are ideal for use in modern systems to build compact high-density storage arrays. These arrays can be stacked together to build a disaggregated storage cluster. With the introduction of the NVMe-over-Fabrics (NVMf) standard [130], low latency remote access to these arrays can be effectively provided. The NVMf standard can take advantage of fast RDMA-enabled networks in modern systems to reduce the network overheads of remote access. Guz *et al.* [60] have shown that at the application level, NVMf has $\sim$10% overhead compared to local IO. These emerging non-volatile memory and networking technologies have the potential to change the design principles of storage systems.

These devices are being increasingly adopted in datacenters and HPC clusters to accelerate data storage and analytics [11, 128]. Unfortunately, existing data storage runtimes have been designed considering the performance characteristics of archaic hardware, such as spinning hard disks and Ethernet networks. Applying the design principles of storage systems developed for these devices to those developed for modern devices leads to inefficient utilization of available hardware resources. In addition, the features and high-performance offered by new hardware make it possible to enable designs previously considered impractical. For example, persistent memory exposed as NVDIMM can be used to build fast persistent data structures, which were considered impractical with hard disks. There is a fundamental need for fault-tolerance in storage systems and ubiquitous use of data structures. Hence, the design of fast persistent data structures will have a significant

impact on the performance of storage systems. This thesis intends to rethink the assumptions and design paradigms of traditional storage systems to take full advantage of new hardware, and propose new algorithms which were considered to be impractical with previous generation hardware.

With emerging memory and networking technologies (PMEM, NVMe, and RDMA) being adopted in datacenters, the challenge is to design new distributed storage systems which can efficiently take advantage of new hardware. From the storage system perspective, four dimensions must be considered – performance, scalability, fault-tolerance, and quality of service (QoS). To satisfy these dimensions, existing storage solutions take a black-box approach to system design. They expect that storage devices are accessible through standardized interfaces, such as the kernel block driver, or the POSIX application programming interface (API). Similarly, networking devices are assumed to be only accessible through the sockets API. Multi-layered software-heavy systems are common in the storage community. However, to attain a synergistic solution, a tighter integration between hardware and software is required. Software should not view hardware as a black box, but rather should take a white-box approach and make optimal use of internal hardware features. The design of storage systems should, therefore, involve cross-layer holistic thinking.

In this thesis, we explore three broad challenges that must be solved to provide the four required dimensions in storage systems – (1) lowering software latency to realize full hardware potential, (2) utilizing new hardware features to design more efficient software, and (3) re-evaluating the design principles of storage systems based on the performance characteristics of new hardware. In the sections that follow, we clearly articulate the specific challenges that we seek to solve in this thesis and present detailed descriptions of the designs we propose to solve them.

1.1 Problem Statement

In this section, we discuss the main problem statement of this thesis.

PMEM Logging. The performance critical nature of logging has led many researchers to propose new log implementations on PMEM as a way of improving end-to-end system performance. Several works have looked at designing both unreplicated and replicated logging protocols on PMEM [14, 70, 75, 95, 162, 172, 178, 182]. Unreplicated protocols, like PMDK's libpmemlog [75] and FLEX [178] are by design not resilient to node or device failures. This makes it hard to apply them in systems with reliable and replicated primary storage. The log is a single point of failure in persistent systems, so its resilience must be at least that of primary storage; therefore, replicated logging is essential for a robust storage system. Unfortunately, prior work on replicated logging places more trust in PMEM hardware's ability to retain data than it should. For instance, Query Fresh [172] does not have any mechanism to handle memory corruption (resulting from software bugs) and undetected media errors, so it is possible to read corrupted data. Tailwind [162] assumes the presence of DMA capable battery-backed DRAM buffers to guarantee persistence. Such special hardware support is impractical in a production system and we are not aware of any commodity system offering such support.

Providing concurrency for log writes is another dimension where prior work falls short. It is challenging to provide concurrent writes to the log because multi-threaded concurrency is non-deterministic and storage systems typically require log writes to preserve a total order to guarantee correctness. Without a total order, consistency of systems cannot be guaranteed on recovery. PMEM makes this problem even more challenging because cache lines may be evicted implicitly at any time. So, if log data is accessed concurrently and in-place, there are no ordering guarantees for data persistence. Prior work [75, 178]

completely sidesteps this issue by fully isolating log writers to preserve the total log order but at the cost of limited or no concurrency.

Reducing Persistence Overhead. Storage systems can be broadly classified into three types – cached, uncached, and decoupled. In a cached system, a volatile cache is used to improve performance by caching a part of the persistent data. This cache is tightly coupled with the persistent backend, i.e., the cache is needed to update the backend. A logical or physical log is used to ensure atomicity and durability of operations. Traditional systems used physical logging which suffered from the large size of log records. To improve logging performance, recent systems [71, 90, 113, 139, 146, 179, 180] have moved to logical or operation logging. Usually, both the cache and the log have limited space. Therefore, checkpoints are necessary to cleanup space in the log, or cache, or both. Herein lies the major problem with cached systems. Data from the cache is used to update the persistent backend and since the backend must be updated atomically, the cached pages cannot be modified until they are made persistent. As a result, during checkpoints the system either becomes temporarily unavailable or clients experience intolerable delay. This compromises tail latency or quiescent freedom, and sometimes both.

In an uncached system, all data is immediately persistent and nothing is cached. To ensure atomic updates, transactions are used. Since data is immediately persistent, checkpoints are not required. While uncached systems were considered impractical with disks and SSDs because of their slow write performance, they are gaining popularity in recent times due to the advent of PMEM. The fast performance of PMEM makes uncached system practical in production scenarios. Unfortunately, the overhead of transactions to atomically update data in PMEM is too high, resulting in performance being compromised. The main reason for this, as mentioned earlier, is that the CPU caches are not persistent and cache

lines can be implicitly evicted. To ensure correct ordering of updates, explicit cache flushes and store fences are needed. By avoiding the need for checkpoints, uncached systems can attain low tail latency and quiescent freedom but at the cost of performance. The overhead of transactions and cache flushes have been well studied [63, 125].

Decoupled systems are variants of cached systems where the persistent backend can be updated without using data in the cache. This can be done, for instance, by recording detailed information in a log, which can then be used to update the backend independent of the frontend operation. DudeTM [107] and NV-HTM [25] are two systems for PMEM that have explored a decoupled approach. These systems rely on expensive physical logging, as opposed to the more efficient logical logging. Further, they require special hardware support for consistency – hardware transactional memory (HTM). Therefore, we believe that these systems are neither performant nor widely applicable. Persimmon [190] is another system which proposes a decoupled design, however it does not provide any concurrency.

Accelerating Checkpoint IO. Application-level checkpointing is a common approach taken to provide insulation from inevitable system failures. In a disaggregated setup, NVMf can be used to reduce checkpoint overhead. However, to take full advantage of this new standard, a new approach is required which is not only transparent but is also able to expose the raw NVMf performance to end applications. This approach must reconsider the need to trap into the OS for every operation, instead providing unprivileged userspace storage access. It is thus important to design *userspace* storage runtimes which can take advantage of these resources and transparently improve application performance. In a disaggregated cluster setup, the most common form of storage runtime provided is a parallel filesystem such as Lustre [149] or GPFS [147]. In recent years, several distributed filesystems [12, 65, 72, 108, 163, 171] have been proposed to alleviate the bottlenecks in parallel

filesystems. However, we find that these systems are still not ideal for highly concurrent checkpoint IO. There are two primary reasons for this. First, these storage systems overlay multiple software layers over POSIX filesystems which decrease the peak attainable bandwidth. Second, the use of a global namespace restricts scalability and reduces efficiency of the storage system.

Quality of Service. In cloud environments, users expect a certain guarantee of service. Considering the new NVMe technology being introduced in enterprise clouds, it is only natural to ask whether a similar guarantee of service can be provided for this emerging hardware. In fact, this issue has been addressed to some extent in the NVMe standard itself. The standard includes provisions to enable request arbitration through mechanisms that are to be provided by hardware. However, there is limited knowledge of using these provisions to enable service guarantees in cloud environments. Prior research [9, 57, 58, 97, 117, 140, 151, 153, 188] has mostly focused on software-based provisions for service guarantees. While previous approaches [85, 155] have considered using such hardware provisions to provide some QoS to users, their approaches are not holistic. The designs proposed do not provide a complete solution for providing service-level agreement (SLA) based guarantees to users. For providing such a solution, there are two key requirements. First, the SLA provisioning should be completely application oblivious, i.e., it should be completely handled by the cloud provider based on the SLA negotiated by the user. Second, there must be mechanisms in place which allow for the provisioning of SLAs without violations. Achieving these requires a holistic approach.

The research proposed in this thesis takes on these fundamental challenges and tries to investigate the answers to the following research questions:

1. Can we re-evaluate the design principles of storage systems based on the performance characteristics of new hardware?

2. How can we reduce software overhead of storage systems to realize the full potential of hardware?

3. Can we design new algorithms and abstractions to take advantage of the features offered by next-generation hardware?

4. How can the end-users take advantage of the proposed designs and optimizations with minimal or no changes to their applications?

5. What are the application-level benefits that can be achieved through the proposed designs?

In this thesis, we address the aforementioned challenges and propose optimized solutions.

1.2 Research Framework

Figure 1.1 illustrates the research framework that we present to address the challenges highlighted above. The main contribution of this thesis is the design of a generic storage sub-system, called Navi Store[1]. In this section, we further elaborate on different components of the proposed sub-system and the specific challenges they address.

1. Can we re-evaluate the design principles of storage systems based on the performance characteristics of new hardware?

[1]Navi means new in Hindi

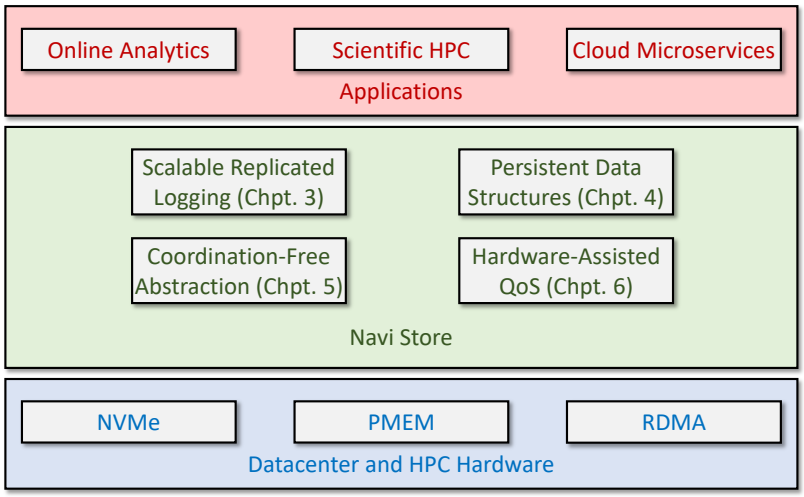


Figure 1.1 Research Framework

The performance characteristics of new hardware make it possible to rethink the design principles of storage systems. For example, we find that the low latency and byte addressability of PMEM makes it possible to design a storage system, where data is accessed in-place. Unfortunately, in-place access complicates data consistency. The weak and complicated persistence properties of PMEM make this process challenging. In this thesis, we present new approaches that make PMEM easier to use for programmers. Specifically, we explore the design of a generic replicated log on PMEM, called Arcadia. Arcadia implicitly handles atomicity, integrity, and replication of log records to reduce programmer burden. Our design has several novel aspects including concurrent log writes to PMEM with in-order commit, atomicity and integrity primitives for local and remote PMEM writes, and a frequency-based log force policy for providing low overhead persistence with guaranteed bounded loss of uncommitted records. We present Arcadia's design and implementation in

Chapter 3. Building upon this log design, we explore the design space of persistent data structures. We first discuss challenges with lock-free data structure design, then we propose a new decoupled approach for designing persistent data structures that can hide the persistence overheads of PMEM. We call this approach DIPPER and present it as a generic approach to design persistent data structures in Chapter 4. The unique performance characteristics of PMEM resulted in several novel ideas in both Arcadia and DIPPER to tame its idiosyncrasies.

2. How can we reduce software overhead of storage systems to realize the full potential of hardware?

   New hardware technologies provide very lower latency. For example, both NVMe SSDs and InfiniBand adapters provide latency at least an order of magnitude faster than hard-disks and Ethernet adapters. As a consequence, software latency now constitutes a significant percentage of overall system latency. This leaves hardware underutilized and increases total cost of ownership (TCO) of datacenters. Storage systems must be designed to peel away unnecessary software layers and directly access hardware (for example, by bypassing the kernel or by eliminating superfluous buffer copies). In this thesis, we present a design template for direct-access coordination-free filesystems, called `microfs`. `Microfs` is an abstraction that allows for synchronization-free control and data planes along with userspace direct-access to storage devices. Building upon this abstraction, we present `NVMe-CR`, an ephemeral userspace storage runtime for NVMf-enabled disaggregated clusters. A description of this abstraction and the design of `NVMe-CR` is presented in Chapter 5.

3. Can we design new algorithms and abstractions to take advantage of the features offered by next-generation hardware?

   Four design dimensions were considered as part of this thesis – performance, scalability, fault-tolerance, and QoS. New hardware offers several new features that can either directly or indirectly used to satisfy one or more of the four desired dimensions. For example, NVMe SSDs provide support for hardware-based arbitration schemes. This serves as a great base for providing fine-grained bandwidth guarantees in multi-tenant cloud environments. The challenge, however, is to provide service guarantees in an application-oblivious manner. Further, there must be some mechanism for cloud providers to accurately estimate a lower bound on bandwidth to be able to negotiate service level agreements (SLAs) with tenants. We show how existing runtimes can be modified for application oblivious QoS provisioning. We also theoretically model the arbitration mechanism available in NVMe and discuss how the model can assist cloud providers in SLA mapping and provisioning. More importantly, we find that the existing weighted round robin (WRR) scheme proposed in the NVMe standard is unable to provide accurate bandwidth guarantees. We then propose a deficit round robin (DRR) scheme which can provide required guarantees and show how it can be easily implemented in hardware. A detailed description of our hardware-assisted QoS work is provided in Chapter 6.

4. How can the end-users take advantage of the proposed designs and optimizations with minimal or no changes to their applications?

   The designs proposed in this thesis have been integrated into systems like `NVMe-CR`, DStore[2], SPDK [76], HBase [2], RocksDB [45], and Masstree [114]. For all systems,

[2]NVMe-CR and DStore were proposed in this thesis

the goal was to improve performance in an application-oblivious fashion. `NVMe-CR` is meant to be distributed as a shared library that implements a complete POSIX interface and can be preloaded to run unmodified MPI binaries. To run non-MPI applications, a simple two-line change to initialize and finalize the `NVMe-CR` runtime, is all that is required. The DStore framework exposes three standardized interfaces via its plugins – key-value (YCSB [34]), filesystem (POSIX through fuse [5]), and HDFS [21]. By using these plugins existing applications can be run unmodified over DStore. Similarly, our HBase, RocksDB, and Masstree implementations require no modification to the application interface and do not break existing applications. Finally, our QoS-aware SPDK runtime also does not require application modification. By only changing how applications are run (by setting their IO priority using `ionice`), users can take advantage of our QoS-aware designs.

5. What are the application-level benefits that can be achieved through the proposed designs?

   Our evaluation has shown that the design dimensions considered in this thesis (performance, scalability, fault-tolerance, and QoS) translate into application-level benefits as well. In particular, we optimized data storage and analytics through the proposed designs. Any application heavily reliant on these two activities is likely to reap significant benefits. To evaluate the effectiveness of proposed designs, we first conducted thorough empirical evaluation using micro-benchmarks, such as YCSB [34], iozone [24], and TPC-H [35]. Based on the performance results, we optimized and fine-tuned our designs iteratively to obtain the best possible performance. Finally, we took two production applications – AdMaster data analytics and CoMD scientific simulation, and showed how their scalability and performance can be drastically

improved. These observations lead us to conclude that other web 2.0/2.0 applications, cloud microservices, and long-running scientific HPC applications can realize similar improvements by leveraging our work.

Chapter 2: Background

2.1 Persistent Memory

PMEM technologies, such as PCM [101, 102], ReRAM [10], STT-RAM [165], 3D-XPoint [61], and memristor [158] have the potential to disrupt storage system design. PMEM allows applications to use byte-addressable *load* and *store* instructions to access data that is persistent across power cycles. PMEM has three orders of magnitude better latency and an order of magnitude better bandwidth compared to flash [121, 183]. It is evident that PMEM is a practical option for storing latency critical log data. Given that PMEM is an emerging technology and its cost is much higher than flash [93], we do not expect PMEM to replace flash as primary storage media in the foreseeable future. Instead, we expect that PMEM available in next generation systems will be used for special purposes, such as latency critical logging and metadata storage.

Despite its performance benefits, PMEM is hard to use correctly. Accessing persistent data in-place means that any update must be done in a crash-consistent manner. This is further complicated by the fact that data in CPU caches is not persistent: atomicity of a CPU store is only at 8-byte granularity, and cache lines can be evicted implicitly. The cache hierarchy may change the order in which data is persisted from what was desired, making the task of keeping data consistent challenging. Ensuring persistence requires explicit cache

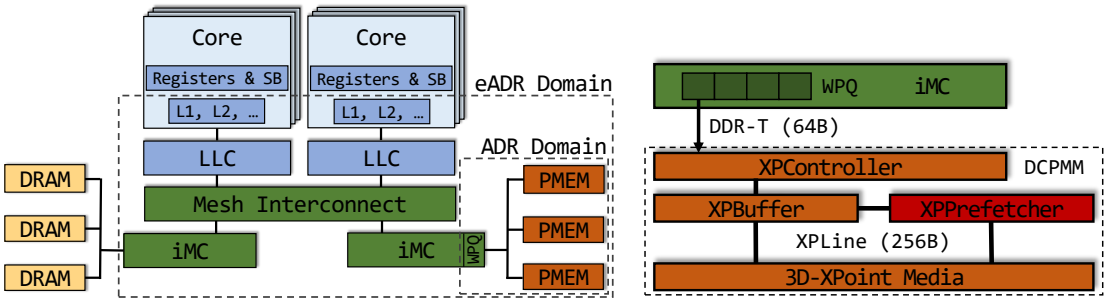


Figure 2.1 (left) Architecture of PMEM-enabled Systems, and (right) DCPMM Internal Architecture. *SB* is store buffer, *ADR* is asynchronous DRAM refresh, *eADR* is enhanced ADR, *WPQ* is write pending queue, and *iMC* is integrated memory controller.

flushes and store fences, while ensuring atomicity requires the use of logging. Further, PMEM is vulnerable to uncorrectable media errors and memory corruption, complicating data integrity. Any practical solution on PMEM must handle these issues with persistence, atomicity, and integrity, which requires non-trivial effort.

PMEM System Architecture. The left side of Figure 2.1 shows the typical architecture of PMEM-enabled systems. We expect the system to consist of one or more identical multi-core NUMA-enabled CPUs. Each CPU has local registers, store buffers, and caches. The last level cache (LLC) is shared across all cores in a CPU. Each CPU has its own memory (DRAM and PMEM) connected to other CPUs through a mesh interconnect. The PMEM system can be designed to support one of two persistence modes – ADR or eADR. In ADR persistence mode, the PMEM DIMMs and the write pending queues (WPQ) in the integrated memory controller (iMC) are part of the persistence domain. On power failure, all stores that have reached the ADR domain will be flushed to the PMEM DIMM. However, the CPU caches are not part of the persistence domain. So, any data left in the CPU cache will be lost in the event of power failure. In contrast, in eADR mode, the

CPU caches are also part of the persistence domain (they will be flushed to PMEM in case of power failure). So, data in cache can be considered to be persistent, but data in CPU registers and store buffers will still be lost.

Optane DCPMM. Optane DCPMM is Intel's PMEM solution. It has much higher capacity than DRAM and is currently available in 128, 256, and 512GB DIMM configurations. The right side of Figure 2.1 shows its internal architecture. Optane DIMMs operate in ADR mode but systems designed for eADR can be tested on them for accurate performance measurements, even though the system may not be crash-consistent. The CPU memory controller uses the DDR-T protocol to communicate with DCPMM. DDR-T operates at cache-line (usually 64B) granularity and has the same interface as DDR4 but uses a different communication protocol to support asynchronous command and data timing. Access to the media (3D-XPoint) is at a coarser granularity of a 256B XPLine, which results in a read-modify-write operation for stores, causing write amplification. The XPLine also represents the Error-Correcting Code (ECC) block unit of DCPMM; access to a single XPLine is protected using hardware ECC. Like SSDs, DCPMM uses logical addressing for wear-leveling purposes and performs address translation internally using an address indirection table (AIT). Optane DIMMs also use an internal cache (XPBuffer) with an attached prefetcher (XPPrefetcher) to buffer reads and writes. The cache is used as a write-combining buffer for adjacent stores and lies within the ADR domain, so updates that reach the XPBuffer are persistent. DCPMM modules can be configured in either Memory mode or App Direct mode. In memory mode, DCPMM is used as main memory without persistence to expand its capacity. In App Direct mode, DCPMM is accessible as byte-addressable persistent memory, separate from DRAM. We only focus on App Direct mode in this thesis because we are only interested in persistence use cases of PMEM.

2.2 Non-Volatile Memory Express

The Non-Volatile Memory express (NVMe) [131] standard is a recent innovation that has significantly impacted research in storage systems. The NVMe standard is a protocol for accessing storage devices over PCIe and replaces the legacy SATA protocol designed for hard disks. The most significant change is the support for up to 64K parallel IO queues with each queue possibly holding up to 64K pending requests. An NVMe IO qpair (QP), which consists of a pair of submission and completion queues, must be allocated to submit requests. This follows the design principles of RDMA networks and allows for highly parallel asynchronous IO. The standard allows flash storage devices such as SSDs to achieve profound improvements in latency and throughput. NVMe-based SSDs have been emerging as the latest storage technology bridging the dreaded performance gap between hard disks and memory. These new devices are built for extremely low latency and achieving high degrees of parallel I/O. The NVMe-over Fabrics (NVMf) protocol [130] additionally allows low-latency access to remote SSDs using the same NVMe command set. The NVMf protocol, in particular, has the potential to fundamentally change the design principles of distributed storage systems.

In cloud environments, users expect a certain guarantee of service. Considering the new NVMe technology being introduced in enterprise clouds, it is only natural to ask whether a similar guarantee of service can be provided for this emerging hardware. In fact, this issue has been addressed to some extent in the NVMe standard itself. The standard includes provisions to enable command arbitration through mechanisms that are to be provided by hardware. Command arbitration is a method used by the SSD controller to determine the next submission queue to process requests from. For example, a basic weighted round robbin scheme for arbitration has been included in the standard. The controller processes

requests from submission queues by giving higher priority to queues with a larger weight. Command arbitration serves as a useful base to provide bandwidth guarantees in multi-tenant cloud environments. The challenge in this context is to enable service guarantees without application modification.

SPDK [76] is a userspace library built for applications with high-performance storage requirements. SPDK moves all necessary IO drivers to userspace and operates in polling mode, thereby enabling high-performance access to storage. In addition to legacy storage protocols, SPDK offers support for the NVMe standard, including NVMf. Each application thread requiring I/O is recommended to create a separate QP, allowing maximum parallel processing and eliminating the need for synchronization through locking. Further, the processing of IO operations is completely asynchronous. Applications need to explicitly ask the SPDK runtime to poll the completion queues. This asynchronous operation allows for a complete or partial overlap of I/O and application processing. The SPDK design solves most of the performance related issues that plague the Linux NVMe driver [186].

SPDK can be used to design a direct-access userspace data plane for fast storage of data. In this manner, the kernel IO stack can be bypassed which not only reduces latency but provides the ability to exploit hardware features such as command arbitration and namespace isolation. Unfortunately, bypassing the kernel has negative side effects. For example, it is easy for multiple processes to share storage devices if they use the POSIX interface since the kernel serves as a common entry point. Userspace runtimes, in contrast, must handle synchronization on their own, perhaps through the use of shared memory for coordination or a client-server model. Another common issue with userspace runtimes is compliance with POSIX API. Unless a complete POSIX abstraction is provided, existing applications cannot be easily run without modification.

2.3 Remote Direct Memory Access

Modern networking interconnects such as InfiniBand [6] and RDMA over Converged Ethernet (RoCE) [15] are heavily used in high-performance computing to achieve high throughput and low latency. These technologies are now being increasingly adopted in datacenters worldwide. One of the key features offered by these interconnects is Remote Direct Memory Access (RDMA). Through RDMA, a process can remotely read or update memory contents of another remote process with no remote involvement. Data transmission is done in a completely OS-bypassed manner, i.e the communication is processed in userspace and carried out in a zero-copy fashion. Leveraging RDMA and using hardware offload for all protocol processing provides high-performance and low latency communication. RDMA provides both two-sided (Send and Recv) and one-sided (Read and Write) primitives. We advocate for the use of these advanced networking interconnects in the design of storage systems, not only for the communication performance benefits but also for the improved reliability that it offers as compared to Ethernet hardware. The Reliable Connection (RC) protocol provided by InfiniBand/RoCE (similar to TCP in Ethernet) provides a data delivery guarantee with data corruption detection and ordered data delivery. In addition, RC supports hardware-level re-transmission and acknowledgements for error recovery. These interconnects use a credit-based link-layer flow control protocol which assures the absence of buffer overflows during communication, a significant advantage over Ethernet. Moreover, the heavy reliance on hardware-based protocol processing in RDMA networks prevents the existence of software-based errors from disturbing network communication.

Like DRAM, PMEM on remote nodes can be directly accessed with RDMA. Similarly, NVMe drives on remote nodes are also accessible via the RDMA conduit of the NVMf

protocol. The combination of RDMA, PMEM, and NVMe, therefore, offers the possibility of fast storage access regardless of the location of data.

Chapter 3: Arcadia: A Scalable Replicated Log on PMEM

The allure and peril of PMEM. PMEM has attractive latency and bandwidth properties for a storage technology. At nearly the speed of DRAM, it provides valuable persistence across power cycles. By being accessible as memory, it efficiently supports byte-addressable access to data. These features promise large performance gains for a number of important workloads.

Unfortunately, the persistence property of PMEM is hard to realize in practice [81, 125]. Only data that has reached the PMEM controller is actually persistent. Data in CPU caches is volatile, thus it is lost on power failure. Moreover, the CPU may reorder store operations and implicitly evict cache lines at any time, so persistence must to be managed explicitly. Even then, the hardware only provides 8-byte atomicity for local writes and *no* atomicity for remote writes over RDMA [82]. Using the memory interface for speed also makes PMEM intrinsically *local*, so node failures and network partitions curtail its availability. The single point of failure also limits its durability as uncorrectable media errors or DIMM failures may result in permanent loss of data [189]. These idiosyncrasies lead us to believe that PMEM is *evil*.

Introducing a log. Most of these problems can be addressed by adding a log. Logging can be used to mange persistence and provide atomicity to preserve consistency of updates

across failures for data stored in PMEM. Before making changes to PMEM, the programmer first records into the log a description of the changes to be made (typically redo or undo records), then makes the changes to primary storage in PMEM. If the system crashes while making the changes to PMEM, log recovery at the next start-up restores consistency of the data in PMEM by completing (with redo) or reverting (with undo) any partial updates [122].

To preserve the performance benefits of using PMEM as the primary storage, the log itself must also reside in PMEM and must be implemented with performance in mind. The log must also be able to handle the difficulties of PMEM so that the log remains consistent through crashes. In order to handle the inherently local nature of PMEM, the log should also be replicated across nodes, and the replication must also have good performance. PMEM's characteristics are an excellent match with RDMA. RDMA provides low-latency remote access to memory, including byte addressable PMEM. It is not without complications, however, because RDMA to PMEM suffers from many of the same atomicity and persistence problems of local PMEM. Thus, the abstractions to tame the perils of using PMEM locally must be extended to PMEM over RDMA. A replicated log on PMEM is a powerful tool benefiting two key scenarios. It allows using fast PMEM to easily and safely add fault-tolerance to a system that was volatile or had weak consistency properties, and it improves system performance of slower disaggregated storage by dramatically reducing the log latency that often limits the update rate.

Why does not prior work solve all problems? Prior work on PMEM logging [14, 70, 75, 95, 162, 172, 178, 182] places inordinate trust in hardware's ability to retain and atomically update data. This is because these works were designed considering that PMEM has similar reliability as flash/disk. However, reliability for PMEM is a more complex problem: media

Log Design	Device/Node Failure	Network Partition	Media Error	Power Loss
PMDK [75]	×	×	×	✔
FLEX [178]	×	×	×	✔
Query Fresh [172]	✔	✔	×	✔
Tailwind [162]	✔	✔	✔	×
Arcadia	✔	✔	✔	✔

Table 3.1 Comparison of resilience to key failure scenarios with related work. Note that Arcadia is the only system that is robust to all of these failure scenarios. In addition, it is the only one to provide concurrent writes with in-order commit.

errors and software bugs can corrupt data and the reliability of each memory-cell degrades as it is used, potentially leading to premature failure [81]. So, in practice, without sufficient redundancy and integrity checks, data may be lost or corrupted and the burden of ensuring data availability and consistency falls onto the programmer. Further, prior work provides limited concurrency for log writes as a simple way to ensure monotonicity, or in-order commit. Providing concurrency while maintaining monotonicity requires preventing holes in the log which is not easy to attain. This is challenging because multi-threaded concurrency is non-deterministic and storage systems typically require log writes to preserve a total order to guarantee correctness. Table 3.1 presents a comparison of resilience to key failure scenarios between Arcadia and related work. Arcadia was designed to overcome these shortcomings of prior work. Through replication, Arcadia can survive device/node failures and network partitions. Further, by leveraging novel integrity and atomicity primitives for PMEM, Arcadia is resilient to memory corruption, media errors, and power loss. Arcadia also provides concurrency while preserving in-order commits. To achieve this goal, Arcadia separates log writes into a set of distinct steps and *only* isolates those that require

serialization. In this manner, accidental or unnecessary synchronization can be avoided and concurrency is maximized.

In this chapter, we present the design and implementation of **Arcadia**[3], a generic replicated log to tame the evils of PMEM. We make the following novel contributions in this chapter:

1. Persistence, atomicity, and integrity primitives for both local and remote accesses to PMEM. These abstract primitives hide the weak consistency of PMEM, ensure reliable persistence, and make PMEM easier to use.

2. A generic interface for Arcadia that has two distinguishing features – it hides the complexities of replication and PMEM + RDMA persistence and it decouples serial steps from those allowing concurrency. This minimizes accidental serialization and allows applications to be as concurrent as possible within the constraints of correct operation. The concurrency serves to overlap application processing, checksum computation, and replication latency across multiple threads.

3. Design and implementation of Arcadia using the proposed primitives to handle atomicity, integrity, and replication of log records. Our design has several novel aspects, which include concurrent log writes on PMEM with in-order commit and a frequency-based log force policy for providing low overhead persistence with guarantees on bounded data loss.

4. We deploy Arcadia on two servers with real PMEM that are connected with RDMA. Our analysis shows that Arcadia significantly outperforms state-of-the-art PMEM logs, such as FLEX [178], PMDK's libpmemlog [75], and Query Fresh [172] while

[3]Named after the Greek province of Arcadia.

providing stronger log record durability guarantees. We expect Arcadia to be used as an off-the-shelf log implementation for any storage system using logging, in particular systems that have weak consistency or disaggregated storage.

3.1 Primitives for Local and Remote Accesses

Handling atomicity, integrity, and persistence of accesses to PMEM is a non-trivial task. In this section, we first present primitives for durable local and remote writes to PMEM. Building upon these, we present primitives for atomically and reliably accessing data in local or remote PMEM. Crucially, these primitives handle the complexities of PMEM + RDMA persistence and hide the weak and complicated persistence properties of PMEM. We distinguish the primitives as Persistence, Replication, Integrity and Atomicity.

Persistence Primitive. To guarantee persistence locally, we construct a primitive using hardware access macros. It takes a location and length in PMEM and makes sure that data previously stored at that location is persistent. It relies on architecture-specific instructions such as `clwb` and `sfence` that can guarantee that data has been flushed from the volatile CPU caches into PMEM.

Replication Primitive. Our goal was to construct a replication primitive which can work on currently available commodity hardware and does not require hardware-modification or BIOS configuration changes. To this end, we design a single round-trip protocol for both replicating and persisting data to remote PMEM. We use the one-sided RDMA-Write-with-Immediate (RDMA-Write-Imm) operation to both replicate data and signal the remote server to force data to storage using the persistence primitive. The 'immediate' value in this operation carries with it the length of the data to force while its starting address is obtained from the completion event. Essentially, the RDMA-Write-Imm acts as an asynchronous

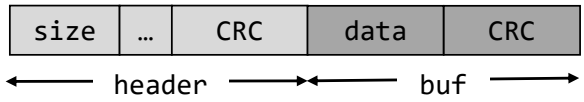


Figure 3.1 Integrity Primitive Data Layout in PMEM

```
# header and buf are preallocated in PMEM
function ReliableWrite(data, size)
    header = generateHeader(data, size)
    header.crc = crc32(header) # Header CRC
    memcpy(buf.data, data, size) # Copy data
    buf.crc = crc32(data) # Data CRC
    # Replicate + Force header and data
    return rdma_write_and_force(header, buf)

function ReliableRead(data)
    rdma_read(header, buf) # Remote read
    if (header.crc != crc32(header))
        return false
    if (buf.crc != crc32(buf.data))
        return false
    memcpy(data, buf.data, header.size)
    return true
```

Listing 3.1 Integrity Primitive

RPC to signal the remote server to persist data. Once the persist operation is complete, the remote server sends an acknowledgement using an RDMA send. The local server can use this acknowledgement as an assurance of remote data persistence. This approach preserves the benefits of one-sided RDMA while also guaranteeing remote persistence.

Integrity Primitive. This primitive is designed for reliably reading and writing data in PMEM. Reliability here means that we should be able to verify the integrity of data. This requires protection against two situations – the first where writes may be torn, i.e., only part of data is persisted, and the second where data is corrupted because of undetected media

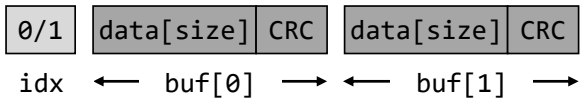


Figure 3.2 Atomicity Primitive Data Layout in PMEM

errors. To achieve reliability, we add a header field to every data item with checksums protecting both of them. Figure 3.1 provides an overview of the data layout for this primitive. The header contains the size of the data buffer and can be used to store additional information, for instance, to identify the type of data stored. Listing 3.1 presents a pseudo-code of the primitive. The `rdma_write_and_force()` function represents the replication primitive and can be replaced by the persistence primitive if replication is not desired. For reliably writing data, checksums (such as CRC32) for both header and data are be generated. By protecting both using checksums, there are no ordering requirements of writes or persistence barriers for the header and data. In addition, there is no requirement on atomicity of writes to PMEM. This is crucial because replicating data using RDMA does not provide any atomicity guarantees. In our approach, a single replicate and persist operation is sufficient for the entire write. When reading data, both the header and data checksum must be checked before data can be safely copied. Header integrity must be validated before data integrity, otherwise the size field in the header may not be correct. As an optimization, the header checksum can be replaced by a special integrity check value, such as a log sequence number (LSN).

Atomicity Primitive. This primitive is designed for atomically accessing data in PMEM and is required when updating an object with a fixed location. It guarantees that the entire data is updated atomically and that data integrity can be validated on reads. Atomically updating data is challenging because writes may be torn, so data may be only updated

```
# idx and buf[] are preallocated in PMEM
function AtomicWrite(data)
    memcpy(buf[!idx].data, data, size)
    buf[!idx].crc = crc32(data) # Generate CRC
    # Replicate + Force data and CRC
    rdma_write_and_force(buf[!idx])
    idx = !idx # Flip index
    # Replicate + Force index
    return rdma_write_and_force(idx)

function AtomicRead(data)
    # Remote read
    rdma_read(idx, buf)
    if (buf[idx].crc != crc32(buf[idx].data))
        return false
    memcpy(data, buf[idx].data, size)
    return true
```

Listing 3.2 Atomicity Primitive

partially. Data cannot be updated in-place because there is no guarantee of atomicity for remote writes to PMEM. To solve this issue, we propose the use of copy-on-write (CoW) for updates to data. Figure 3.2 provides an overview of the data layout and Listing 3.2 presents a pseudo-code of the primitive. We use two buffers to implement the CoW approach and switch between the two for every update. The index flag is used to identify the current valid buffer for reads. Data integrity is protected using checksums. For any update, data must be written to the invalid buffer. Once data and its checksum have been written and persisted, the index can be updated and persisted accordingly. As an optimization, the index flag can be placed in volatile storage as long as the valid data buffer can be identified on recovery using the contents of the data. Another possible optimization is to always write to a new dynamically allocated buffer and discard the old buffer once the write has successfully completed.

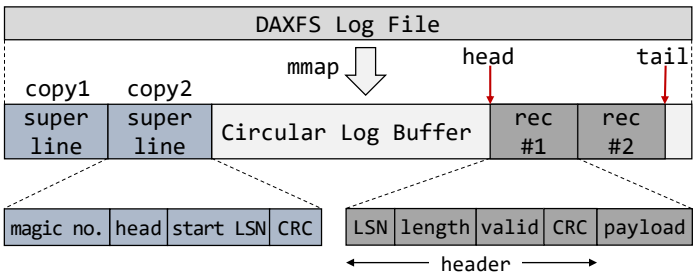


Figure 3.3 Arcadia's Layout on PMEM

Use Cases. The proposed primitives are useful in making PMEM easier to use. The integrity primitive can be used to reliably write data once, or when the data being overwritten is known not to be needed anymore. The atomicity primitive can be used when updating existing data in place, where we need to make sure that if there is a failure during write, the old data remains readable. We show how these primitives can be applied in Arcadia to achieve the persistence, integrity, and atomicity requirements of log writes. Both primitives are used in Arcadia for accessing different portions of the log – the integrity primitive is used for accessing log records while the atomicity primitive is used for accessing the log header.

3.2 Arcadia Design

In this section, we present the design and implementation of Arcadia. We design Arcadia as a replicated log with a single-primary, multi-backup model. It has a single multi-threaded writer doing updates (logger) and a single reader during recovery.

3.2.1 Design Overview

PMEM Layout. We place the entire log as a single file on a DAX filesystem formatted on PMEM. The entire file is `mmap`ed into the address space of the process to enable load/store

access to log records. Figure 3.3 shows an overview of the log layout on PMEM. The log itself consists of a circular buffer of fixed size to hold the log records and a header containing information necessary to both identify and recover instances of the log. We call this header the *superline* because it fits within a single cache line and its functionality is analogous to the superblock in a filesystem. The superline stores the head and start LSN of the log which are used on recovery to find the first valid record. We do not include the log tail in the superline to avoid the overhead of having to update it on each log append. Instead, we find the log tail on recovery by iterating over all valid records to locate the end of the log. Each log record in Arcadia consists of a header and payload pair. The header contains the length and checksum of the payload as well as a monotonically increasing LSN and a valid flag. Arcadia relies on special values and basic consistency checks like magic numbers, LSNs, checksums, and valid flags to validate the content of data in persistent storage.

Operation Modes. Arcadia can be deployed in one of three operation modes – *local*, *local+remote*, or *remote only*. In the *local* mode, log data is placed and accessed only on PMEM available on the local server. In *local+remote* mode, the local copy is the primary replica of the log with one or more secondary servers serving as backups. Finally, in *remote only* mode, all durable copies of the log reside on one or more remote backups, while the client only holds a volatile copy of the log. As long as there is any replication, at least one log is remote. Latency of parallel log appends is limited by the slowest backup in a write quorum, thus having a persistent primary does not significantly help performance. This means that log clients can be located anywhere there is good RDMA connectivity and are not required to have PMEM.

API Function	Description
[id, ptr] reserve(size)	Reserve space for a record
int copy(id, data, size)	Copy data into a record
int complete(id)	Mark a record as completed
int force(id, freq=1)	Force a record to PMEM
REC append(data, size)	Append a record
LSN getLSN(id)	Get record LSN
int cleanup(id)	Reclaim record space
int cleanupAll()	Reclaim entire log
ITERATOR begin/end()	Recovery iterator

Table 3.2 Arcadia's Interface

Interface. Arcadia's interface was designed with three goals – (1) minimizing unintended synchronization, (2) providing direct access to PMEM, and (3) enabling use in any situation requiring logging. An overview of the interface is presented in Table 3.2. Conventionally, log interfaces only include an *append* method for log writes. Bundling the entire write as a single append method limits any overlap of application compute with log writes and data persistence. To achieve the first goal, we propose a set of fine-grained methods to write data in the log. These methods break the append method into four separate stages – reserve, copy, complete, and force. This separation allows users increased flexibility; it enables them to call these methods only if required and place them at the best possible location. An append method is also provided as a combination of the four fine-grained methods wrapped up into a convenient high-productivity operation. The second goal is achieved by providing a direct pointer to the PMEM region allocated for a record using reserve. The benefit of having a direct pointer is that it allows the user to assemble the log record contents directly in PMEM without need to first build it in DRAM and then copy it. It avoids the need for a data copy in cases where the data to be logged is not already sitting

in DRAM. Finally, to achieve the third goal, we provide several generic methods to add records, reclaim space used by records, and iterators to traverse valid records for recovery. More details on how these methods operate can be found in section 3.2.3.

3.2.2 Replication and Recovery Protocol

Arcadia implements a quorum-based protocol for replication and recovery. The write quorum (W) is a configurable parameter that can be adjusted based on desired performance and fault-tolerance. The read quorum (R) is automatically selected based on W and the number of durable replicas (N) to satisfy quorum requirements ($R + W > N$). Therefore, the system can tolerate up to $N - W$ replica failures when writing log records.

Replication. When log data needs to be replicated, RDMA writes are initiated to all backups (remote log replicas) in parallel. Next, we wait for the persistence acknowledgement from all backups. Once complete, the data can be considered to be durably replicated. Network partitions and backup failures can disrupt this process. To handle such cases, Arcadia uses a timeout-based approach. If the backup write times out, we consider the backup to be failed and immediately close the connection with that server. This also avoids the problem of an inconsistent backup in cases with a transient network partition. As long as the number of failed backups does not impact the write quorum (i.e., W backups are in complete sync), the replication process is successful. In situations where write quorum cannot be achieved, replication will fail, and as a consequence any call to *force* will also fail. We delegate responsibility for these failures to the application. An easy fix that the application can use in such cases is to gracefully shutdown, add new backup servers (by copying the PMEM log files), and restart the application.

Recovery. On recovery, Arcadia checks that a sufficient number of copies of the log are available and consistent to meet the read quorum, R. If the read quorum is not met because backups are unavailable, Arcadia fails recovery and lets the user retry after more backups come online. If read quorum fails because too many copies are corrupted, then Arcadia cannot guarantee data consistency and recovery cannot proceed safely. During the recovery process, we use data in the superline to detect which copies of the log are consistent and contain the most recent data. Before accepting any new requests, Arcadia repairs any inconsistent or failed log copies using the consistent copies. Only inconsistent copies are modified during recovery, therefore, the recovery process is idempotent and invulnerable to repeated failures. Once recovery is complete, the primary and all backups are up to date, so new requests can be safely accepted.

Handling Primary Failure. Primary failures are handled by using a third-party coordinator, such as Apache ZooKeeper [4]. On primary failure, the coordinator triggers leader election, and selects a new primary. The user application is then migrated to the new primary and restarted once log recovery is complete.

Handling Diverging Histories. Diverging histories can happen due to repeated failures of different backups. To solve this problem, we add an epoch field to the superline in each log copy. The epoch values start out at 1. On each recovery, all available copies are read and the maximum value is taken. A majority of copies must be readable, or recovery cannot continue. Recovery increments the number by 1 and writes the new value to all available copies. Writes to a majority of nodes must be successful to continue. Only log copies with the highest epoch are considered valid, thereby avoiding diverging histories.

3.2.3 How Arcadia Works

In this sub-section, we describe how Arcadia works in detail. Arcadia uses the proposed integrity primitive for accessing log records and the atomicity primitive for accessing the superline. As an optimization to the integrity primitive, we use the LSN for validating the header rather than a checksum. For optimizing the atomicity primitive, we keep the index in volatile memory and identify the valid superline on recovery based on which copy has the latest start LSN.

Concurrent Log Writes and Monotonicity. Arcadia allows concurrent writes despite having in-order commits. We describe how each write method is handled internally to enable this. The **reserve** method allocates space for log records and returns a direct pointer to the allocated space in PMEM. It is also responsible for generating the LSN for the record. To ensure monotonicity of LSNs, this method synchronously allocates log space and generates LSNs. The **copy** method is a convenience method to copy data into a reserved record. It uses non-temporal stores that bypass the CPU caches to copy data to PMEM efficiently. This method can be safely called multiple times to copy different data chunks into the record. Note that users may choose not to use this method and instead use `memcpy` or CPU stores to copy data themselves. The **complete** method is used once all data has been written to the record. Complete generates and writes the checksum for the record payload and also sets the valid flag in the record header. The **force** method guarantees that a record is durable, i.e., it will be available after a crash. It does this by replicating and persisting records. Importantly, it ensures that there are no holes in the log. To do this, force waits, if necessary, until all previous records have been marked valid (complete) and forces them first. This is done by spinning on the valid flag, which is part of the record

header, until it is set. Doing so ensures that records are always persisted in-order despite concurrent accesses.

The key insight here is that reserve and force are the only synchronous methods required during a log write. Multiple threads can safely call copy and complete in parallel. Although data is concurrently written into the log, the reserve and force methods carefully track the status of all records to ensure that monotonicity is still maintained.

Log Space Reclamation. Space reclamation is important when records are no longer necessary for recovery because the changes they describe have already been made durable. Depending on the type of logging algorithm used, records may be invalidated at different times. However, a cleanup method operating at the granularity of a record is sufficiently generic to be applicable in all cases. This method is responsible for unsetting the valid flag of the record. It also checks if there is a contiguous section of the log starting from the head (the oldest valid record) that can be reclaimed. If so, it advances the log head and updates its value in the superline. A cleanupAll method is also provided to reclaim the entire log. This method simply re-initializes the entire log and updates the superline accordingly.

Recovery Iterator. This iterator is used to access all valid records upon recovery from a crash. It is useful in bringing the system back to a consistent state using data in the log. Before it can be used, the recovery protocol (see section 3.2.2) is triggered to repair any corruption and ensure consistency of all copies. The recovery iterator operates on the local log copy and obtains the log head from its superline to find the oldest valid record. To validate records, three integrity checks are performed – (1) the LSN must be monotonically increasing, (2) the valid flag must be set, and (3) the checksum of the payload must match. The iterator ends when any one of these checks fails. These integrity checks guarantee that any partially written or corrupted records are not applied during recovery.

3.2.4 Force Policy

Up until now, we have assumed that log writers would always do an explicit force on the log after writing a log record, i.e. that the thread doing the logging would not be able to continue until the log record was safely committed. But doing a force operation on the log after every write is expensive. The high cost of persisting data and the coarse granularity of writes for traditional block storage motivated researchers to propose techniques to reduce the number of synchronous, small writes required by relaxing *freshness* requirements (how up to date the state is after crash recovery). One popular approach is called group commit [66]. In this approach, log records are persisted in batches (or windows), such that the number of unpersisted records at any time is limited by a threshold, often called the group commit size. By controlling the group commit size, the desired level of freshness can be achieved.

With PMEM, the performance characteristics of persistent writes have changed drastically. PMEM write latency and granularity is much lower than other PCIe connected storage devices, so the cost of persistence is lower too. This seems to imply that there is no merit in limiting write frequency with PMEM. However, we find that when adding replication overhead to overall performance characteristics, the overhead of persistence remains relatively high. This is because the cost of replication over RDMA is often an order of magnitude higher than local writes to PMEM. To reduce the cost of replication, we first explore if group commit can be a viable approach. We find that although it improves overall latency, it is not scalable; at high concurrency, group commit has significant overhead. This is because a shared global counter is required to maintain the current window size. Concurrent access to this counter across threads results in significant cache thrashing and reduces the effectiveness of group commit. This effect was not observed in prior work because

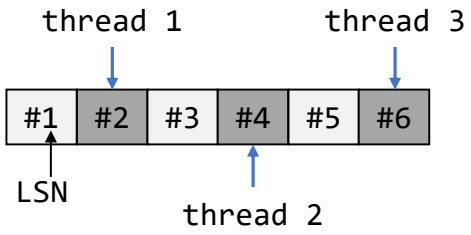
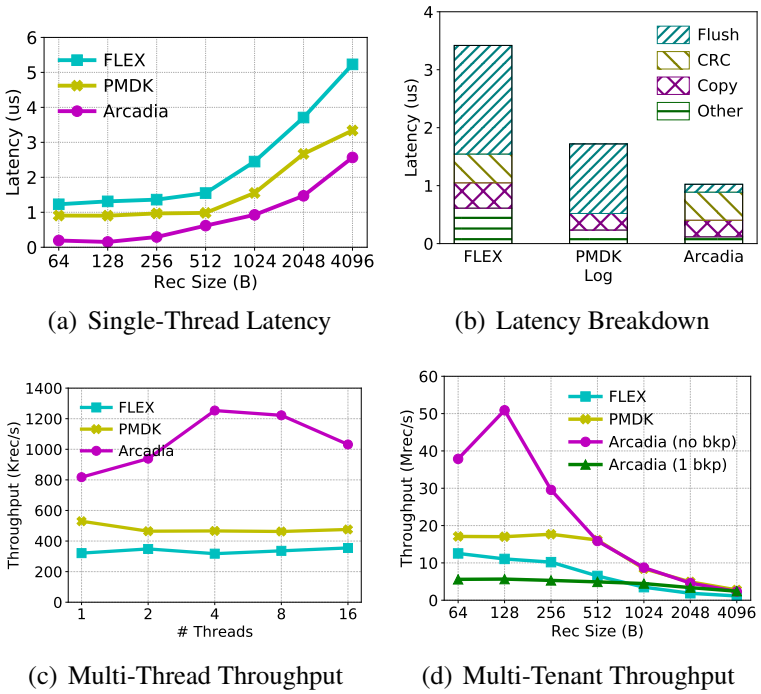


Figure 3.4 Example showing the worst case scenario for the frequency-based force policy with T=3 and F=2, where all threads are blocked in trying to force persistence. All 6 records have been completed and may be lost on crash.

they did not allow concurrent access to the log. Further, with fast PMEM hardware the software and synchronization overhead constitutes a significant portion of overall latency, so the effects of synchronization are much more pronounced.

To reduce the overhead of replication and persistence without adding synchronization overhead, we propose a novel frequency-based force policy. The key idea is as follows. Instead of maintaining a window of unpersisted records, we force records at intervals determined by a predefined 'update frequency'. The crux of the idea lies in the fact that we can leverage the monotonicity of record LSNs to determine when to force data. For instance, given a frequency of F, data will be persisted every time a record with an LSN divisible by F is forced. Each thread that completes a record with such an LSN is responsible for persisting records, which distributes this effort over all active threads. In this manner, we do not need a shared counter to maintain the window size, but instead piggy back on existing synchronization to generate monotonic LSNs as consecutive integers. In our design, threads can force records concurrently, which could leave holes in the log if a subset of them complete the force before a crash. To prevent this issue, we require any thread calling force on a record to wait until all pending forces on records with lower LSNs complete *and* all previous records have been completed. In the worst case, all threads can



(a) Single-Thread Latency (b) Latency Breakdown

(c) Multi-Thread Throughput (d) Multi-Tenant Throughput

Figure 3.5 Microbenchmark Comparison with FLEX and PMDK

get blocked doing a force. This can happen if, for instance, the first thread doing a force gets context switched, or if a record with an earlier LSN that is not a multiple of F has not been completed yet. When threads force with the configured frequency, they will each be separated by F records. This gives us the theoretical upper bound of the number of completed records that can be lost as $F \times T$, where T is the maximum number of threads or concurrent forces allowed. Figure 3.4 shows an example of the worst case scenario with T=3 and F=2. All three threads are blocked in this scenario. It is easy to see that the maximum number of records that may be lost on crash[4] can be correctly calculated using the formula described here.

[4]We call this the vulnerability window.

Use Cases. There are two use cases for this policy. The first is where explicit guarantees of persistence are not required for ensuring consistency (e.g., when recording logical operations in the log for a volatile key-value store). In this case, the frequency-based force policy can be used for all log updates to reduce the overhead of persisting records. A few of the recent updates may be lost on crash (the policy provides a bounded guarantee for how many updates can be lost), but consistency is not compromised. The second use case is where explicit guarantees of persistence are required for ensuring consistency (e.g., when recording state updates in the log for a filesystem). Even in this case, the frequency-based force policy can be used for all log updates to reduce the overhead of persistence (by calling force with freq=F for every record). When users require an explicit guarantee of record persistence, they may either issue a synchronous force (a force with freq=1) *or* check if the difference between the LSN of a newly allocated record and the record they want forced is greater than the theoretical upper bound of the size of the vulnerability window, $F \times T$.

3.3 Experimental Analysis

In this section, we present the evaluation of Arcadia. We conduct a comprehensive analysis of the impact of the novel aspects of our design. We also compare performance and resilience of Arcadia with state-of-the-art PMEM-optimized logs, FLEX [178], PMDK's libpmemlog [75] (referred to as simply PMDK in the evaluation), and Query Fresh [172]. We were unable to compare with Tailwind [162] because it requires special hardware support and its source code is unavailable.

3.3.1 Experimental Testbed

Our experimental testbed consists of two Linux (4.18) nodes, each equipped with two Cascade Lake CPUs (5218@2.30GHz), 192GB DRAM (2 x 6 x 16GB DDR4 DIMMs),

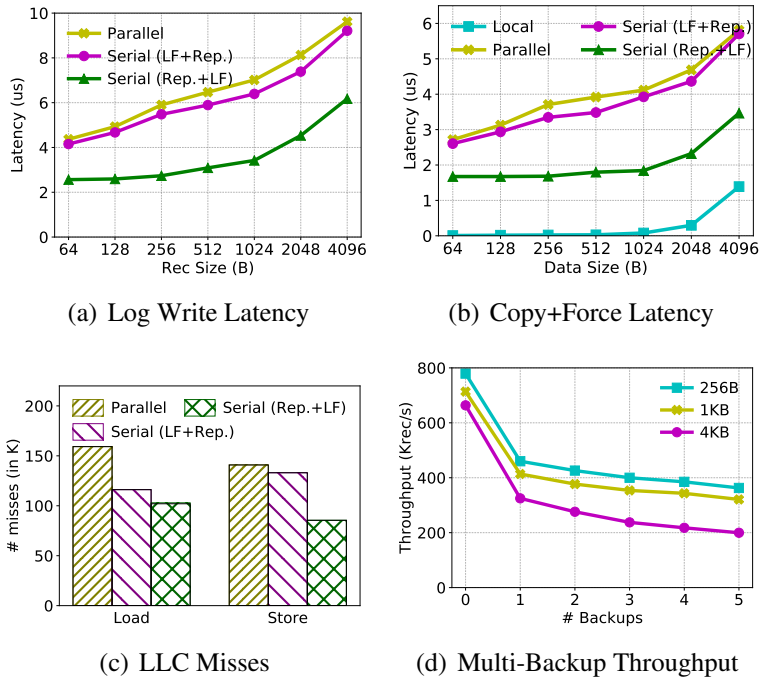
and 1.5TB PMEM (2 x 6 x 128GB DCPMMs) configured in App Direct mode. Each CPU has 16 physical cores (with hyperthreading enabled) and 22MB of L3 cache (LLC). All available PMEM is formatted as two `ext4` DAX filesystems, each utilizing the memory of a single CPU socket as a single interleaved namespace. The two nodes are connected using 100Gbps EDR InfiniBand. One node is used as the primary while the other is used as a backup.

All code was compiled using GCC 8.3.1. PMDK [75] 1.8 was used across all evaluations to keep comparisons fair. Hardware counters were obtained using a combination of Intel VTune Profiler [74], Intel PCM [79], and Linux perf utility. DCPMM hardware counters were collected using the `ipmwatch` utility, a part of the VTune Profiler.

3.3.2 Microbenchmark Evaluation

We first present a microbenchmark-level comparison of Arcadia with FLEX and PMDK. Since FLEX and PMDK do not support replication, we compare them with Arcadia deployed in local mode.

Latency: We evaluate the latency of log writes with a single thread while varying the record size. Figure 3.5(a) shows this evaluation. We find that Arcadia has the lowest latency out of all three designs, up to 6x faster than PMDK and 8x faster than FLEX. One of the main reasons for this trend is that Arcadia does not maintain the pointer to the tail (most recently added record) of the log in PMEM (superline) and thus does not need to update it on every record write. FLEX performs the worst because it has high software overhead and it appends the record header and payload in separate operations. To confirm the reasons for these observations, we analyzed the breakdown of log writes for a 1KB record in Figure 3.5(b). We can clearly observe that flushing data to PMEM takes much



(a) Log Write Latency

(b) Copy+Force Latency

(c) LLC Misses

(d) Multi-Backup Throughput

Figure 3.6 Replication Overhead Analysis

more time in FLEX and PMDK. This is because they both update the log tail for each write. The flush time is especially high because it also includes the additional store fence required to wait for data being copied into PMEM before the tail update. By avoiding a tail update, Arcadia shows much lower latency despite computing checksums.

Throughput: Figure 3.5(c) presents throughput curves in a multi-threaded setting. We measure the overall log throughput while increasing the number of threads concurrently adding records to the log. Both FLEX and PMDK have flat curves because they fully isolate writers using coarse-grained locks and do not provide any concurrency. Arcadia only isolates steps that require serialization (reserve and force), and can provide some level of concurrency. Its throughput is maximum at 4 threads, but reduces slightly at higher concurrency. There are two reasons for this – (1) serialization overhead in reserve and force,

and (2) poor bandwidth for highly concurrent accesses to PMEM. We also measure the aggregate log throughput in a multi-tenant setting. This is a common usage scenario where multiple tenants are sharing the same resources. Figure 3.5(d) shows the throughput for different record sizes with 16 concurrent single-threaded tenants, each writing to a separate log. Without replication, Arcadia performs the best in all cases. As the record size increases, throughput is bound by the PMEM bandwidth, and all log implementations converge to that limit. An observation we make is that Arcadia's throughput is lower for 64B records than 128B records. This is because of the write amplification effects in DCPMM for small-sized writes. With replication, Arcadia has lower throughput than FLEX and PMDK for small record sizes. In these cases, replication overhead bounds overall throughput. However, for large record sizes (¿ 2KB), throughput is indistinguishable from the local mode.

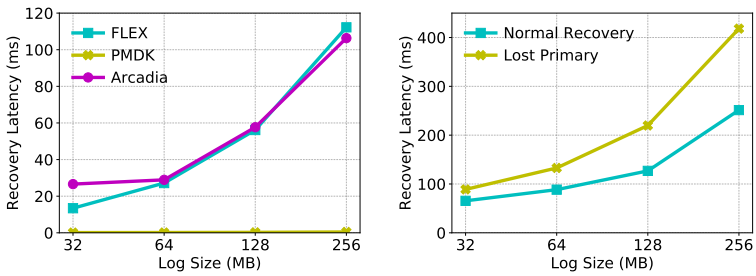
3.3.3 Replication Overhead Analysis

To understand the performance characteristics of replication with PMEM and RDMA, we conduct an experiment to measure the replication overhead of different policies with Arcadia deployed in local+remote mode. We compare three policies – (1) parallel, in which the local cache flush and RDMA replication are done in parallel, (2) serial (LF+Rep.), in which the two operations are done sequentially with the local flush performed first, and (3) serial (Rep.+LF), which is similar to serial (LF+Rep.) except the order of operations is reversed. Figure 3.6(a) analyzes the log write latency across record sizes for these three policies. Surprisingly, we find that the polices have significant impact on overall latency. Contrary to expectation, parallel has the worst latency, while serial (Rep.+LF) has the best. To understand the reasons for this trend, we analyzed the overhead of just the copy and

force operations of log writes (shown in Figure 3.6(b)) as well as the LLC miss counts of the three policies (shown in Figure 3.6(c)). The copy+force overhead curves mirror the trends of the log write latency curves. This confirms that the overhead of the replication policies are the reason for these trends. We can also observe that a local flush is an order of magnitude faster than remote flush. Therefore, when using the parallel policy, the local cache flush invalidates data from the LLC.[5] So, RDMA writes need to read data back from PMEM instead of reading it directly from the LLC. This additional read is manifested as additional LLC misses and is confirmed by Figure 3.6(c). The serial policy where the local flush happens first, suffers from the same issue, and therefore shows similar performance. Its performance is only marginally better than the parallel policy. We suspect that this is because concurrent reads and writes to PMEM in the parallel policy, conflict and slightly reduce performance. The serial (Rep.+LF) policy has the best performance because RDMA writes can get data directly from the LLC; the LLC misses graph confirms this reason.

Next, we analyze the impact of number of synchronous backups on log throughput. Our testbed only has two nodes, so we conduct this experiment on another cluster with Skylake nodes and EDR InfiniBand. Figure 3.6(d) shows the impact of number of backups on throughput for different record sizes. A common theme across record sizes is that adding just one backup results in a significant drop in throughput. This is because the overhead of replication is significant. Interestingly, going beyond one backup does not impact throughput as much. The asynchronous nature of RDMA enables data to be sent to multiple backups in parallel, minimizing impact on throughput. This is a significant result as it shows that as long as replication is enabled, throughput is not significantly impacted by the number of backups. Therefore, it is not necessary to compromise fault-tolerance for

[5] At the time of writing this thesis, cache write-back instructions had not been implemented in Intel processors.



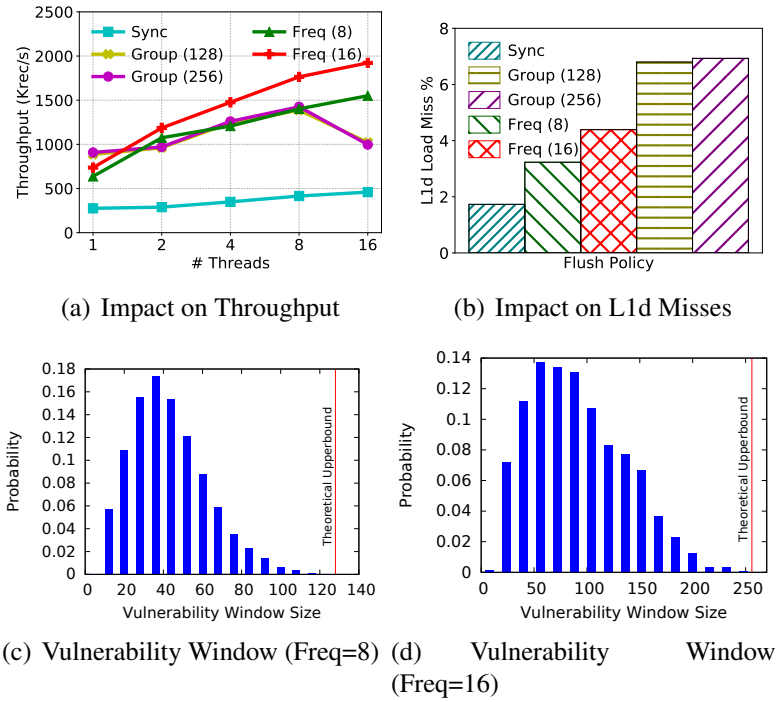
(a) Recovery Latency (b) Arcadia Recovery Latency

Figure 3.7 Recovery Evaluation

performance when replication is enabled. These results also indicate that clients without local PMEM can work efficiently in cases where PMEM is restricted to a subset of server nodes.

3.3.4 Recovery Evaluation

To evaluate the recovery performance of Arcadia, we measured the total time taken to recover log state and iterate over all valid records on recovery, calling a null recovery function for each record. Figure 3.7(a) compares Arcadia's recovery latency in local mode with FLEX and PMDK. Both Arcadia and FLEX rely on checksums to verify record integrity, so they have similar performance. This is because the time taken to verify checksums dominates recovery latency, which is also why latency increases linearly with log size. PMDK does not use checksums, so its recovery procedure only consists of calling null functions for each record. This is why it is able to recover so fast. However, by not relying on any integrity checks, it is unable to handle undetected media errors and may end up reading corrupted data. Figure 3.7(b) compares Arcadia's recovery latency with replication enabled in normal recovery and when the primary log copy fails or is lost. These represent the best



(a) Impact on Throughput (b) Impact on L1d Misses

(c) Vulnerability Window (Freq=8) (d) Vulnerability Window (Freq=16)

Figure 3.8 Force Policy Analysis

and worst case situations for recovery, respectively. We observe that when the primary copy is lost, the latency is higher. In this scenario, Arcadia needs to recover the primary copy using data in the backup which increases recovery time. However, this process uses the fast one-sided RDMA reads for copying data which does not add significant overhead. Even for a 256MB log, recovery takes less than 500ms in the worst case; so, we conclude that recovery is fast enough for most practical scenarios.

3.3.5 Force Policy Analysis

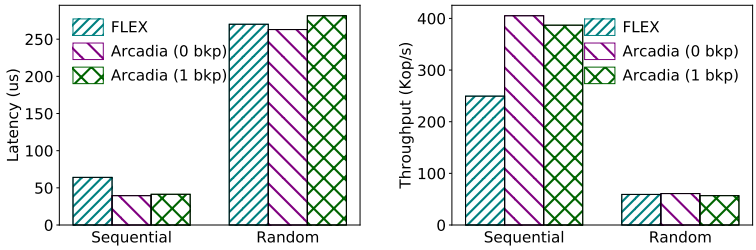
We compare group commit and frequency-based force policies in this experiment. We compare log throughput for the two policies in Figure 3.8(a). The group size or frequency is listed in parenthesis. We evaluate with two different values for each policy; these values

have been chosen such that their theoretical vulnerability window sizes are comparable. So, group (128) is comparable to freq (8), while group (256) is comparable to freq (16). We also evaluate the sync policy in which each log write is synchronously forced. Results demonstrate the group commit has significant overhead at high concurrency and its throughput drops by 30% at 16 threads. Concurrent access to the window size variable is behind this drop. On the other hand, the frequency-based policy is much more scalable because it avoids this synchronization overhead. To confirm that synchronization overhead is the reason for poor performance of group commit, we analyze the L1d miss rate for all policies (see Figure 3.8(b)). We can clearly observe that group commit has much higher miss rates because of constant cache thrashing as a result of concurrent accesses to the shared counter.

We also measure the distribution of the vulnerability window size (from latest completed record to the most recently forced record) for the frequency-based policy. Unlike group commit, the vulnerability window size is not fixed because if a thread gets blocked doing a force, other threads can keep adding new records and increase the window size. Figures 3.8(c) and 3.8(d) show this distribution for frequencies of 8 and 16, respectively. Interestingly, we find that the probability distribution is skewed towards smaller sizes, far below the theoretical upper bound. This shows that on average, threads are unlikely to block on a force. Overall, we conclude that the frequency-based policy is not only more scalable, but in practice it provides better resilience than its theoretical limit.

3.3.6 Arcadia Applications

RocksDB. To demonstrate the practical benefits of Arcadia, we integrate it with RocksDB, a popular key-value database used at Facebook, by swapping out its log with Arcadia. We



(a) Latency (b) Throughput

Figure 3.9 Comparison with FLEX using RocksDB

use the fine-grained interface for adding log records. We evaluate this version in both local and local+remote modes and compare it with the FLEX integration of RocksDB [141]. Figure 3.9 compares the latency and throughput for both sequential and random key-value puts at full-subscription (16 threads). By reducing the log append latency, Arcadia improves latency by up to 38% and throughput by up to 62% in local mode (0 bkp). In local+remote mode (1 bkp), Arcadia is better than FLEX (which is local-only) for sequential puts and only marginally worse for random puts, despite enabling replication. Replication overhead is much less than the overhead of the entire put operation. Hence, there is little difference in performance between the different modes. Also, random puts are much slower than sequential puts, so the benefits of using Arcadia are less.

Masstree. We integrate Arcadia with Masstree [114], another popular key-value database, to enable comparison with Query Fresh. Like the RocksDB application, we use the fine-grained interface for adding log records as well as the frequency-based force policy. This experiment shows the practical benefits of the frequency-based force policy. Figure 3.10(a) compares the throughput of read-modify-writes and Figure 3.10(b) compares the theoretical vulnerability window size for Query Fresh and Arcadia (with both group commit and

frequency-based force policies). From the results, we observe that Arcadia's group commit policy has high synchronization overhead at 8 and 16 threads which reduces performance bringing it close to Query Fresh's throughput. Query Fresh also uses group commit but it only enables limited log concurrency. Therefore, it delivers lower throughput than Arcadia but is also less impacted by the synchronization overheads of group commit. The frequency-based policy is able to deliver the best performance (up to 65% faster than Query Fresh) and fault-tolerance by avoiding unnecessary synchronization and forcing records more frequently.

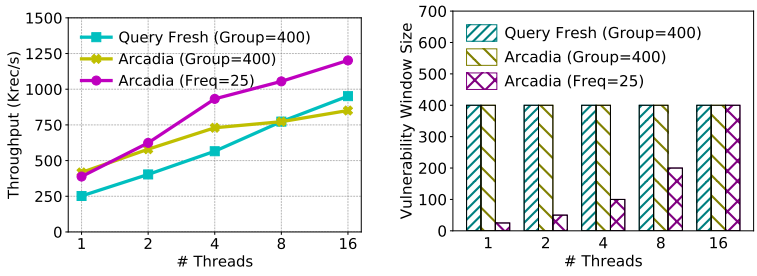
3.3.7 Key Insights

Our measurements demonstrate the performance benefits of our approach compared to PMDK, FLEX, and Query Fresh. The measurements confirm the benefits of specific innovations and design choices in Arcadia, such as avoiding the superline tail pointer, and using concurrency to limit the impact of checksums and replication. They also highlight subtle factors such as the serialization implied by group commit and the interaction between local flush and remote replication, giving confidence in the overall conclusions.

Arcadia's log interface crucially decouples steps requiring serialization from those allowing concurrency to maximize performance of parallel applications. It also illustrates PMEM's byte-addressability to avoid unnecessary data copying. The proposed frequency-based force policy gives flexibility in the freshness and performance trade-off, while allowing more concurrency than the traditional group commit method.

3.4 Related Work

Logging on PMEM. A number of prior works [14, 70, 75, 95, 162, 172, 178], have leveraged PMEM performance for logging as a means to improve overall system performance.



(a) Throughput (b) Fault-Tolerance

Figure 3.10 Comparison with Query Fresh using Masstree

Some of these works (Query Fresh [172], WBL [14], NV-Logging [70]) propose PMEM-based logs that are tightly integrated with a particular system or logging algorithm. Some of them (PMDK [75], FLEX [178], NVWAL [95]) propose unreplicated logs that cannot provide high availability. Others (e.g., Tailwind [162]) rely on special hardware support. Moreover, none of these are able to satisfy the robustness requirements of large-scale production systems because they do not provide resilience to some key failure scenarios.

Logging on Flash/Disk. Corfu [17], Raft-based log [8], Apache Kafka [170], DistributedLog [59], GNR [38], and JBD2 [166] are relevant log implementations on traditional flash and disk storage that are deployed in production systems. These designs are considered to be reliable and robust. However, they were designed considering the performance characteristics, access methods, and reliability of flash/disk which are considerably different than those of PMEM. Therefore, naively applying them to PMEM is unlikely to result in a performant or robust solution.

Combining RDMA and PMEM. Octopus [111], Forca [69], File-MR [182], Orion [181], pDPM [164], AsymNVM [112], NVFS [83], Hotpot [150], Mojim [191], and RDMP-KV [105] have worked on combining PMEM with RDMA for fast access to remote persistent storage. Each takes a different approach to remote persistence and atomicity that are not broadly applicable in other scenarios because they are either tied to a particular data format (key-value or file) or the complications of RDMA + PMEM persistence are not fully solved. In contrast, in this chapter, we present abstract primitives for ensuring remote persistence, atomicity, and integrity, that can be applied generically on commodity hardware.

3.5 Summary

In this chapter, we described Arcadia, a generic log stored on PMEM and replicated using RDMA with an easy to use interface. Implementing fast, fault-tolerant systems is challenging, but logging is a crucial abstraction to ease this task. The PMEM shortcomings with atomicity, persistence, and locality, with replication over RDMA put the implementation of robust systems at risk of overcomplexity. Arcadia's design encapsulates lessons learned uncovering and addressing problems using these technologies safely and realizing the alluring performance and reliability benefits this combination offers. Arcadia takes a pragmatic approach to the performance benefits of PMEM and RDMA while accommodating practical requirements such as concurrency and reliability to provide a realistic assessment of their promise. By presenting a usable log interface we hope to foster wider adoption of these technologies to improve robust storage systems.

Chapter 4: DIPPER: A Decoupled Model for Persistent Data Structure Design

The emerging PMEM technology offers unprecedented high performance while supporting data persistence, and has fueled a renaissance in re-evaluating the design of persistent storage systems [30, 33, 41–43, 50, 70, 71, 86, 139, 175, 176, 179]. In the previous chapter, we presented the design of a generic replicated log on PMEM and showed how such a design is useful in making PMEM easier to use and improving the performance of a wide range of storage systems. In this chapter, we build upon our work on the replicated log and take up the challenge of designing a storage system which is simultaneously fast, tailless, and quiescent-free. We believe that these are important requirements for modern systems, particularly considering the high demands of applications in a cloud setting. Providing these three features is essential for predictable and consistent performance, which is important in satisfying latency and throughput service level objectives [103, 160]. Inconsistent user experience has been shown to directly result in loss of revenue [84].

Most file and database systems cache important data and employ a journal or write-ahead log (WAL) to support fault-tolerance. Numerous studies [13, 26, 28, 32, 48, 63, 70, 95, 110, 116, 132, 148, 168, 184, 195] have focused on providing PMEM-aware data structures and logging schemes to reduce latency and guarantee failure recovery. These designs can indeed provide much better performance than SSD- or disk-based schemes,

but they are unable to provide a performant quiescent-free storage system, which means that users' requests in the frontend may have to wait for completion of data persistence activities in the backend, particularly during checkpoints. The reason for this is that the cache must be write-protected during checkpoints to ensure that the persistent backend is updated in a consistent manner. This limitation results in significant delay for requests arriving during checkpoints. As a result, the system must either quiesce for a short time or accept high tail latency.

On the other hand, the storage and memory like nature of PMEM has spawned a new class of storage systems that place and access data in-place [28, 32, 33, 41, 42, 48, 50, 63, 86, 110, 139, 148, 168, 169, 175, 179, 195]. Such systems unburden themselves from the limitations of checkpoints since data is always persistent. However, the challenge in designing such systems is that updates to PMEM must be done in a crash-consistent manner. This is further complicated by the fact that data in CPU caches is not presistent and cache lines can be evicted implicitly. Ensuring consistency requires explicit cache flushes and store fences, while ensuring atomicity requires the use of transactions or journaling. Any practical solution on PMEM must deal with both atomicity and consistency issues. This requires expensive, and often complex, protocols to be used which significantly lower end-to-end performance [63, 125]. The high cost of atomic data persistence compromises performance.

In this chapter, we propose a new approach for storage system design, called **D**ecoupled, **I**n-memory, and **P**arallel **PER**sistence (DIPPER) that exploits the byte addressability of PMEM to efficiently achieve the decoupling of system and checkpoint spaces. The system space to store data structures is entirely in DRAM while the checkpoint space (including

an operation log) is entirely in PMEM. Some recent systems like DudeTM [107], NV-HTM [25], and Bullet [71] have also proposed partially or fully decoupling operation and persistence phases to lower the cost of persistence and improve performance. However, these systems still rely on expensive physical logging which is particularly bad when the working set of transactions is large. The novelty of DIPPER is in its unique architecture which allows the use of operation logging without limiting frontend concurrency. The key idea is to let the frontend operate independently on DRAM while recording operations in the log and applying these updates to the backend asynchronously on an identical backend in PMEM. Operations need not wait for the updates to be applied to the backend to be considered durable. DIPPER ensures that the log replay is deterministic, so the checkpoint space can be updated in the background without involvement of the system space. The frontend and backend are kept consistent by applying the concept of observational equivalence [145]. The goal is to hide the latency of persistence by using the volatile system space for all requests and taking checkpoints in the background. This approach solves the limitations of prior work – By keeping the frontend in fast DRAM, and only recording operations in a log, the cost of persistence is significantly lowered. In addition, the decoupled persistence process prevents the need to quiesce the system and can deliver low tail latency.

DIPPER can be used to design fast and crash-consistent storage systems with PMEM. Our approach uses PMEM to store identical persistent shadow copies of DRAM structures. We use shadow updates for backend atomicity to avoid costly transactions and cache flushes. This process is seamlessly handled by our PMEM allocator. In this manner, the same code can be used to perform operations on both structures and the need to serialize

Name	Data Structure	Persistence Mode	Design Technique
Link-free [195]	linkedlist	both	dirty-bit
SOFT [195]	linkedlist	both	dirty-bit
TLog*	linkedlist	both	per-thread logging
Log-free*	linkedlist	eADR	atomic operations
Volatile [99]	ring buffer	none	atomic operations
TX*	ring buffer	both	transactions
TX-free*	ring buffer	eADR	per-thread status buffer

*Proposed in this thesis

Table 4.1 Summary of Lock-free Designs Evaluated. Persistence mode 'both' is equivalent to both ADR and eADR.

data is avoided. This also simplifies the backend design and prevents the need to hand-modify code to add cache flushes and transactions. Our design not only increases productivity, but also makes the process of keeping the volatile and persistent copies consistent much easier. PMEM bandwidth ($\sim$30GB/s for read and $\sim$10GB/s for write on our cluster) is comparable to DRAM, which implies that the checkpoint space can keep up with the system space.

We design a generic storage sub-system, called **DStore**, short for Decoupled Store which uses DIPPER to implement its control plane. We deploy DStore on a server with Intel Optane DCPMM. With the aforementioned techniques, DStore can reduce software overhead to $\sim$10%. Experimental results demonstrate that DStore can deliver up to 6x lower tail latency service level objectives (SLO) and up to 5x higher throughput SLO compared to state-of-the-art PMEM optimized systems like PMEM-RocksDB [141, 178], MongoDB-PMSE [78], and NOVA [179, 180].

4.1 Lock-Free Persistent Data Structures

In this section, we first discuss challenges associated with persistent lock-free data structure (PLFS) design. Then, we present the design and evaluation of two PLFSs – ring buffer and linkedlist. The intention of this case study is two fold. First, we want to highlight the difference in PLFS design between eADR and ADR persistence modes. Our findings show that designs which assume the presence of ADR may not provide the best performance in eADR mode, particularly for update heavy workloads. So, there is merit in tailoring designs for eADR mode. Second, we want to compare and contrast PLFS design techniques. Our results show that selecting an appropriate technique can significantly impact performance and should be done considering the workload IO pattern and persistence mode. Most importantly, our analysis shows that transactions and cache flushes on PMEM have high overhead. This observation inspired the design of DIPPER for data structure design as a means to avoid transactions completely and avoid the persistence cost of PMEM in the critical path.

4.1.1 Visibility v/s Persistence

Designing persistent lock-free data structures is challenging because commonly held assumptions about data structure behavior break down when recovering from a crash. In traditional concurrent volatile data structures, it is a common assumption that *visible* operations are complete. This guarantee is provided by hardware – visible memory changes are cache coherent, so they are visible to all threads. When adding persistence to the picture, this assumption no longer holds in ADR mode. Even if an operation is *visible*, it may not be *persistent*, because data may still be in volatile cache. A crash before persistence will prevent the operation from being visible upon recovery. Therefore, operations can only be

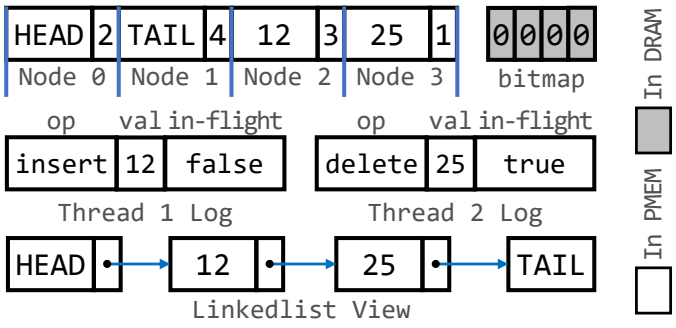
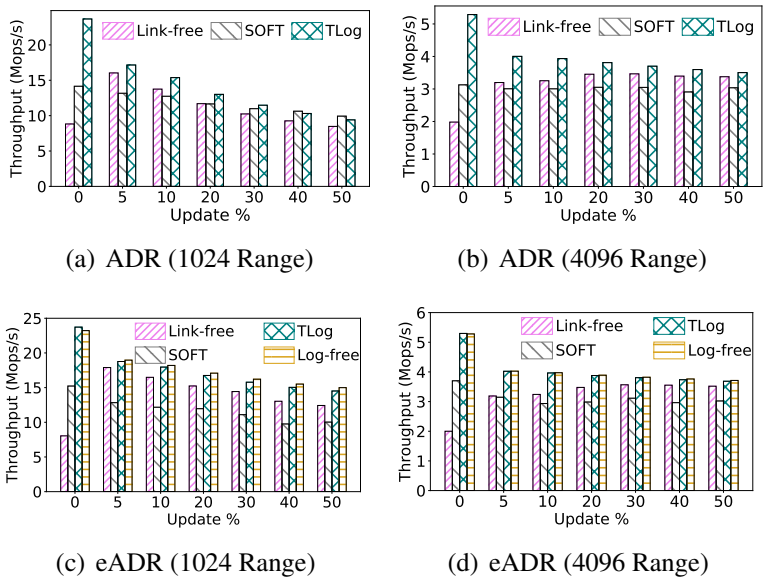


Figure 4.1 TLog Design Overview with 2 Threads. Thread 1 has completed its operation whereas Thread 2 is in-flight. 0 in the bitmap indicates that the memory region is allocated.

considered complete if they are both *visible* and *persistent*. The biggest challenge in the design of lock-free persistent data structures is to guarantee that visibility and persistence are atomic, i.e., an operation is either both or none. An exception to this is with eADR mode. In eADR mode, the volatile cache hierarchy is within the persistence domain. So, *visible* operations can be immediately considered to be *persistent*. In this case there is no need to flush the cache but store fences may still be required to force data out of the per-core store buffers, which are not part of the persistence domain. Even then, a compound operation, consisting of multiple atomic *visible* operations, can only be made atomic by using transactional techniques, such as journaling. In other words, if an operation does not have a single linearization point (instruction), then it is not atomic.

4.1.2 Design Techniques

There are several existing techniques in literature which can be used to implement PLFS. Providing atomicity and isolation guarantees requires the use of complex transactional systems, which often use locks for mutual exclusion. The use of locks implies that the data structure is no longer lock-free. Transactional abstractions which use redo/undo



(a) ADR (1024 Range) (b) ADR (4096 Range)

(c) eADR (1024 Range) (d) eADR (4096 Range)

Figure 4.2 Lock-free Linkedlist Throughput Evaluation

logging are one of the most common techniques but as many of these rely on locks internally they are not applicable. Some abstractions, such as PMDK's libpmemobj library [80] do not use locks but rely on per-thread buffers to maintain transaction state. Such abstractions can be used to maintain consistency and lock-freedom. Another recently developed technique is the *dirty-bit* design, as used in [173, 195]. In this approach, data is marked *dirty* when it is updated; threads which find the dirty bit set flush data to PMEM so that only committed data is read. Using per-thread scratch buffers for logging or status tracking is another approach that can be used. Since threads operate on independent memory regions, they can operate in a lock-free manner. Finally, for eADR mode, relying on atomic operations is sufficient to maintain consistency. Note that only using atomic operations may not be sufficient in this case because some information may be required to identify the data structure state on recovery from crash. Other techniques, such as per-thread logging may

also be necessary. Also note that in all designs atomic operations are required to maintain mutual exclusion.

In this section, we present designs for two truly lock-free persistent data structures – a multi-producer, multi-consumer ring buffer and a concurrent linked list. We chose these data structures because they are commonly used in database systems for several different purposes. Table 4.1 shows a summary of the different designs we evaluate. Designs with persistence mode 'both' were designed for ADR mode but can also be run in eADR mode by replacing all persistence barriers with store fences. The main point of this study is not to implement the best possible design but rather to compare and contrast different design approaches.

4.1.3 Lock-free Linkedlist

We consider a sorted linkedlist with any arbitrary structure as the value. Our lock-free linkedlist designs are extensions of Harris' algorithm [64] and support the same basic operations – `insert`, `delete`, and `contains`. The original algorithm uses the atomic `compare-and-swap` operations to maintain lock freedom. Node deletion is logical and requires subsequent garbage collection to reclaim memory. The logical deletion process involves marking nodes as *removed* by setting the least significant bit of the node's pointers. Our designs build upon the original algorithms to provide persistence. We propose one design for ADR systems, called TLog, and another for eADR systems, called Log-free. Both designs use the same memory allocator and base design.

TLog Design. TLog adds persistence barriers to ensure durability. A new node is flushed to PMEM before being added to the list. In addition, changes to a node's next pointer are also flushed. For maintaining operation atomicity, we rely on per-thread scratch buffers for

micro-logging operations. Before initiating a list operation, each thread stores the operation type and its parameters in its scratch buffer and marks it as in-flight. Once the operation is completed, the operation is marked as completed. On recovery, scratch buffers of all threads are examined and any in-flight operations are redone. Recovery is idempotent because insert and delete operations are both idempotent themselves.

Log-free Design. In eADR mode, the volatile cache is in the persistence domain. Insert and delete operations have a single linearization point (the `compare-and-swap` instruction). Therefore, we do not need to use transactions or journaling to maintain atomicity. We can use Harris' algorithm as is, but with store fences at linearization points. However, we still need to prevent PMEM leaks and allow PMEM address space relocation. Our memory allocator solves these two problems.

Memory Allocator Design. The memory allocator consists of a fixed size PMEM slab indexed using a volatile lock-free bitmap. The bitmap relies on atomic `fetch-and-and/or` instructions to retain lock freedom. Our memory reclamation algorithm uses epoch-based reclamation (EBR) [47] to maintain correctness and garbage collect unused memory. To prevent PMEM leaks, we keep the bitmap in volatile DRAM and rebuild its state upon recovery. This is done by walking through the linkedlist and marking the memory region for each node as allocated in the bitmap. In this manner, memory allocation does not require transactional support. Finally, to safeguard against PMEM address space relocation, we use relative pointers and pointer swizzling [32, 156].

Figure 4.1 shows an overview of the TLog design. As can be inferred from the figure, linkedlist nodes are allocated from the PMEM slab and use node offsets instead of pointers. In the example shown, two threads operate on the linkedlist. Thread 1 has completed its insert operation and marked the in-flight flag in its log as `false`. Thread 2, on the other

hand, has not completed its delete operation and its in-flight flag is still set to `true`. If there is a crash at this moment, then on recovery Thread 2's pending operation will be completed using data in the log. The Lock-free design is the same as shown in the figure, except that the per-thread logs are not required.

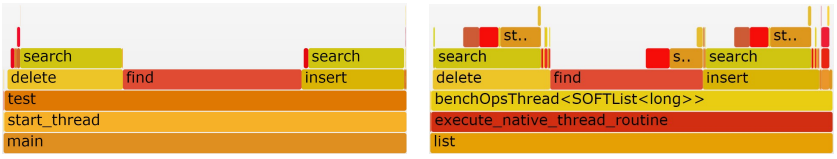
Evaluation. We compare our designs with two current state-of-the-art approaches, SOFT and Link-free, proposed in [195]. We measure the total throughput achieved by all implementations while varying the update ratio and value range at full subscription (28 threads) to simulate a heavy load scenario.

Figures 4.2(a) and (b) compare list throughput in ADR mode for 1024 and 4096 value ranges. TLog outperforms Link-free in all cases and SOFT in all but two cases. Link-free works similar to TLog but does not use per-thread logs for atomicity. Instead, find operations are required to flush the node (if not already persisted) before returning it. This increases the latency of find operations and reduces overall throughput. SOFT is an optimization of Link-free in that it does not flush node pointers but uses a valid flag per-node to indicate which nodes are part of the list. On recovery, all PMEM nodes are scanned to find valid nodes and are added to the list. SOFT reduces the number of persistence barriers required, so insert and delete operations are fast. However, it still requires find operations to flush nodes, increasing find latency. On the other hand, TLog uses micro-logging for atomicity and does not rely on find operations doing persist barriers. Therefore, TLog find latency is much lower than SOFT or Link-free but insert and delete latency is higher. This also explains why TLog performance gets closer to Link-free and SOFT as the update percentage is increased.

Figures 4.2(c) and (d) compare list throughput in eADR mode for 1024 and 4096 value ranges. TLog outperforms both Link-free and SOFT in all cases. This is because in eADR

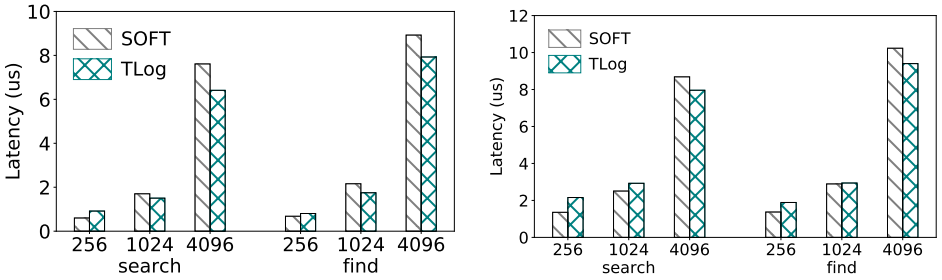
mode there is no need to flush cache-lines. So, insert and delete operations do not incur persistence overheads. SOFT and Link-free optimize insert/delete over find and are hence outperformed by TLog. We also observe that Log-free outperforms TLog in all but two cases, albeit only by a small margin. The Log-free design takes advantage of the eADR mode to avoid the micro-logging operation which TLog performs. By eschewing logging, Log-free achieves better throughput, particularly for 1024 value range. The Log-free design's improvement over TLog is only marginal because a majority of time is spent in search and find operations as opposed to persistence barriers.

To make complete sense of the ADR results, we first analyze the time-wise breakdown of each function call using a flame graph. Figure 4.3 shows the flame graphs for TLog and SOFT with 1024 value range and 50% updates. The stacked boxes show the function call trace and the width of each box is proportional to total time spent in that function. For both implementations, a majority of time is spent in iterating over the list (search and find operations). Persistence barriers and other operations comprise a very small fraction of overall time. Increasing value range or decreasing update percentage will further reduce this fraction of time. According to [68], a typical application using list-based sets performs 90% reads. This indicates that optimizing linkedlist traversal is more important than minimizing persistence operations. Both SOFT and Link-free focus on optimizing persistence, resulting in a sub-optimal design. On the other hand, TLog does not change search and find operations (as compared to Harris' algorithm), and shows better performance for find intensive cases. To verify this reason, we measure the latency of search and find operation while varying value range (256, 1024, and 4096) and update percentage (5 and 50). Results are shown in Figure 4.4. We find that on increasing value range SOFT latency increases



(a) TLog (b) SOFT

Figure 4.3 Flame Graph of Persistent Linkedlist Implementations with 1024 Value Range and 50% Updates



(a) 5% Updates (b) 50% Updates

Figure 4.4 Lock-free Linkedlist Latency for Search and Find

more for both operations as compared to TLog. These results confirm the reasons for the performance trends in Figure 4.2.

4.1.4 Lock-free Ring Buffer

We base our designs on Krizhanovsky's algorithm [99] for volatile lock-free ring buffers.

Volatile Design. The original algorithm uses per-thread head and tail pointers to maintain lock freedom. To perform a push or pop operation, each thread increments the global head/tail pointer using a `fetch-and-add` instruction and stores the old value in its local head/tail pointer. The local pointer indicates the queue slot which the thread will operate

on. Before operating on the location, the thread must make sure that it is safe to push or pop at that slot. To ensure this, two global variables (last tail and last head) are maintained to indicate the earliest push and pop operations still in-flight. Threads compute theses variables for each operation by iterating over all thread-local head/tail pointers. These variables are used to ensure that we do not push at a slot still being popped or vice-versa. Once each thread completes its operation, it sets its local head/tail pointer to `INT_MAX`. This allows other threads to push/pop that slot. Essentially, this store instruction is the linearization point of an operation where it becomes visible to other threads.

Persistent Design Overview. We propose two designs for our lock-free persistent ring buffer – a transaction-based design, TX, for ADR systems and a transaction-free design, TX-free, for eADR systems. For both designs, the ring buffer, global head/tail pointers, and thread-local pointers are placed in PMEM. Last head and last tail pointers are placed in DRAM instead of PMEM because they can be computed using thread-local pointers and hence do not need to be persisted. We also add two thread-local pointers in PMEM (push and pop position) to indicate the slot position where the thread is operating on. These pointers are used to identify the slots for which push/pop operations were interrupted in case of a crash. Memory management is fairly straightforward since the ring buffer is of a fixed size. We allocate PMEM for all necessary structure and pointers statically on application startup. The simplified memory allocation avoids PMEM leaks. In addition, we use indices instead of real pointers to allow PMEM address space relocation. Figure 4.5 shows an overview of the ring buffer design with 2 producers and 2 consumers. As shown in the figure, each producer/consumer has a private buffer which contains the local head/tail and position pointers. The position variable always indicates the slot which was being operated on. On recovery, the head/tail pointer is examined. If it is not `INT_MAX` (∞), then

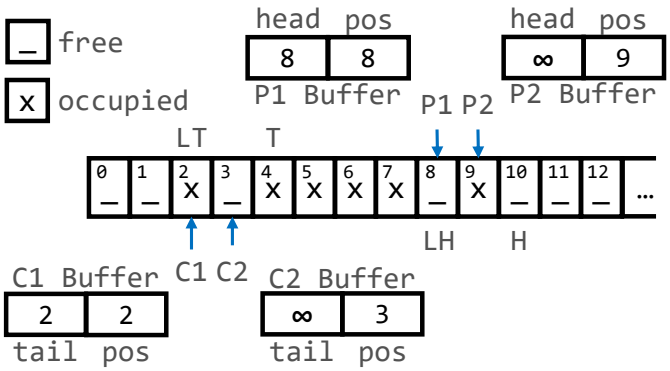


Figure 4.5 Ring Buffer Design Overview with 2 Producers and 2 Consumers: LT is last tail, T is tail, LH is last head, and H is head.

the corresponding operation was left incomplete. In this manner, all incomplete operations can be detected by scanning the buffers of all producers and consumers and appropriate steps can be taken to restore the state to produce a consistent view of the ring buffer.

TX Design. In this design, we wrap the critical section of push/pop operations with transactions for atomicity. We use PMDK's libpmemobj transactional bindings for this purpose, which use a combination of redo and undo logging to achieve atomicity. The use of transactions guarantees that there are no partially completed operations in case of a crash. On recovery, we examine the push/pop position pointers to determine which buffer slots were being operated on at the time of crash. We use this information to consolidate the buffer, i.e., copy data to remove holes, which can occur as a result of a subset of threads initiating their operations at the time of crash. Using PMDK transactions does not compromise lock-freedom because transactions use thread-local buffers to store internal state and avoid synchronization.

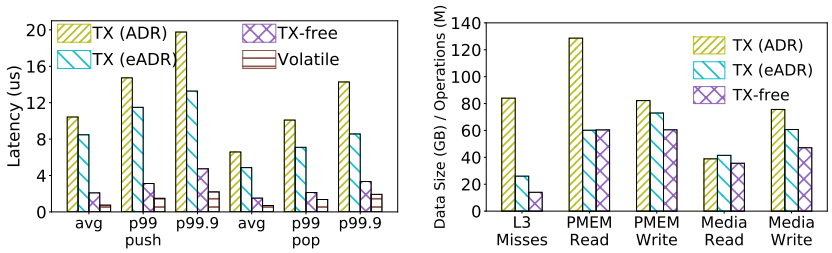
TX-free Design. The TX-free design follows the same principle as the TX design but does not require the use of transactions. Only store fences are required to implement persistence

barriers. This is because the volatile cache is within the persistence domain in eADR mode. Further, there is only a single linearization point, which involves setting the push/pop position to `INT_MAX`. On recovery, checking the values of the position pointers for each thread enables us to identify in-flight operations and roll them back. After the roll-back is complete, we consolidate the buffer, just as in the TX design.

Evaluation. To the best of our knowledge, TX and TX-free are the first lock-free persistent ring buffer solutions available. Thus, we compare the performance of the proposed designs with the volatile implementation in both ADR and eADR persistence modes. We use 4KB slot size and 32K slots so that the working set is 3-4 times larger than the LLC size.

We first examine the effects of flushing data with low temporal locality in eADR mode. We modify our designs to flush slot data (but not per-thread buffers or transaction state) on each push operation and measure latency. Figure 4.6 shows the normalized latency for TX and TX-free designs in eADR mode. Except for average push TX latency, all other latency values are similar or higher when slot data is not flushed from the cache. We find that not flushing non-critical data can increase latency. The reason for this is that data in the ring buffer has low temporal locality and competes with other critical data for cache space. This reduces the cache hit rate of the per-thread buffers and transaction state which have high temporal locality and increases overall latency. Push latency is less affected than pop latency because not flushing the slots removes an expensive persistence operation off the critical path for push operations. This is also the reason why TX has slightly better push latency. Finally, we can also observe that TX-free is less affected as compared to TX. This is because TX-free does not use transactions and hence has a lower overall memory footprint, so the cache hit rate of critical data is less affected. **Overall, flushing data with low temporal locality is good for lowering latency**. Although this observation appears

Figure 4.6 Effect of flushing data with low temporal locality in eADR mode – Results show ratio of latency when no data is flushed to latency when only slot data is flushed.



(a) Operation Latency (b) PMEM Hardware Counters

Figure 4.7 Lock-free Ring Buffer Evaluation – (a) Latency for 50% pops and 50% pushes with 4KB slot size at 28 threads, (b) Performance trends via low-level PMEM counters. The y-axis represents number of operations (in millions) for L3 misses and data size (in GB) for PMEM and media read/write.

trivial and has been observed in prior literature, its impact is more pronounced with PMEM because its latency is much higher than DRAM. Therefore, applying this observation can significantly impact performance. Based on these results, we flush slot data in eADR mode for all subsequent experiments.

We also measure the latency of each operation with 50% pops and 50% pushes at full subscription (28 threads). Figure 4.7(a) shows the average, p99, and p99.9 latency of different implementations. We observe that TX in ADR mode shows the worst performance.

This is expected because it requires expensive transactions and cache-line flushes to ensure atomicity and durability. All changes to PMEM data need to be logged and PMEM writes have high overhead. We can also observe that in eADR mode, TX-free has much better performance than TX. Even though cache-line flushes can be avoided for TX in eADR mode, the overhead of transactions is high. However, TX-free avoids both cache flush and transaction overheads and achieves near volatile performance. In fact, P99 latency for TX-free is only 2x that of the volatile design. To better understand the reasons for these performance trends, we examine low-level PMEM counters for the entire duration of our experiments. Results are presented in Figure 4.7(b). PMEM read/write represent data exchanged with the PMEM controller while media read/write represent data exchanged with the internal 3D-XPoint media in DCPMM. We find that TX (ADR) has the highest number of L3 misses and PMEM operations. In comparison, TX (eADR) significantly reduces L3 misses and PMEM reads. This is because we do not flush transaction data and per-thread buffers to PMEM, so load operations are more likely to be serviced from the CPU cache. PMEM and media writes are also reduced by avoiding some cache-flush operations. Finally, TX-free further reduces PMEM and media writes because transactional PMEM updates are not required anymore. An interesting observation is that the media reads remain largely unchanged for all implementations. This implies that reads are mostly serviced from the XPBuffer in DCPMM.

4.1.5 Key Insights

The main takeaways from our analysis are twofold. First, transactions on PMEM have high overhead; so, **transactions should be avoided to the extent possible**. Per-thread logging and dirty-bit design are two alternatives that can be used to avoid transactions. As

System	Persistence Technique	Type	Low Tail Latency	Fast Performance	Quiescent Freedom
MongoDB-PM [123], sqlite [139]	Periodic Async Checkpoint	Cached	×	×	✔
PMEM-RocksDB [141, 178]	Continuous Async Checkpoint	Cached	×	×	×
NOVA [180], Pronto [116]	Copy on Write (CoW)	Cached	×	✔	✔
MongoDB-PMSE [78]	Inline Persistence	Uncached	✔	×	✔
DudeTM [107], NV-HTM [25]	Decoupled Durability + HTM	Decoupled	✔	×	✔
DStore (proposed)	DIPPER	Decoupled	✔	✔	✔

Table 4.2 Comparison of related work

we observed from our evaluation, per-thread logging is more optimal for read-heavy workloads while the dirty-bit design is better for update-heavy workloads. Also, in some cases it may not be possible to avoid transactions. Therefore, choosing the correct design technique is important for both performance and correctness. Second, we conclude that ADR-based designs do not necessarily provide the best performance in eADR mode. **To attain optimal performance, algorithms should be specifically designed for the eADR persistence mode by taking advantage of the immediate persistence of any visible operation**. Our results show that this is more applicable for update-heavy workloads.

Hiding the persistence overhead. Evaluation of our lock-free designs has shown that they provide good performance. However, as we saw from the comparison in Figure 4.7(a), the overhead of persistence is quite high for the ADR design compared with the volatile design. To reduce this performance gap, we propose DIPPER, a decoupled approach to hide the overhead of persistence. We discuss DIPPER's design in detail in the next section.

4.2 DIPPER Design

This section proposes a new approach, called **D**ecoupled, **I**n-memory, and **P**arallel **PER**sistence (or DIPPER) for implementing persistent data structures that *fully decouples* the frontend and backend. We compare DIPPER with existing approaches in Table 4.2.

4.2.1 Key Ideas

We present DIPPER as an approach on PMEM to make a set of DRAM data structures persistent by only logging information necessary to perform an operation on this set. DIPPER generates a linear order for all operations, preserved in the log, and operates deterministically on the data structures, according to this linear order. In case of recovery from failure, this linear order, as preserved in the log, can be easily recovered and the same scheme can be used to reconstruct the state of DRAM data structures by treating all operations in the log as new requests. To be applicable in practical scenarios, DIPPER must solve two challenges. First, deterministic operation on data structures should not pose as a scalability bottleneck. Second, DIPPER must provide an atomic quiescent-free checkpoint solution to ensure uninterrupted service to end-users.

To solve these challenges, DIPPER fully decouples volatile system space and persistent checkpoint space both logically and physically. The system space to store data structures is entirely in DRAM while the checkpoint space (including the log) is entirely in PMEM. When a checkpoint is triggered, the checkpoint space is updated using the operations recorded in the log. Since the log replay is deterministic, the checkpoint space can be updated in the background without affecting the system space. The main idea is to hide the latency of persistence by using the volatile system space for all requests and taking checkpoints in the background. In this manner, requests operate directly on the volatile

version and experience DRAM-like latency, though write operations will experience a minor overhead for operation logging. The expectation is that the rate of write requests is low enough that the persistent shadow copy can be updated quickly and kept consistent with the volatile version. This is generally true since most requests arriving at storage systems are read requests [23, 29, 87]. Furthermore, write rates vary hugely, with some periods of low activity and some periods of bursty traffic [87, 89]. The volatile frontend can absorb bursty traffic easily and the persistent backend can then be updated during the periods of low activity in the background.

Our approach reduces the size of each log record, since only high-level operations and their parameters need to be logged. This reduces the log fill up rate. Further, the high bandwidth of PMEM lowers checkpoint cost and improves the rate of log clean up. DIPPER works with a PMEM backend and not SSD or disk backend, because only then is the rate of log fill up lower than the rate of the log cleanup (checkpoint). Thus, the checkpoint process can be completely overlapped with system operation.

The background checkpoint process does not compromise crash consistency for two reasons. First, the system logs all updates and log records are not discarded until a checkpoint is complete. Second, the checkpoint process is designed to be completely atomic. In this manner, we achieve quiescent-free checkpoints while maintaining fault-tolerance.

4.2.2 Architecture and Abstraction

Figure 4.8 shows the architecture of DIPPER. DIPPER is a *write-ahead* logging approach. Log records are filled *before* the start of a request and marked as committed *after* completion of the request. DIPPER relies on the concept of observational equivalence to allow concurrency while guaranteeing determinism and correctness (see section 4.2.7).

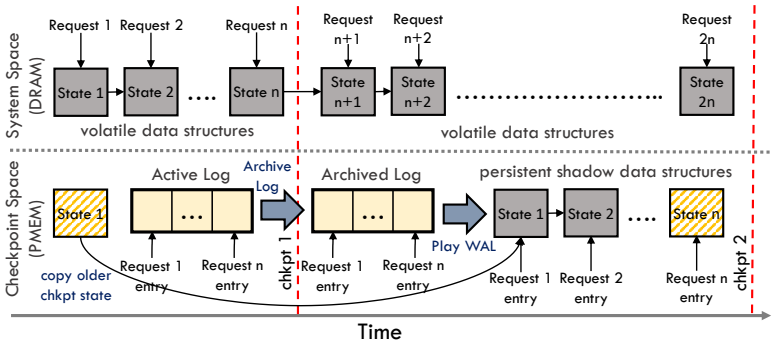


Figure 4.8 DIPPER and its atomic quiescent-free checkpoint architecture. Pattern filled states represent a checkpoint image.

DIPPER treats the set of DRAM data structures as a black box, logging only logical operations performed on this box and the input required from outside the box. To achieve fault-tolerance, DIPPER proposes a novel abstraction of shadow persistent data structures (or simply shadow copies) in PMEM, that represent a shadow copy of the volatile system space located in the persistent checkpoint space. We call these shadow copies because their state lags behind their volatile counterparts. These represent a snapshot of the system at a particular point in time and are only used for recovery in case of failure. During normal operation, the system operates purely on DRAM data structures while logging operations in the log. Each logical operations translates to a set of functions to be performed on each data structure. This mapping needs to be statically defined as it will be used by the recovery logic to update the shadow copies. In the background and in parallel to frontend operation, the operations in the log are applied to the persistent backend.

DIPPER is a derivative of logical logging. However, DIPPER differs from prior algorithms utilizing logical logging in one critical aspect. DIPPER leverages its determinism property along with byte addressable PMEM to decouple normal system operation

from checkpointing. This allows DIPPER to provide fault-tolerance while allowing requests to only operate on DRAM data structures. This is in stark contrast to other systems [70, 116, 123, 141] that only cache pages in memory that are eventually persisted.

4.2.3 Memory Management

Memory management is an important component of our design. We delegate several important functions to the memory allocator to make it easy for DIPPER to be applied to any data structure. Our backend design uses shadow updates for atomicity, so the PMEM allocator need not be crash consistent itself. This means that DIPPER can work with any off-the-shelf DRAM allocator. The same allocator can be used for both DRAM and PMEM management. Keeping both allocator designs the same makes it easier to reconstruct the volatile space from the persistent space in the event of a crash. We expect the allocator to implement two additional functions – one to iterate over all allocated memory regions and flush them to PMEM and the other to create a copy of the allocator state. The first function is used to ensure durability at the end of a checkpoint. The second function is used to for two purposes. The first is to avoid persistent memory leaks. During a checkpoint, a copy of the PMEM allocator is created along with copies of other data structures for recovery in the event of a crash. The second is to recover volatile space from the persistent space after a crash. In addition, to allow the data structures to be seamlessly copied and work inspite of PMEM address space relocation, we use relative pointers and pointer swizzling [32, 156] for both DRAM and PMEM structures. This means that we store offsets to the base address instead of pointers. On each pointer dereference, the base address is added to the offset to obtain the actual pointer to data.

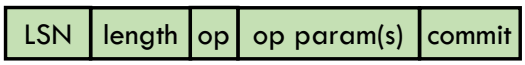


Figure 4.9 DIPPER log record: *LSN* is log sequence number, *length* is length of log record, *op* is operation type, and *commit* is committed flag. The generic nature of the record allows any arbitrary operation to be captured in the log.

4.2.4 Logging on PMEM

DIPPER relies on a PMEM resident log to record all processed operations. We exploit the byte addressability of PMEM to access log records in-place. Figure 4.9 shows the structure of a DIPPER log record. We capture each *operation* and its parameters within the log record. The LSN is used to verify the validity of log records. Consequentially, the LSN must be persisted atomically. With PMEM, only single word writes (usually 8B) are atomic. Furthermore, spurious cache line evictions can change the order in which portions of a log record are made persistent. To ensure that the LSN is persisted atomically, we flush cache lines containing each log record in the reverse order. We *write* and *flush* the LSN only after all other cache lines in the log record have been persisted. LSN is the first field in the log record. Thus, the log record will only be considered valid once the first cache line containing the LSN is flushed as the last step of the log write. Cache lines are flushed by calling `clflushopt` or `clwb`, followed by a store fence. Note that this is just one possible implementation for the log. DIPPER can work with any log implementation as long as arbitrary operations can be added to the log and records can be written atomically to PMEM. The Arcadia log design proposed in the previous section can also be used here directly, and indeed, the approach we discuss here does borrow several design choices made in Arcadia.

4.2.5 Atomic Quiescent-Free Checkpoint

To prevent the log from growing without end, checkpoints are triggered once the free space in the log fall below a pre-defined threshold. As shown in Figure 4.8, the checkpoint process begins by swapping the active and archived logs (this is fast and only involves a pointer swap), and moving any uncommitted log records to the new active log. Once this process is complete, frontend operations can proceed on DRAM data structures and the checkpoint is processed asynchronously. The checkpoint procedure involves playing *committed* records from the archived log on the shadow copies. A dedicated checkpoint thread pool is responsible for operating on the persistent backend in parallel with the frontend.

We leverage the byte addressability of PMEM to reuse the functions defined for each DRAM data structure operation. In this manner, the shadow copies iterate through the same states that the volatile copies went through. We guarantee durability by flushing all modified cache lines upon completion of the checkpoint process. This is done by iterating over all allocated pages in the PMEM allocator (including the allocator state) and flushing each cache line in the page to PMEM. Once all operations in the log have been processed, the shadow copies will have the same state as the DRAM structures had at the start of a checkpoint. Thus, the state of the shadow copies at the end of a checkpoint represents the checkpoint image. Correctness is guaranteed by leveraging the determinism property of DIPPER. To guarantee idempotency during a checkpoint, we always create a new copy of the shadow copies. In case of a crash during a checkpoint, the recovery logic uses the old persistent versions for recovery. A root object, placed in a well known offset in PMEM contains pointers to current and old copies of the shadow copies as well as the current state of the checkpoint process. Our PMEM allocator is responsible for flushing cache lines to ensure durability and creating copies of backend structures to ensure atomicity.

Finally, to achieve atomicity, we update the locations of shadow copies in the root object atomically and *only* upon successful completion of the checkpoint process. As is evident, checkpoints are processed in the background without significant impact on system throughput. Our evaluation (see section 4.5.3) verifies this claim.

This approach for implementing the backend has several benefits. First, since the representations of the DRAM and PMEM data structures are the same, the same code can be used for both. Further, by using shadow updates to maintain backend atomicity, the cost of consistency is significantly lowered. There is no need to ensure that data structures are updated in a consistent and durable manner for each operation. Therefore, complicated techniques, such as transactions, which are often employed to attain atomicity are not required. This simplifies the implementation of the backend and allows code written for volatile structures to be used as is for the persistent structures.

4.2.6 Idempotent System Recovery

The recovery process involves recovering the state of volatile data structures from their persistent counterparts. For recovery, the root object provides pointers to the data structures and the state of the checkpoint process. If we detect that the system crashed during a checkpoint, we first need to reconstruct the latest versions of the shadow copies (otherwise, this step is skipped). This is done by playing records from the archived log on the old copies of the shadow copies. Essentially, we redo the checkpoint procedure ongoing at the time of crash. Next, we recover the volatile state. This involves replicating the PMEM allocator state in the DRAM allocator and copying pages from PMEM to DRAM. Finally, we replay log records in the active log on the data structures to restore system state to what it was before the crash.

```
global ongoing-ops <- empty # list of ongoing operations
global logical-log <- empty # logical log
global log-lock <- free     # global log lock

function high-level-operation(op)
retry:
  grab log-lock
  for o-op in all ongoing-ops
    # if we do not violate system concurrency control
    # requirements, let o-op and op proceed in parallel
    if conflict(o-op, op) == true
      release log-lock
      wait until o-op completes
      goto retry

  # no conflict or conflict resolved
  add op to ongoing-ops
  add op to logical-log
  release log-lock

  # now we can execute op
  ...

  # once complete, remove it from ongoing-ops
  remove op from ongoing-ops
```

Listing 4.1 Algorithm for Applying Observational Equivalence in DIPPER

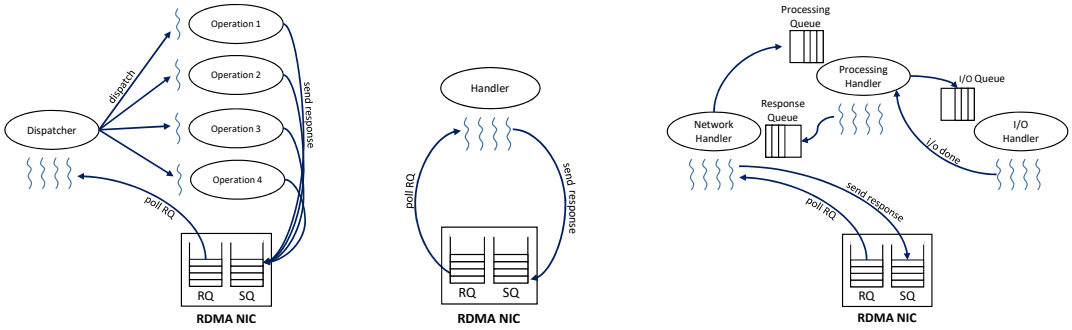
After recovery is complete, the system state is restored and new requests can be accepted. During recovery, we only play *committed* log records on the shadow copies. So, there is no need to log undo operations since the shadow copies represent a consistent checkpoint image (or in database terms, a transaction consistent checkpoint [142]). Hence, DIPPER can be thought of as a *redo-only* logging algorithm. The state of the system is defined exclusively by volatile structures which means that the recovery process is guaranteed to be idempotent.

4.2.7 Observational Equivalence and Concurrency

Supporting concurrent operations is a must for any practical solution. DIPPER supports concurrency through the notion of *observational equivalence* [145]. Using this notion, we can state that two data structure states are observationally equivalent if they both give the same answer to any observation from a prespecified set. Two operations on a data structure are said to be commutative or non-conflicting if reordering the operations results in observationally equivalent states. Commutative operations are permitted to operate concurrently on a data structure. The use of commutativity for concurrency is not new and has been widely studied [31, 157]. However, applying it to logical logging has been problematic. This is because if the volatile and persistent domains are not fully decoupled, then different portions of the persistent backend must be updated atomically since the log only contains operations and not actual data. Concurrent modification of data structures could lead to an inconsistent backend. Prior work [109] has shown how a write graph can be used to delay the persistence of data objects and increase concurrency. Nevertheless, this solution requires costly maintenance of the write graph state. Further, concurrency is limited when data are being made persistent. DIPPER overcomes these challenges by fully decoupling volatile and persistent domains. The volatile structures do not need to be involved to update the persistent backend. Therefore, commutativity can be fully exploited to increase concurrency. For instance, operations on distinct keys in a hashtable are non-conflicting and can be executed in parallel. The hashtable must, of course, support concurrency and avoid concurrent modification of a single bucket. In DIPPER, the implication is that log records are not required to be in serialized order but only conflicting order. So, not only can a single data structure be updated concurrently, by non-conflicting requests but different data structures can also be operated on in parallel. In case of recovery after a crash the exact

representations of the data structures will not be the same as those before crash, but the observable state of the data structures will be the same. This guarantee is sufficient to ensure correctness of the system [145]. A high-level overview of this algorithm is presented in Listing 4.1.

4.3 Thread Models



(a) Threaded Server Design (TS)

(b) Shared Threaded Server Design (STS)

(c) Functional Partitioning-based Design (FP)

Figure 4.10 Thread Models for System Design

Web services and applications are becoming more complex and their demand is steadily increasing. This requires the design of new systems that can handle this ever-increasing load and provide a robust and responsive service platform. Much of the previous work [104, 174] highlights the need for a highly decoupled and functionally partitioned (FP) model, such as Staged Event-Driven Architecture (SEDA). Such models are believed to ensure fair response times to clients while ensuring high request processing throughput. These

models are designed using processes and threads as the models for concurrent programming. Different functional activities are assigned to different pools of threads, thus utilizing the plethora of cores available in modern processors. However, such designs overlook the overhead of thread synchronization and data transfer between cores, particularly when each request takes only a brief amount of time to be processed. This is particularly true when using RDMA-based communication, where the advanced networking hardware provides extremely low latency communication and offload capabilities. The performance characteristics of RDMA networks require a re-evaluation of the concurrency architecture to ensure optimal response time and usage of hardware.

Traditional storage architectures were based on the popular functional partitioning design approach [104]. However, we argue that it might not be the best approach for achieving low latency of operations. Functional partitioning is considered to be particularly useful for ensuring fair response times to clients. This argument holds true when each request takes a significant amount of time to be resolved but breaks down if the request only takes a brief amount of time. In this scenario, the overheads of thread synchronization and data transfer between cores (cache thrashing) can lead to significant performance degradation. This is particularly significant when using fast storage devices such as NVMe drives or PMEM, where the latency is extremely low. This calls for re-designing the storage architecture, particularly for latency-critical operations.

To avoid any performance degradation and scaling to a large number of clients, we believe that a run-to-completion approach (i.e. single thread to process entire request) is the best solution. Using a single thread to process the complete request ensures that we do not run into any thread synchronization issues and prevents frequent cache thrashing. The thread-per-request model and bounded thread pool models are two of the most common

single-threaded designs. The thread-per-request model is well known for its simplistic design and application in web services [27]. The bounded thread pool model is a variant of the thread-per-request model which imposes a limit on the total number of threads. Our goal here is to show that these models work well for querying workloads on HBase [2], a NoSQL database. We evaluate three designs based on these single-threaded models as well as the default functional partitioning approach for scan operations. These designs are presented in Figure 4.10 and discussed in detail below.

Threaded Server Design (TS). This design is based on the thread-per-request model, utilized in several web servers. In this design, each operation consumes a thread, with the OS responsible for switching between threads to ensure fair response times for clients. While this approach is relatively easy to program, it can lead to significant performance degradation when the number of threads is large, owing to the overheads associated with threading. In this design, we have a dispatcher thread pool for reading operation requests from the network adapter and launching a thread to process the request in. Each operation thread is responsible for processing the complete operation including sending the result back to the client.

Shared Threaded Server Design (STS). While the threaded server design suffers from the threading overhead for a large number of threads, this design aims to solve this issue by using a bounded thread pool for operation processing. The entire design uses just one thread pool to process the entire operation. Incoming requests wait in the NIC's queues until one of the handler threads pick them up for processing. Limiting the number of concurrent threads leads to a more scalable and robust design as compared to the unconstrained threaded server design. It should be noted that the number of threads can and should be tuned.

Functional Partitioning-based Design (FP). Functional partitioning typically involves partitioning the application's activities into a set of distinct functions which are each processed by separate threads. The interaction between functional units usually takes place through shared queues and variables. In our case, network, computation, and IO can easily be identified as the main activities. Separate thread pools are assigned for these three tasks with communication between the thread pools using thread-safe queues.

To test our design with a real-world application, we use data provided by AdMaster Inc. [1], a marketing data technology company that uses Big Data to provide businesses with useful marketing data. The data consists of a set of events for users interacting with ads on smartphones. The overall dataset has more than 50 fields for each record. After discussion with AdMaster data scientists, we came up with four workloads which represent queries that are daily used by AdMaster to analyze their data.

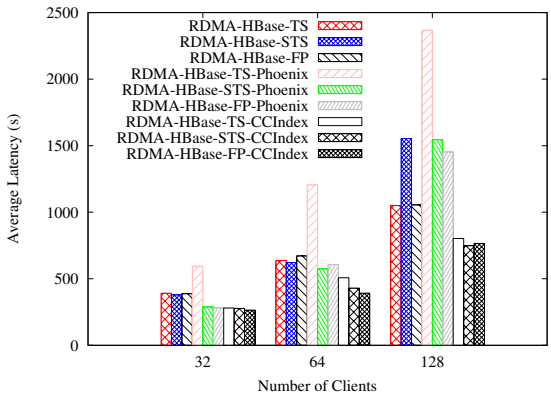


Figure 4.11 Performance Comparison of Thread Models with Multi-Client AdMaster Workload

To evaluate the performance characteristics of different thread models, we integrate them into HBase and design a multi-client workload which incorporates all four application workloads. In this workload, we run multiple clients such that each workload is being run by a quarter of the clients. We feel that this multi-client workload accurately embodies the characteristics of the load and request distribution received by AdMaster databases. This is because, in real world applications, users typically run different types of queries concurrently. We use 15M records from the AdMaster data for this evaluation. Results for this evaluation are presented in Figure 4.11. We also observe that STS and FP have similar performance for CCIndex [194] and Phoenix [3], while for HBase FP has the least average latency for 128 clients. TS performance suffers from the overheads of threading, particularly for 64 and 128 clients. To achieve optimal performance with STS, we observe that we need to increase the number of main handler threads from two to four. This is primarily because clients are running different workloads, thus more handlers are needed to ensure that a big request does not block other requests. Thus, while STS and FP performance are similar, STS still uses significantly less number of threads. Based on these results, we use the STS thread model in the design of DStore, which is discussed in the next section.

4.4 DStore

In this section, we present the design and implementation of DStore and its integration with DIPPER.

4.4.1 Overview

To demonstrate the effectiveness and efficiency of DIPPER, we propose a new fast and durable object storage sub-system, called DStore. DStore is designed with the goal of being a generic fault-tolerant embedded storage sub-system.

API Function	Type	Description
ctx ds_init()	environment	Initialize *context* for application thread
void ds_finalize(ctx)	environment	Terminate *context*
OBJECT oopen(ctx, name, size, op)	filesystem	Open an *object* for reading, writing, or both
void oclose(OBJECT)	filesystem	Close an *object*
ssize_t oread(OBJECT, buf, size, offset)	filesystem	Partial reads on *object*
ssize_t owrite(OBJECT, buf, size, offset)	filesystem	Partial writes on *object*
ssize_t oget(ctx, key, value)	key-value	Get *value* for *key*
ssize_t oput(ctx, key, value, size)	key-value	Set *value* for *key*
int odelete(ctx, name)	key-value	Delete *object* or *key*
int olock(ctx, name)	concurrency control	Acquire *lock* on *object*
int ounlock(ctx, name)	concurrency control	Release *lock* on *object*

Table 4.3 DStore Interface Overview

API and Semantics. We do not opt for POSIX compliance so as to avoid the shortcomings of POSIX IO [16, 187]. The growing popularity of simpler cloud services which offer access to objects instead of files needs to be recognized. As a consequence, we propose a new set of APIs to provide scalable access to objects. Table 4.3 lists the main API for accessing DStore.

The API and semantics of DStore have been specifically constructed to handle a wide variety of use cases and requirements. We provide both key-value and filesystem style APIs while storing the data as *objects*. Unlike object stores such as OpenStack Swift [135], DStore treats *objects* as modifiable entities. The primary difference between the key-value and filesystem API is statefulness. The key-value API does not require an *object* handle, releasing DStore of the arduous task of tracking open handles. This allows applications to achieve much higher scalability.

The *oopen*, *oclose*, *oread*, and *owrite* primitives are semantically related to their filesystem counterparts. The *oget* and *oput* primitives are derivatives of the standard kev-value functions, *get* and *set*. Each thread submitting IO needs to initialize a context for submitting

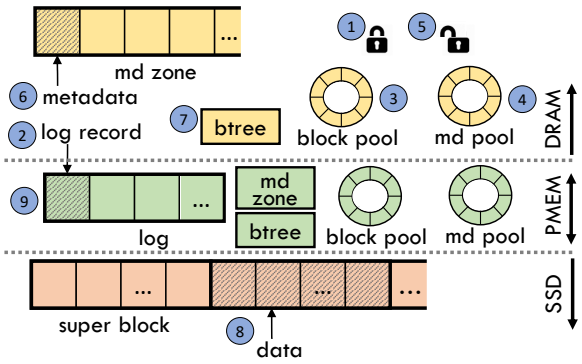


Figure 4.12 DStore layout and steps followed by a write request

requests using *ds_init*. For concurrency control, *olock* and *ounlock* primitives are provided to specify complex dependencies between objects. Concurrency control for a single object is implicitly handled by DStore.

Use Cases. Due to lack of a complete POSIX interface, we do not envision that DStore can be a direct filesystem replacement, but rather a system complementary to a filesystem which can handle workloads requiring higher scalability and performance. It can also be integrated into existing file and database systems. The key-value API enables DStore to be directly used as a persistent key-value store. The low latency and scalability of DStore can be utilized in checkpoint-recovery systems [20, 62]. In addition, distributed file and object storage systems can easily be modified to use DStore. We also provide plugins for YCSB, HDFS, and fuse over DStore to allow existing applications to run seamlessly.

4.4.2 Data Layout

DStore distributes data and internal structures between DRAM, PMEM, and SSD (see Figure 4.12). Data are stored purely on SSD because of its capacity and non-volatility. SSD pages are grouped into *blocks* which are the unit of data allocation in DStore. The

first block is reserved for the superblock, which contains relevant recovery information about the system. Most importantly, it contains the PMEM root object which is required for recovery in case of failure. A block pool is used to manage the SSD *blocks*. To store metadata pages, an metadata zone is used along with an metadata pool to allocate free entries in the metadata zone. The metadata and block pools are circular buffers containing free blocks and metadata pages. An PMEM-based log is used for recovery in case of failure. For maintaining an index of objects in the system, we utilize a btree. All data structures including the metadata and btree are stored in DRAM with their shadow copies stored in PMEM. Essentially, DStore implements its control plane using DIPPER, with the frontend in DRAM and backend in PMEM, while the data plane is placed on a high-capacity SSD.

We use a DAX filesystem formatted on PMEM for space management. We directly map a part of PMEM into the address space of the system by using `mmap` on a file located on the filesystem. To allocate memory for shadow copies, we use a simple slab-based memory allocator. This allocator uses the `mmap`ed memory and creates slabs in different size classes that are a power of two. For DRAM management, we use a similar slab-based allocator but with slabs allocated from the volatile heap.

DStore does not utilize a write cache, but writes directly to the internal DRAM-based write cache in SSDs, providing significant data transfer time savings. We find that SSDs with internal DRAM have enhanced power-loss data protection [77]. In the event of power failure, device capacitors will assist in flushing write cache data to non-volatile storage. DStore transparently leverages device capacitance to reduce the overhead of crash consistency.

4.4.3 Exploiting DIPPER in DStore

DStore uses DIPPER to make all DRAM structures shown in Figure 4.12 persistent. Data are updated *in-place* and are thus omitted from the log. We write log records for *oopen*, *owrite*, *oput*, and *odelete* operations. Log records for *oopen* and *owrite* are only written if they modify an metadata. The input parameters (excluding data) for all operations are stored in the log record. In our design, the size of each log record is just 32B plus the object name. In practice, we expect most log records to fit within a single cache line.

In DIPPER, a write operation works as follows (see Figure 4.12): ① Lock the block and metadata pool, ② allocate and write the log record, ③ allocate blocks from block pool, ④ allocate pages from metadata pool, ⑤ unlock the block and metadata pool, ⑥ write metadata with allocated blocks in metadata zone, ⑦ write btree record to memory, ⑧ write data to SSD, and ⑨ commit and flush log record to PMEM. Steps ③, ④, ⑥, and ⑦ are responsible for constructing a metadata and btree entry. Exactly the same set of steps is used to reconstruct metadata and btree state upon recovery from failure.

As we can infer from the description above, the btree and metadata zone are updated in parallel outside the synchronous region. This parallelism is achieved by using the observational equivalency property of DIPPER. Our concurrency control algorithm (see section 4.4.4) ensures that log records are added in conflicting order to maintain correctness.

4.4.4 Concurrency Control

We propose a concurrency control (CC) algorithm, which forwards information readily available in the PMEM log to determine concurrently executing operations. Our goal while designing this algorithm is to minimize additional memory usage and keep the latency for detecting conflicting requests minimal. Most systems use per-object locks to provide

concurrency control which increase linearly with the number of *live* objects in the system. In contrast, our CC algorithm embeds a lock flag within the log records for each request. The number of locks is therefore limited by the size of the log. Conflicting requests do not use a hold and wait approach, but rather spin on dedicated flags corresponding to their conflict.

Write-Write Conflicts. The log contains records of all operations currently in execution. When a new request arrives, we scan the log to check if any operations in execution are operating on the same object. If so, we spin on the committed flag of the conflicting record until the operation finishes. This ensures that we do not have two concurrently executing requests on the same object. However, scanning the complete log to check for conflicts is inefficient and unnecessary. Scanning from the first uncommitted record until the end of the log enables us to detect conflicting operations *without* adding significant overhead.

Read-Write Conflicts. Since read requests are not added to the log, read-write request conflicts can still occur. For resolving read-write concurrency, we introduce a new in-memory hash table that maps object names to their current read count. The read count is updated using the atomic `fetch-and-add` instruction to ensure consistent count values during parallel operation by threads. By looking at the read count at the start of a write request, we can be sure that no request is reading that object at that time. In case the read count is non-zero, we simply poll on it until it is zero.

4.4.5 Additional Design Considerations

In this subsection, we discuss some of the additional considerations that were made while designing DStore.

Inter-Object Dependencies. Having the ability to specify complex inter-object dependencies is invaluable for many applications. For example, in a filesystem, dependencies between a file and its directory are captured by locking the directory before modifying the file. Such cases can be handled by using the *olock* and *ounlock* primitives to lock objects. To implement these primitives we introduce a novel `NOOP` log operation. This operation represents a `NOOP` on the DRAM data structures and is ignored by DIPPER recovery logic. The *olock* primitive places a `NOOP` record in the log and *ounlock* marks this record as committed. Thus, a log scan can correctly identify locked objects as conflicts.

Request Processing Model. For improving scalability and latency, we use the run-to-completion STS thread architecture, proposed in section 4.3, to implement the IO processing pipeline completely in userspace. For optimal IO performance, we utilize the NVMe [131] protocol to interact with flash hardware directly.

Durability and Consistency. DStore writes data directly to storage device-level RAM (if available, otherwise to non-volatile media). In the event of an unexpected crash, device capacitors will safely flush data to non-volatile media. By eschewing buffering, DStore provides strong guarantees of data durability. Further, DStore only marks log records as committed once data is made durable. This implies that metadata will always be consistent, i.e., objects can never contain garbage data.

CoW Design. To enable fair comparison of DIPPER with CoW checkpoints used in related work, we implement CoW in DStore. This works as follows. When a checkpoint is triggered, all volatile pages in the frontend are marked as read only. As soon this is done, the frontend can process write operations again. When a client tries to modify a read-only page, a page fault is triggered and a handler copies the page to PMEM. Clients can assist in

this copying process, but must wait until the page is copied before making any modification to it.

4.5 Experimental Analysis

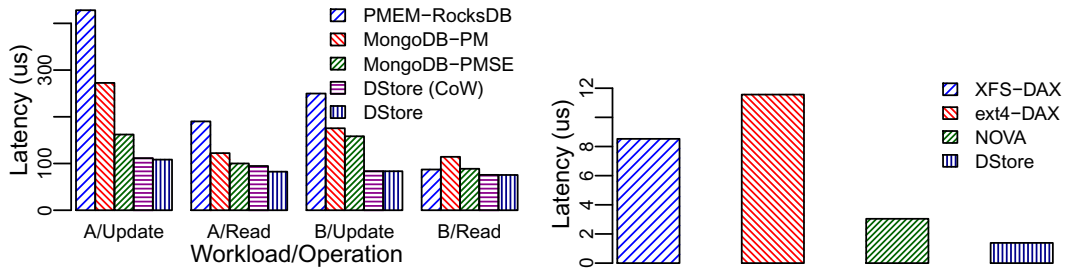
In this section, we present the evaluation of DStore.

4.5.1 Experimental Testbed

Our experimental testbed consists of a Linux (3.10) server equipped with two Cascade lake CPUs (8280L@2.70GHz), 384GB DRAM (2 x 6 x 32GB DDR4 DIMMs), and 6TB PMEM (2 x 6 x 512GB DCPMMs) configured in App Direct mode, and a 750GB Intel P4800X NVMe drive. Each CPU has 28 cores (with hyperthreading disabled) and 38.5MB of L3 cache (LLC). All available PMEM is formatted as two `xfs`-DAX filesystems, each utilizing the memory of a single CPU socket as a single interleaved namespace.

All code was compiled using gcc 9.2.0. PMDK [75] 1.8 was used across all evaluations to keep comparisons fair. Hardware counters were obtained using a combination of Intel VTune Profiler [74] and Linux iostat utility. DCPMM hardware counters were collected using the `ipmwatch` utility, a part of the VTune Profiler.

We decided to compare performance with at least one system from each category in Table 4.2. We directly compare DStore with three popular PMEM-optimized NoSQL storage systems, PMEM-RocksDB [141, 178], MongoDB-PM [123], and MongoDB-PMSE [78]. PMEM-RocksDB is an optimized version of the log-structured merge (LSM) tree-based vanilla RocksDB [45] and uses a PMEM resident log to improve performance. MongoDB-PM uses an optimized btree-based WiredTiger engine with the btree index and journal placed in a DAX filesystem formatted on PMEM to improve performance. MongoDB-PMSE uses PMEM optimized data structures to store data in-place and uses PMDK's



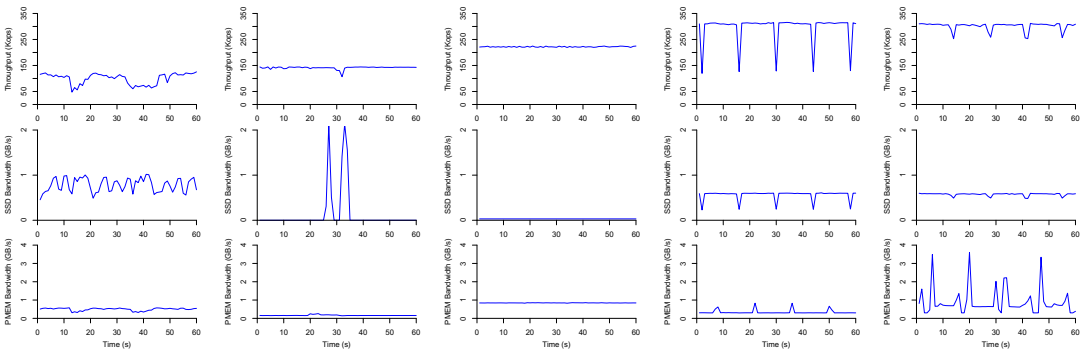
(a) YCSB Operation Latency (b) Metadata Overhead

Figure 4.13 Latency Evaluation

pmemobj-cpp library for crash consistency. We also implement the copy-on-write checkpoint scheme used in NOVA [180] and Pronto [116] in DStore for comparison purposes. Finally, we also compare metadata overhead with the Linux direct-access (DAX) filesystems `xfs` [161], `ext4` [115], and NOVA [179].

4.5.2 Is DStore Fast?

To evaluate DStore performance, we measure the average latency of 4KB read and update operations at full-subscription (28 cores) with YCSB [34] workloads A (50% read, 50% write) and B (95% read, 5% write). We compare DStore latency with that of other PMEM-optimized systems. From Figure 4.13(a), we can observe that DStore provides the best latency in all cases, up to 4x lower than other systems. The reason for this is that metadata requests experience ultra-low latency since the frontend is entirely in DRAM. In contrast, other systems must access persistent storage for metadata updates, which increases latency. For instance, MongoDB-PMSE must update both data and metadata in PMEM for each operation. This is also the reason that we see higher improvement for update than read operations. In general, update latency is lower for workload B compared to



(a) PMEM-RocksDB (b) MongoDB-PM (c) MongoDB-PMSE (d) DStore (w/ CoW chkpt) (e) DStore

Figure 4.14 System throughput and storage bandwidth over a 1 minute window for a full-subscription (28 cores) 50% read, 50% write workload

A for all systems because the high read:write ratio implies that the cost of persistence can be more easily overlapped with updates to the volatile cache. Finally, we also observe that DStore with copy-on-write checkpoints provides nearly the same latency as DStore. This is because checkpoint design only impacts tail latency and not average latency.

We also compare with PMEM-optimized DAX filesystems (`xfs`-DAX, `etx4`-DAX, and NOVA) to evaluate the filesystem interface of DStore. Since these filesystems place data in PMEM while DStore does so in SSD, we were unable to make a direct comparison. Instead, we measure the metadata overhead of 4KB writes to a file for each system. Figure 4.13(b) shows this comparison. Just like the previous experiment, we find that DStore is the fastest in terms of updates to metadata. This is because updating metadata only requires making changes to in-memory data structures and recording the operation in the log. In contrast, other systems need to update changes to PMEM because their volatile and persistent domains are not decoupled. For instance, NOVA must update the file's inode as well as add the operation to the inode's log, both of which must be made in PMEM for durability.

Size	Time	NVMe Write	BTree	Metadata	Log Flush	Total
4KB	Time (cycles)	24029	789	807	1663	27288
	Time (ns)	8899.63	298.89	292.22	615.93	10106.67
	% of Total	**88.06**	2.96	2.89	**6.09**	100
16KB	Time (cycles)	108844	1284	789	1945	112862
	Time (ns)	40312.60	475.56	292.22	720.38	41800.74
	% of Total	**96.44**	1.14	0.70	**1.72**	100

Table 4.4 Time breakdown of write requests

`xfs`-DAX and `ext4`-DAX suffer from similar limitations. Overall, we find that by keeping metadata in DRAM and using compact logical logging, DStore significantly reduces operation latency compared to other systems. Through experimental analysis, we also discover that our userspace run-to-completion pipeline is successful in avoiding context switches in the critical path, which also contributes to latency reduction.

Why is DStore Fast? To better understand why DStore is fast, we analyze its write pipeline in more detail. Table 4.4 shows the latency breakdown of 4KB and 16KB writes. The time spent in each component is given in cycles, nanoseconds, and as a percentage of total time. The right-most column shows the total time for the write request. Metadata indicates the time required to allocate blocks and update the corresponding metadata. Log flush represents time spent in flushing the log record to PMEM. What stands out the most is the percentage of time spent doing NVMe writes. This indicates that we are successfully able to reduce software overhead to $\sim$10% of total time. We also observe that a log flush takes less than 2000 cycles (or 740 ns), minimizing the impact of logging on write performance. This also indicates that 3D-XPoint technology provides extremely low latency for a single cache line flush. Finally, we find that the metadata and log flush overheads are similar for both IO sizes. This is because the use of logical logging in DIPPER leads to request size

agnostic software overhead. Therefore, we conclude that having a DRAM frontend with the entire metadata in the reason for DStore's fast performance.

4.5.3 Is DStore Quiescent-Free?

To measure the effects of checkpoints on system throughput, we conduct an experiment with a full-subscription (28 cores) 50% read, 50% write workload. We measure the aggregate throughput over a 1 minute window. We chose a 1 minute window because it was long enough for each system to trigger a checkpoint at least once. The first row of graphs in Figure 4.14 shows the result of this analysis. We can clearly observe that DStore is able to sustain higher throughput than other systems for the entire time window. The troughs in the graph represent periods of checkpoint. DStore throughput drops slightly during a checkpoint because of the impact of background threads applying operations to the backend. Nevertheless, even the lowest throughput achieved is greater than the highest of any other system, i.e. DStore is able to satisfy a high throughput SLO. Importantly, the system never fully quiesces for both read and write operations. Other systems are unable to achieve high throughput because of the reasons discussed earlier (see section 4.5.2). Apart from MongoDB-PMSE, all systems experience throughput drops during checkpoints. MongoDB-PMSE uses inline persistence, so no checkpoints are required and the throughput is consistent over time. Despite this, the overheads of cache flushes and transactions prevent it from achieving good performance even though it places data on PMEM. DStore delivers 15% higher throughput SLO compared to MongoDB-PMSE and is more cost effective because it places data on SSD. Another observation we make is that the copy-on-write design significantly lowers throughput during checkpoints. This is because client threads need to block and wait until the pages they want to modify are made

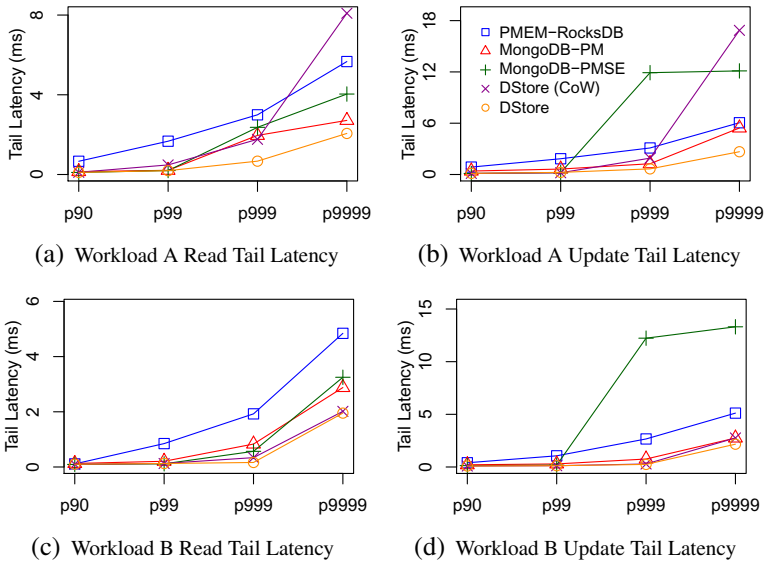


Figure 4.15 Tail latency curves at full-subscription (28 cores) for YCSB workloads A (50% read, 50% write) and B (95% read, 5% write)

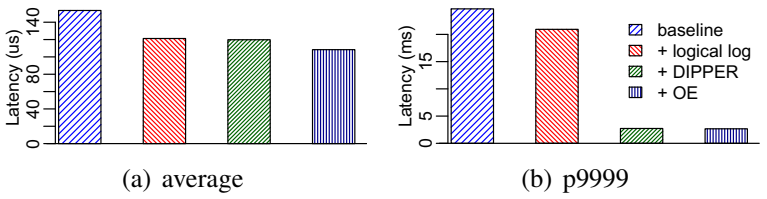


Figure 4.16 Effect of optimizations on write latency

durable. Finally, we find that the continuous background compaction in RocksDB prevents the frontend from achieving consistent throughput. In fact, for a short duration, it was unable to serve any update requests, violating quiescent freedom.

Why is DStore Quiescent-Free? To understand our observations further, we measured the SSD and PMEM bandwidth during the experiment. The SSD bandwidth for PMEM-RocksDB and MongoDB-PM clearly show the activity of the asynchronous checkpoints,

which explains the dips in throughput during them. MongoDB-PMSE does not use the SSD at all as it places all data in PMEM. Interestingly, for DStore (including CoW case), the SSD bandwidth curve mirrors the throughput curve. This is because data is directly updated in the SSD for each request and metadata is only placed in DRAM/PMEM. If we look at the PMEM bandwidth, we notice that except DStore, other systems utilize only a small percentage of available bandwidth. The reason for this is that their throughput is limited by other factors, such as checkpoints or transactions. These systems are unable to effectively utilize the byte-addressability and performance of PMEM. Even CoW does not utilize PMEM effectively because pages are flushed individually when page faults are triggered and not in batches. DStore is able to better exploit its performance by using shadow updates for the backend. In this manner, the backend throughput is not limited since transactions are not needed. We also observe that the PMEM backend is not always active which indicates that DStore can handle workloads with an even higher write:read ratio without quiescing. We conclude that logical and physical decoupling of normal operation and checkpoints allows them to be completely overlapped. As a result, DStore can provide uninterrupted service to end users.

4.5.4 Is DStore Tailless?

We measure the tail latency of 4KB read and update operations at full-subscription (28 cores) with YCSB workloads A (50% read, 50% write) and B (95% read, 5% write). Figure 4.15 shows the tail latency curves for both operations. Overall, DStore has flatter tail latency curves and also the lowest latency values for all cases, up to 6x lower than other systems. The reason for this trend is the decoupled design, which is successful in hiding the latency of persistence. In contrast, other systems have longer tails because of the effects

of checkpoints on client requests. Requests arriving during checkpoints must wait for data to be persisted, resulting in this trend. Some other interesting observations we make are as follows. CoW shows high p9999 latency for the update heavy workload A but close to DStore latency for the read heavy workload B. This is due to a reduction in the frequency of checkpoints for workload B as compared to A. We find that read tail latency is also worse for other systems as compared to DStore. This implies that checkpoints impact both read and write requests. Finally, we observe that MongoDB-PMSE has high p999 and p9999 latency despite using an uncached design. We believe this trend is because of the high tail latency of PMEM itself and not the software design. The high tail latency of Optane DCPMM has been verified in [183].

Why is DStore Tailless? For improving write latency, we proposed effective optimizations, including PMEM-based logical logging, decoupled persistence (DIPPER), and observational equivalency (OE). To determine the impact of these optimizations on system performance, we first evaluate the baseline design, adding optimizations one-by-one and measuring performance again. The naïve baseline uses ARIES-style physical logging [51] with CoW checkpoints. Figure 4.16 gives us an idea of the impact of optimizations on both average and p9999 latency at full-subscription. The naïve design has the worst performance. This is due to the high log write latency and overhead of CoW checkpoints. Moving from physical to compact logical logging improves average latency by 21% and tail latency by 15%. Incorporating DIPPER (+DIPPER) on top of this improves tail latency significantly (∼7.6x). DIPPER impacts tail and not average latency because it only improves response times for requests during a checkpoint. Finally, adopting OE completely removes any synchronization overhead, further improving average and p9999 latencies by 9% and 2%, respectively. OE particularly helps at high concurrency by allowing parallel

System	Shutdown Type	Metadata	Replay	Total Time
PMEM-RocksDB	clean	0	156	156
MongoDB-PM	clean	474	139	613
MongoDB-PMSE	clean	985	0	985
DStore	clean	846	841	1687
PMEM-RocksDB	crash	5013	150	5163
MongoDB-PM	crash	7525	12786	20311
MongoDB-PMSE	crash	1708	0	1708
DStore	crash	11580	861	12441

Table 4.5 System recovery time (in ms): metadata is time to recover metadata and replay is time to replay log records.

operation on the btree and metadata zone. We conclude that logical logging is the most beneficial for average latency while DIPPER is the most beneficial for tail latency.

4.5.5 Recovery Performance

To test recovery performance, we evaluate two cases: one with a normal shutdown and the other with an unexpected crash. We simulate system failure by forcing a hard shutdown (`SIGKILL`) and restarting the process. For this experiment, we load two million 4KB objects into each system. Table 4.5 shows system recovery times for both cases mentioned earlier. When recovering from a clean shutdown, DStore must reconstruct its volatile space from the persistent space. Other systems do not have this overhead because they only bring data into the cache on-demand. For this reason, recovery from a clean shutdown takes longer for DStore. In case of failure during a checkpoint, all systems take longer to recover. This is because any operations in-flight need to be re-executed and lost data must be reconstructed using the log. For DStore, the ongoing checkpoint process during a crash must be redone upon recovery in addition to the normal recovery process. MongoDB-PM and PMEM-RocksDB must recover any lost volatile data by replaying records from the

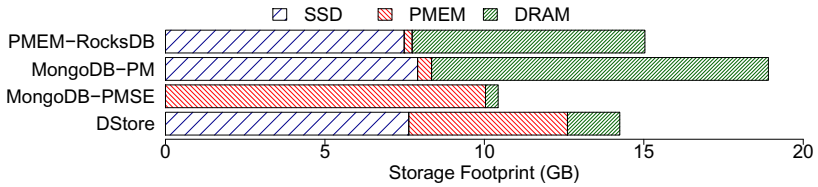


Figure 4.17 Storage footprint with 2M 4KB objects

log. All three have a similar recovery procedure and therefore show similar performance. MongoDB-PMSE only needs to re-execute in-flight operations using transaction data and recovers the fastest. In general, the two-level design prevents instant recovery for DStore. Since recovery is not the common case, we believe that the current design provides adequate recovery times but leaves room for improvement in the future.

4.5.6 Storage Footprint

DStore maintains two copies of metadata (in DRAM and PMEM) and uses shadow updates for the PMEM copy. We consider it important to measure the storage overhead and compare it with other systems. To evaluate the physical storage footprint, we load two million objects into the system and then measure the total space (DRAM, PMEM, and SSD) consumed by each system. Figure 4.17 shows the result of this analysis. Default settings are used for all systems. Interestingly, we find that all systems have similar storage footprint. Overall, DStore only consumes more space than MongoDB-PMSE. MongoDB-PMSE consumes the least space because it does not require a volatile cache. While the actual data storage footprints for all systems are virtually the same, the metadata overheads differ significantly. Both PMEM-RocksDB and MongoDB-PM reserve a large chunk of DRAM as their cache space but only actually utilize a small portion of it. For this reason, both have a higher storage footprint than DStore. In the worst case, DStore may need space

System	Throughput	p9999 Lat.	Recovery Lat.	Space Ampl.
MongoDB-PM	106704 IOPS	5439 us	20311 ms	2.47
MongoDB-PMSE	219221 IOPS	12119 us	**1708 ms**	**1.36**
PMEM-RocksDB	47532 IOPS	6055 us	5163 ms	1.97
DStore (CoW)	119685 IOPS	16863 us	12441 ms	1.86
DStore	**252832 IOPS**	**2665 us**	12441 ms	1.86

Table 4.6 Summary of achievable service level objectives

for three copies of metadata. However, since the space is allocated ad-hoc, this overhead is kept to a minimum. Further, the performance benefits of the decoupled approach far outweigh the drawbacks of additional PMEM usage.

4.5.7 Evaluation Summary

To get a complete picture of how different systems might perform in a cloud setting, we analyzed their achievable SLO. Table 4.6 provides a summary of these for throughput, p9999 and recovery latency, and space amplification[6]. These represent the worst case values we obtained in our experiments. For each metric, the best values have been highlighted. DStore achieves the best throughput and p9999 SLO. This is because the use of DIPPER prevents sudden drops in throughput and keeps tail latency low, resulting in consistent and predictable performance. For recovery and space amplification, MongoDB-PMSE is able to delive the best SLO. This is expected because it does not utilize a cache and data is persisted inline. Therefore, there is no storage overhead of the cache and recovery can be near instantaneous. DStore (CoW) has the same recovery and storage overhead as DStore because it uses the same recovery and memory allocation design. However, the CoW design is unable to deliver the same performance characteristics as DIPPER.

[6]We define space amplification as the ratio of size of application data to the size of space utilized by the storage system across DRAM, PMEM, and SSD.

Key Takeaways. Our results indicate that using a decoupled design is optimal for throughput and latency SLO while an uncached design is optimal for recovery and space SLO. Cached approaches can provide fast performance, but because of inconsistent performance during checkpoints, they are ineffective in providing reasonable SLO. Depending on the required tradeoffs, a decoupled or uncached design should be chosen for storage systems. Overall, we conclude that DStore provides the best performance SLO, good space SLO, and adequate recovery SLO and satisfies the three requirements we set out to achieve.

4.6 Related Work

Over the last decade, a significant body of work has attempted to design transactional abstractions [25, 32, 49, 107, 169], persistent data structures [28, 36, 48, 63, 110, 116, 148, 168, 195], and file, key-value, and database systems [30, 33, 41–43, 50, 70, 71, 86, 139, 175, 176, 179] for PMEM. All of these systems can be classified as either cached, uncached, or decoupled.

Cached Systems. Despite the byte-addressability and performance benefits of PMEM, cached systems remain a popular class of storage systems. Several cached systems [30, 36, 43, 70, 116, 123, 141, 176, 178] have been recently proposed to leverage PMEM. Despite differences in the design and implementation of these systems, all of them suffer from the same flaw. Clearing cache or log space is a costly operation and typically requires the cache to be write-protected for the duration of this operation. Ultimately, this design is unable to provide consistent performance that is desired by cloud providers.

Uncached Systems. Several systems [28, 32, 33, 41, 42, 48, 50, 63, 86, 110, 139, 148, 168, 169, 175, 179, 195] have been proposed to take advantage of the storage and memory characteristics of PMEM to modify and access data in-place. Unfortunately, ensuring crash

consistency for updates is non-trivial, requiring complicated transactional or journaling algorithms to be used. It has been shown that durable transactions have high overhead and significantly lower end-to-end performance [63, 125].

Decoupled Systems. Some recent systems like DudeTM [107], NV-HTM [25], and Bullet [71] have proposed partially or fully decoupling operation and persistence phases to lower the cost of persistence and improve performance. However, these systems still rely on expensive physical logging which is particularly bad when the working set of transactions is large. DudeTM and NV-HTM both use HTM for atomicity. Unfortunately, HTM requires hardware support which limits their wide-scale applicability. Further, DudeTM requires hardware changes to support its timestamp design, making it incompatible with commodity HTM. In contrast, DStore does not require special hardware support and uses more efficient logical logging. DStore is influenced and inspired by Bullet. Like DStore, Bullet separates volatile and persistent domains and proposes a cross-referencing technique to keep the two consistent. However, Bullet requires both the key and value to be added to the log. In contrast, DStore's design allows values to be omitted from the log and be placed only in persistent storage. Furthermore, DStore's DIPPER is more generic than Bullet's cross-referencing logs in that it can be applied to any storage system and not just key-value stores.

Logical Logging. Some recent works [113, 139] have successfully applied logical logging to database systems, while WAFL [90] and NOVA [179, 180] have done the same to filesystems. However, unlike DStore, these works do not provide a truly atomic quiescent-free checkpoint solution.

4.7 Summary

In this chapter, we first presented the design of a lock-free persistent linkedlist and ring buffer [53]. We demonstrated the effect of persistence mode on the performance of persistent lock-free data structures and that data structures optimized for ADR mode will not be optimal in eADR mode unless re-architectured. Next, we presented the design and implementation of DStore, a high-performance fault-tolerant userspace storage sub-system. DStore provides fast performance, low tail latency, and quiescent freedom simultaneously through the proposed DIPPER approach which decouples system and checkpoint space. We showed that by using PMEM to store the checkpoint space, we can leverage its performance to effectively overlap the operation of the two spaces. The novelty of our approach was in the design of the backend, which allowed us to use the same code for both spaces and provide low overhead persistence. We also explored different thread models [56] for our design and showed that a single thread run-to-completion design is more suited for providing low latency. Evaluation with Intel Optane DCPMM demonstrates the clear superiority of DStore over other state-of-the-art storage systems.

Chapter 5: Microfs: A Coordination-Free Abstraction for Ephemeral Storage

With enormous compute power, upcoming exascale systems [137] will bring with them crippling frequencies of system failure. Prior work estimates that their mean time between failure (MTBF) will be less than 30 minutes [7]. Exascale applications must protect themselves from unavoidable failures by checkpointing internal state to persistent storage. System-level checkpoint dumps on existing multi-petaflop systems have been shown to have significant overhead [22]. This problem will be exacerbated on exascale systems as not only will checkpoint time increase, but checkpoint frequency will also increase to account for the decrease in MTBF. The IO runtime must bear the burden of storing vast amounts of data in as little time as possible. Consequently, the storage components of exascale systems must be redesigned.

Newly introduced NVMe SSDs based on flash and 3D-XPoint memory offer unprecedented performance and concurrency. For example, new SSDs offer write bandwidth up to 2.5GB/s [91], an order of magnitude faster than SATA SSDs. These devices are ideal for use in HPC systems to build high density storage arrays. These arrays can be stacked together to build a disaggregated storage cluster. With the introduction of the NVMe-over-Fabrics (NVMf) standard [130], low latency remote access to these arrays can be effectively provided. The NVMf standard can take advantage of fast RDMA enabled networks in HPC

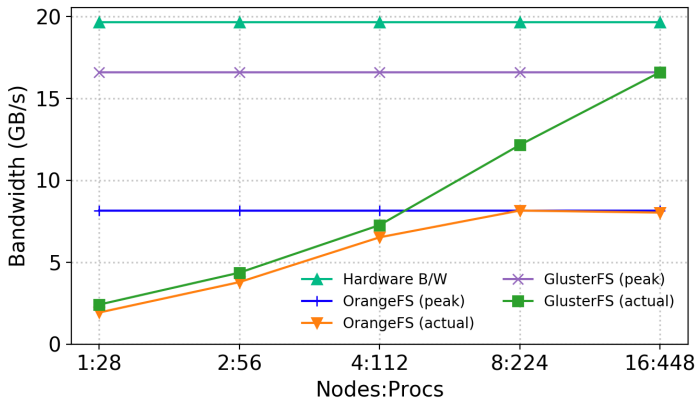


Figure 5.1 Weak scaling checkpoint bandwidth with OrangeFS and GlusterFS. Note the gap between the peak and available hardware IO bandwidth.

systems to reduce the network overheads of remote access. Guz *et al.* [60] have shown that at the application level, NVMf only has ∼10% overhead compared to local IO.

In a disaggregated HPC cluster setup, the most common form of storage runtime provided is a Parallel File System (PFS), such as Lustre [149]. In recent years, several distributed filesystems [12, 65, 72, 108, 163, 171] have been proposed to alleviate the bottlenecks in PFS. However, we find that these systems are still not ideal for highly concurrent checkpoint IO. To demonstrate the shortcomings of existing filesystems, we measure the sustained bandwidth of checkpoint IO on NVMe SSDs while varying the number of concurrent application processes.

Figure 5.1 shows the checkpoint bandwidth for OrangeFS [163] and GlusterFS [65] with the ECP CoMD application [44]. At best, OrangeFS and GlusterFS can only achieve 41% and 84% of the peak hardware bandwidth, respectively. There are two primary reasons for this. First, these storage systems overlay multiple software layers over POSIX

filesystems which decrease the peak attainable bandwidth. Second, these filesystems expose significant overhead at high concurrency, which is because the strict POSIX semantics require `open` and `creat` syscalls to be atomic. The need for operation atomicity and metadata consistency requires complicated distributed synchronization mechanisms which suffer from scalability limitations [119, 120]. Note that at lower process counts, GlusterFS is unable to deliver their peak bandwidth. This is because GlusterFS uses consistent hashing to distribute files between storage devices which has high standard deviation of load under low concurrency, as shown in [100].

There has been work [192, 193] to alleviate these bottlenecks by reducing or eliminating the need for synchronous namespace (or directory hierarchy) access. Unfortunately, these works are unable to extract the best possible performance from fast NVMe devices. Control and data operations have to trap into the OS and go through multiple software layers, negating the benefits of using low latency NVMe devices. As such, these storage systems are only suitable for storing long-lived input and output data and not latency critical ephemeral checkpoint data. The characteristics of checkpoint IO are different. The data is ephemeral and the IO bandwidth required depends on the job scale. Therefore, for storing checkpoint data, we need a storage runtime which can be configured during a job's runtime based on its load factor. The runtime itself must be ephemeral and should terminate with the job. To the best of our knowledge, no storage runtime is available in the HPC community yet, which can offer direct access to storage using NVMf as well as synchronization-free control and data planes. Therefore, there is a clear need to design a coordination free storage runtime for checkpoint/restart, which can provide low latency direct access to remote SSDs using NVMf. An ephemeral storage runtime for checkpoint data must satisfy three requirements:

(1) full utilization of available bandwidth, (2) high resiliency of data, and (3) support for large number of concurrent clients.

5.1 The microfs Abstraction

We introduce micro filesystem (or simply `microfs`) as a powerful design template for ephemeral distributed filesystems that wish to provide the minimal functionalities expected of a storage system. `Microfs` is designed to peel off the unnecessary software layers that hinder performance and allow applications to directly access storage devices. As opposed to conventional kernel filesystems, `microfs` runs purely in userspace which allows it to bypass the kernel virtual filesystem (VFS) and block driver, eliminating the need to trap into the kernel for every operation. Furthermore, `microfs` abstains from providing a shared namespace, instead only providing a private one for each application process. It is this choice which obviates the requirement to synchronize across all `microfs` instances, a common challenge for user-level filesystems. With this observation, we present `microfs` as a high-performance alternative for storing ephemeral application data.

To attain the goals of `microfs`, we formulate the following design principles.

Principle 1: Direct userspace access to storage devices. To enable unprivileged userspace access to devices, `microfs` uses the `vfio` kernel module. IO operations can then be submitted using memory mapped IO and completed by issuing DMA operations purely in userspace. This allows for bypassing the kernel VFS and block driver while also providing fine-grained control over the IO pipeline. We expect `microfs` instances to implement a run-to-completion request pipeline (by avoiding locks and using polling instead of interrupts) to reduce IO latency.

Principle 2: Maintaining storage device integrity. Storage devices are shared across several `microfs` instances. Integrity is maintained by logically and physically partitioning the device between the instances. This partitioning is done at initialization time using a shared communication runtime. We create group communicators for processes sharing hardware to enable coordinated access to storage devices. Once the partitioning is complete, each runtime instance can access its portion of the device without any need for coordination. In this manner, unprivileged direct access to storage can be provided efficiently.

Principle 3: Synchronization-free control and data planes. Data plane operations are designed to be synchronization-free by allocating a separate hardware IO queue for each `microfs` instance. This enables parallel IO by exploiting the large number of queues in modern storage devices. For example, the Intel Optane P4800X SSD controller can support up to 32 hardware queues. The use of a single IO queue per instance guarantees that IO operations are completed in the order they are received. Control plane (metadata) operations are synchronization-free since no inter-`microfs` coordination is required, by definition.

Principle 4: Data and metadata durability. Traditional kernel filesystems rely on an OS-level cache to buffer IO, journaling both data and metadata to maintain durability. Instead, `microfs` writes data directly to device-level RAM and transparently leverages device capacitance to guarantee durability. Metadata consistency is maintained using lightweight operation logging. Furthermore, the runtime periodically checkpoints internal filesystem states to the storage device to prevent the log from growing without end.

5.2 NVMe-CR Design

In this section, we present the design of `NVMe-CR`.

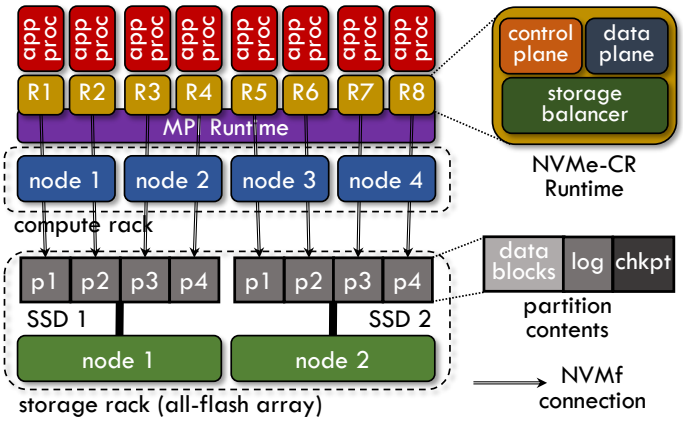


Figure 5.2 NVMe-CR Runtime Architecture. Each runtime instance directly accesses its own remote SSD partition via NVMf.

5.2.1 Architecture

`NVMe-CR` is a storage runtime designed for scalable C/R capabilities using the emerging NVMf protocol. `NVMe-CR` is built upon the `microfs` abstraction to enable low latency IO. The goal of `NVMe-CR` is to provide C/R support for systems with disaggregated storage. `NVMe-CR` does not provide a global namespace but only private per-process namespaces. This is a conscious design decision to eliminate synchronization. Figure 5.2 shows the architecture of `NVMe-CR`. Each application process runs its own storage runtime which is mapped to a single partition of a remote SSD, connected using NVMf. Each runtime instance implements three critical functionalities – the control plane, data plane, and storage balancer. The control plane is responsible for creating and storing metadata of files and directories. The data plane provides a block device like interface to access the remote SSD partition using NVMf. The storage balancer partitions available storage devices between compute processes and provides a conflict free many-to-one mapping. These three components work in tandem to expose an ephemeral storage runtime to disaggregation oblivious

applications. The sub-sections that follow provide a detailed description of the working of these components.

5.2.2 Application Obliviousness

One of the primary goals we want to achieve is portability, i.e., enhancing the storage runtime without application modification. We use the common approach of symbol interception provided by the GNU ld linker to achieve this. We intercept all the standard POSIX IO library calls and redirect them to the NVMe-CR runtime. NVMe-CR implements all of the IO library calls while remaining purely in userspace by following the design principles of microfs. For management of the NVMe-CR runtime, we also intercept the MPI_Init and MPI_Finalize calls. Runtime initialization and finalization is handled by these wrappers. Internally, we leverage the MPI runtime for coordination between multiple instances as well as for identification purposes. It should be noted that coordination is only necessary in the initialization routine. Subsequent control and data plane requests are not required to synchronize with other runtime instances. By using the symbol interception method, we can efficiently separate the filesystem API from remote data storage and run unmodified application binaries over NVMe-CR.

5.2.3 Data Storage using NVMf – Data Plane

NVMe-CR can transparently leverage NVMf as the data plane conduit to access the high density storage arrays. Currently, our implementation uses the RDMA transport for data exchange. Figure 5.3 compares the proposed and existing data conduits. The entire software stack is shifted to userspace from the kernel. The entire software stack is shifted to userspace from the kernel. To enable userspace access to remote SSDs via NVMf we use

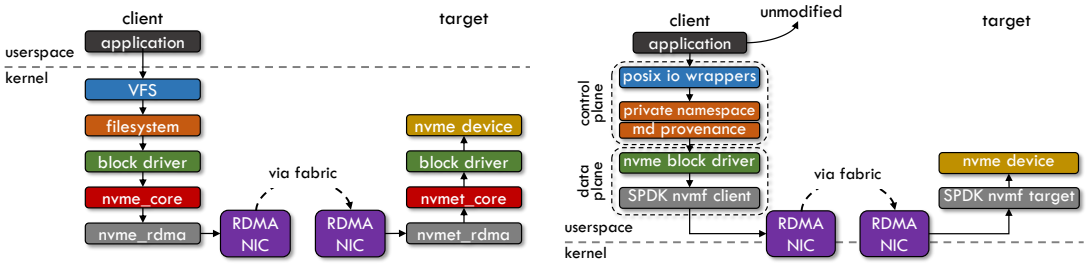


Figure 5.3 (left) Existing application-oblivious approach to access remote storage using NVMf – Note that the entire data plane lies in kernel space, and (right) NVMe-CR application-oblivious approach to access remote storage using NVMf – Note that the entire data plane lies in userspace, a stark contrast from the existing approach.

Intel's SPDK library [76]. We chose SPDK as opposed to other systems like NVMeDirect [92], and libaio [106] because it is the current state-of-the-art solution. Further, it has negligible software overhead and its NVMf server is multi-tenant. SPDK NVMf server daemons are deployed on each storage node for handling client NVMe requests. SPDK NVMf clients, embedded within the `NVMe-CR` runtime are responsible for communication with server daemons. In this manner, all data plane operations are internally translated into NVMf requests and handled by the `NVMe-CR` runtime.

Data Durability. `NVMe-CR` eschews buffering write requests, instead writing data directly to internal device-level RAM, if available. Otherwise, it is directly written to flash memory. Flash devices with RAM usually support enhanced power-loss data protection [77]. In the event of power failure, device capacitors will safely flush volatile data to non-volatile flash memory. Writing to device RAM does not limit the overall data storage capacity but only improves performance in cases where data fits within device RAM. If data does not fit in RAM, performance gets limited to SSD bandwidth. By avoiding data buffering, we assure that data is always persistent and can survive temporary power failures. This design choice

was made based on the observation that buffered IO reduces overall application progress rate.

5.2.4 Metadata Management – Control Plane

The control plane is responsible for metadata management, including block allocation, data and metadata consistency, and exposing a POSIX compatible interface. Its goal is to eliminate the need for complex distributed synchronization and minimize network IO. `NVMe-CR`'s control plane design has several advantages. First, by exposing only private namespaces, metadata operations never have to coordinate across processes. In contrast, other systems must use distributed locking algorithms for each metadata operation to maintain a consistent global namespace. Second, metadata is entirely maintained in DRAM with lightweight operation logging used for consistency and durability. The only network IO involved is for reading/writing file data and writing compact log records. Other systems must transmit additional data over the network (such as inodes and large sized physical log records), reducing overall system efficiency. In addition, the control plane uses a combination of several techniques and designs to maximize performance. These are explained in detail below.

Hugeblocks. For space management, we divide the SSD into *blocks*, the smallest unit of storage allocation. Given that checkpoint files are several MBs, if not GBs, we allow files to be managed in large block sizes. We call these *hugeblocks* because of their similarity to *hugepages*. In our design, we use a *hugeblock* size of 32KB as opposed to kernel filesystems which support block sizes only up to 4KB. We use a circular block pool for $O(1)$ *hugeblock* allocation. The use of *hugeblocks* significantly lowers the amount of information that must be kept to track file blocks, which not only reduces the space overhead but

also makes the process of block allocation faster. Further, we submit NVMf IO requests in *hugeblock* units as well, which allows us to attain peak hardware bandwidth regardless of the number of clients sharing an SSD. The reason for this is that NVMe SSDs internally break up large IO requests into several smaller (usually hardware block sized) requests which can then be split across the available flash channels for highly parallel IO. The entire SSD bandwidth can then exploited even if there is only a single client accessing the device.

POSIX Semantics. By using SPDK for remote IO, we can indeed bypass the kernel, but we need to provide a POSIX filesystem like interface to applications. `NVMe-CR` implements wrappers for commonly used POSIX IO syscalls. For providing a filesystem like runtime with POSIX semantics, we borrow several conventional filesystem concepts and techniques, such as *inodes* to store file metadata and *directory* files to store directory entries. The control plane handles failure recovery using a write-ahead log. All metadata operations executed during the following functions are logged: `mkdir`, `open`, `write`, and `unlink`. The log is flushed before a subsequent operation is processed. Since we do not buffer writes and we journal metadata operations for `open` and `write`, our runtime provides stronger data durability guarantees than required by POSIX. As a consequence, metadata will always be consistent, even with unexpected failures. This is an important property because it guarantees that a completely written checkpoint file will never hold corrupted data and can safely be used for recovery.

Per-process Private Namespace. If we take a look at common checkpointing patterns used by applications, we find that two patterns are prevalent – N-1 and N-N [19]. In the N-1 pattern, processes write to a single shared file, whereas in the N-N pattern each process writes to a unique file. Recent work [167] has estimated that 90% application runs use the N-N pattern. Due to this reason, the designs proposed in this chapter are specifically

targeted towards the N-N pattern. The concurrent creation of millions of files is a scalability challenge for filesystems. The choice of private per-process namespaces in `NVMe-CR` allows us to overcome the scalability bottlenecks with the N-N pattern. Each runtime instance is only tasked with the creation of a single file per checkpoint. Each runtime instance exposes a private root directory which is not accessible to other processes. The root directory is a file which resides on the remote SSD partition for the process. Since the root directory files are independent for each process, the namespace exposed by them are private. The directory hierarchy is constructed using a set of directory files indexed by a DRAM resident B+Tree. The B+Tree contains mappings of directory and file names to their root inode. Any file manipulation operation (`creat`, `open`, `unlink`, `read`, or `write`) is handled exclusively by the `NVMe-CR` runtime associated with the originating process. Each runtime not only handles the maintenance of the private namespace but also block and inode allocation for files. In this manner, no coordination is required for any operation.

Metadata Provenance. To ensure that checkpoint files are available despite application failure, we must ensure that metadata is safely made durable. A simple approach could involve storing metadata on the remote SSDs and updating it on each `fsync` or `write` call. This would lead to unnecessarily high remote IO operations over NVMf. To reduce the metadata overhead imposed on each runtime instance, we allow the control plane to store metadata (inodes, block pool, and B+Tree) locally, within the memory of compute nodes. To guarantee metadata durability and consistency, we journal filesystem metadata operations using a compact operation log stored on remote SSDs. Each syscall that modifies an inode needs to be logged. Only the syscall type and its parameters need to be added to the log. The use of operation logging significantly reduces the amount of data that must be sent over network to the remote SSDs. We call this approach metadata provenance because

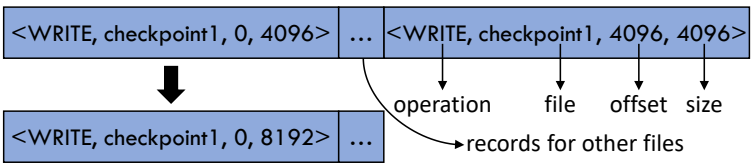


Figure 5.4 Log Record Coalescing

we store every operation in the log, giving us the ability to track fine-grained changes to metadata. During recovery in the event of a crash, the runtime reconstructs metadata by replaying operations recorded in the log. An in-memory B+Tree is used to keep mappings of filenames to their inodes allowing fast lookups of files. The state of the B+Tree can also be reconstructed upon recovery from a crash.

To limit the size of the log, the runtime checkpoints internal DRAM state (which includes the inodes, block pool, and B+Tree) to a reserved region on the remote SSD. To ensure that checkpoints do not affect user IO requests, the checkpoint process is overlapped with the application compute phase. In this manner, the checkpoint process is completed in the background using a dedicated checkpoint thread without affecting application performance. The background thread can exactly determine when the application checkpoint process is complete by monitoring the number of open files. When the number of open files is zero and the number of free log records is below a predefined threshold, the background thread kickstarts the process of checkpointing internal volatile state. Further, the checkpoint process is designed to be atomic. Log records are only discarded once the checkpoint is complete. A failure during checkpoint will not affect the durability and consistency of data.

Log Record Coalescing. To reduce the number of log records written, we propose a technique called log record coalescing. In this technique, we take advantage of the sequential

nature of checkpoint IO to combine near-adjacent log records as long as they represent consecutive writes to the same checkpoint file. Figure 5.4 shows how this process works. Instead of adding new log records for each write, we can simply update the log record for the previous write. We use a sliding window to find the log record for the previous write and update it accordingly. In this manner, the log fillup rate is significantly lowered, which means that filesystem state does not need to be checkpointed frequently to clean log space. In addition, the number of records that must be replayed on recovery is significantly lowered too, which provides near-instantaneous recovery of the `NVMe-CR` runtime itself.

5.2.5 Load-aware IO – Storage Balancer

The storage balancer is responsible for both allocation of storage nodes for a job and balancing IO load between available storage devices. `NVMe-CR` considers two vital factors while distributing data between remote SSDs – load balancing and fault-tolerance. Load balancing is important to ensure that we utilize all of the available IO bandwidth and all processes get an equal distribution of the bandwidth. Fault-tolerance is another important factor because we need to place checkpoint data in a separate failure domain than what the process is in. Otherwise, it is likely that failure of a process coincides with loss of its checkpoint data. To account for both of these factors, `NVMe-CR` uses a novel data distribution algorithm.

First, we identify the failure domains for each node by using the network topology. Nodes which share hardware are placed in the same domain. For example, all nodes within a rack and all nodes sharing a power distribution unit are placed in the same domain. Next, we create partner failure domains, such that nodes in both partners are in separate failure domains. For each failure domain, we create a list of partner domains sorted by the number

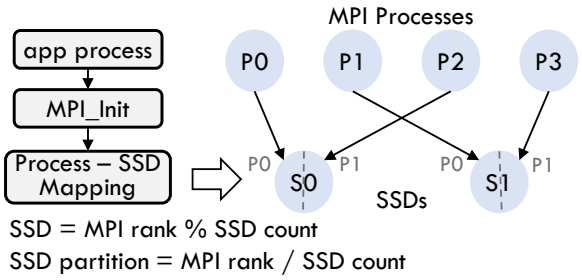


Figure 5.5 NVMe-CR Storage Allocation Process

of switch hops between them. Finally, we create a mapping of processes and storage nodes, such that their respective failure domains are partners and the load on each storage node is as equal as possible. We construct the mapping using a greedy algorithm to minimize communication cost. Storage devices for a job are allocated on the closest (fewest hops away) available partner domain. Processes within a job are assigned to the allocated SSDs in a round robbin manner to achieve load balancing. Once this mapping is defined, we create an MPI communicator for all processes sharing an SSD, called `MPI_COMM_CR`. The SSD is then partitioned between processes in the associated communicator. Each process gets a contiguous segment of the SSD based on the its rank and the communicator size. Figure 5.5 shows how the available storage is partitioned and allocated to MPI processes.

Security Model. To ensure device integrity despite userspace access, we rely on the namespace feature in the NVMe standard. All SSDs are divided into at least two namespaces. The job scheduler assigns storage to jobs at the granularity of a namespace. If there are no free namespaces, new ones are created from unused SSD space. Although the number of namespaces supported by each SSD is limited, the number of concurrent jobs an SSD can support is only limited by its bandwidth. This is because a few concurrent jobs can easily saturate its bandwidth. This approach allows SSDs to be shared between applications while relying on the isolation property of namespaces to maintain security.

This enforcement requires integration with the cluster job scheduler. In practice, we do not anticipate this to be a problem. For example, by using Slurm's generic resources plugin [154], we were able to support this design on our cluster easily. Further, NVMe-CR's control plane acts as a trusted intermediary between the application and SSD. The control plane performs access control checks for file IO so that POSIX permissions are respected and unauthorized users cannot corrupt or read checkpoint data.

For each job, the user must specify the number of storage devices required for checkpoint data. This number can be determined by ensuring that the process:SSD ratio is in the range 56–112. This is based on our experimental results showing that at this ratio NVMe SSD bandwidth is utilized to its maximum. The storage balancer works along with the job scheduler to allocate SSDs based on allocated compute nodes and the network topology, as explained earlier. The load balancing does not require maintenance of additional information because job schedulers already have topology information readily available. When the user runs an application, the storage balancer is again invoked to partition the allocated devices among the application processes. Once the partitioning (which is done during runtime initialization) is complete, the load balancer does not need to be involved during the lifetime of the application.

Handling Cascading Failures. For NVMe-CR, our storage balancer tries to keep application processes and their checkpoint data on separate failure domains to minimize the possibility of both failing. Although rare, cascading failures can happen on large-scale clusters and must be handled to protect checkpoint data. This is challenging because such failures may disrupt the application and also availability of its checkpoint data. To protect data despite cascading failures, we use multi-level checkpointing [124]. In this solution, most checkpoints are still handled by NVMe-CR, but every so often, one checkpoint is put on a

slower but more reliable parallel filesystem, such as Lustre. Through redundancy mechanisms, such as replication, such systems can guarantee that data is available even with cascading failures. Therefore, performance is not compromised because most checkpoints are handled by the fast NVMe-CR runtime, and fault-tolerance to cascading failures is also not compromised by relying on the redundancy of parallel filesystems. Our approach is complimentary to existing multi-level checkpointing approaches for fault-tolerance.

5.2.6 Implementation Notes

We implement NVMe-CR purely in C11 and distribute it as a shared library. This library can be preloaded during runtime using the LD_PRELOAD environment variable to transparently run any MPI application binary. The remote NVMf devices to connect to can be specified as a list exported using the NVMF_DEVICES environment variable. The root directory path can be specified using the the CHKPT_DIR environment variable. NVMe-CR's control plane is implemented on top of DStore's control plane, which was presented in section 4.4.

5.3 Experimental Analysis

In this section, we present the evaluation of NVMe-CR.

5.3.1 Experimental Testbed

We consider a disaggregated cluster for our evaluation. Our cluster has one storage rack with 8 nodes and one compute rack with 16 nodes. Each node in the storage rack has a 28 core skylake (Gold 6132@2.6GHz) CPU with 192GB memory running Linux 3.10 and an Intel P4800X Optane SSD. Each node in the compute rack has a 28 core broadwell (E5-2680v4@2.4GHz) CPU with 128GB memory running Linux 3.10. All nodes are equipped

with Mellanox ConnectX-5 adapters and are connected using 100Gbps EDR InfiniBand. Lustre is used as the PFS and is configured with 4 separate storage servers, each using one 12Gbps RAID controller.

We use ECP CoMD [44] as a representative HPC application to show the practical benefits of our proposed `NVMe-CR` runtime. Most applications in the ECP application suite, including AMG, Ember, ExaMiniMD, and miniAMR have similar behavior and are likely to show similar improvements as CoMD. We compare `NVMe-CR` performance with OrangeFS, GlusterFS, Crail [12], XFS, and ext4.

We only compare with Crail in a single server configuration because its publicly available version only supports a single NVMf server. Although we could not compare with Crail in a multi-server setting, we expect it to perform worse than NVMe-CR because Crail uses a single metadata server which becomes a bottleneck at high-concurrency. We were also unable to compare with DeltaFS; despite significant effort, we were unable to run it on our cluster.

5.3.2 Optimal Hugeblock Size

Chosing an optimal *hugeblock* size is important for obtaining the best performance. If the size is too small, metadata overhead and IO request count will be high. On the other hand, a large block size will increase the waiting time for each hardware IO queue, reducing multi-process performance. To determine the optimal *hugeblock* size, we measure the overhead of writing 512MB checkpoint files for a full subscription run. Results (see Figure 5.6(a)) show that 32KB is the optimal size for achieving the lowest latency. Using a 32KB block size instead of the standard 4KB delivers 7% improvement in latency. It

also results in an 8x reduction in the size of the block pool and the number of inodes used. Henceforth, we use a 32KB *hugeblock* size for all experiments.

5.3.3 Load Imbalance Evaluation

Load balancing is important to ensure that available hardware resources are efficiently utilized. To measure load imbalance for different storage systems, we compare the coefficient of variation (ratio of standard deviation and mean) of load (size of data stored) on each storage server. Figure 5.6(b) shows the value of this coefficient when CoMD is run at different process counts. GlusterFS uses a consistent hashing algorithm to distribute data, which has been shown to have high standard deviation at low concurrency [100], just as observed. OrangeFS stripes file data across servers which works much better than consistent hashing at low concurrency but has noticeable overhead at higher process counts. In contrast, `NVMe-CR` achieves perfect load balancing regardless of the level of concurrency. The reason for this is that our storage balancer creates a mapping between processes and storage devices using a round-robbin policy. Since each process creates a file of the same size, the load on each server is then exactly equal. GlusterFS and OrangeFS cannot achieve perfect load balancing because they are not associated with a single application. A mapping of processes to storage devices cannot be created since the filesystem does not know which application each client belongs to.

5.3.4 Direct Access Evaluation

To quantify the benefits of direct access, we measure the dump time for different checkpoint sizes using `NVMe-CR`, `XFS`, `ext4`, and SPDK on a local NVMe SSD for a full subscription (28 cores) run. Figure 5.6(c) shows the results of this analysis. Checkpoint data is written using the `write` system call and persistence is guaranteed by calling `fsync`.

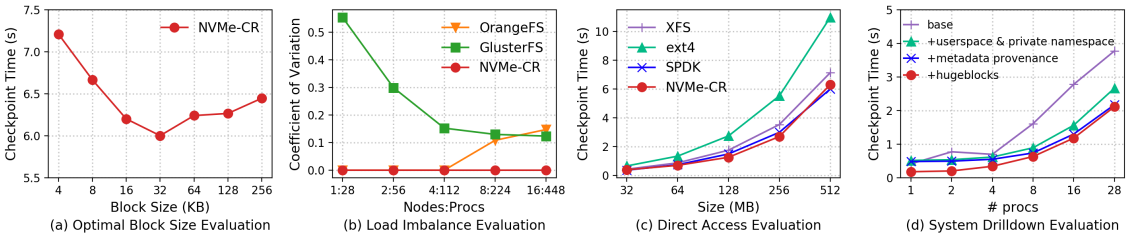


Figure 5.6 (a) Checkpoint times for different *hugeblock* sizes, (b) Load imbalance for `NVMe-CR`, OrangeFS, and GlusterFS, (c) Full subscription performance of `NVMe-CR`, `ext4`, `XFS`, and SPDK, and (d) Drilldown evaluation showing impact of optimizations.

We find that direct access is crucial in lowering the latency for large sized checkpoints. For 512MB data, we see 19% and 83% improvement compared to `XFS` and `ext4`, respectively. This improvement comes from the ability to bypass the kernel as well as the use of *hugeblocks* and metadata provenance, which significantly reduce metadata overhead. For example, `hugeblocks` lowers the number of SSD blocks to be allocated and tracked by 4x. Compared to SPDK, `NVMe-CR` has no noticeable overhead. This clearly highlights the elimination of software and metadata overheads. Note that SPDK alone cannot handle all the IO challenges (POSIX compliance, metadata management, and private namespace) of an NVMf-enabled distributed storage system as we have discussed earlier. We also measure the percentage of benchmark time spent in the kernel. By enabling userspace device access in `NVMe-CR`, the benchmark only spends 10% of its time in the kernel compared to 76.5% for `XFS` and 79% for `ext4`. This reduction is because all POSIX IO syscalls are now resolved completely in userspace. `NVMe-CR` provides userspace implementations of all IO syscalls. The 10% time spent in the kernel is because of non-IO syscalls made by the benchmark itself and during the initialization and finalization routines in `NVMe-CR` (for

example as a result of calling `malloc`). We also note that on increasing data size, the performance gap increases. This is because metadata overhead has a linear correlation with file size.

5.3.5 Drilldown Evaluation

To understand the impact of the different optimizations and designs in `NVMe-CR`, we measure the checkpoint time with the CoMD application on a single node. We start with a base design resembling a traditional kernel filesystem and add optimizations one-by-one, measuring the checkpoint time for each case. Figure 5.6(d) shows this analysis. Bypassing the kernel and eliminating the global namespace provides up to 44% improvement compared to baseline. The improvement is also higher at scale which is because the overheads of global namespace synchronization are high. Metadata provenance improves performance up to 17% (v/s + userspace & private namespace) by lowering the size of log records. At low concurrency it improves performance by reducing software overhead while at high concurrency, it improves performance by increasing the available data bandwidth. Finally, using hugeblocks improves performance up to 62% (v/s + metadata provenance). The improvement is mostly noticeable at low concurrency because performance is dictated more by software overhead than IO bandwidth. Overall, we conclude that the private namespace improves performance at high concurrency, hugeblocks improves performance at low concurrency, and metadata provenance improves performance in all cases. Combining all three optimizations delivers the best performance at all levels of concurrency.

5.3.6 NVMf Overhead

We measure the overhead of NVMf by comparing checkpoint performance on a local and remote SSD for a full subscription (28 cores) run. We also compare with Crail,

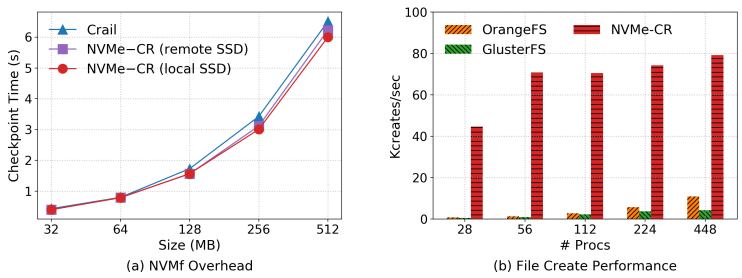


Figure 5.7 (a) Full subscription performance of `NVMe-CR` on a local and remote SSD, and (b) File create performance of `NVMe-CR`, OrangeFS, and GlusterFS.

OrangeFS*	GlusterFS*	NVMe-CR#
2686.25	3.5	445.25

Table 5.1 Metadata overhead with CoMD in MB. * per storage node # per runtime

a userspace storage runtime which supports NVMf via SPDK. The remote access is as demonstrated in Figure 5.3 over EDR InfiniBand fabric. Results (see Figure 5.7(a)) show that there is no noticeable overhead in latency for writing checkpoints. In fact, the maximum overhead we observe is below 3.5% and is independent of the size of the checkpoint. The benefits of metadata provenance are clear when we compare `NVMe-CR` and Crail. Even though both use SPDK for NVMf support, `NVMe-CR` consistently provides up to 5-10% lower overhead for remote access. There are two factors which contribute to achieving negligible overhead. First, using the SPDK NVMf driver instead of the kernel NVMf driver eliminates kernel space overhead. Second, the use of metadata provenance reduces the amount of additional data that must be sent via NVMf, minimizing the overhead of metadata operations. These results highlight the advantages of enabling userspace remote access over fast RDMA networks. It is clear that the proposed userspace data and control planes are successful in creating a low latency pipeline.

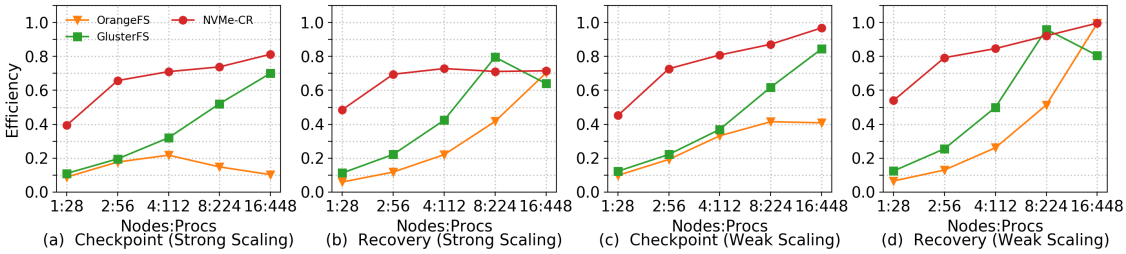


Figure 5.8 Efficiency of storage systems during checkpoint and recovery of the CoMD application state. Efficiency of a storage system is defined as the ratio of application perceived IO bandwidth and the hardware IO bandwidth.

5.3.7 Metadata Overhead

In the N-N checkpoint pattern, the number of files scales linearly with the job size. The number of file creates that a storage system can perform is an important metric for checkpoint IO. We compare the create performance of storage systems at different job scales to determine how well they can perform under heavy load. Results are presented in Figure 5.7(b). `NVMe-CR` provides 7x and 18x higher create performance at 448 processes. The primary reason for this is the absence of a global namespace. Each process can create files in parallel and avoid serialization of operations. For each file create, a corresponding entry must be added to the directory file stored on the remote SSD. Hence, the file create performance is only limited by hardware bandwidth and not software latency. OrangeFS and GlusterFS use consistent hashing to lower metadata overhead. Despite this optimization, both must add file entries to a single common directory file which effectively serializes file creates, leading to poor performance.

We also measure the storage overhead of metadata and checkpoints compared to OrangeFS and GlusterFS. As we can see from Table 5.1, `NVMe-CR` requires ∼450MB storage per runtime, OrangeFS requires ∼2.6GB, and GlusterFS only requires 3.5MB per storage

node. The total overhead for NVMe-CR can be calculated by multiplying this value with the number of application processes. For high concurrency runs, the total overhead for NVMe-CR is higher than both OrangeFS and GlusterFS, but remains under acceptable limits. For instance, on our cluster only 1.7% of SSD space is utilized for metadata and checkpoint storage, in the worst case. We also measure the DRAM footprint and find that NVMe-CR consumes less than 512MB per-instance – 404MB for inodes and 102MB is for B+Tree. This minor increase in metadata overhead can be justified by the significant performance benefits of our design. For GlusterFS, the overhead is minimal because it uses consistent hashing which requires little metadata to be maintained. On the other hand, OrangeFS has high overhead as it needs to store both file metadata and striping information.

5.3.8 Application Evaluation

We measure the overhead of checkpointing the CoMD application state under both weak and strong scaling configurations. We compare checkpoint as well as recovery efficiency with NVMe-CR, OrangeFS, and GlusterFS. We do not compare with Crail because it currently only supports a single storage server for its NVMf tier. We define the *efficiency* of a storage runtime as the ratio of the peak IO bandwidth visible to applications to the peak theoretical bandwidth offered by hardware. Checkpoint performance is only dependent on overall bandwidth of the storage system. Therefore, we use efficiency as a metric which allows us to compare the relative checkpoint performance of different storage systems. For all cases we use the aggregate SSD bandwidth as the hardware peak.

Strong Scaling. For strong scaling analysis, the problem size is fixed to 16,384K atoms for a total fixed checkpoint size of 86GB (for 10 checkpoints). Figures 5.8(a) and 5.8(b) show the checkpoint and recovery efficiency for multi-node full subscription runs. Overall,

we find that NVMe-CR achieves better efficiency for both cases than other storage systems. Synchronization-free control and data planes are responsible for good scalability. The load balancer in NVMe-CR allows it to attain better efficiency than other systems at lower process counts. This is because our greedy load balancing algorithm ensures that all of the SSDs are utilized efficiently. Apart from GlusterFS, other systems are unable to handle the metadata burden when 448 processes concurrently checkpoint data. The reason is the use of a global namespace and high software overhead. During recovery, however, they perform much better since there is no metadata overhead involved. GlusterFS is better than other systems we compare with because of its decentralized design. Nevertheless, its checkpoint performance is still $\sim 13\%$ lower than NVMe-CR because of its poor create performance (see Figure 5.7(b)) and reliance on POSIX IO (see Figure 5.6(c)). At high concurrency, NVMe-CR achieves the best efficiency of all systems because of its coordination-free design.

Weak Scaling. We fix the problem size to 32K atoms per process while scaling up to 448 processes. We take 10 periodic checkpoints during application runs for a total checkpoint dump of 700GB. This experiment was designed to show that NVMe-CR can handle large data volume. Figures 5.8(c) and 5.8(d) show the results of this analysis. Overall, we find that NVMe-CR achieves near perfect efficiency (0.96 for checkpoint and 0.99 for recovery) at 448 processes. Recovery efficiency is so high because of log record coalescing which allows the runtime to recover instantaneously and fully utilize available bandwidth. For checkpoints, other systems suffer from the high synchronization overhead of creating a large number of concurrent files, leading to poor efficiency. During recovery, just like in the strong scaling case, their performance improves significantly because there is less synchronization involved to read files. However, GlusterFS performance dips at 448 processes because its

Metric	OrangeFS	GlusterFS	NVMe-CR
Checkpoint Time (s)	85.9	44.5	39.5
Recovery Time (s)	3.6	4.5	3.6
Progress Rate	0.252	0.402	0.423

Table 5.2 CoMD evaluation with multi-level checkpointing at 448 processes. For progress rate, higher values are better.

metadata server is unable to handle the large influx of read requests. In contrast, `NVMe-CR` achieves good efficiency at both low and high concurrency by avoiding synchronization and achieving direct device access.

5.3.9 Multi-Level Checkpointing Evaluation

To evaluate `NVMe-CR` in a real-world use case with cascading failures, we conduct an experiment with CoMD at 448 processes. We use different systems for the first checkpoint level and Lustre as the second level, where one checkpoint in ten is written to Lustre. We compare checkpoint time, recovery time, and application progress rate for different systems in Table 5.2. It is clear that `NVMe-CR` is the best for all metrics, which is because its efficiency is near ideal. Compared to GlusterFS, it reduces checkpoint time by 11%, recovery time by 20% and progress rate by 5%. The large improvement in recovery is due to log record coalescing (without coalescing, recovery takes 4s) which allows very fast metadata recovery. Therefore, using `NVMe-CR` directly results in benefits at the application level, reducing overall runtime and increasing the probability of applications running successfully.

5.4 Related Work

There have been several works which focus on reducing the overheads of application checkpointing. `CRUISE` [144] is a userspace filesystem backed by DRAM to enable fast checkpoints. Zest [129] uses a log structured design with a burst buffer layer to reduce checkpoint overhead. CRFS [138] improves checkpoint IO by aggregating small IO operations into larger operations which are written asynchronously to disk. PLFS [19] is another filesystem aims at optimizing checkpoints which follow the N-1 pattern. Other works like PapyrusKV [94], UnifyCR [108], and BurstFS [171] present a burst buffer design using node local storage to accelerate C/R IO as opposed to `NVMe-CR` that is targeted towards a disaggregated setup. Storage systems like Hermes [98], DDN IME [37], Cray DataWarp [67], GlusterFS [65], and OrangeFS [163] overlay multiple software layers on kernel filesystems to access data. While these works have improved checkpoint overhead, they suffer from two basic limitations. First, these filesystems still suffer from the use of POSIX filesystems to access devices. Direct userspace access to devices via NVMf, as achieved by our proposed `NVMe-CR`, is not supported. Second, serialization of metadata operations and synchronization between clients could prevent applications from fully utilizing available IO bandwidth. BSCFS [72] and Crail [12, 159] do support an NVMf-based data plane, however, both require applications to be modified to use their specific non-POSIX API. BLCR [62] is an orthogonal approach for system-level C/R as opposed to application C/R, which we focus on.

`NVMe-CR`'s `microfs` abstraction is most related to the design of DeltaFS [193]. DeltaFS does not expose a single namespace, but a collection of snapshots that can be used by applications to construct their own namespace view. This provides the ability to execute large numbers of parallel metadata operations with minimal coordination. `Microfs` extends this

idea to completely remove coordination requirements for both metadata and data operations. It assumes the complete absence of a global namespace and partitions SSDs between processes to achieve logical isolation. To reduce the amount of data to be checkpointed, techniques such as *incremental checkpointing* [46], *cooperative checkpointing* [133], *multi-level checkpointing* [124] and *data compression* [73] have been proposed. While these approaches reduce checkpoint overhead, they still rely on existing inefficient IO subsystems. Thus, these works are complementary to the designs proposed in this chapter and can be combined for improved performance.

5.5 Summary

In this chapter, we presented the design of `NVMe-CR` [54], a storage runtime for disaggregated clusters with NVMf. First, we presented a powerful design template for filesystems meant for storing ephemeral application data. By following the design principles of this template, filesystems can achieve low latency direct access to device. Built upon these principles, `NVMe-CR` provides synchronization-free control and data planes as well as a fault-tolerance and load-aware storage balancing. By proposing techniques like metadata provenance, log record coalescing, and hugeblocks, `NVMe-CR` is well suited for storing checkpoint data. Experimental analysis with CoMD shows that on a local cluster our runtime can achieve near perfect (> 0.96) efficiency at 448 processes. As a result, our runtime lowers checkpoint overhead by up to 2x, while increasing job progress rates by as much as 1.6x.

Chapter 6: HAQ: Hardware-Assisted Quality of Service for NVMe Drives

The architecture of enterprise clouds is rapidly changing. Novel hardware and software innovations constantly drive the evolution of cloud environments. The NVMe standard is a recent innovation which has significantly impacted research in storage systems. The standard allows flash storage devices such as SSDs to achieve profound improvements in latency and throughput. NVMe-based SSDs have been emerging as the latest storage technology bridging the dreaded performance gap between hard disks and memory. These new devices are built for extremely low latency and achieving high degrees of parallel I/O. This makes them ideal to be used in cloud environments, where sharing of resources is a given.

In cloud environments, users expect a certain guarantee of service. Considering the new NVMe technology being introduced in enterprise clouds, it is only natural to ask whether a similar guarantee of service can be provided for this emerging hardware. In fact, this issue has been addressed to some extent in the NVMe standard itself. The standard includes provisions to enable request arbitration through mechanisms which are to be provided by hardware. However, there is limited knowledge on using these provisions to enable service guarantees in cloud environments. Prior research [9, 57, 58, 97, 117, 140, 151, 153, 188] has mostly focused on software-based provisions for service guarantees. While previous

approaches [85, 155] have considered using such hardware provisions to provide some QoS to users, their approaches are not holistic. The designs proposed do not provide a complete solution for providing SLA-based guarantees to users. For providing such a solution, there are two key requirements. First, the SLA provisioning should be completely application oblivious, i.e., it should be completely handled by the cloud provider based on the SLA negotiated by the user. Second, there must be mechanisms in place which allow for the provisioning of SLAs without violations. Achieving these requires a holistic approach which we propose in this chapter. We show how existing runtimes can be modified for application oblivious QoS provisioning. We also theoretically model the arbitration mechanism available in NVMe and discuss how the model can be used for SLA mapping and provisioning.

NVMe SSDs are still considered as emerging hardware. While costs have been rapidly declining in recent years, they are still high enough to prevent wide-scale adoption in cloud environments [52, 126, 185]. Having a system which can provide NVMe device emulation can prove to be very useful. Emulation allows for testing NVMe related code without the need for buying expensive hardware. In addition, emulation also allows for approximate performance modeling of such applications. Existing schemes for NVMe emulation do not provide any mechanisms to test and evaluate the arbitration mechanisms available in the standard. Moreover, no flash device has implemented weighted arbitration schemes yet [97]. Thus, we propose designs for accurate modeling of QoS schemes in the NVMe part of Quick Emulator (QEMU) [18, 134]. With this solution, cloud providers can not only verify their middleware and schedulers, but can also use it for performance modeling and benchmarking.

To summarize, the main contributions of this work are as follows:

- Design of QoS-aware NVMe emulator which provides support for weighted round robin and deficit round robin arbitration
- Theoretical modeling of arbitration schemes with queuing theory
- Extension of Storage Performance Development Kit (SPDK) runtime to allow for application oblivious SLA provisioning

Our evaluation shows that by combining our QoS-aware NVMe emulator and enhanced SPDK runtime, we can achieve I/O bandwidth SLA guarantees in an application oblivious manner. We also observe that our QoS-aware emulator can deliver similar or better performance as compared to the existing emulator in QEMU. To the best of our knowledge, our proposed emulator is the first NVMe emulator to offer support for QoS.

6.1 QEMU NVMe Emulation

NVMe SSDs are still considered as emerging hardware. Although NVMe SSDs have been commercially available for several years now, their cost is a significant barrier to their adoption in large-scale cloud environments. We believe that over time, as the performance of SSDs increases and their cost decreases, their adoption will see an exponential increase. While it is important to design runtime and middleware that can take advantage of the NVMe standard, it is also important to provide mechanisms to emulate NVMe devices. Emulation allows for testing NVMe related code without the need for buying expensive hardware. In addition, emulation also allows for approximate performance modeling of such applications.

There exist many solutions that provide the ability to emulate NVMe devices. The most robust and stable of these is provided as part of QEMU. QEMU [143] is a popular

open-source hypervisor for hardware virtualization. QEMU introduced NVMe emulation support a few years ago which has now developed into a stable tool for NVMe device virtualization. The reason that we chose QEMU for this purpose is that it is a popular and well-known framework for hardware virtualization and it already includes mechanisms to virtualize Memory Mapped IO (MMIO), block IO, hardware interrupts (e.g. MSI-X), and PCIe devices. This makes it perfect for emulation of NVMe devices.

6.1.1 The Need for QoS-Aware Emulation and Runtime

QoS is an extremely important part of the cloud computing paradigm. In fact, this is one of the primary reasons for the popularity of cloud computing. Most cloud providers these days offer SLAs to their clients as a basic way of achieving QoS. In the context of NVMe storage, it is paramount to have a design that provides some guarantee of service (e.g., latency or bandwidth) to applications or VMs depending on the service granularity. NVMe provides hardware-based mechanisms for command arbitration. Schemes like round robin and weighted round robin arbitration (WRR) (see Section 6.2.1) have been included in the NVMe standard, while NVMe SSD vendors are free to implement vendor specific arbitration mechanisms. However, none of these schemes have been implemented in commercially available SSDs.

While the QEMU NVMe emulator supports the entire NVMe command set, the hardware-based weighted round robin arbitration mechanism specified in the NVMe standard is not supported. Cloud providers providing service guarantees for storage might want to test their scheduling solutions using hardware virtualization. In this context, having an emulator for NVMe devices which can provide command arbitration mechanisms as described in the

standard can prove to be extremely useful. We believe that making such a solution available will tremendously benefit cloud providers in designing scheduling solutions. We do not focus on improving the emulator to accurately model the *performance* characteristics of NVMe devices, but rather on accurate modeling of the *command processing* and *arbitration* mechanisms. This will allow us to provision service guarantees using the improved QEMU emulator.

6.2 QoS-Aware NVMe Emulation

In this section, we describe our proposed design for QoS-aware NVMe emulation. We first briefly describe the Weighted Round Robbin and Deficit Round Robbin (DRR) arbitration scheme, then present our solution for implementing these schemes in QEMU, and finally demonstrate via experimental evaluation the superiority of our solution over the existing NVMe emulation in QEMU.

6.2.1 WRR Arbitration

The WRR arbitration mechanism in the NVMe standard provides a useful mechanism to implement QoS support in cloud middleware. The biggest advantage of this mechanism is that it is implemented completely in hardware resulting in low latency arbitration and reduced complexity of drivers and runtimes. This mechanism works as follows. There are three different priority classes for NVMe submission queues, high, medium, and low. Each class of priority is assigned a numerical weight. The NVMe controller processes commands for submission queues in order of their priorities. The maximum number of commands that can be processed for queues of a certain priority in one arbitration round is determined by the weight of that priority class. For a single submission queue, the maximum number of commands that can be processed in one arbitration round is determined by the arbitration

burst setting. By adjusting the weights of different classes of priority, the desired level of QoS can be achieved. The current NVMe emulator in QEMU does not provide support for the WRR arbitration mechanism. In this section, we describe our proposed design for a QoS-aware NVMe emulator.

6.2.2 DRR Arbitration

While the WRR scheme is efficient in providing a guarantee of throughput, it requires the sizes of requests to be fixed or previously known. Otherwise, applications with different sized requests will result in a higher than intended weight for large requests. In this context, significant research has been done to provide solutions which can provide optimal bandwidth QoS despite request size variation. Schemes like deficit round robin (DRR) and weighted fair queuing (WFQ) are popular models widely used in networking. Both DRR [152] and WFQ [39] can provide bandwidth guarantees. However, WFQ requires $O(log(n))$ time to process each request, while DRR only requires $O(1)$, where n is the number of priority classes. Even though there are just three priority classes (as defined in the NVMe standard), DRR is simpler and satisfies our requirements for QoS. DRR is a modification of WRR, where instead of giving each request equal cost, its given a cost equal to its size. This effectively ensures that the overall bandwidths for each priority class are in the ratio of their weights. It is practical to extend WRR to DRR because it is easy for the NVMe controller to determine the size of each request by quickly peeking at the size entry in the request structure. We believe that DRR is an effective alternative to WRR which can provide better QoS with the same overhead. We implement the DRR scheme in the QEMU NVMe emulator and show that it is much more effective in providing bandwidth guarantees to cloud users than WRR.

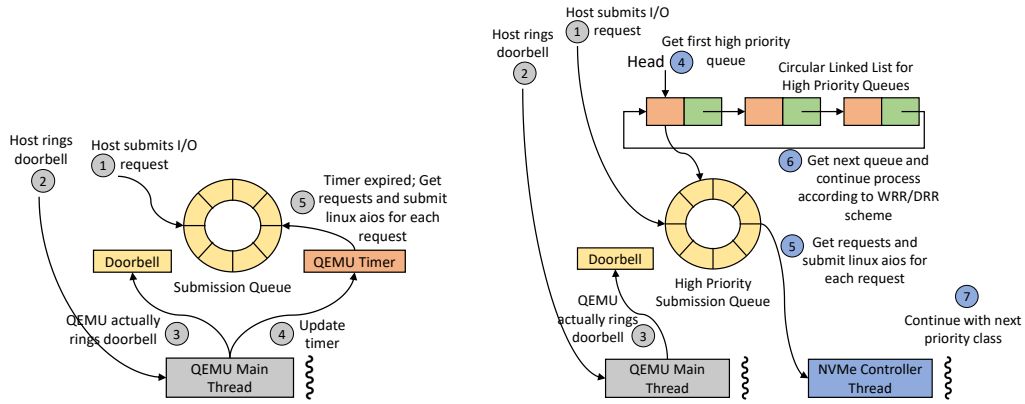


Figure 6.1 (left) Existing command submission procedure for QEMU NVMe emulation, and (right) Proposed command submission procedure for QoS-aware NVMe emulation

6.2.3 Designing Hardware-Based Arbitration Schemes

For basic NVMe emulation, the existing designs in QEMU can be re-used. However, for providing QoS, the software-based arbitration mechanism in QEMU needs to be re-designed.

Existing emulation design. The left side of Figure 6.1 shows the command submission process for the existing emulator. The working can be detailed by the following steps: ① The host puts an I/O request in the submission queue, ② The host rings the queue doorbell, ③ This triggers an interrupt which is processed by the QEMU main thread, ④ The QEMU main thread updates the timer of the submission queue, and ⑤ The timer expires and the main thread submits Linux aios for each request. The default design uses the QEMU main thread to execute the entire I/O processing pipeline. The fundamental flaw with this approach is that as soon as a command is placed in the submission queue, the main thread will start processing it. Submitting an I/O request involves ringing a doorbell which generates an interrupt. This interrupt is also handled by the QEMU main thread. Thus,

at one time, only one command can be in any submission queue. This design allows no conception of QoS and is fundamentally different from the way commands are processed in an actual NVMe SSD.

Design Considerations. Emulation is an effective tool for testing the performance and functionality of hardware. A good emulation requires accurate modeling of hardware, i.e., the software implementation should mimic the behavior of hardware. In addition, specially while proposing new hardware logic, the functionality, complexity, and memory usage should be kept as minimal as possible. This allows for cost savings and latency benefits. In this context, we emulate arbitration schemes using simplistic space efficient data structures. Each priority class is assigned a circular linked list for holding pointers to the submission queues in the priority class. In addition, 8 bit unsigned integers are used to store remaining and pre-defined weights for each priority class which will map to hardware registers or dedicated buffers on device memory for each queue. After each arbitration round, the remaining weights are updated to the pre-defined weights. In addition, a single bit is kept for each submission queue to indicate if it has a pending request. This allows for efficient arbitration by allowing the controller to quickly skip over empty queues. We believe that this design closely mimics hardware behavior. We now describe our arbitration scheme designs in QEMU.

Proposed Design. For emulating the NVMe device in a more realistic manner, we introduce a dedicated thread for executing the functions of the NVMe controller. We also make sure that data structures shared with the QEMU main thread are locked using a mutex before access. In addition, to allow for software-based arbitration, we introduce a circular linked list for each priority class. Each submission queue is added to the appropriate linked list when it is created. We maintain just one head pointer for each linked list which points

to the submission queue that should be used next by the controller for command arbitration. The controller processes commands from each priority class one by one. For each priority class, the controller will use the appropriate linked list and start processing commands for the queue pointed to by the head pointer, moving the pointer each time a queue is serviced. It will continue processing commands for the priority class until either all queues have been serviced or the total cost of commands processed is equal to the weight of the priority class. Thus in one arbitration round, all priority classes will be served, but the maximum cost of commands that can be processed for each class is equal to its weight.

The right side of Figure 6.1 shows the command submission process for the proposed emulator. The following steps describe the complete process: ① The host submits requests directly to the submission queues, ② The host then rings the doorbell, ③ This generates an interrupt which is handled by the QEMU main thread, ④ The NVMe controller thread first uses the high priority linked list, ⑤ It picks up each submission queue with an outstanding request and submits Linux aios for requests, ⑥ Controller thread moves onto next submission queue, and ⑦ After processing requests for all submission queues in a priority level, the controller moves to next priority linked list. The NVMe controller thread in parallel continuously scans the circular linked lists for each priority class. Whenever it finds pending requests, it submits a Linux aio operation for each request while ensuring WRR/DRR arbitration. Our NVMe controller thread can execute independently of the QEMU main thread. This allows the request submission and processing to proceed in parallel, thereby allowing for the possibility of multiple outstanding requests in submission queues. This emulates the actual behavior of an NVMe SSD more accurately. Applications submit and place a request in the submission queues independent of the processing of requests by the NVMe hardware. Thus, our solution provides a better emulation of NVMe hardware.

There are many possible solutions for designing arbitration in QEMU. For example, the QEMU aio interface can be used to process commands for each submission queue parallelly. We chose our design with the goal of emulating the NVMe device behavior as accurately as possible with minimal overhead. In our design, we use just one additional thread, but the arbitration mechanism and NVMe controller are precisely emulated. Our experimental analysis also confirms this claim.

6.3 Performance Modeling of Arbitration Schemes

In this section, we model the performance of the WRR and DRR arbitration schemes.

6.3.1 Performance Modeling

In the previous section, we presented our QoS-aware NVMe emulator design. To allow cloud providers to fully utilize this new tool, it is necessary to provide a model for performance prediction. This will allow for accurate SLA provisioning with minimal violations.

Symbol	Description
$h/m/l$	high/medium/low priority class weight
λ	input request rate
γ	average SSD throughput
μ	SSD processing rate
ρ	queue utilization
p	SSD parallelism
q	queue depth
a	average request latency

Table 6.1 Notation Used

We first model the WRR throughput. The presence of many varying situations and variables makes the performance modeling of WRR challenging. For simplicity, we assume that the average I/O size for each priority class is the same. For each priority class, the maximum commands processed in one round will be equal to its weight. However, in case the submission queues in a priority class do not have more commands than its weight, other priority classes will be able to get higher throughput. Thus, the minimum throughput for each priority class should be in the ratio of their weights. Assuming that γ is the average throughput the NVMe SSD can deliver and h, m, and l are the weights of the corresponding priority classes, the minimum throughput for the high priority class can be calculated out as

$$\gamma_h \geq \frac{h}{h+m+l} \times \gamma \tag{6.1}$$

Minimum throughput for other priority classes can be calculated similarly. For estimating the actual throughput of each priority class, queuing theory can be used. While each priority class can have multiple queues, they can be considered to constitute one combined big queue. If λ_h, λ_m, and λ_l are the input request rates for the respective priority classes, we know from queuing theory that $\gamma = \lambda$, i.e. output rate is equal to input rate if input rate is not greater than the processing rate μ. This is generally true for NVMe since the command submission will fail if the submission queue is full. The input rate will then automatically adjust to be equal to the output rate. For a general case, the weights in Equation 6.1 need to be multiplied by the utilization (ρ) of each priority class so that we can determine the average commands processed from each class in one arbitration round. The utilization of each class can be computed as $\frac{\lambda}{\gamma}$ using queuing theory. Thus, the average throughput for each class can be calculated as

$$\gamma_h = min(\lambda_h, \frac{h}{h + \rho_m m + \rho_l l} \times \gamma) \tag{6.2}$$

It is easy to see that if we do not submit any requests in the medium and low priority classes, i.e ρ_m and ρ_l are zero, then γ_h will be γ or the complete throughput of the SSD as long as the input rate λ_h is high enough to keep the SSD busy. The utilization of each priority class (ρ) is the ratio of its input rate to max throughput. Utilization determines how many requests are available to be processed for the priority class in one arbitration round. A utilization of one means that the number of requests available for processing is equal to the weight of the priority class. For DRR, replacing γ with the average SSD bandwidth should suffice since it is bandwidth based. In addition, no guarantees on the request size are required.

To calculate the latency for each operation of a priority class, we need to account for two factors. First, queuing delay and second, internal parallelism in the SSD. Queuing delay accounts for the time a request waits to be serviced by the SSD. In general, a request in a particular priority class must wait for requests before it in its own class as well as requests in other priority classes to finish. Assuming requests take an average of time of a to complete, p requests can be submitted in parallel, and each priority class can have a maximum of q outstanding requests, we can calculate the maximum latency for the high priority queue as

$$t_{max} = \frac{q}{h} \times \frac{h + \rho_m m + \rho_l l}{p} \times a \tag{6.3}$$

This equation should hold true for both DRR and WRR as long as the average request size for all classes is the same. Otherwise, a will not remain a constant. The effective parallelism p of the SSD can approximately be estimated based on the number of flash chips

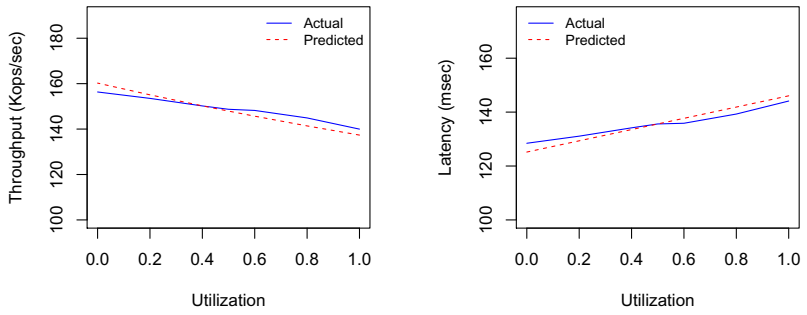


Figure 6.2 Model Validation: actual and predicted high priority class throughput and latency with varying low priority queue utilization (ρ_l)

it has. This is assuming that requests are uniformly distributed over the logical address space. For workloads with different request distributions, conflicts between requests will likely reduce the parallelism. Write workloads will be adversely affected by garbage collection activities further reducing the parallelism. Assuming no conflicts between requests, the effective parallelism can be derived by dividing the number of flash channels by the write amplification factor. The write amplification can accurately estimated using equations proposed in [40]. In some scenarios, it may be infeasible to accurately know the IO request pattern of workloads. In these cases, a better solution would be to estimate the a/p ratio empirically by running the workload once.

6.3.2 Model Validation

To prove the validity of our proposed model, we run experiments on an NVMe SSD to confirm the latency and throughput characteristics under different load conditions. Our testbed consists of one node with a 8 core, 16 thread sandy bridge CPU with 32GB DRAM and an Intel P3700 NVMe SSD. We use CentOS 7.1 with the 4.9 Linux kernel. However, as mentioned before, there does not exist an actual flash device offering either WRR or

DRR arbitration in hardware. Hence, we build a thin software layer over SPDK, providing software-based queues offering DRR and WRR schemes. This layer is designed to be light weight with minimal locking to ensure hardware-like performance. To validate our model, we run a random read benchmark with three threads using the three separate priority classes. DRR weights are set to (32k, 64k, 128k) and each thread submits 32k IO requests. Each priority class is assigned a separate hardware queue, although, request submission and completion is handled by a single thread to ensure DRR compliance. We vary the utilization of the low priority thread and measure the throughput and latency of the high priority thread. For our model, we estimate γ and a/p through empirical analysis. Both of these parameters can be estimated by running a simple multi-client benchmark with one thread per physical core utilizing a separate hardware NVMe queue. γ was found to be around 240k while $\frac{a}{p}$ was around 81 μs. We compare these results with those predicted by our model. This comparison is presented in Figure 6.2. Figure 6.2(a) shows the high priority throughput with varying low priority utilization (ρ_l). ρ_h and ρ_m are set to one. Figure 6.2(b) shows the latency of the high priority class with varying low priority utilization. We observe near perfect correlation between the observed and predicted values with a maximum deviation of less than 5%. Thus, we believe that our model works well in practical situations and is a useful tool for performance prediction. Given the estimated values of γ, a, and p, we can accurately predict the latency and throughput of any priority class.

6.3.3 Model Usage Scenarios

Calculating the average throughput and maximum latency for each priority class can prove to be useful in multiple scenarios while provisioning I/O bandwidth and latency SLAs in cloud environments. Consider a scenario where some job is already using an

NVMe device and the cloud resource scheduler would like to schedule another job to use the same device. By using Equation 6.2, the scheduler can calculate the I/O bandwidth for both jobs and determine whether their respective SLAs will be violated. A decision can then be made about scheduling the new job to use the NVMe device. Now consider another scenario where the scheduler would like to assign weights to each priority class. Equations 6.2 and 6.3 can be used to calculate the weights, assuming that the bandwidth and latency SLAs and utilization of each job are known beforehand.

We have shown in this section that the performance modeling of arbitration schemes in NVMe can serve to be extremely useful in cloud environments.

6.4 QoS-aware SPDK Runtime

In this section, we describe our proposed approach for designing a QoS-aware SPDK runtime. We start by presenting our solution for enabling service guarantees in NVMe-based cloud environments, before moving on to our application oblivious design for QoS provisioning using SPDK. Finally, we present evaluation results with synthetic application scenarios to demonstrate the benefits of our design.

6.4.1 Enabling Service Guarantees

In modern cloud environments, users expect a guarantee of service for their applications. Cloud providers typically negotiate SLAs with users as a way to provide these guarantees. In such a scenario, cloud middleware and runtime should be able to provide mechanisms to satisfy these guarantees as QoS. From an end user's perspective, the SLA provisioning should be completely transparent. So, the service guarantee mechanisms should be completely application oblivious. In the context of NVMe storage, our goal is to use the

hardware provided arbitration mechanisms as a way of providing applications with a guarantee of IO bandwidth. Since these schemes should be application oblivious, we propose to modify the (SPDK) runtime to allow for QoS support.

A cloud environment typically charges users based on the level of priority they desire. Since, the NVMe WRR arbitration scheme allows for three priority classes as discussed before, we propose the same priority classes for application users as well. The corresponding priority classes will be mapped to each other such that an application with high priority will submit IO requests to the high priority submission queue and so on. The DRR scheme is also based on the same three-priority system. The weight of each priority class, which determines the maximum number of IO commands that can be processed from one class in an arbitration round, can be determined by the cloud provider based on the SLA negotiated with the user. Similarly, for scheduling multiple users to share an NVMe SSD, the IO priority requested by each user and the priority class weights need to be considered.

6.4.2 Application Oblivious QoS Provisioning

The SPDK runtime has support for the NVMe WRR scheme. This is however left to the discretion of the user himself. To use the WRR scheme, the user has to explicitly enable it using an SPDK function and set the priority for each submission queue created. This existing mechanism does not satisfy our application oblivious requirements. We thus propose a new priority mapping design in SPDK which does not require application changes to modify priority. To this end, we propose to use the Linux IO priority framework as a means to transfer the priority class information from the application to the SPDK runtime, similar to the approach proposed in [85].

The IO priority class for an application can be set using the *ionice* command which expects a value from 0 to 3. We use the 1-3 classes and map them to high, medium, and low priority classes. 0 is mapped to the urgent priority class and is left for cloud administrative purposes. The priority class for the application can then be obtained by SPDK using the Linux IO priority interface (`ioprio_get` syscall). This design works because the SPDK runtime will be run in the context of the thread submitting IO. We modify the SPDK runtime to always use the WRR or DRR scheme and set the priority for a QP based on the IO priority of the application it is associated to. The IO priority of each application can be set by the cloud middleware based on the SLA with each user. In this manner, any application built using SPDK can be provided any level of service without the need to modify the application.

6.4.3 Handling Rogue Tenants

Our QoS-aware runtime provides tenants with an interface to provide a certain guarantee of service. In this regard, it is important to ensure that a rogue tenant cannot influence the performance of others. In our design, each tenant is allocated a separate QP for IO request processing, resulting in performance isolation. The actual request processing is done by the SSD controller which ensures that the service guarantees of each priority class are maintained. A rogue tenant might submit requests at a rate higher that its service guarantee. This will only result in its submission queue getting filled up, and eventually it will be unable to submit requests. This will result in its throughput getting limited to its service guarantee. Thus, QoS will be ensured regardless of how the tenants submit requests.

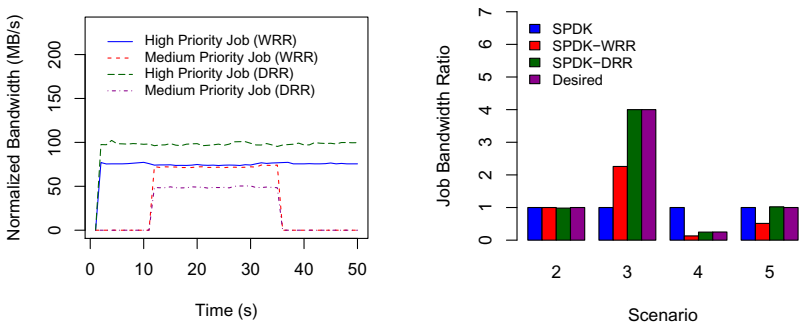


Figure 6.3 Evaluation with Synthetic Application Scenarios: (a) Bandwidth over time with Scenario1, (b) Job bandwidth ratio for Scenarios 2-5

6.4.4 Synthetic Application Scenarios

We use the same testbed as described in Section 6.3 for our evaluations. We use a modified version of QEMU [134] and SPDK v17.10 for our evaluations and as a base for our designs.

To show the benefits of a QoS-aware SPDK runtime, we simulate five application scenarios and measure the bandwidth achieved by each job over time. For all scenarios, we set the priority class weights to (32, 16, 8) for WRR and (128k, 64k, 32k) for DRR, ensuring that the weight ratios are the same. For the first scenario, we use an experiment similar to the one in Section 6.1. In this scenario, we run one high priority job with 4k requests and one medium priority job with 8k requests. Figure 6.3(a) shows the results of this experiment. With the WRR, both jobs receive the same bandwidth despite having different priorities. Each request is given equal priority regardless of its size, leading to a skewed bandwidth distribution. However, DRR is able to achieve near perfect bandwidth distribution as per priority weights. We also note that the bandwidth over time is relatively stable pointing to a robust hardware emulation.

The remaining scenarios all have 2 simultaneous jobs submitting back-to-back requests. Scenario 2 has two high priority jobs, both with 4k requests. Scenario 3 has a high priority job with 4k requests and a low priority job with 8k requests. Scenario 4 is the same as Scenario 3 with the priorities exchanged. Scenario 5 has two high priority jobs, one submitting 4k and 8k requests and the other 8k and 16k requests. We measure the total average bandwidth for both jobs in each scenario and calculate the bandwidth ratio. This analysis is presented in Figure 6.3(b). We also provide the expected ratio as per the priority weights. We are more interested in the bandwidth ratios rather than their actual values since we are focused on QoS and the hardware performance is only emulated. In all scenarios, the difference in request sizes leads WRR to incorrectly favor the job with larger request size, while DRR is able to achieve close to the expected ratio. The only case where WRR provides the desired ratio is Scenario 2, where the request sizes for both jobs are the same. Vanilla SPDK just provides equal throughput distribution, not providing any service guarantees whatsoever. This analysis clearly demonstrated the superiority of DRR over WRR in achieving bandwidth guarantees.

Although our results are based on NVMe emulation, we expect similar behavior with actual hardware. There are two reasons for this expectation. One, both WRR and DRR have been shown to be easy to implement in hardware [88, 152] and provide accurate bandwidth ratios. Two, usage of separate hardware queues for each application along with lockless request submission and completion paths ensure that the hardware performance characteristics are reflected in the application performance. **We thus believe that DRR should either become part of the NVMe standard or be accepted by vendors as a good implementation choice for vendor specific arbitration schemes.**

6.5 Related Work

Several prior works have studied QoS support over shared cloud and data-center storage systems [57, 58, 117, 153, 188]. For flash-based storage, in particular, specific cost model based I/O schedulers such as FIOS [140] and FlashFQ [151] designed for fairness and throughput guarantees have been proposed. On the other hand, providing QoS-aware runtimes for NVMe devices has been a topic of recent research. Joshi et al. [85] propose to implement WRR support for NVMe in the Linux driver. They employ a similar I/O priority-based approach for application oblivious QoS provisioning. However, their design suffers from two fundamental flaws. First, they implement their design in the Linux driver which we have shown to perform poorly as compared to SPDK. Second, they provide no mechanism for cloud providers to provision SLAs using their solution. We argue that our hardware-assisted QoS solution is more applicable in terms of performance and usability in cloud environments.

With the availability of fast network interconnects, storage disaggregation-based technologies and systems are being extensively explored [12, 96]. Along these lines, software-based systems for accessing remote NVMe Flash at latencies as low as local NVMe access, such as ReFlex [97], have been proposed. Open-source disaggregated I/O architectures, such as Crail [12], are built exclusively with user-level I/O support (e.g., NVMf, SPDK, RDMA), allowing heterogeneous storage and networking hardware to interact with each other in an optimal manner within the data processing engine.

6.6 Summary

NVMe-based devices are gaining popularity in cloud environments. This trend motivated us to work on analyzing, modeling, and provisioning QoS for NVMe SSDs [55].

In this chapter, we first evaluated the existing QEMU-based NVMe emulator using performance and QoS as our metrics. We concluded that the emulator is insufficient for providing any service guarantees. We then proposed designs for accurate modeling of hardware-based WRR and DRR in the QEMU NVMe emulator. We theoretically modeled the arbitration schemes and showed how the analysis can be used for SLA provisioning in cloud environments. Finally, we discussed a new approach for providing service guarantees using a QoS-aware SPDK runtime. We demonstrated through experimental evaluation that our QoS-aware NVMe emulator with DRR scheme and SPDK runtime can deliver bandwidth service guarantees in cloud environments in an application oblivious manner. This work should prove useful to vendors while designing arbitration schemes in SSDs and to cloud providers in provisioning SLAs for tenants.

Chapter 7: Future Research Directions

In this section, we present future research directions that the community is moving towards.

7.1 Distributed Persistent Pools

A good avenue for future research is the concept of distributed persistent memory. Prior research [150] has shown that having such a solution is incredibly useful in different scenarios. Distributed persistent pools can effectively aggregate PMEM across several nodes and be used to design distributed filesystems. However, there still remain several challenges that must be solved to have a practical and scalable solution. Transferring data between PMEM on different nodes can be directly achieved via RDMA. However, making sure that data is actually persisted is still an ongoing research problem. There is not yet an efficient mechanism to guarantee remote persistence. Software-based solutions, which we have proposed in this thesis, have noticeable latency overhead, while hardware changes have yet to be standardized. However, our prior work [105] has shown that proposed hardware primitives, when available, can be used to efficiently solve this problem. Second, PMEM is vulnerable to uncorrectable media errors and memory corruption. In this context, it is imperative to ensure data availability and durability despite corruption. Commonly used

techniques like replication and erasure coding can be applied to solve this issue, but the design of such schemes for PMEM is still an emerging research direction. As a future work of this thesis, we plan to investigate possible practical solutions that solve these issues. We also plan to explore how these distributed pools can be effectively utilized in storage systems.

7.2 Computational Storage/Memory

The conventional computational paradigm involves moving data from storage to compute (usually the CPU), processing that data using a kernel and moving the results back to storage. As the size of data has exploded in recent times, distributed and disaggregated storage systems have become commonplace to manage and store the vast volume of data. In such systems, the traditional computational paradigm results in a majority of application time being spent in moving data between the CPU and storage, reducing the effectiveness of compute capabilities and increasing application runtime. To solve this problem, computational storage/memory is a new paradigm that has been proposed and is gaining popularity in recent years. In this paradigm, instead of data being moved to compute, compute is moved to data. The key advantage is that movement of compute is significantly less expensive and less frequent. Therefore, compute capabilities can be utilized to their maximum. Recently, Samsung and Xilinx have introduced SmartSSDs [177] which house programmable FPGAs to allow certain computational functionalities to be offloaded to these devices. The major research challenges here include coming up with abstractions to allow code shipping and to program near-storage computational units, such as SOCs and FPGAs.

Chapter 8: Impact on the Storage Community

8.1 Impact on the Design of Distributed Storage Systems

The main contributions of this thesis are new models, algorithms, data structures, and abstractions which enable both programmers and end-users to efficiently utilize new hardware technologies. Therefore, our work is not tied to any particular storage system or application. The proposed designs are generic enough to be applicable to any distributed storage system. For instance, the `microfs` abstraction presented in Chapter 5 can be utilized to design coordination-free ephemeral storage systems. Furthermore, as we showed in Chapter 4, our DIPPER algorithm can be exploited to build fast, persistent crash-consistent data structures to store file system metadata. Finally, the Arcadia log interface and design is generic and can be used for any log implementation on PMEM.

8.2 Impact on Scientific and Web Applications

The designs presented in this thesis are specifically targeted towards long-running scientific applications and short-running web services. We have taken one production application from both these classes of applications – CoMD is representative of scientific applications while AdMaster is representative of web services. Through experimental analysis, we

showed that the proposed designs can successfully improve the performance of these applications and allow them to be run at large scale efficiently. We expect that other applications in these domains will be able to utilize the proposed designs and get similar improvements. Increasing the performance and scalability of applications can also make it possible to run simulations at scales that were earlier impractical. This is especially true for time-sensitive simulations, such as weather prediction. The low progress rates with current generation storage systems imply that by the time a simulation is complete, it may no longer be useful. In these scenarios, the highly efficient `NVMe-CR` design can be utilized to reduce the simulation time and make the results useful.

8.3 Impact on NVMe SSD Design

Hardware-based arbitration schemes for NVMe SSDs serve as a great base to provide basic service guarantees when accessing storage in cloud environments. Relying on hardware schemes as opposed to software schemes not only results in better performance but also provides fine-grained control over QoS. The hardware-assisted QoS work in this thesis demonstrates the shortcomings of the WRR arbitration scheme proposed in the NVMe standard. We showed that DRR is a more suitable scheme to provide bandwidth guarantees and is easy to implement in hardware. The NVMe standard allows SSD vendors to implement custom arbitration schemes. We expect that vendors will implement DRR or similar schemes in future generations of SSDs.

8.4 Software Release and Wide Acceptance

The High-Performance Big Data (HiBD) project [127] aims to make scalable high-performance designs of popular Big Data stacks publicly available to the community. Packages provided by the project include optimized versions of Hadoop, Spark, and HBase. The HiBD packages are being used by more than 335 organizations worldwide in 37 countries to accelerate Big Data applications. More than 38,000 downloads have taken place from this project's site. Some of the designs proposed in this thesis, including our thread-based architecture work, have been integrated into the RDMA-based Apache HBase package and can be used on RDMA clusters to seamlessly accelerate data analytics. In addition, code for our lock-free persistent data structure designs [53] has been open-sourced and is available at `https://github.com/padsys/PMIdioBench`. In the future, we also plan to make the Arcadia, DStore, and `NVMe-CR` code open-source to further promote academic research and industry collaboration.

Chapter 9: Conclusion and Contribution

Modern datacenters and HPC clusters are increasingly adopting non-volatile memory-based storage devices and RDMA-based high-throughput interconnects. This trend is expected to grow as the amount of data to be stored is growing exponentially. To be able to utilize these new technologies effectively, storage systems design principles need to be re-evaluated and re-designed to exploit the full potential of modern architectures. Unfortunately, existing storage runtimes are designed considering the performance characteristics of archaic hardware like spinning disks and Ethernet networks. Therefore, these systems are unable to extract optimal performance from modern hardware and lead to poor application-level performance. In this thesis, we have proposed a three-pronged approach to fully exploit the architectural features of new technologies – (1) reducing software overhead, (2) utilizing new features of hardware, and (3) rethinking storage system design principles.

The work proposed in this thesis has provided a thorough analysis of existing storage system designs on emerging non-volatile memory systems and identified various limitations and associated bottlenecks. To circumvent these problems, we proposed Navi Store, a generic storage sub-system, with four key design components. These four proposed components take a cross-layer holistic approach to system design.

The first major contribution of this thesis is Arcadia, a scalable replicated log on PMEM. This design hides the data consistency, persistence, and integrity problems with PMEM and makes it easier to use. Our analysis shows that Arcadia significantly outperforms state-of-the-art PMEM logs, such as FLEX, PMDK's libpmemlog, and Query Fresh while providing stronger log record durability guarantees. We expect Arcadia to be used as an off-the-shelf log implementation for any storage system using logging, in particular systems that have weak consistency or disaggregated storage.

The second major contribution is DIPPER, an approach for designing persistent data structures. This algorithm utilizes the byte addressability of PMEM to hide its persistence overhead without impacting concurrency or fault-tolerance. We design a generic storage sub-system, called **DStore**, short for Decoupled Store which uses DIPPER to implement its control plane. Experimental results demonstrate that DStore can deliver up to 6x lower tail latency service level objectives and up to 5x higher throughput service level objectives compared to state-of-the-art PMEM optimized systems like PMEM-RocksDB, MongoDB-PMSE, and NOVA.

The third major contribution is microfs, a coordination-free abstraction for storage system design. Using this abstraction serves as a base, we designed NVMe-CR, a direct-access ephemeral storage system with minimal coordination requirements for accelerating checkpoint IO. Compared to these state-of-the-art systems, NVMe-CR can reduce checkpoint overhead by as much as 2x. Furthermore, through increased efficiency and low software overhead, our runtime can lower the required hardware IO bandwidth (and the overall TCO as a consequence) by as much as 2x.

The fourth and final major contribution is a hardware-based QoS approach. This design provides request-size agnostic latency and bandwidth service guarantees in an application-oblivious fashion. We demonstrated through experimental evaluation that hardware-based arbitration with DRR scheme and SPDK runtime can deliver bandwidth service guarantees in cloud environments in an application oblivious manner. This work should prove useful to vendors while designing arbitration schemes in SSDs and to cloud providers in provisioning SLAs for tenants.

The designs proposed in this thesis have shown empirically better performance at micro-benchmark as well as at application-level in comparison to state-of-the-art solutions employed by production storage systems when running on modern datacenters and HPC clusters. In particular, evaluation with YCSB shows that DStore can improve application throughput by up to 2x. Further, using the ECP CoMD application, our results show that NVMe-CR can achieve near perfect (> 0.96) efficiency at 448 processes.

Bibliography

[1] AdMaster. http://www.admaster.com.cn/en/ (accessed Dec. 2020).

[2] Apache HBase. http://www.hbase.apache.org (accessed Dec. 2020).

[3] Apache Phoenix. http://phoenix.apache.org/ (accessed Dec. 2020).

[4] Apache ZooKeeper. http://www.zookeeper.apache.org (accessed Dec. 2020).

[5] FUSE: Filesystem in Userspace. http://fuse.sourceforge.net/ (accessed Dec. 2020).

[6] InfiniBand Trade Association. http://www.infinibandta.com (accessed Dec. 2020).

[7] Abhinav Agrawal, Gabriel H Loh, and James Tuck. Leveraging Near Data Processing for High-Performance Checkpoint/Restart. In *SC*, page 60, 2017.

[8] Jung-Sang Ahn, Woon-Hak Kang, Kun Ren, Guogen Zhang, and Sami Ben-Romdhane. Designing an Efficient Replicated Log Store with Consensus Protocol. In *HotCloud*, 2019.

[9] Sungyong Ahn, Kwanghyun La, and Jihong Kim. Improving I/O Resource Sharing of Linux Cgroup for NVMe SSDs on Multi-core Systems. In *HotStorage*, 2016.

[10] Hiroyuki Akinaga and Hisashi Shima. Resistive Random Access Memory (ReRAM) Based on Metal Oxides. *Proceedings of the IEEE*, 98:2237–2251, 2010.

[11] David G Andersen and Steven Swanson. Rethinking Flash in the Data Center. *IEEE Micro*, 30(4):52–54, 2010.

[12] Apache. Crail. `https://crail.incubator.apache.org/` (accessed Dec. 2020), 2017.

[13] Joy Arulraj, Andrew Pavlo, and Subramanya R Dulloor. Let's Talk About Storage and Recovery Methods for Non-Volatile Memory Database Systems. In *SIGMOD*, pages 707–722, 2015.

[14] Joy Arulraj, Matthew Perron, and Andrew Pavlo. Write-Behind Logging. *Proceedings of the VLDB Endowment*, 10(4):337–348, 2016.

[15] InfiniBand Trade Association. Supplement to InfiniBand Architecture Specification Volume 1, Release 1.2. 1: Annex A16: RDMA over Converged Ethernet (RoCE), 2010.

[16] Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh. POSIX Abstractions in Modern Operating Systems: The Old, the New, and the Missing. In *EuroSys*, page 19, 2016.

[17] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobbler, Michael Wei, and John D Davis. CORFU: A Shared Log Design for Flash Clusters. In *NSDI*, pages 1–14, 2012.

[18] Fabrice Bellard. QEMU, A Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.

[19] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. PLFS: A Checkpoint Filesystem for Parallel Applications. In *SC*, page 21, 2009.

[20] Wesley Bland, Aurelien Bouteiller, Thomas Herault, George Bosilca, and Jack Dongarra. Post-failure Recovery of MPI Communication Capability: Design and Rationale. *The International Journal of High Performance Computing Applications*, 27(3):244–254, 2013.

[21] Dhruba Borthakur et al. HDFS Architecture Guide. *Hadoop Apache Project*, 53:1–13, 2008.

[22] Jiajun Cao, Kapil Arya, Rohan Garg, Shawn Matott, Dhabaleswar K Panda, Hari Subramoni, Jérôme Vienne, and Gene Cooperman. System-level Scalable Checkpoint-Restart for Petascale Computing. In *ICPADS*, pages 932–941, 2016.

[23] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *FAST*, pages 209–223, 2020.

[24] Don Capps and William Norcott. IOzone Filesystem Benchmark. `http://www.iozone.org/` (accessed Dec. 2020), 2003.

[25] Daniel Castro, Paolo Romano, and Joao Barreto. Hardware Transactional Memory meets Memory Persistency. *Journal of Parallel and Distributed Computing*, 130:63–79, 2019.

[26] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *OOPSLA*, pages 433–452, 2014.

[27] David A Chappell and Tyler Jewell. *Java Web Services*. Tecniche Nuove, 2002.

[28] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D Viglas. Rewind: Recovery Write-Ahead System for In-Memory Non-Volatile Data-Structures. *Proceedings of the VLDB Endowment*, 8(5):497–508, 2015.

[29] Yanpei Chen, Kiran Srinivasan, Garth Goodson, and Randy Katz. Design Implications for Enterprise Storage systems via Multi-Dimensional Trace Analysis. In *SOSP*, pages 43–56, 2011.

[30] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *ASPLOS*, pages 1077–1091, 2020.

[31] Austin T Clements, M Frans Kaashoek, Nickolai Zeldovich, Robert T Morris, and Eddie Kohler. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. In *SOSP*, pages 1–17, 2013.

[32] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *ASPLOS*, pages 105–118, 2011.

[33] 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 *SOSP*, pages 133–146, 2009.

[34] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*, pages 143–154, 2010.

[35] Transaction Processing Performance Council. TPC-H Benchmark Specification. *Published at http://www.tcp.org/hspec.html*, 2008.

[36] Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. Log-Free Concurrent Data Structures. In *USENIX ATC*, pages 373–386, 2018.

[37] DDN. Infinite Memory Engine. `https://www.ddn.com/products/ime-flash-native-data-cache/` (accessed Dec. 2020), 2019.

[38] Veera Deenadhayalan. GPFS Native RAID for 100,000-Disk Petascale Systems. In *LISA*, 2011.

[39] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and Simulation of a Fair Queueing Algorithm. *ACM SIGCOMM Computer Communication Review*, 19(4):1–12, 1989.

[40] Peter Desnoyers. Analytic Modeling of SSD Write Performance. In *Proceedings of the International Systems and Storage Conference (SYSTOR)*, page 12, 2012.

[41] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. Performance and Protection in the ZoFS User-Space NVM File System. In *SOSP*, pages 478–493, 2019.

[42] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System Software for Persistent Memory. In *EuroSys*, page 15, 2014.

[43] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing DRAM Footprint with NVM in Facebook. In *EuroSys*, page 42, 2018.

[44] ExMatEx. CoMD: Classical Molecular Dynamics Proxy Application. `https://github.com/ECP-copa/CoMD` (accessed Dec. 2020), 2016.

[45] Facebook. RocksDB. `https://rocksdb.org/` (accessed Dec. 2020).

[46] Kurt B Ferreira, Rolf Riesen, Ron Brighwell, Patrick Bridges, and Dorian Arnold. libhashckpt: Hash-Based Incremental Checkpointing Using GPU's. In *EuroMPI*, pages 272–281, 2011.

[47] Keir Fraser. *Practical Lock-Freedom*. PhD thesis, University of Cambridge, 2004. UCAM-CL-TR-579.

[48] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. A Persistent Lock-Free Queue for Non-Volatile Memory. In *PPoPP*, pages 28–40, 2018.

[49] Kaan Genç, Michael D Bond, and Guoqing Harry Xu. Crafty: Efficient, HTM-Compatible Persistent Transactions. In *POPL*, pages 59–74, 2020.

[50] Yonatan Gottesman, Joel Nider, Ronen Kat, Yaron Weinsberg, and Michael Factor. Using Storage Class Memory Efficiently for an In-Memory Database. In *SYSTOR*, page 21, 2016.

[51] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Elsevier, 1992.

[52] Laura M Grupp, John D Davis, and Steven Swanson. The Bleak Future of NAND Flash Memory. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, pages 2–2. USENIX Association, 2012.

[53] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. Understanding the Idiosyncrasies of Real Persistent Memory. *Proceedings of the VLDB Endowment*, 14(4):626–639, 2021.

[54] Shashank Gugnani, Tianxi Li, and Xiaoyi Lu. NVMe-CR: A Scalable Ephemeral Storage Runtime for Checkpoint/Restart with NVMe-over-Fabrics. In *IPDPS*, 2021.

[55] Shashank Gugnani, Xiaoyi Lu, and Dhabaleswar K Panda. Analyzing, Modeling, and Provisioning QoS for NVMe SSDs. In *International Conference on Utility and Cloud Computing (UCC)*, pages 247–256, 2018.

[56] Shashank Gugnani, Xiaoyi Lu, Houliang Qi, Li Zha, and Dhabaleswar K Panda. Characterizing and Accelerating Indexing Techniques on Distributed Ordered Tables. In *International Conference on Big Data (Big Data)*, pages 173–182, 2017.

[57] Ajay Gulati, Irfan Ahmad, and Carl A. Waldspurger. PARDA: Proportional Allocation of Resources for Distributed Storage Access. In *FAST*, pages 85–98, 2009.

[58] Ajay Gulati, Arif Merchant, and Peter J. Varman. mClock: Handling Throughput Variability for Hypervisor IO Scheduling. In *OSDI*, pages 437–450, 2010.

[59] Sijie Guo, Robin Dhamankar, and Leigh Stewart. DistributedLog: A High Performance Replicated Log Service. In *ICDE*, pages 1183–1194, 2017.

[60] Zvika Guz, Harry Huan Li, Anahita Shayesteh, and Vijay Balakrishnan. NVMe-over-Fabrics Performance Characterization and the Path to Low-Overhead Flash Disaggregation. In *SYSTOR*, page 16, 2017.

[61] Frank T Hady, Annie Foong, Bryan Veal, and Dan Williams. Platform Storage Performance with 3D XPoint Technology. *Proceedings of the IEEE*, 105(9):1822–1833, 2017.

[62] Paul H Hargrove and Jason C Duell. Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters. *Journal of Physics: Conference Series*, 46(1):494–499, 2006.

[63] Swapnil Haria, Mark D Hill, and Michael M Swift. MOD: Minimally Ordered Durable Data Structures for Persistent Memory. In *ASPLOS*, pages 775–788, 2020.

[64] Timothy L Harris. A Pragmatic Implementation of Non-Blocking Linked-Lists. In *International Symposium on Distributed Computing*, pages 300–314, 2001.

[65] Red Hat. GlusterFS. `https://www.gluster.org/` (accessed Dec. 2020), 2019.

[66] Pat Helland, Harald Sammer, Jim Lyon, Richard Carr, Phil Garrett, and Andreas Reuter. Group Commit Timers and High Volume Transaction Systems. In *International Workshop on High Performance Transaction Systems*, pages 301–329, 1987.

[67] Dave Henseler, Benjamin Landsteiner, Doug Petesch, Cornell Wright, and Nicholas J Wright. Architecture and Design of Cray DataWarp. *Cray User Group*, 2016.

[68] Maurice Herlihy and Nir Shavit. The Art of Multiprocessor Programming, 2008.

[69] Haixin Huang, Kaixin Huang, Litong You, and Linpeng Huang. Forca: Fast and Atomic Remote Direct Access to Persistent Memory. In *ICCD*, pages 246–249, 2018.

[70] Jian Huang, Karsten Schwan, and Moinuddin K Qureshi. NVRAM-aware Logging in Transaction Systems. *Proceedings of the VLDB Endowment*, 8(4):389–400, 2014.

[71] Yihe Huang, Matej Pavlovic, Virendra Marathe, Margo Seltzer, Tim Harris, and Steve Byan. Closing the Performance Gap Between Volatile and Persistent Key-Value Stores Using Cross-Referencing Logs. In *USENIX ATC*, pages 967–979, 2018.

[72] IBM. BSCFS. `https://github.com/IBM/CAST` (accessed Dec. 2020), 2018.

[73] Dewan Ibtesham, Kurt B Ferreira, and Dorian Arnold. A Checkpoint Compression Study for High-Performance Computing Systems. *The International Journal of High Performance Computing Applications*, 29(4):387–402, 2015.

[74] Intel. Intel VTune Profiler. `https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html` (accessed Dec. 2020), 2014.

[75] Intel. PMDK. `https://github.com/pmem/pmdk` (accessed Dec. 2020), 2014.

[76] Intel. SPDK. `https://github.com/spdk/spdk` (accessed Dec. 2020), 2015.

[77] Intel. Enhanced Power-Loss Data Protection in the Intel Solid-State Drive 320 Series, November 2016. Available at: `https://newsroom.intel.com/wp-content/uploads/sites/11/2016/01/Intel_SSD_320_Series_Enhance_Power_Loss_Technology_Brief.pdf` (accessed Dec. 2020).

[78] Intel. Persistent Memory Storage Engine for MongoDB. https://github.com/pmem/pmse (accessed Dec. 2020), 2016.

[79] Intel. Processor Counter Monitor. https://github.com/opcm/pcm (accessed Dec. 2020), 2017.

[80] Intel. libpmemobj-cpp. https://github.com/pmem/libpmemobj-cpp (accessed Dec. 2020), 2018.

[81] Intel. 300 nanoseconds. https://pmem.io/2019/12/19/performance.html (accessed Dec. 2020), 2019.

[82] Intel. Understanding Remote Persistent Memory. https://software.intel.com/content/www/us/en/develop/articles/persistent-memory-replication-over-traditional-rdma-part-1-understanding-remote-persistent.html (accessed Dec. 2020), 2019.

[83] Nusrat Sharmin Islam, Md Wasi-ur Rahman, Xiaoyi Lu, and Dhabaleswar K Panda. High Performance Design for HDFS with Byte-Addressability of NVM and RDMA. In *ICS*, page 8, 2016.

[84] Jake Brutlag. Speed Matters. https://ai.googleblog.com/2009/06/speed-matters.html (accessed Dec. 2020), 2009.

[85] Kanchan Joshi, Kaushal Yadav, and Praval Choudhary. Enabling NVMe WRR support in Linux Block Layer. In *HotStorage*, 2017.

[86] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *SOSP*, pages 494–508, 2019.

[87] Anil Kashyap. Workload Characterization for Enterprise Disk Drives. *ACM Transactions on Storage (TOS)*, 14(2):1–15, 2018.

[88] Manolis Katevenis, Stefanos Sidiropoulos, and Costas Courcoubetis. Weighted Round-Robin Cell Multiplexing in a General-Purpose ATM Switch Chip. *IEEE Journal on selected Areas in Communications*, 9(8):1265–1279, 1991.

[89] Swaroop Kavalanekar, Bruce Worthington, Qi Zhang, and Vishal Sharda. Characterization of Storage Workload Traces from Production Windows Servers. In *IISWC*, pages 119–128, 2008.

[90] Ram Kesavan, Rohit Singh, Travis Grusecki, and Yuvraj Patel. Algorithms and Data Structures for Efficient Free Space Reclamation in WAFL. In *FAST*, pages 1–14, 2017.

[91] Byungseok Kim, Jaeho Kim, and Sam H Noh. Managing Array of SSDs When the Storage Device is No Longer the Performance Bottleneck. In *HotStorage*, 2017.

[92] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. NVMeDirect: A User-space I/O Framework for Application-specific Optimization on NVMe SSDs. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*. USENIX Association, 2016.

[93] Hyojun Kim, Sangeetha Seshadri, Clement L Dickey, and Lawrence Chiu. Evaluating Phase Change Memory for Enterprise Storage Systems: A Study of Caching and Tiering Approaches. In *FAST*, pages 33–45, 2014.

[94] Jungwon Kim, Seyong Lee, and Jeffrey S Vetter. PapyrusKV: A High-Performance Parallel Key-Value Store for Distributed NVM Architectures. In *SC*, pages 1–14, 2017.

[95] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. NVWAL: Exploiting NVRAM in Write-Ahead Logging. In *ASPLOS*, pages 385–398, 2016.

[96] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash Storage Disaggregation. In *EuroSys*, pages 29:1–29:15, 2016.

[97] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. ReFlex: Remote Flash $\approx$ Local Flash. In *ASPLOS*, pages 345–359, 2017.

[98] Anthony Kougkas, Hariharan Devarajan, and Xian-He Sun. Hermes: A Heterogeneous-aware Multi-tiered Distributed I/O Buffering System. In *HPDC*, pages 219–230, 2018.

[99] Alexander Krizhanovsky. Lock-Free Multi-Producer Multi-Consumer Queue on Ring Buffer. *Linux Journal*, 2013(228):4, 2013.

[100] John Lamping and Eric Veach. A Fast, Minimal Memory, Consistent Hash Algorithm. *arXiv preprint arXiv:1406.2294*, 2014.

[101] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting Phase Change Memory as a Scalable DRAM Alternative. In *ISCA*, pages 2–13, 2009.

[102] Benjamin C Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. Phase-Change Technology and the Future of Main Memory. *IEEE Micro*, 30(1):143, 2010.

[103] Lucas Lersch, Ivan Schreter, Ismail Oukid, and Wolfgang Lehner. Enabling Low Tail Latency on Multicore Key-Value Stores. *Proceedings of the VLDB Endowment*, 13(7):1091–1104, 2020.

[104] Min Li, Sudharshan S Vazhkudai, Ali R Butt, Fei Meng, Xiaosong Ma, Youngjae Kim, Christian Engelmann, and Galen Shipman. Functional Partitioning to Optimize End-to-End Performance on Many-core Architectures. In *SC*, pages 1–12, 2010.

[105] Tianxi Li, Dipti Shankar, Shashank Gugnani, and Xiaoyi Lu. RDMP-KV: Designing Remote Direct Memory Persistence based Key-Value Stores with PMEM. In *SC*, pages 722–735, 2020.

[106] Linux. libaio. `https://docs.oracle.com/cd/E36784_01/html/E36873/libaio-3lib.html` (accessed Dec. 2020), 2014.

[107] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. In *ASPLOS*, pages 329–343, 2017.

[108] LLNL. UnifyCR. `https://github.com/LLNL/UnifyFS` (accessed Dec. 2020), 2018.

[109] David Lomet and Mark Tuttle. Logical Logging to Extend Recovery to New Domains. In *SIGMOD*, pages 73–84, 1999.

[110] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. Dash: Scalable Hashing on Persistent Memory. *Proceedings of the VLDB Endowment*, 13(8):1147–1161, 2020.

[111] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: An RDMA-enabled Distributed Persistent Memory File System. In *USENIX ATC*, pages 773–785, 2017.

[112] Teng Ma, Mingxing Zhang, Kang Chen, Zhuo Song, Yongwei Wu, and Xuehai Qian. AsymNVM: An Efficient Framework for Implementing Persistent Data Structures on Asymmetric NVM Architecture. In *ASPLOS*, pages 757–773, 2020.

[113] Nirmesh Malviya, Ariel Weisberg, Samuel Madden, and Michael Stonebraker. Rethinking Main Memory OLTP Recovery. In *ICDE*, pages 604–615, 2014.

[114] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache Craftiness for Fast Multicore Key-Value Storage. In *EuroSys*, pages 183–196, 2012.

[115] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 Filesystem: Current Status and Future Plans. In *Proceedings of the Linux Symposium*, volume 2, pages 21–33, 2007.

[116] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. Pronto: Easy and Fast Persistence for Volatile Data Structures. In *ASPLOS*, pages 789–806, 2020.

[117] Arif Merchant, Mustafa Uysal, Pradeep Padala, Xiaoyun Zhu, Sharad Singhal, and Kang Shin. Maestro: Quality-of-Service in Large Disk Arrays. In *ICAC*, pages 245–254, 2011.

[118] Rick Merritt. 3D XPoint Steps Into the Light. `http://www.eetimes.com/document.asp?doc_id=1328682` (accessed Dec. 2020), 2016.

[119] Vilobh Meshram, Xavier Besseron, Xiangyong Ouyang, Raghunath Rajachandrasekar, Ravi Prakash Darbha, and Dhabaleswar K Panda. Can a Decentralized

Metadata Service Layer Benefit Parallel Filesystems? In *Cluster*, pages 484–493, 2011.

[120] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. Understanding Manycore Scalability of File Systems. In *USENIX ATC*, pages 71–85, 2016.

[121] Sparsh Mittal and Jeffrey S Vetter. A Survey of Software Techniques for Using Non-Volatile Memories for Storage and Main Memory Systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1537–1550, 2016.

[122] C. Mohan, Donald J. Haderle, Bruce G. Lindsay, Hamid Pirahesh, and Peter M. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems (TODS)*, 17:94–162, 1992.

[123] MongoDB Inc. MongoDB. `https://www.mongodb.com/` (accessed Dec. 2020).

[124] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R De Supinski. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In *SC*, pages 1–11, 2010.

[125] Sanketh Nalli, Swapnil Haria, Mark D Hill, Michael M Swift, Haris Volos, and Kimberly Keeton. An Analysis of Persistent Memory use with WHISPER. In *ASPLOS*, pages 135–148, 2017.

[126] Dushyanth Narayanan, Eno Thereska, Austin Donnelly, Sameh Elnikety, and Antony Rowstron. Migrating Server Storage to SSDs: Analysis of Tradeoffs. In *EuroSys*, pages 145–158, 2009.

[127] OSU NBCL. High-Performance Big Data. http://hibd.cse.ohio-state.edu.

[128] NERSC. Perlmutter Supercomputer. https://www.nersc.gov/systems/perlmutter/ (accessed Dec. 2020), 2019.

[129] Paul Nowoczynski, Nathan Stone, Jared Yanovich, and Jason Sommerfield. Zest Checkpoint Storage System for Large Supercomputers. In *Petascale Data Storage Workshop*, pages 1–5. IEEE, 2008.

[130] NVMe Express. NVMe over Fabrics. http://www.nvmexpress.org/wp-content/uploads/NVMe_Over_Fabrics.pdf (accessed Dec. 2020), 2016.

[131] NVMe Express Inc. NVM Express. http://nvmexpress.org/ (accessed Dec. 2020), 2017.

[132] Gihwan Oh, Sangchul Kim, Sang-Won Lee, and Bongki Moon. SQLite Optimization with Phase Change Memory for Mobile Applications. *Proceedings of the VLDB Endowment*, 8(12):1454–1465, 2015.

[133] Adam J Oliner, Larry Rudolph, and Ramendra K Sahoo. Cooperative Checkpointing: A Robust Approach to Large-scale Systems Reliability. In *ICS*, pages 14–23, 2006.

[134] OpenChannelSSD. QEMU for Open-Channel SSDs. https://github.com/OpenChannelSSD/qemu-nvme (accessed Dec. 2020), 2017.

[135] OpenStack. Swift. http://swift.openstack.org/ (accessed Dec. 2020).

[136] Tim O'Reilly. What is Web 2.0: Design Patterns and Business Models for the Next Generation of Software. *Communications & strategies*, (1):17, 2007.

[137] ORNL. U.S. Department of Energy and Cray to Deliver Record-Setting Frontier Supercomputer at ORNL. https://www.ornl.gov/news/us-department-energy-and-cray-deliver-record-setting-frontier-supercomputer-ornl (accessed Dec. 2020), 2019.

[138] Xiangyong Ouyang, Raghunath Rajachandrasekar, Xavier Besseron, Hao Wang, Jian Huang, and Dhabaleswar K Panda. CRFS: A Lightweight User-Level Filesystem for Generic Checkpoint/Restart. In *ICPP*, pages 375–384, 2011.

[139] Jong-Hyeok Park, Gihwan Oh, and Sang-Won Lee. SQL Statement Logging for Making SQLite Truly Lite. *Proceedings of the VLDB Endowment*, 11(4):513–525, 2017.

[140] Stan Park and Kai Shen. FIOS: A Fair, Efficient Flash I/O Scheduler. In *FAST*, page 13, 2012.

[141] Peifeng Si. Persistent Memory Storage Engine for RocksDB. https://github.com/pmem/pmem-rocksdb (accessed Dec. 2020), 2019.

[142] Slawomir Pilarski and Tiko Kameda. Checkpointing for Distributed Databases: Starting from the Basics. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):602–610, 1992.

[143] QEMU. QEMU: the FAST! Processor Emulator. https://www.qemu.org/ (accessed Dec. 2020), 2017.

[144] Raghunath Rajachandrasekar, Adam Moody, Kathryn Mohror, and Dhabaleswar K Panda. A 1 PB/s File System to Checkpoint Three Million MPI Tasks. In *HPDC*, pages 143–154, 2013.

[145] Donald Sannella and Andrzej Tarlecki. On Observational Equivalence and Algebraic Specification. In *Colloquium on Trees in Algebra and Programming*, pages 308–322. Springer, 1985.

[146] Mohit Saxena, Mehul A Shah, Stavros Harizopoulos, Michael M Swift, and Arif Merchant. Hathi: Durable Transactions for Memory using Flash. In *DaMoN*, pages 33–38, 2012.

[147] Frank B Schmuck and Roger L Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *FAST*, pages 231–244, 2002.

[148] David Schwalb, Markus Dreseler, Matthias Uflacker, and Hasso Plattner. NVC-Hashmap: A Persistent and Concurrent Hashmap for Non-Volatile Memories. In *Proceedings of the 3rd VLDB Workshop on In-Memory Data Mangement and Analytics*, pages 1–8, 2015.

[149] Philip Schwan. Lustre: Building a File System for 1000-node Clusters. In *Proceedings of the Linux Symposium*, pages 380–386, 2003.

[150] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed Shared Persistent Memory. In *SoCC*, pages 323–337, 2017.

[151] Kai Shen and Stan Park. FlashFQ: A Fair Queueing I/O Scheduler for Flash-based SSDs. In *USENIX ATC*, pages 67–78, 2013.

[152] Madhavapeddi Shreedhar and George Varghese. Efficient Fair Queuing using Deficit Round-Robin. *IEEE/ACM Transactions on Networking*, 4(3):375–385, 1996.

[153] David Shue, Michael J. Freedman, and Anees Shaikh. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *OSDI*, pages 349–362, 2012.

[154] Slurm. Slurm Generic Resources. `https://slurm.schedmd.com/gres.html` (accessed Dec. 2020), 2019.

[155] Xiang Song, Jian Yang, and Haibo Chen. Architecting Flash-based Solid-State Drive for High-performance I/O Virtualization. *IEEE Computer Architecture Letters*, 13(2):61–64, 2014.

[156] Steve Stargall. *Programming Persistent Memory: A Comprehensive Guide for Developers*. 2019.

[157] Guy L Steele Jr. Making Asynchronous Parallelism Safe for the World. In *POPL*, pages 218–231, 1989.

[158] Dmitri B Strukov, Gregory S Snider, Duncan R Stewart, and R Stanley Williams. The Missing Memristor Found. *Nature*, 453(7191):80, 2008.

[159] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Ana Klimovic, Adrian Schuepbach, and Bernard Metzler. Unification of Temporary Storage in the NodeKernel Architecture. In *USENIX ATC*, pages 767–782, 2019.

[160] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection. In *NSDI*, pages 513–527, 2015.

[161] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS File System. In *USENIX ATC*, pages 1–14, 1996.

[162] Yacine Taleb, Ryan Stutsman, Gabriel Antoniu, and Toni Cortes. Tailwind: Fast and Atomic RDMA-based Replication. In *USENIX ATC*, pages 851–863, 2018.

[163] The OrangeFS Project. OrangeFS. `http://www.orangefs.org/` (accessed Dec. 2020), 2019.

[164] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores. In *USENIX ATC*, pages 33–48, 2020.

[165] AA Tulapurkar, Y Suzuki, A Fukushima, H Kubota, H Maehara, K Tsunekawa, DD Djayaprawira, N Watanabe, and S Yuasa. Spin-Torque Diode Effect in Magnetic Tunnel Junctions. *Nature*, 438(7066):339, 2005.

[166] Stephen C Tweedie. Journaling the Linux ext2fs Filesystem. In *The Fourth Annual Linux Expo*, 1998.

[167] Sudharshan S Vazhkudai, Bronis R de Supinski, Arthur S Bland, Al Geist, James Sexton, Jim Kahle, Christopher J Zimmer, Scott Atchley, Sarp Oral, Don E Maxwell, et al. The Design, Deployment, and Evaluation of the CORAL Pre-Exascale Systems. In *SC*, page 52, 2018.

[168] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H Campbell. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *FAST*, volume 11, pages 61–75, 2011.

[169] Haris Volos, Andres Jaan Tack, and Michael M Swift. Mnemosyne: Lightweight Persistent Memory. In *ASPLOS*, pages 91–104, 2011.

[170] Guozhang Wang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, Mammad Zadeh, Neha Narkhede, Jun Rao, Jay Kreps, and Joe Stein. Building a Replicated Logging System with Apache Kafka. *Proceedings of the VLDB Endowment*, 8(12):1654–1655, 2015.

[171] Teng Wang, Kathryn Mohror, Adam Moody, Kento Sato, and Weikuan Yu. An Ephemeral Burst-Buffer File System for Scientific Applications. In *SC*, page 69, 2016.

[172] Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. Query Fresh: Log Shipping on Steroids. *Proceedings of the VLDB Endowment*, 11(4):406–419, 2017.

[173] Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. Easy Lock-Free Indexing in Non-Volatile Memory. In *ICDE*, pages 461–472, 2018.

[174] Matt Welsh, David Culler, and Eric Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. *ACM SIGOPS Operating Systems Review*, 35(5):230–243, 2001.

[175] Xiaojian Wu and AL Reddy. SCMFS: A File System for Storage Class Memory. In *SC*, page 39, 2011.

[176] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *USENIX ATC*, pages 349–362, 2017.

[177] Xilinx. SmartSSD. `https://www.xilinx.com/applications/data-center/computational-storage/smartssd.html` (accessed Dec. 2020).

[178] Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson. Finding and Fixing Performance Pathologies in Persistent Memory Software Stacks. In *ASPLOS*, pages 427–439, 2019.

[179] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *FAST*, pages 323–338, 2016.

[180] 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 *SOSP*, pages 478–496, 2017.

[181] Jian Yang, Joseph Izraelevitz, and Steven Swanson. Orion: A Distributed File System for Non-Volatile Main Memory and RDMA-Capable Networks. In *FAST*, pages 221–234, 2019.

[182] Jian Yang, Joseph Izraelevitz, and Steven Swanson. FileMR: Rethinking RDMA Networking for Scalable Persistent Memory. In *NSDI*, pages 111–125, 2020.

[183] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *FAST*, pages 169–182, 2020.

[184] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *FAST*, pages 167–181, 2015.

[185] Zhengyu Yang, Manu Awasthi, Mrinmoy Ghosh, and Ningfang Mi. A Fresh Perspective on Total Cost of Ownership Models for Flash Storage in Datacenters. In *International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 245–252, 2016.

[186] Ziye Yang, Luse E Paul, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, and Vishal Verma. SPDK: A Development Kit to Build High Performance Storage Applications. In *International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 154–161, 2017.

[187] Erez Zadok, Dean Hildebrand, Geoff Kuenning, and Keith A Smith. POSIX is Dead! Long Live... errr... What Exactly? In *HotStorage*, pages 12–12, 2017.

[188] Jianyong Zhang, Anand Sivasubramaniam, Qian Wang, Alma Riska, and Erik Riedel. Storage Performance Virtualization via Throughput and Latency Control. *ACM Transactions on Storage (TOS)*, 2(3):283–308, 2006.

[189] Lu Zhang and Steven Swanson. Pangolin: A Fault-Tolerant Persistent Memory Programming Library. In *USENIX ATC*, pages 897–912, 2019.

[190] Wen Zhang, Scott Shenker, and Irene Zhang. Persistent State Machines for Recoverable In-memory Storage Systems with NVRam. In *OSDI*, pages 1029–1046, 2020.

[191] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. Mojim: A Reliable and Highly-Available Non-Volatile Memory System. In *ASPLOS*, pages 3–18, 2015.

[192] Qing Zheng, Kai Ren, and Garth Gibson. BatchFS: Scaling the File System Control Plane with Client-Funded Metadata Servers. In *9th Parallel Data Storage Workshop*, pages 1–6, 2014.

[193] Qing Zheng, Kai Ren, Garth Gibson, Bradley W Settlemyer, and Gary Grider. DeltaFS: Exascale File Systems Scale Better Without Dedicated Servers. In *Proceedings of the 10th Parallel Data Storage Workshop*, pages 1–6. ACM, 2015.

[194] Yongqiang Zou, Jia Liu, Shicai Wang, Li Zha, and Zhiwei Xu. CCIndex: A Complemental Clustering Index on Distributed Ordered Tables for Multi-dimensional Range Queries. In *IFIP International Conference on Network and Parallel Computing*, pages 247–261. Springer, 2010.

[195] Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. Efficient Lock-Free Durable Sets. In *OOPSLA*, pages 1–26, 2019.

Acronyms

DIMM Dual In-Line Memory Module.

HPC High Performance Computing.

IO Input/Output.

NVMe Non-Volatile Memory Express.

PCIe Peripheral Component Interconnect Express.

PMEM Persistent Memory.

QoS Quality of Service.

RDMA Remote Direct Memory Access.

SLA Service Level Agreement.

SLO Service Level Objective.

SSD Solid State Drive.

TCO Total Cost of Ownership.