The Dissertation Committee for Soujanya Ponnapalli
certifies that this is the approved version of the following dissertation:

Minimizing I/O Bottlenecks to Achieve Scalable and High-Throughput Systems

Committee:

Vijay Chidambaram, Supervisor

Emmett Witchel

James Bornholt

Jonathan Goldstein

Natacha Crooks

Minimizing I/O Bottlenecks to Achieve Scalable and High-Throughput Systems

by

Soujanya Ponnapalli

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2023

Dedication

To my loving parents and every guru in my life.

To Amma and Daddy, my guiding light, for being the greatest inspirations of my life, and to my teachers for their instrumental role in shaping my mind.

Acknowledgments

I sincerely thank everyone who has inspired me to choose this path, supported me along the way, made my academic pursuit enjoyable, and wished me the best.

I am forever indebted to my advisor, Vijay Chidambaram, for his incredible mentorship. Vijay taught me crucial skills like conducting, writing, and presenting research. Whenever I lost sight of the big picture, Vijay course-corrected me and helped me refocus. During challenging times, his confidence in me transformed into my strength to persevere. I deeply admire his ability to strike the perfect balance between giving me freedom, nurturing my productivity, offering critical feedback, and boosting my confidence as a researcher. I am grateful for his strong sense of responsibility and dedication to his students. It is remarkable how he could tolerate my nocturnal work shifts and my musically challenged yet undoubtedly the loudest voice in the hallway. I am fortunate to have been his plant guardian, house sitter, and, most importantly, his PhD student, for he is an exceptional advisor.

I am grateful to my Ph.D. committee — Emmett Witchel, Jonathan Goldstein, Natacha Crooks, and James Bornholt, including Isil Dillig — for their investment in my academic growth. Emmett's timeless advice helped me overcome one of my biggest struggles with writing and has inspired clarity in my thinking. Jonathan's generous collaboration opportunity has significantly improved my academic journey and fostered an appreciation for simple yet impactful ideas. Natacha's flexibility and will to prioritize my best interests have become a source of comfort during tough times. She has always encouraged me to work smart, be kind, and to give back. I am thrilled to begin the next phase of my research career by collaborating with and learning from her. James has been consistent in helping me navigate the defense process smoothly. Isil has served on my committee during previous graduate school milestones and has inspired me to aim higher. I am genuinely thankful for such a supportive committee, and I hope to pay their kindness forward.

I am grateful for my mentors', who are exceptional researchers with endless passion, and for every opportunity to learn from them. My first project, RainBlock, was synthesized at VMware Research Group (VRG), where I worked with Michael Wei and Dahlia Malkhi. I was frequently amazed by Michael and Dahlia's thorough understanding and objective perception of blockchain systems. The RainBlock team taught me fundamental skills like asking the right questions, explaining ideas simply but clearly, maintaining code for large-scale projects, designing the API before the system, etc., which proved invaluable in the later stages. In subsequent summers, I interned with Microsoft Research in Cambridge, UK, and Redmond, WA. Working with Dushyanth Narayanan, Ant Rowstron, and the Project HSD team, to design a holographic storage device for the cloud has revealed the beauty of cross-disciplined researchers working toward a shared goal. Collaborating with Anirudh Badam and Ranveer Chandra on harvesting the under-utilized resources in the cloud, like storage and memory for caching, has broadened my perspective on the potential of good research and its impact on the environment and economy. Working with Jonathan Goldstein and Phil Bernstein on Project Cascades to design IO-efficient databases that can afford simple recovery in data centers has allowed me to appreciate their ability to deconstruct seminal work into a few key insights and to discuss them in unique contexts. This experience unveiled the importance of simple abstractions to support and scale complex applications. I am forever grateful for these learning experiences; I hope for many fruitful collaborations with them even in the future.

I'm grateful to my collaborators for allowing me to lean on their expertise. The CrashMonkey team – Jayashree Mohan, Ashlie Martinez, and Pandian Raju – for introducing me to the fundamentals of systems building, starting with tmux. The RainBlock and mLSM teams – Dahlia Malkhi, Michael Wei, Amy Tai, Aashaka Shah, Souvik Banerjee, Ittai Abraham, Pandian Raju, Gilad Oved, and Zachary Keener – for guiding me to understand blockchain systems, authenticated data structures, distributed scripting and benchmarking. The Skye team – Sekwon Lee and Rohan Kadekodi – for answering my queries on Persistent Memory and CXL since

the project's inception. The Cascades team – Jonathan Goldstein, Phil Bernstein and Jose Falerio – for comprehensive discussions on ACID transactions and the bottlenecks in distributed databases, and Anuj Kalia for generously helping me set up eRPC. I appreciate the opportunity to work with Kimberly Keeton, Marcos Aguilera, and Sharad Singhal on project Dinomo, alongside Aasheesh Kolli, Greg Ganger, and Saurabh Kadekodi on project WineFS, and Zsolt Istvan on data protection project. Each of these collaborations has been a remarkable learning experience.

I'm thankful for my interactions with outstanding and well-rounded people during the most impressionable years of my life. Vijay, Emmett, Simon Peter, Chris Rossbach, Isil Dillig, Alison Norman, Swarat Chaudhuri, and my professors at UT, Suresh Purini, Vivek Vellanki, Kannan Srinathan, Kishore Kothapalli and my professors at IIIT-Hyderabad, the UTCS administrative department with Katie Traughber, Gabrielle Bouzigard, and Violet Cantu for their support in meeting various deadlines, Irene Zhang, Miguel Castro, Muthian Sivathanu and Sriram Rajamani, for extending internship opportunities, Mahesh Balakrishnan, Remzi Arpaci-Dusseau, Jay Lorch, Lalith Suresh and others, for making networking at conferences fun and invaluable, and my teachers at St. Gabriel's High School including Nageswara Rao, Madhusudhan Reddy and D. K. Jha for always encouraging a curious mind.

I'm thankful for the lasting friendships that have formed during internships. I will cherish the impromptu plank sessions, soccer and foosball breaks, radio talks *etc.* with the VRG interns. Special thanks to Arjun Singhvi, a fellow night owl who's always only a phone call away, Alex Conway and Reto Acchermann, for the hiking trips, Yizou Shan, Sanidhya Kashyap and Alin Tomescu, for our conversations spanning research and beyond, and Yanfang Lee, my caring flatmate that summer. I will also treasure playing badminton, ping pong and board games, and conquering moderate hikes in the UK with MSR interns. Special thanks to Sheena Panthaplackel, my flatmate across continents who had kept up with my drifting sleep schedule, and to Arjun Kashyap and Praneeth Chakravarthula for matching my competitiveness.

Special thanks to Lori Blonn and Deepak Vashist for making internships more interesting.

I'm blessed to have lab mates who remind me of the depths to research and seamlessly transition into loving friends outside work. Thanks to my brilliant peers, I understand the importance of building quickly and prioritizing tasks with the big picture in mind. Karaoke evenings, late nights with ice cream or frozen yogurt, walks in Hancock golf course and Domain, lunch and coffee chats, *etc.* are some of my best memories in grad school. Rohan, for our endless conversations on the ideal color scheme and the freedom of artistic expression, Aashaka, for truly understanding me to the extent of gamifying chores for me, Sekwon, for introducing me to karaoke and sharing my journey from passionate reviewers to eventually understanding writing preferences, Jayashree, for figuring coursework and serving us home-cooked meals, Hayley, for providing feedback on drafts (and their versions) and for the relentless head nods during my presentations, Ashlie, for meticulously reviewing my pull requests and for teaching me how to code per standards, Yeonju, for your effort in getting me to work from the lab post COVID, Supreeth, for opting for a cubicle so that we could all hang out together, Devvrit and Kevin, for being almost-cubies and bringing a sense of community to us all, Amanda, for sharing the grievances of fellow PM researchers, and Pandian, Chandana, Dhatri, Nayan and Mit, for being the seniors that have warmly welcomed me into their circle, and to the LASR lab, for fun lunches on Fridays – thanks for a lot for these memories and more.

I'm thankful to my extended family and friends for demonstrating their support in unique ways – my grandparents, aunts and uncles, cousins, the Ponnapallis' and the Koettings' and Pravallika Rao, Sanjana Reddy, Divya Jyothi, Satvik Chekuri, Arjun Sanjeev, Tinku Monish, Ravikiran Chanumolu, Sowmith Manepalli, Aditya Srinivas, Samyukta Mogily, Hari Narayana, Krishna Chaitanya, Sukumar Mokkarala, Meghana Manusanipalli, Deeraj Kaki, and to everyone who has helped me evolve.

I'm also immensely grateful to a version of myself for manifesting this journey

at a very young age; this experience has prepared me for life beyond research. I hope to sustain the determination, passion, confidence, and to keep that inner child within me alive for the next phase of my research career.

I am incredibly privileged to have a supportive family that simplifies pursuing dreams for me and agrees to my hard asks in a heartbeat. They encourage me to navigate life in my own way and reassure me with an environment that frees me from fearing failure. My elder brother, Chaitanya Sai Ponnapalli, instills the value of hard work in me and sets high standards for excellence. I am fortunate to have had him as my first flatmate in Austin and, in a full-circle moment, also as my last. He remains steadfast in motivating me to keep going. My sister-in-law, Mackenzie Ponnapalli, indulges me with delicious meals and is determined to ensure that I never face a dull moment. Amma and Daddy have closely witnessed my journey with pure conviction; they are my biggest cheerleaders. My journey is a testament to their warmth, love, patience, and support. Whenever I am conflicted, they find the time to serve as my sounding boards and provide me with advice that stands the test of time. My family inspires me to live up to my full potential, more so with humility and kindness. It is impossible to express my gratitude for them in words; I *am* because of my loving family, and this work is as much theirs as it is mine.

Abstract

Minimizing I/O Bottlenecks to Achieve Scalable and High-Throughput Systems

Soujanya Ponnapalli, PhD
The University of Texas at Austin, 2023

SUPERVISOR: Vijay Chidambaram

Modern applications store large volumes of data and access this data at high throughput. Fortunately, advanced hardware meets applications' demands by offering low-latency and high-bandwidth storage and datacenter networks. However, state-of-the-art systems infrastructures underutilize available hardware resources and fail to meet the throughput and scalability requirements of I/O-intensive applications.

This dissertation studies the performance limitations of three distinct systems: monolithic key-value stores, distributed transactional stores, and public blockchains. First, it attributes their low throughput and poor scalability to I/O-bottlenecks that are inherent to the systems' design and architecture. Next, it addresses the question: *How do we architect systems to minimize I/O-bottlenecks and simultaneously achieve high throughput and scalability?* It proposes a fundamental redesign of systems by carefully crafting the roles and responsibilities of each system component to improve the utilization of underlying resources.

This dissertation employs a few key ideas to achieve scalable, high-throughput systems. The first idea aims at customizing the storage subsystem to align with

the performance characteristics of underlying media and its applications' data layout. The second centers on co-designing data processing along with its storage with a clear demarcation of the roles of each system component. The third focuses on leveraging asynchrony and batching to reconstruct how and when I/O is performed for improving the utilization of available resources. This dissertation revisits these well-known ideas, however, from the perspective of minimizing I/O-bottlenecks to achieve high throughput and scalability.

This dissertation combines these ideas to architect three novel systems; Skye: a monolithic key-value store, Cascades: a distributed transactional store, and RainBlock: a distributed and decentralized database. Skye is tailored for Persistent Memory (PM), a novel storage-class memory technology that offers high throughput and low latency. Cascades is optimized for cloud servers and SSDs interconnected through high-speed datacenter networks. Finally, RainBlock is designed for commodity hardware and targets public environments with untrusted servers. This work presents the design and architecture of each of these systems, discusses their trade-offs, and evaluates their end-to-end performance and scalability.

Contents

List of Tables

List of Figures

Chapter 1: Introduction

Modern applications have become I/O-intensive *i.e.,* they store and process large volumes of data at high throughput [1]. These applications leverage the growing ease of data collection [162], the high availability of data stores [9, 10, 92, 187, 97], and explore new opportunities to efficiently process and analyze data at scale [197, 217, 100, 134]. Thus, I/O-intensive applications are increasing the demand for scalable, high-throughput systems infrastructure.

Fortunately, recent innovations in hardware offer low-latency, high-bandwidth storage devices [65, 82, 171] and network adapters [157], and are meeting the demands of modern I/O-intensive applications. For instance, Optane DC Persistent Memory (PM) [171] presents a new storage-class memory technology that durably stores data across power cycles at DRAM-comparable low latency and high bandwidth. Further, eRPC [119] introduces a new Ethernet-based general-purpose communication layer that achieves microsecond-scale round-trip time and supports gigabit transfers per second, like specialized networks with Infiniband and RDMA [125, 6]. Table-1.1 reviews prior research [117, 72], highlights the advancements over the past decade and summarizes today's hardware landscape.

	Until 2010			2020s		
	DRAM	Storage	Network	DRAM	Storage	Network
Latency (ns)	100	10000	300000	80	100	2000
Bandwidth (GB/s)	20	0.9	0.1	100	6	12

Table 1.1: **Network and storage advancements**. Network latency and bandwidth has improved by 120× and 25× respectively over the past decade. Storage latency and bandwidth has improved by 100× and 6× respectively with PM relative to SSDs. Today, network and storage have performance comparable to main memory.

However, in this dissertation, we discuss how existing systems infrastructures fail to effectively utilize available hardware resources and *do not* meet the throughput

and scalability requirements of modern I/O-intensive applications.

1.1 Systems for I/O-intensive applications

This dissertation analyzes three unique systems: key-value stores designed for PM, distributed transactional stores built for datacenter networks and SSDs, and distributed decentralized databases or public blockchains designed for commodity servers and networks. These systems represent distinct data points that are defined by their architectures and the underlying assumptions about their execution environments, as shown in Table-1.2. With the following three distinct systems, this dissertation seeks to emphasize the generality of its approach.

Architecture	Monolithic	Distributed	
Trust	Centralized	Centralized	Decentralized
System	PM key-value stores	Transactional stores	Public blockchains

Table 1.2: **Systems**. We consider three systems: Public blockchains, PM key-value stores, and distributed databases; they make unique design choices considering their architectures and assumptions on their execution environments.

PM key-value stores. PM key-value stores support simple put and get interfaces and are designed for monolithic servers with multiple non-uniform memory access (NUMA) nodes; each node hosts up to six non-volatile DIMMs (NVDIMMs). In this dissertation, we study state-of-the-art PM key-value stores [238, 122, 116, 185, 214, 48, 61] and highlight that they do not fully utilize the available I/O bandwidth of PM. Further, they suffer from poor performance which does not scale with the increasing number of application threads or the available PM capacity.

Distributed transactional stores. Distributed transactional stores and databases are designed for datacenter networks with replicated SSDs to support transactions from applications. The state-of-the-art distributed databases [191, 241, 206, 209,

33, 205, 192, 198, 87, 103] either employ expensive commit protocols like two-phase commit (2PC) [64, 129] to guarantee atomic, durable, and serializable transactions and thereby suffer from poor performance, or forgo synchronous 2PC and support weaker guarantees. With 2PC or faster alternatives [81], these distributed stores synchronously write to networked SSDs in the critical path of committing transactions. Thus, distributed stores incur the high I/O latency of networked SSDs [8, 5], suffer from low throughput and thereby underutilize the available network bandwidth in datacenters. Further, for highly contented workloads, distributed stores have limited performance scalability across multiple cores and servers [109, 234, 170].

Public blockchains. Public blockchains are distributed decentralized databases that are designed for commodity servers that support transactions from applications. Public blockchains [16, 13, 236, 144, 229] rely on a network of untrusted servers termed miners to order and execute transactions. Therefore, trust in public blockchains is implemented via authenticated data structures like Merkle trees [150, 19], that provide data along with proofs, and via consensus protocols that help replicate results across miners. In state-of-the-art public blockchains like Ethereum [16], miners store the system state in Merkle trees on their local disks. As a result, miners must perform slow I/O in the critical path of processing transactions and cannot fully utilize their compute capacity. In comparison to centralized distributed databases [241, 204, 192, 209] and payment systems like Visa [11], public blockchains suffer from orders of magnitude lower throughput and scalability.

1.2 Performance limitations from I/O bottlenecks

This dissertation highlights the low throughput and poor scalability of these three systems, traces their limitations to I/O bottlenecks that are fundamental to the systems' design and architecture, and details the nature of these I/O bottlenecks.

PM stores: Direct-access architectures for providing low latency. In this dissertation, we analyze the performance and scalability of state-of-the-art PM key-

value stores that are either retrofitted for PM [238] or designed from the ground-up for PM [61, 48]. We show that these PM stores utilize <45% of available PM bandwidth and their throughput drops on increasing the number of application threads beyond a certain threshold. These limitations result from the *direct-access architectures* of PM stores. Fundamentally, PM stores allow application threads to perform I/O directly on NVDIMMs to provide low latencies to applications, and no longer retain fine-grained control over all PM accesses. Thus, PM stores underutilize available PM bandwidth and suffer from poor performance and scalability. Further, prior studies [228, 107, 70] and our empirical study outline the nuanced performance characteristics of PM and show that PM stores incur I/O bottlenecks from adopting hardware-agnostic designs. Thus, we need a PM key-value store that effectively utilizes available PM bandwidth and can simultaneously achieve high throughput and scalability.

Distributed databases: Synchronous logging to disk for tolerating failures. In this dissertation, we review how state-of-the-art distributed databases support durable, serializable, and atomic, distributed transactions that span across servers. Distributed stores rely on two-phase commit (2PC) protocol [64] or its variants [154, 129] and log the progress of distributed transactions on networked disks [8, 5]. In the event of failures, they use these logs to recover all servers to a consistent state. Thus, distributed databases perform synchronous I/O to their recovery logs in the commit path of transactions to tolerate failures. Further, for workloads with high contention, processes hold locks on hot data for the entire duration till their writes to recovery logs are durable; this prevents other processes from executing transaction in parallel and limits scalability. We notice that prior solutions that tackle these I/O bottlenecks do not address the complexity of recovering from failures [74, 80]. Thus, we need distributed databases that avoid I/O in the critical path of transactions and can achieve high throughput and scalability without complicating recovery.

Public blockchains: Expensive data authentication for decentralizing trust. In this dissertation, we analyze a popular public blockchain like Ethereum [16] and

discuss its low throughput and scalability. Ethereum has orders of magnitude lower throughput and scalability relative to centralized databases [2, 241, 198] or payment systems like Visa [11]. These performance limitations result from employing authenticated data structures like Merkle trees [150, 19] which are crucial for enabling any untrusted server to join the blockchain network and process user transactions; authenticated data structures provide data along with proofs that verify the correctness of data. Miners in Ethereum store the system state in a logical Merkle tree and rely on key-value stores like RocksDB [24] or LevelDB [93] to persist them on disk; RocksDB and LevelDB induce their own I/O overheads [176, 175, 180]. Further, miners execute one transaction at a time and serialize disk I/O, introducing I/O bottlenecks that fundamentally limit the throughput and scalability of Ethereum; processing a single block of 100 transactions in Ethereum requires performing more than 10K random I/O operations (100× higher and takes hundreds of milliseconds even on a datacenter-grade NVMe SSD); these overheads increase with the growing system state on disk [222]. Overall, the inefficient on-disk layout of authenticated storage in Ethereum cause miners to process fewer transactions per second and to underutilize their compute resources. However, naively redesigning public blockchains can compromise their safety, security, or their decentralized nature. Thus, we need a public blockchain that allows miners to process more transactions per second without compromising its safety, security or its decentralized mode of operation.

I/O bottlenecks. Thus, state-of-the-art PM key-value stores [238, 48, 61], transactional stores [2, 191, 241, 206, 209, 33, 205, 192, 198, 87, 103], and public blockchains like Ethereum [16], incur I/O bottlenecks that (a) are inherent to their design and architecture, (b) induce the poor utilization of underlying storage, network, and compute resources respectively, and (c) limit their end-to-end throughput and scalability.

1.3 Minimizing I/O bottlenecks

To address the I/O bottlenecks across these three systems, and to synthesize a systematic approach for achieving scalable, high-throughput systems, we refer to prior research through the lens of minimizing I/O bottlenecks. First, write-optimized and log-structured data structures [158], key-value stores [24, 90], file systems [182], and distributed systems [218, 40], customize their accesses to the ideal access patterns of the slow storage devices like disks and SSDs. Next, operating systems (OS) research [147] deconstruct and rethink the responsibilities of an OS for achieving higher performance. Also, database systems research [81] continues to restructure its I/O operations by leveraging batching [101] and asynchrony [179] to achieve scalability across multiple cores and high throughput. We highlight that all of them diverge from general-purpose designs to exploit their full potential via specialization.

Main ideas: Following prior research, this dissertation employs several key ideas to build specialized systems that minimize I/O bottlenecks. The first idea aims at customizing the storage subsystem to align with the performance characteristics of underlying media and with the data layout of its target application. The second idea focuses on co-designing the data processing and data storage layers through a clear demarcation of the roles of each system component. The third idea centers on leveraging asynchrony and batching to reconsider how and when I/O is performed with an objective to improve the utilization of underlying resources.

This dissertation seeks an answer to: *How do we architect systems to minimize I/O bottlenecks and simultaneously achieve high throughput and scalability?* It proposes a fundamental redesign of systems by carefully crafting the roles and responsibilities of each component to achieve *specialized systems* that minimize I/O bottlenecks and improve resource utilization.

1.4 Contributions and overview

This dissertation combines these ideas to build three novel systems: Skye, Cascades, and RainBlock. This dissertation makes the following contributions:

1. Introduces Skye, a PM key-value store that saturates PM write bandwidth; discusses the novel *indirect-access* architecture and nvLOG, an interface to PM that accounts for its complex performance characteristics.

2. Presents Cascades, a novel distributed transactional store that saturates datacenter networks while processing distributed transactions; Cascades uses Lattice, an existing framework that enables logging at high throughput without trading off the simplicity of recovery.

3. Introduces RainBlock [167, 166] a public blockchain that tackles I/O bottlenecks from data authentication and achieves high throughput and scalability; presents the novel Distributed, Sharded Merkle Tree (DSM-Tree) data structure that is custom-designed for storing the system state of public blockchains.

This dissertation presents the design and architecture of the three systems, discusses their trade-offs, and evaluates their end-to-end performance improvements.

1.4.1 Skye: Fine-grained control over all PM accesses

Skye is the first PM store that effectively utilizes available PM write bandwidth and achieves high throughput and scalability. The central idea in Skye is to minimize I/O bottlenecks by maintaining fine-grained control over all PM accesses for achieving high PM bandwidth utilization. Skye deviates from current practices and provides *indirect-access* to applications; applications send requests to Skye, which uses dedicated threads to access PM on their behalf. Instead of relying on hardware-managed PM, Skye controls how data is placed on individual NVDIMMs. Skye leverages multiple media to avoid overloading PM and limits remote NUMA accesses

to achieve scalable and high throughput. On a single NVDIMM, Skye outperforms state-of-the-art PM stores by 2.5-5× on standard Yahoo Cloud Serving Benchmark (YCSB). With four NVDIMMs across four NUMA nodes, its write throughput scales by 3.9×; Skye utilizes ≈86% of available PM write bandwidth.

1.4.2 Cascades: Asynchronous durability with efficient recovery

Cascades is a distributed transactional store that effectively utilizes available network bandwidth and simultaneously achieves high throughput and scalability without trading off efficient recovery. Cascades introduces a novel way of handling failures while persisting data asynchronously. Cascades relies on an existing logging infrastructure Lattice for the persistence and replication of data. Lattice notifies Cascades once the log records are *durably persisted*; Lattice receives log records and their dependencies and considers a log record to be durably persisted if the record and all its dependencies are durable. Cascades notifies its clients only after a transaction's commit record is durably persisted. However, Cascades can speculatively execute newer transactions before the previous log records are durably persisted. Thus, Cascades makes forward progress without waiting on (consensus-replicated) I/O to complete, and achieves high throughput and scalability. On zone-replicated ultra SSDs [8], Cascades performs 36× higher when using Lattice instead of synchronous logging in the critical path. With premium SSDs [5], which are relatively cheaper than ultra SSDs, Cascades achieves 160× higher throughput. Overall, Cascades leverages asynchrony and achieves strong consistency without complicating recovery with an additional ≈30% overhead in its end-to-end throughput.

1.4.3 RainBlock: Faster transaction processing in public blockchains

RainBlock [166, 167] presents a new architecture for public blockchains and is the first to address the I/O bottlenecks in public blockchains like Ethereum. The main idea in RainBlock is *to increase the transaction processing rate at miners* which

allows miners to safely pack more transactions per block. And the central insight that allows faster transaction processing at miners is to customize the data authentication layer for high scalability and to decouple the responsibility of maintaining the latest state from the miners. RainBlock deconstructs miners into storage nodes, miners, and I/O-Helpers. Storage nodes manage the system state and employ the novel Distributed, Sharded Merkle Tree (DSM-Tree) data structure. The DSM-Tree stores data memory-optimized format and allows concurrent reads and writes at high throughput. I/O-Helpers prefetch data from storage nodes on behalf of miners outside the critical path. In RainBlock, miners can typically process transactions without performing I/O in the critical path. On workloads that emulate Ethereum mainnet transactions, a single miner in RainBlock processes 27× more transactions than in Ethereum. In geo-distributed settings, with miners across three continents, miners in RainBlock process 20K transactions per second. RainBlock finalizes 20× higher transactions (with the same latency) relative to Ethereum.

1.5 Outline

This dissertation has the following outline:

1. **Background and motivation** (chapter 2) provides the necessary background for each system, outlines their throughput and scalability limitations, and highlights the source of I/O bottlenecks in each system, and motivates the need for scalable, high-throughput systems.

2. **Minimizing I/O bottlenecks** (chapter 3) outlines a set of well-established ideas that are crucial for achieving scalable, high-throughput systems.

3. **Core contributions**. The next three chapters describe the design and architecture of three novel systems. Skye (chapter 4), a PM store that effectively utilizes available PM bandwidth, Cascades (chapter 5), a transactional store that effectively utilizes the available network bandwidth, and RainBlock (chapter 6),

a public blockchain that effectively utilizes compute capacity at miners. These chapters also discuss a performance evaluation of each system.

4. **Related work** (chapter 7) places Skye, Cascades, and RainBlock, in the context of prior research on PM stores, distributed databases, and public blockchains.

5. **Conclusion** (chapter 8) discusses potential directions for extending this work and concludes this dissertation.

Chapter 2: Background and Motivation

In this chapter, we provide relevant background for the core contributions of this dissertation. First, we describe Persistent Memory (PM) and outline the design choices of state-of-the-art PM key-value stores (§2.1). Next, we review state-of-the-art distributed stores and their common practices to achieve high throughput, scalability, reliability, and strong consistency (§2.2). Finally, we describe public blockchains and their operation without centralized trust (§2.3). Note that we discuss the inherent I/O bottlenecks in each system and outline their impact on overall performance to motivate the need for novel, scalable and high-throughput systems.

2.1 Persistent Memory key-value stores

First, we describe Persistent Memory (PM) and outline its hardware and performance characteristics. Next, we discuss state-of-the-art PM key-value stores and their design choices. Finally, we discuss the inherent I/O bottlenecks in these PM stores and motivate the need for scalable, high-throughput PM stores.

2.1.1 Persistent Memory

Persistent Memory (PM) is a new storage-class memory technology that offers durability and byte-addressability. Intel's Optane DC Persistent Memory Module [171] was the first commercially-available media of this kind; other companies are also working on persistent memory technologies (*e.g.,* PCM [54], STT-MRAM [34], Memristor [227], Samsung's memory-semantic SSD [82]). PM is available as individual non-volatile DIMMs (NVDIMMs) which can be directly connected to the memory-bus like DRAM, as shown in Figure-2.1. PM has low latency (similar to DRAM) and high bandwidth (10× compared to modern SSDs) [228]. It is cheaper and denser than DRAM; a single server with 4 non-uniform memory access (NUMA) nodes can

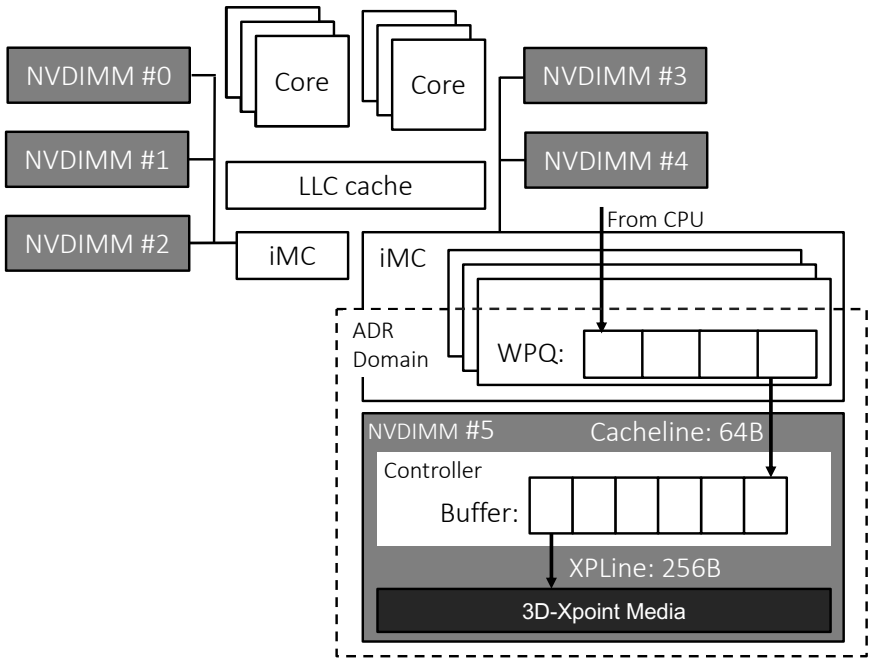


Figure 2.1: **PM hardware configuration**. Up to 6 NVDIMMs per node each with an internal XP-Buffer and 256B XPLine.

support up to 12 TB of PM (six 512 GB-sized NVDIMMs per node). Therefore, PM, with its large capacity, low latency and high bandwidth, is a promising building block for scalable high-throughput persistent key-value stores.

Latency and bandwidth. PM has asymmetric latency and bandwidth for reads and writes. Loads on PM incur 2–3× higher latency than on DRAM, while stores incur similar latency on PM and DRAM [171]. The write bandwidth of a single NVDIMM is 2.3 GB/s ($1/6^{th}$ of DRAM's write bandwidth) and its read bandwidth is 6.6 GB/s ($1/3^{rd}$ of DRAM's read bandwidth) [107]. PM has high I/O bandwidth relative to disks (10× compared to modern SSDs). However, PM bandwidth is sensitive to the access patterns, sizes, and the number of concurrent operations.

Setup: AppDirect or Memory mode. PM can be setup either in MemoryMode or AppDirect mode. With AppDirect, PM is exposed as a block device and offers data durability; we focus on the AppDirect mode of PM.

Configuration: Interleaved or non-interleaved PM. With the AppDirect mode, individual NVDIMMs can be setup in interleaved or non-interleaved mode. Non-interleaved mode exposes each NVDIMM as its own PM device. Interleaving exposes multiple NVDIMMs within a NUMA node as a single PM device where 4KB blocks are spread out (round-robin) across available NVDIMMs.

PM block size and write amplification. Each NVDIMM has a write-combining buffer (XPBuffer) that batches writes into 256 bytes before writing to the underlying media. Writes smaller than 256 bytes force the XPBuffer to perform read-modify-writes which results in write amplification and introduce I/O overheads.

Thread scaling and concurrent I/O. Using too many threads to perform writes on NVDIMMs causes writes to queue quickly (introducing head-of-line blocking overheads) at the XPBuffer as it combines writes and manages the load [107, 228, 70, 240].

Mixed I/O. NVDIMMs are connected to channels which in turn are connected to an integrated memory controller (iMC) on the CPU. Read and write requests are inserted into the read or write pending queues inside the iMC. Due to the asymmetric latencies of PM, processing reads and writes concurrently at peak bandwidth utilization incur blocking overheads at the iMC [107, 228, 70].

CXL and PM expansion. Compute Express Link (CXL) [65] is the first open multi-protocol method to support a cache-coherent interconnect for processors and memory devices. CXL is built upon PCIe and supports memory expansion [113, 148, 190, 94]. With CXL, NVDIMMs can also be hosted and accessed over the PCIe-bus like SSDs. CXL allows CPU cacheable loads and stores to PM over the PCIe-bus. Recent studies on real CXL hardware show that like PM, CXL performance also suffers with too many threads, is sensitive to access patterns and how data is distributed to CXL devices [200]. With CXL, available PM capacity and bandwidth in a server will scale up significantly, making PM a promising building block for large-scale, scalable, high-throughput key-value stores.

2.1.2 PM key-value stores and direct-access for applications

PM offers an order of magnitude higher bandwidth than existing storage devices like SSDs and persists data at DRAM-comparable, low latencies. Hence, it is important to build PM key-value stores that can effectively utilize the high bandwidth of PM; applications like Flink [32] and CockroachDB [131] can leverage the high write bandwidth of PM and support high write throughput. Further, PM stores with high PM bandwidth utilization enable disaggregated data centers [94] to keep their costs in check [178]. Therefore, we explore this part of the design space in this dissertation: scalable PM key-value stores that achieve high PM bandwidth utilization.

Direct-access architecture to provide low latency. The state-of-the-art PM key-value stores [238, 48, 61] allow applications to directly read and write to PM and provide low latency to applications. They minimize additional accesses in the critical path of processing requests; applications memory-map a file on PM and directly perform loads and stores. However, through this approach, current PM stores *give up control*: applications decide how many concurrent threads read and write to PM. Thus, existing PM stores *offload* fine-grained control over the I/O performed on PM by allowing application threads to directly access PM.

Interleaving NVDIMMs as a single PM device. All PM key-value stores [238, 48, 61, 224, 98, 122, 116, 185] and even PM file systems [240, 115, 226] interleave PM; they rely on hardware to manage multiple NVDIMMs in a NUMA node and employ a single, unified PM device. With this approach, the hardware controls the striping and placement of data across individual NVDIMMs; a large file round-robins at 4 kB granularity across NVDIMMs in a NUMA node. Thus, PM stores *offload* the control of how data is placed across individual NVDIMMs to memory controllers.

2.1.3 I/O bottlenecks from PM-agnostic design choices

PM stores that assume an interleaved PM device and provide direct-access to applications incur I/O bottlenecks that limit their throughput and scalability. They

incur I/O bottlenecks due to the nuanced performance characteristics of PM.

Performance characteristics of PM. Recent PM studies [70, 107, 228] and our analysis (§4) highlight that PM is sensitive to the number of concurrent reads and writes to NVDIMMs and the data placement across individual NVDIMMs. If PM stores do not carefully control PM accesses and data placement, then the memory-controller gets overloaded, reducing PM bandwidth and limiting the peak throughput of PM stores. Our empirical study on PM shows that increasing the number application threads beyond a certain threshold reduces PM bandwidth and limits overall throughput. Further, managing individual NVDIMMs provides ≈20% higher throughput for reads and writes. Therefore, the design choices of existing PM key-value stores are directly at odds with obtaining high throughput on PM.

Performance limitations and poor PM bandwidth utilization. PM stores prioritize low latency, offload fine-grained control over PM, and obtain only a fraction of PM bandwidth and suffer from low throughput. The best-performing key-value store, FlatStore [61] achieves only ≈45% of the write bandwidth on a single NVDIMM; furthermore, its throughput does not scale (and drops) beyond eight threads (§4). The low throughput and scalability of existing PM stores is a fundamental consequence of their latency-oriented designs. With direct-access architectures, PM stores *give up control*: applications decide how many reads and writes happen concurrently to PM, and the hardware controls the striping of data across individual NVDIMMs. Therefore, we need a PM key-value store that avoids I/O bottlenecks from PM and effectively utilizes PM bandwidth to achieve high and scalable write throughput.

2.2 Distributed transactional stores

In this section, we provide some background on distributed databases and describe a few of their common practices to support ACID transactions with strong consistency and isolation guarantees. We explain how they support transactions that span partitions, handle failures and outline their inherent I/O bottlenecks.

System	Strong consistency	Concurrency control	Strict isolation
Spanner [68]	✓	TrueTime API	✓
FoundationDB [241]	✓	MVCC w/ serializable txs	✓
Chardonnay [81]	✓	MCC w/ serializable txs	✓
Hekaton [75]	×	OCC w/ latch-free data structures	(SSI) ×
VoltDB [198]	✓	Serial execution (2PC-variant)	✓

Table 2.1: **Design choices of modern databases**. Concurrency Control and Isolation; Optimistic Concurrency Control (OCC), Multi-Version Concurrency Control (MVCC), Last-Write-Wins (LWW), Snapshot Isolation (SSI), External (externally-consistent), Multiversion (Bounded Staleness)

2.2.1 Common practices in distributed databases

First, we discuss how distributed stores handle distributed transactions at high throughput while providing strong consistency and isolation.

In-memory or on-disk databases. In-memory databases [198, 118, 75] store entire data in DRAM and can support distributed transactions with ACID properties [220, 237]. However, DRAM is 10–50× more expensive than regular SSDs. Therefore, applications rely on distributed databases [241, 209, 68, 33, 191, 81] that store data on disk using storage engines like RocksDB [24] or LevelDB [93]. Table 2.1 lists a few on-disk and in-memory databases (separated by a horizontal line respectively).

Partitioning for scalability. Distributed databases employ a sharded cluster of servers where each server stores a subset of the data for achieving scalable performance. Databases may allow servers to share/replicate some data (shared-something) or adopt shared-nothing architectures. Cassandra [33], Chardonnay [81], NuoDB [204], and FaunaDB [87], *etc.* are on-disk databases with shared-nothing architectures.

Replication for availability. Distributed databases rely on replication to handle failures (*e.g.,* power, hardware, software environment, failures) and support high data availability. Each shard has standbys (replicas) with consensus protocols to manage

the shard and its replicas (like Paxos [132] in Spanner [68]). Databases replicate across several zones in a datacenter since cross-datacenter latencies are too high.

Strong consistency and distributed transactions. With partitioning for scalability and replication for availability, databases must support transaction span across shards, or distributed transactions, and provide ACID guarantees. Strong consistency is a serializable and linearizable execution of transactions *i.e.,* all executions have a serial order of transactions; for any two concurrent transactions TX1 and TX2, the database appears to commit TX1 either before or after TX2.

Two-phase commit (2PC). Distributed transactions employ protocols like two-phase commit (2PC) to support strong consistency and isolation properties. The 2PC protocol [64, 130] and its variants [154, 129] enable cross-shard or distributed transactions. They ensure that distributed transactions are either committed or aborted across all participating servers while maintaining consistency and isolation; 2PC with Two-Phase Locking (2PL) provides isolation while avoiding deadlocks.

2.2.2 Synchronous I/O to recovery logs for fault tolerance

Distributed databases rely on write-ahead or ARIES-style [155] logs to support ACID distributed transactions and for consistent recovery while handling failures.

Logging for fault tolerance. Distributed databases rely on logs to achieve strongly consistent distributed transactions and to recover from crashes or power failures. Protocols like 2PC [64] and its variants [154, 129] ensure that distributed transactions are either atomically committed or aborted in a coordinated manner, despite failures. With such protocols, servers log their decisions to commit or abort a transaction before communicating that to other participants; their decisions are recovered from logs while handling failures. With these logs, databases can handle incomplete transactions at the time of a failure; they either choose to complete or abort these transactions based on the information in the logs.

2PC protocol and transaction commit path. The basic 2PC protocol proceeds in the following manner. For a transaction (TX) that is to be executed on all the participating servers (participants), a coordinating server (coordinator) starts the first phase by sending a `PREPARE` message to each participant. Participants can vote either `YES` or `NO` in response, where `YES` is a promise that the participant will not unilaterally abort TX and can (eventually) commit TX when requested. Before voting `YES`, the participant typically persists all the TX's writes to a durable log to recover consistently in the event of a failure. Note that if any participant votes `NO` (or fails to respond before coordinator timeouts), then the coordinator decides to abort TX. Otherwise, coordinator commits TX by logging this decision and initiating the second phase of the protocol by issuing a `COMMIT` message to every participant. In response, participants apply and commit TX and release locks. A well-known problem of 2PC is its blocking nature [50, 193] *i.e.,* a coordinator's failure can prevent participants from making forward progress; to address this Spanner replicates coordinators [68].

2.2.3 I/O bottlenecks from recovery logs

Distributed databases rely on protocols like 2PC to atomically commit distributed transactions while tolerating failures; servers log their decisions to disk before communicating them to other servers over the network. The end-to-end latency in such distributed databases is determined by the network latency for exchanging `PREPARE/COMMIT` messages or votes amongst participants and the I/O latency for each participant to durably persist votes/decisions to its recovery log on disk. With low-latency and high-bandwidth, general-purpose RPC frameworks like eRPC, distributed databases predominantly suffer from I/O bottlenecks [81].

Storage more dominant than network. With eRPC, databases can send messages at single-digit μs latencies [117]. Thus, I/O latencies become the dominant cost. Note that Persistent Memory and 3D-Xpoint media can support DRAM comparable low latencies, however, they are more expensive than traditional SSDs. More com-

monly, databases either use network-replicated disks *i.e.,* more-expensive ultra SSDs that provide ≈2ms latency or cheaper premium SSDs that provide ≈450ms latencies. Thus, networks have become 1–2 orders of magnitude faster than network-replicated SSDs and introduce I/O bottlenecks in the critical path of processing transactions.

Group commit and batching. Modern databases exploit batching; they commit large batches of transactions to reduce I/O bottlenecks and achieve higher throughput [2, 191]. With batch commits and by handling non-conflicting transactions optimistically, databases trade off low latency and achieve high throughput and scalability.

Poor scalability for highly-contented workloads. Distributed databases incur higher overheads for highly contended workloads with distributed transactions. With 2PC, distributed databases wait until the logs on disk are successfully updated before releasing their locks. Thus, with high contention, synchronous logging for recovery serializes conflicting transactions and limits the end-to-end throughput and poor scalability of distributed databases.

Early lock release (ELR) proposal and its limitations. Prior research [74] proposes releasing locks before updating the logs and discusses the correctness of this approach [194]; it proposes logging asynchronously while ensuring that the logging order matches the execution order of transactions. However, this proposal does not extend its solution to distributed databases with shared-nothing architectures and multiple coordinators and hence suffers from poor scalability. Further, ELR requires a complex mechanism to recovery from failures; in scenarios where databases have multiple coordinators, ensuring a consistent rollback across multiple coordinators and participants is challenging. Hence, this proposal has not seen practical implementations due to several unaddressed concerns [80]. Thus, we need a new approach to achieve scalable and high-throughput distributed databases that support ACID transactions with strong consistency and can minimize I/O bottlenecks without trading off the simplicity of recovery.

2.3 Public blockchains

In this section, first we provide some background on public blockchains. Next, we discuss how popular public blockchains like Ethereum [16] rely on Merkle trees to authenticate data and to decentralize trust. Finally, we outline the inherent I/O bottlenecks in Ethereum and highlight its throughput and scalability limitations.

2.3.1 Overview

Public blockchains, like databases, maintain some system state and support transactions on that system state. Public blockchains are widely used because of their open networks, decentralized architectures, and immutable, auditable state.

Open Networks. Public blockchains allow untrusted servers to join their network and process transactions. These untrusted servers are responsible for storing and advancing the system state and the blockchain. Every block in the blockchain has an ordered list of transactions. Further, blocks store cryptographic hashes of their previous blocks, creating a chain of blocks that is cryptographically secure and immutable. In public blockchains, the system state advances from one snapshot to the other with every new block of transactions, providing consistent snapshots of the system state.

Decentralization. In public blockchains, the untrusted servers in their networks can be malicious and can provide incorrect information about the latest system state. Thus, public blockchains follow the State Machine Replication (SMR) [186, 132] paradigm to tolerate a minority of malicious servers. Each untrusted server in the network acts as a state machine replica that starts from a fixed initial state. Following SMR, every non-malicious server that begins with the same initial state and processes the same blocks of transactions, arrives at the same final system state. Public blockchains rely on this non-malicious quorum to serve the correct system state after processing the transactions in the blockchain.

Consensus. In public blockchains, servers that create new blocks and extend the

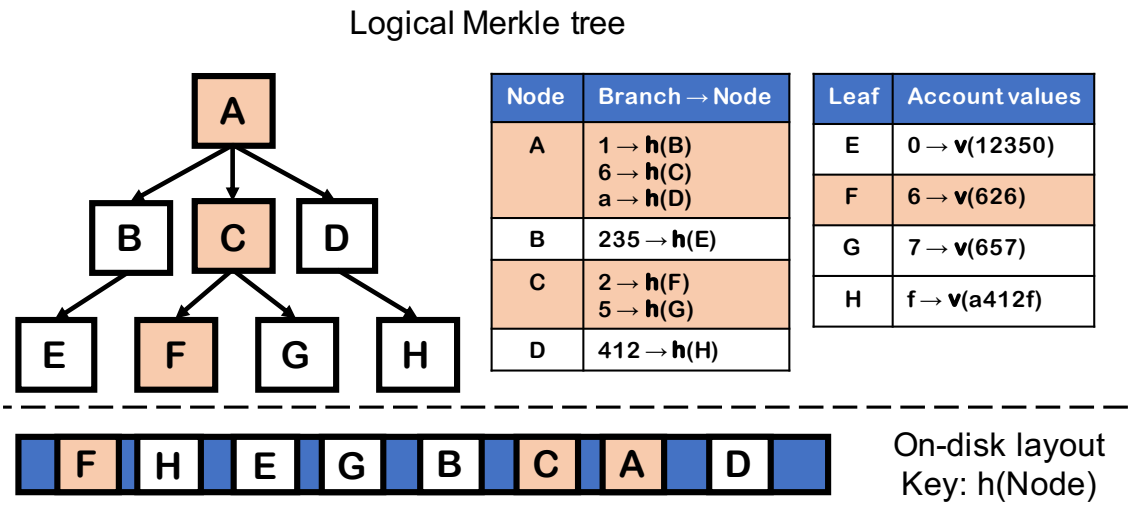


Figure 2.2: **Merkle Patricia Trie**. Reading the account of an address 626, for example, would require reading node A, traversing branch 6, and looking up C using $h(C)$. Then, traversing branch 2 and looking up F using $h(F)$. Notice that the nodes in the MPT at random locations on disk.

blockchain (termed miners) can also be malicious. As a result, public blockchains rely on consensus protocols like Proof-of-Work (PoW) [108] to maintain their correctness and liveness. Broadly, PoW ensures that miners create a new block (roughly once every 12 seconds) only after a majority (about 95%) of the servers in the network receive and process the previous block [211]. This rate limit avoids forking of the blockchain, *i.e.,* it restrains multiple miners from creating new blocks on top of different previous blocks. Thus, PoW prevents forks that ambiguate the latest system state or halt its progress, maintaining the security and liveness of public blockchains.

2.3.2 Merkle trees for data authentication

In public blockchains, with untrusted servers maintaining the system state, users cannot trust the data they receive from such servers. Many public blockchains [16, 104, 45] use *authenticated data structures* such as Merkle trees [150] to provide proofs that verify that this data is correct.

State authentication. With authenticated data structures, users can read data along with proofs (called witnesses) from untrusted servers. These witnesses allow users to verify if the data is correct and if it belongs to the latest snapshot of the system state. Similarly, new servers can request the system state from an untrusted server and verify its correctness, without having to reconstruct it locally by replaying the entire blockchain.

Merkle tree. To authenticate the system state, a public blockchain like Ethereum [16] relies on a variant of the Merkle tree called the Merkle Patricia Trie (MPT) [19]. In Ethereum, the system state is a key-value mapping from unique addresses to user accounts. The MPT stores these accounts in the leaf nodes and indexes them with their addresses, as shown in Figure-2.2. In this dissertation, we use the terms MPT and Merkle tree interchangeably.

Merkle root. In a Merkle tree, every non-leaf node stores the cryptographic hashes of all its children. Thus, the hash of the root node, called the Merkle root, hashes of all the values in the system state, effectively summarizing a snapshot of the system state. In Ethereum, as the system state changes with every new block of transactions, each block stores a Merkle root. The Merkle root in each block represents the expected system state after executing all the transactions in that block.

Merkle proof or witness. Witnesses allow users to identify stale or incorrect data from untrusted servers. A witness is a vertical path in the Merkle tree and has all the nodes from the root to the leaf storing the data. To verify the data, users recompute a Merkle root locally using its witness and cross-check if that Merkle root matches with the Merkle root published in the latest block. A Merkle root mismatch indicates that the data is stale or incorrect.

2.3.3 I/O bottlenecks from inefficient data authentication

Public blockchains have orders of magnitude lower throughput relative to centralized databases [131, 241] and payment systems like Visa [11]. Further, their

transaction throughput drops as the system state increasing over time.

On-disk layout of Merkle trees. Public blockchains store the logical Merkle tree using key-value stores like RocksDB [24] or LevelDB [93]. Each node in the logical Merkle tree is stored as a value in the key-value store, and nodes are indexed with their node hashes. With this approach, Merkle tree traversals result in multiple random disk reads and updates to the tree result in multiple disk writes. Due to random reads, this approach is not suitable for disks and induces poor performance. Fundamentally, by design, the top layer of the Merkle tree is more frequently accessed and modified; however, due to hash-indexed Merkle nodes updated Merkle nodes are written as new key-value pairs on disk. Thus, this approach also does not cater to the common access patterns of Merkle trees.

I/O in the critical path of processing transactions. To process a transaction, miners read and modify the system state which translates to multiple random reads and multiple writes to disk. Moreover, as the system state increases over time, miners take more time to process a new block of transactions, further slowing down public blockchains. Although PoW limits the block creation rate, PoW typically allows servers to receive newer blocks after they process the previous block, and does not limit the number of transactions per block. Therefore, if miners in public blockchains can process more transactions per second without modifying PoW then miners can release larger blocks and achieve higher throughput. There has been active research on increase the throughput of public blockchains by sharding the blockchain [128, 236, 144, 213] or with alternatives for PoW [91, 85, 151, 145, 17]. Tackling the I/O bottlenecks from authenticated storage is orthogonal to these approaches. Therefore, we need a public blockchain that addresses these I/O bottlenecks and achieves high throughput and scalability.

Chapter 3: Minimizing I/O Bottlenecks with Specialized Systems

In this chapter, we revisit prior systems research from the perspective of minimizing I/O bottlenecks and summarize three core ideas. These ideas in combination seek an answer to: *How can we architect systems to minimize I/O bottlenecks and simultaneously achieve high throughput and scalability?*

First, we summarize relevant research on append-only and write-optimized storage to emphasize customizing the storage subsystem to the underlying hardware (§3.1). Next, we review operating systems research to encourage redefining roles and rethinking the responsibilities of each system component (§3.2). Then, we discuss how systems adopt asynchrony and other optimizations to highlight redistributing the roles amongst components (§3.3). Finally, we outline how these ideas metamorphize in three unique and specialized systems: Skye, Cascades, and RainBlock (§3.4).

3.1 Customizing storage to hardware characteristics

Modern systems leverage log-structured designs to limit I/O bottlenecks and to align with the performance characteristics of disks and SSDs.

Log-structured data structures like Merge trees (LSMs) [158], LSM-based key-value stores [24, 93], replicated databases atop shared logs [40, 218, 42], distributed protocols [42, 76] and concurrent data structures [41] over shared logs, *etc.*, continue to gain traction. The shared log paradigm has become the heart of modern distributed systems and applications. Log-structured file systems [182] and journaling [169] also leverage logs to support atomic and crash-consistent system calls.

Fundamentally, logs translate application traffic into sequential writes to disks or SSDs. Since persistent media is optimized for sequential writes, logs enable higher bandwidth utilization for high-throughput systems. Therefore, modern systems lever-

age logs to customize to the performance characteristics of the underlying storage device and demonstrate throughput and scalability improvements.

3.2 Co-designing data processing and storage

Operating systems (OS) highlight the benefits of co-designing; responsibilities that were exclusive to an OS have shifted to hardware controllers and applications.

Operating systems have offloaded specific tasks to specialized hardware to reduce the burden on CPUs. Today, SSD controllers manage wear-leveling, garbage collection, bad block management *etc.*, dedicated network interface cards (NICs) and network processing units (NPUs) handle checksum calculation, TCP/IP segmentation and reassembly, encryption/decryption *etc.*, and GPUs render graphics.

Further, applications and deployment settings have also shaped OS research. For example, UNIX [181] has enabled multiple users to share tasks and resources. However, to improve resource utilization across diverse workloads, Exokernel [83] has provided application with direct access to hardware and achieves higher performance; with Exokernel, applications itself are responsible for managing multiple resources. Further, embracing a similar minimal design, operations systems for decentralized or disaggregated environments [188, 184] have proposed supporting only fundamental functions like security, data integrity, reliable communication *etc.* Later, operating systems also began supporting multiple guest operating systems to enable virtualization in the clouds [44, 99, 147] to support cloud-scale applications in datacenters [164]; they addressed the resulting I/O overheads.

Thus, traditional operating systems have transformed into being distributed, direct-to-application and hardware-aware by rethinking the responsibilities of subsystems and co-designing them for applications and deployment settings. This approach has demonstrated better resource utilization and higher performance.

3.3 Restructuring I/O operations

Modern systems leverage several techniques that uniquely modify their I/O behavior in the pursuit of achieving high throughput and scalability. We revisit a few common practices like caching, batching, scaling out, and employing asynchrony.

Distributed databases and key-value stores use caches to absorb latest writes (write-back caches) and handle reads from main memory [7, 149] to avoid random disk I/O whenever possible. Often, they process a batch of requests to reduce the number of I/O operations and to avoid small, inefficient I/O. Further, these systems scale out to multiple cores, NUMA nodes, and even across servers via sharding, for processing multiple requests in parallel; thus, systems support higher IOPS and address the I/O bottlenecks for I/O-intensive applications. Finally, most systems leverage asynchrony; instead of waiting for I/O operations to complete they make progress on other requests and comeback to older requests once I/O completes. These systems leverage asynchronous and non-blocking I/O to handle multiple requests in parallel and thereby maximize resource utilization and achieve high throughput. With asynchrony, systems detangle I/O from processing requests.

These techniques have their own limitations and trade-offs. Caching introduces weaker-levels of data consistency. Asynchronous durability and relaxed consistency enhance the performance and responsiveness of distributed system. However, prior work investigates the implications of asynchronous durability on data reliability [68], and the inherent trade-offs in adopting relaxed consistency for higher performance [207]. Sharding and partitioning require careful attempts to maintain strong consistency and to balanced load. Batch processing of operations often comes at the expense of increased latency; prior research outlines the intricate relationship between latency and throughput trade-offs in the context of distributed systems [95, 73].

Balancing these techniques is intricate and results in a dynamic landscape of modern systems. Thus, specialized systems that cater to specific workloads, applications, data formats and hardware are crucial for obtaining I/O-efficient systems.

3.4 Specialized systems

These techniques in combination provide a holistic approach towards minimizing I/O bottlenecks and improving the utilization of resources in modern systems. In this dissertation, we follow these ideas to fundamentally re-architect systems by re-defining the role of each component and reconsidering its responsibilities to improve the utilization of underlying resources.

Custom storage. In this dissertation, Skye (§4) introduces a novel logging interface nvLOG that customizes to PM characteristics and is scalable across NVDIMMs. Next, Cascades (§5), writes transaction commit records to networked or replicated disks in a scalable manner; Cascades simultaneously achieves asynchronous durability and efficient recovery without blocking on a synchronous Paxos-style replicated write in the critical path of processing transactions. Finally, in RainBlock and DSM-Tree (§6) introduces a novel Distributed, Sharded Merkle Tree for data authentication that customizes its design to the application data layout and access patterns (optimizes for reads) and is scalable and memory-optimized.

New architectures and design. Skye (§4) adopts a new architecture for PM key-value stores that leverages multiple media and nvLOG to avoid overwhelming PM. Cascades (§5) discusses the API of Lattice that enables it to achieve high throughput and scalability and to simultaneously simplify recovery and replication in distributed transactional stores. RainBlock (§6) deconstructs the responsibilities of a miner and proposes a new architecture for public blockchains.

Restructuring I/O operations. All three systems employ batching to restructure I/O to the underlying storage media and achieve high throughput. Cascades (§5) and RainBlock (§6) avoid I/O overheads from the critical path by performing I/O asynchronously. Further, they speculate on the successful completion of these operations and continue to make progress on other requests.

Chapter 4: Skye

In this chapter, we first present an empirical study to discuss the performance limitations of PM key-value stores and emphasize the I/O bottlenecks in PM stores (§4.1). We study the performance characteristics of PM and summarize four design recommendation to minimize I/O bottlenecks. Next, following these design recommendations, we introduce the novel design and architecture of Skye (§4.2). Finally, we discuss the implementation of Skye (§4.3) and evaluate its throughput and scalability across various workloads and in comparison to state-of-the-art PM stores (§4.5). We show that on a single NVDIMM, Skye outperforms state-of-the-art PM stores by 2.5–5× on the standard Yahoo Cloud Serving Benchmark (YCSB). With four NVDIMMs across four NUMA nodes, Skye obtains ≈86% of PM write bandwidth, and its write throughput scales by 3.9×.

4.1 I/O bottlenecks from PM media

In this section, we first describe the performance and scalability limitations of state-of-the-art PM key-value stores (§4.1.1). Next, we summarize our empirical study that highlights the nuanced performance profile of PM (§4.1.2). Then, we outline four design recommendations for PM key-value stores that are crucial for minimizing I/O bottlenecks from PM media (§4.1.3). Finally, we discuss a few strawman solutions and their drawbacks (§4.1.4) to motivate our approach.

4.1.1 Performance limitations of PM stores

Existing PM stores [61, 238, 48] support direct-access for applications and provide low latencies (2–5 us/op). However, state-of-the-art PM key-value stores [238, 214, 48, 61] suffer from poor performance; these PM stores utilize <45% of PM write bandwidth and <10% of PM read bandwidth with varying access sizes (256B–1kB),

PM key-value stores	Write bandwidth (% utilization)	Read bandwidth (% utilization)
ChameleonDB	4 GB/s (29%)	2 GB/s (7%)
FlatStore	6 GB/s (44%)	3 GB/s (10%)
Viper	2 GB/s (12%)	3 GB/s (9%)

Table 4.1: **PM stores**. Peak PM bandwidth utilization of PM stores on a single node (with 6 NVDIMMs, up to 48 threads) for writes and reads with 8B keys and across 8B–1kB values.

as shown in Table-4.1. Further, their throughput drops on increasing the number of application threads, as shown in Figure-4.1.

We measure the peak bandwidth utilization of these PM stores for pure-write (YCSB LoadA) and pure-read (YCSB RunC) workloads using 8B keys and with varying value sizes (from 8B up to 1kB). We use a single NUMA node with six NVDIMMs and up to 48 threads. For this study, we categorize existing PM key-value stores into two groups: key-value stores that are retrofitted for PM and ones that are designed from the ground up for PM.

Retrofitted PM stores. Retrofitted PM stores are derived from key-value stores that are originally designed for block devices *e.g.,* stores like RocksDB [24] and LevelDB [90] that are based on log-structured merge trees [158], or from in-memory key-value stores *e.g.,* pmem-Redis [165]. ChameleonDB [238], MatrixKV [231], NoveLSM [122], SLM-DB [116], and ListDB [126] are a few retrofitted LSM-based PM key-value stores. These stores do not cater to the unique characteristics of PM but take advantage of its byte-addressability and durability.

We observe that the peak write bandwidth achieved across NoveLSM, SLM-DB and pmem-Redis [165] is 215 MB/s which is <2% of the available PM write bandwidth. Amongst the retrofitted stores, ChameleonDB obtains ≈29% of available write bandwidth, as shown in Table-4.1. Since these PM stores are not NUMA-aware, their performance does not scale to multiple NUMA nodes.

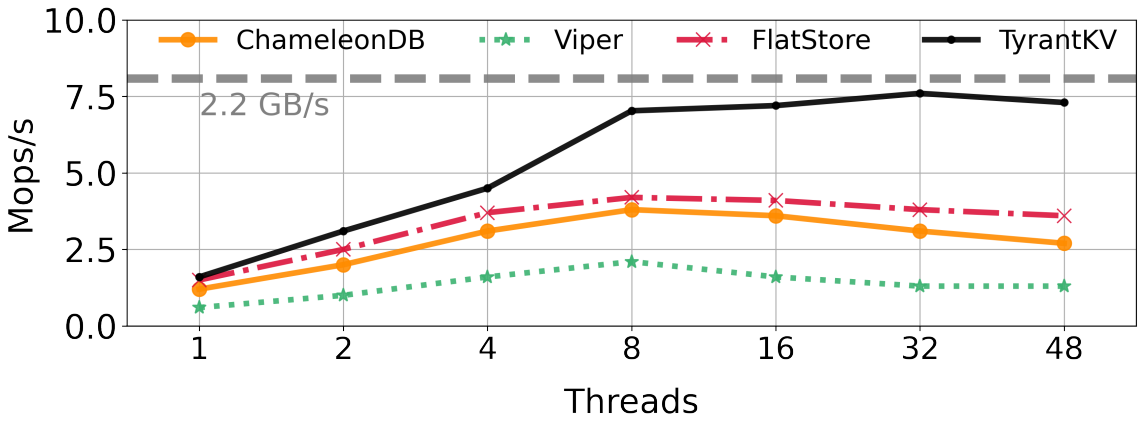


Figure 4.1: **Throughput and scalability**. Existing PM stores have low write throughput which drops beyond 8 threads. Skye achieves high and scalable write throughput.

New PM stores. Key-value stores that are custom-built for PM like FlatStore [61] and Viper [48] achieve better write-bandwidth utilization *e.g.,* with larger 1 kB values they obtain <45% of available write bandwidth, as shown in Table 4.1. These PM stores assume a single interleaved PM device. Viper customizes to PM by allowing application threads to write directly to PM without intermediate DRAM buffers, and has NVDIMM-aligned logs and evenly distributes threads across logs. FlatStore proposes efficient batching to avoid small writes to PM and uses per-core logs that span across multiple NVDIMMs. Thus, FlatStore and Viper do not limit the number of threads accessing a single NVDIMM. Thus, these PM stores do not maintain fine-grained control over the I/O on NVDIMMs, and hence utilize a small fraction of PM bandwidth. Further, they have low throughput that degrades with increasing number of application threads (beyond 8), as shown in Figure-4.1.

4.1.2 PM empirical study

We perform an empirical study to understand the throughput and scalability of PM, under various workloads and configurations.

Setup. We use a four-socket machine with 3 TB of Intel Optane DC Persistent

Memory across 24 NVDIMMs, 224 cores, 790 GB DRAM, and with Ubuntu 20.04 and Linux 5.16 kernel for this study. Our experiments use DAX-enabled ext4 to memory-map 12 GB sized PM files and use direct load and non-temporal store instructions [183] to read and write to PM. Memory-mapping PM avoids I/O overheads from the underlying file system [226, 114, 115]. We page fault and zero-out PM pages before using them to avoid their overheads in our experiments.

In our experiments, we perform reads and writes to PM in different configurations (interleaving vs non-interleaving modes, single vs multiple NUMA nodes), with varying I/O sizes, I/O patterns (sequential vs random), and the number of reader and writer threads. We make the following observations.

O1: Sequential access gives better performance than random. With random instead of sequential 512B accesses, single-threaded read throughput drops by ≈2× from 3.9 GB/s to 2.1 GB/s, and write throughput drops by ≈3× from 2.2 GB/s to 0.7 GB/s. Thus, sequential access pattern is better for reads and writes; sequentiality is more important for writes than reads.

O2: 512B access size gives the best performance. We note a peak single-threaded throughput of ≈2.2 GB/s with 512B and 1kB sized writes, and ≈3.9 GB/s with 256B and 512B sized reads; 512B is preferable for both reads and writes.

O3: Maximum of 3 writers and 6 readers per NVDIMM. On a single NVDIMM with 3 writers, we observe a sequential write throughput of 2.2 GB/s; with 6 readers, we observe a sequential read throughput of 6.9 GB/s. However, increasing the number of threads decreases the write and read throughput due to head-of-line blocking overheads. However, we observe random write throughput of 0.6 GB/s with 3 writers and random read throughput of 5.85 GB/s with 6 readers; random reads scale better than random writes.

O4: Mixed I/O on NVDIMMs. Contrary to prior PM analysis [70], we observe that performing concurrent reads and writes to PM yields better throughput.

Concurrent reads and writes to PM obtain higher cumulative bandwidth, although they reduce the peak read and write PM bandwidth utilization. With 2 readers and 1 writer, we observe a peak I/O throughput of 3.31 GB/s per NVDIMM with mixed I/O. However, serializing reads and writes to NVDIMMs, requires 8 readers and 1 writer to read at 6.8 GB/s and write at 2.14 GB/s but achieves a cumulative throughput of ≈3.26 GB/s. At peak PM write and random-read bandwidth utilization, avoiding mixed I/O provides <5% higher throughput. Thus, concurrent I/O uses 3× fewer threads and allows higher throughput in most cases.

O5: Non-interleaved NVDIMMs provide better performance. We observe consistent and higher bandwidth utilization with the non-interleaved mode. For example, with a single thread, interleaving NVDIMMs has up to 50% lower throughput for reads and writes across varying value sizes (512B–4kB writes). Even with a higher number of threads, managing individual non-interleaved NVDIMMs yields up to 20% higher throughput relative to interleaving. With interleaving and write sizes ranging from 256B to 2MB, write throughput drops from 12.7 GB/s to 10.5 GB/s (≈85% of the maximum write bandwidth), while it remains consistently above 11.7 GB/s (≈97% utilization) without interleaving.

O6: Avoid cross-node traffic. We observe 3× lower write throughput, and 2.5× lower read throughput if threads accessing NVDIMMs are not pinned to their local NUMA node; cross-node traffic limits throughput and scalability.

Summary. With 4 nodes and 24 NVDIMMs we observe a peak aggregate throughput of 162 GB/s for sequential reads (96% of the theoretical limit), 124.2 GB/s for random reads, and 48.3 GB/s for sequential writes (92% of the theoretical limit); we estimate theoretical limits by assuming perfect NVDIMM and NUMA scalability. On a single NVDIMM, we also observe a maximum random read and sequential write throughput of 5.85 GB/s and 2.2 GB/s. With 6 NVDIMMs on a single node, we observe a peak PM throughput of 12.7 GB/s for sequential writes and 31.5 GB/s for random reads.

Note that we use these as our baselines for the maximum attainable read and write bandwidth in the rest of the paper.

4.1.3 Design recommendations for PM stores

We summarize four recommendations (from our analysis and prior work) that enable high PM bandwidth utilization.

R1. Avoid interleaved PM. Managing individual non-interleaved NVDIMMs provides high throughput which remains stable across varying value sizes and scales with the available NVDIMMs. However, existing PM key-value stores (§2.1) and PM file systems [240, 226, 114, 115] use the interleaving approach for simplicity and suffer from low throughput and scalability.

R2. Limit concurrent PM access. Increasing the number of reader or writer threads without bounds causes the buffers in NVDIMMs to fill quickly and induces head-of-line blocking overheads. Thus, it is critical to limit the maximum number of concurrent threads on each individual NVDIMM.

R3. Mix read and write I/O. Performing concurrent reads and writes provides better throughput and requires fewer threads to achieve peak bandwidth utilization (3.31 GB/s per NVDIMM).

R4. Limit cross-node traffic. Memory and thread sharing across multiple NUMA nodes results in cross-node traffic and introduces overheads from the high-latency low-bandwidth NUMA interconnect. Thus, minimizing cross-node traffic is crucial for high PM bandwidth utilization and scalable throughput.

4.1.4 Strawman solutions

Running a key-value store on a PM file system. One might wonder if running a traditional key-value store on a file system designed for PM results in good performance. We ran RocksDB [24] on OdinFS [240], a PM-customized POSIX file system

which aims to achieve high performance and scalability by retaining fine-grained control over PM. We find that RocksDB does improve by using OdinFS (up to 35%), but it is still significantly lower than the performance of PM key-value stores such as FlatStore. One reason behind this is that OdinFS does not support the `mmap` operation; most PM key-value stores access PM via memory mapping, since accessing it via read and write systems calls can be 6–16× slower. More fundamentally, even if OdinFS extends support for mmap, it cannot provide hugepages (as it relies on stripping PM across NUMA nodes for thread parallelism and scalability); hugepages directly impact the performance of memory-map applications [115]. Thus, one cannot obtain a write-optimized PM store by running a key-value store built for solid state drives on top of a PM file system.

Summary. We need a new PM store that takes advantage of the high and expandable bandwidth of PM and provides scalable write throughput to applications. Such a high-throughput PM store must be tailored to the nuanced performance of PM and must account for the best practices that enable high PM bandwidth utilization.

4.2 Skye: Design

We present Skye, a write-optimized key-value store for persistent memory (PM). Skye provides a simple interface with `put`, `get`, and `delete` operations. Skye supports strong consistency (linearizable reads and writes). Skye maintains crash consistency in the event of a crash and recovers efficiently.

The key insight powering Skye is that PM stores must maintain *fine-grained control over all PM accesses* to achieve high and scalable throughput. This idea follows observations from prior PM studies and the recommendations from our analysis (§4.1). Existing PM stores offload the control over PM I/O to hardware and applications. They rely on memory-controllers to distribute data across individual non-volatile DIMMs (NVDIMMs) and allow application threads to directly access the data on PM. Thus, existing PM stores harvest the low latency of PM. However, this

fundamentally trades off high throughput as PM stores no longer control how data is placed on individual NVDIMMs (violates R1), and how many threads access an NVDIMM concurrently (violates R2).

SKYE is the first PM store to have fine-grained control over all PM accesses and to leverage the high bandwidth of PM. It proposes an architecture with *indirect-access to PM for applications*. SKYE uses three main components to implement this architecture and follows the four design recommendations from our analysis.

Log interface for NVDIMMs. Instead of using a single hardware-managed PM device, SKYE manages individual NVDIMMs and implements a log abstraction on top of each NVDIMM. By writing to a NVLOG provided log, SKYE writes sequentially to NVDIMMs and makes data placement decisions at the granularity of each put request (R1).

Dedicated worker threads. Instead of allowing applications to access the data on PM directly, SKYE uses dedicated workers that perform I/O on behalf of applications. SKYE ensures that a fixed number of workers perform I/O on a single NVDIMM (R2).

Leveraging other media. With NVLOG and workers, SKYE efficiently process write requests. However, to enable efficient reads, SKYE leverages DRAM and disks. SKYE maintains in-memory indexes and checkpoints periodically to disks. Thus, SKYE avoids overloading PM and obtains good read and write throughput simultaneously.

NUMA awareness. SKYE adopts a NUMA-aware design. SKYE partitions data and metadata across NUMA nodes and uses workers from the same node to access the data. This enables scalable throughput across multiple NUMA nodes (R4).

4.2.1 Architecture: Indirect-Access to PM

SKYE is designed for machines with multiple non-uniform memory access (NUMA) nodes and multiple NVDIMMs per node. SKYE manages individual non-

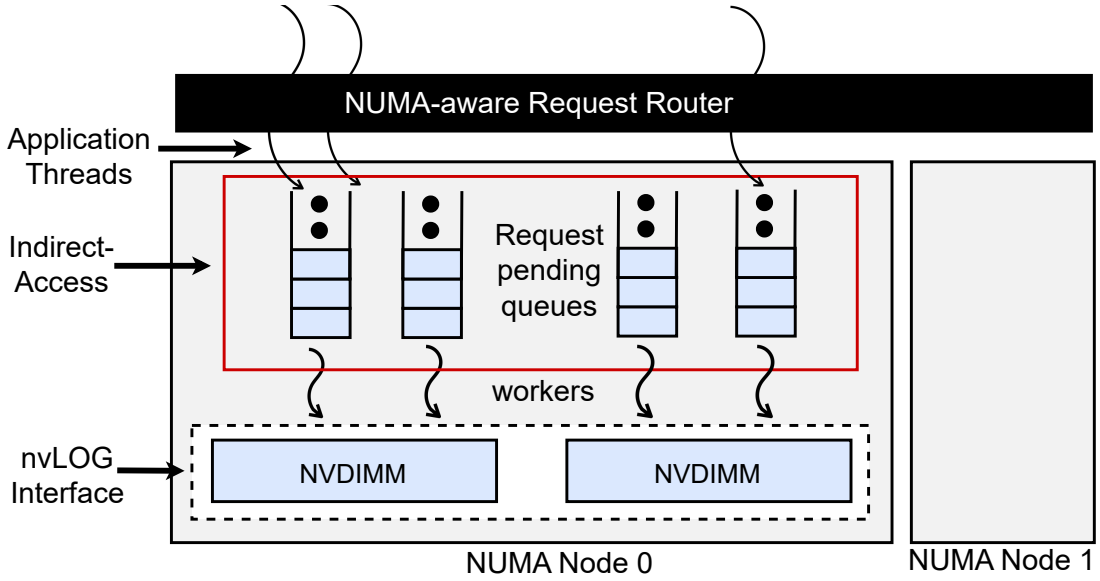


Figure 4.2: **Architecture**. SKYE uses a log interface to NVDIMMs (NVLOG) and provides indirect-access for applications. SKYE uses dedicated workers to process requests on behalf of applications and SKYE is NUMA-aware.

interleaved NVDIMMs and provides indirect-access for applications. Figure 4.2 illustrates the indirect-access architecture of SKYE.

First, SKYE exposes a log interface on top of individual NVDIMMs (NVLOG). Each NVLOG log tailors I/O to the nuanced performance profile of PM.

Next, SKYE uses dedicated workers that read and write to NVDIMMs using NVLOG. SKYE ensures that the number of concurrent threads accessing a single NVDIMM does not exceed a threshold. Each worker has its own read and write pending queues and processes the requests in those queues.

Application threads can enqueue requests into a worker's request pending queue and wait for their completion. SKYE employs a NUMA-aware router that directs the application threads to a specific worker's request pending queues. This indirect-access architecture allows SKYE to reclaim the fine-grained control from hardware and applications, which is crucial for achieving high and scalable throughput.

4.2.2 Log Interface to NVDIMMs

Skye manages PM as multiple, individual, non-interleaved PM devices and implements a log interface for NVDIMMs (nvLOG).

nvLOG presents a high-level interface on top of NVDIMMs. nvLOG allows Skye to create, read, write, and delete logs (append-only files) on individual NVDIMMs without worrying about low-level details like the complex performance characteristics of PM.

Traditionally, PM key-value stores have used interleaved PM. Interleaving combines multiple NVDIMMs into one PM device; sequential PM regions map across NVDIMMs in round-robin fashion at 4kB granularity. A file system is mounted on top of the interleaved PM device. PM stores memory-map files and store their data and metadata on PM. While using a single interleaved PM device per NUMA node is convenient, it gives up fine-grained control over the I/O at each NVDIMM, which is crucial for obtaining high throughput (§4.1).

nvLOG replaces this stack (interleaved PM and file system). Instead of a file abstraction, nvLOG exposes the log abstraction (reads and appends which translate to random reads and sequential writes to NVDIMMs). nvLOG is tailored for the performance characteristics of PM and the needs of Skye. Instead of one interleaved PM device, nvLOG manages multiple NVDIMMs each as its own PM device. nvLOG mounts a file system on each NVDIMM that it manages. nvLOG pre-allocates large files, zeroes-out and pre-faults them (so that page faults and the overheads from zeroing pages are not incurred in the critical path of writing to the logs). As a result, obtaining a new log from nvLOG is inexpensive.

nvLOG also implements efficient log reads and appends. For instance, nvLOG uses non-temporal stores [183] and AVX-enabled memcopies [105] as they enable high PM bandwidth utilization. nvLOG is aware of the block size of PM, so nvLOG batches log appends and writes to the media at 256 bytes or larger granularity. This prevents write amplification and avoids PM I/O bottlenecks.

4.2.3 Workers and Request Queues

Skye maintains a fixed number of workers per NVDIMM. Skye associates each data and metadata log with a worker.

Worker threads process requests on behalf of application threads. Application threads talk to a centralized router, which points them to a worker. The application thread then directly enqueues its request in the worker's read or write pending queue. To process writes, workers dequeue a write request, write values to the data log and corresponding metadata to the metadata log; nvLOG batches small writes to NVDIMMs. To process reads, workers dequeue a read request, and read its value from data logs.

Skye couples data storage with data accesses (by associating a set of data and metadata logs to one worker). Skye decouples PM accesses from application threads using dedicated workers and request queues. This may seem counter-intuitive as one of the advantages of PM is that applications can directly read or write to PM at low latency. However, to achieve high throughput and leverage the high bandwidth of PM, PM stores must carefully control I/O to PM; Skye maintains its own threads instead of allowing applications to access the data on PM. With indirect-accesses, Skye achieves high throughput that scales with increasing number of application threads.

NUMA awareness. Skye associates workers with a single data and metadata log on a particular NVDIMM. Workers always access the NVDIMMs in their own NUMA node.

CPU utilization. Using dedicated workers increases the CPU utilization of Skye. Prior PM studies recommend avoiding concurrent reads and writes on a single NVDIMM to ensure high write-bandwidth utilization; however, we observe that concurrent read and write I/O provides higher throughput with fewer number of workers. We use this in the design to lower the CPU utilization of Skye. Skye allows multiple work-

ers with data and metadata logs on the same NVDIMM to process read and write requests simultaneously.

4.2.4 Request Routing

Skye has a centralized request router that is shared across NUMA nodes. The router in Skye has two main functions: serializing requests, and routing application threads to workers.

Serializing requests. The router is the serialization point in Skye. All put requests are assigned a serial number that is persisted as part of the metadata, and get requests are assigned metadata.

Routing. For put requests, the router distributes application threads uniformly across the available workers. The router ensures that the write request load is distributed uniformly across NUMA nodes and NVDIMMs. For get requests, application threads are routed to a specific worker that manages the corresponding data log.

Read throughput. Skye partitions data and metadata across NUMA nodes and NVDIMMs. Thus, the router in Skye must lookup the latest metadata to locate the data log with the latest data to direct application threads to that worker's request queues. Skye returns the metadata of a write request once it is processed and allows applications to cache this metadata. Thus, application threads can pre-populate get requests with metadata and can enqueue the requests into the worker's queue; in this fast-path, router does not have to perform metadata lookups while routing the get request. Request routing is performed outside the throughput-critical path of Skye.

4.2.5 Leveraging DRAM and Disks

Skye maintains in-memory indexes that store metadata and help locate the data in nvLOGs efficiently.

We observe that PM indexes [135, 156, 244, 146, 102, 127, 215, 62, 242, 143]

provide low throughput (<10 Mops/s) as they do not account for the I/O bottlenecks from inefficient PM accesses. For example, PM hash tables perform small, random, mixed I/O on PM, resulting in low overall throughput. Thus, Skye employs DRAM indexes. These indexes offer considerably higher throughput relative to PM indexes and do not overload PM.

However, updating in-memory indexes in the critical path of processing put requests leads to poor performance. Instead, Skye writes index updates to an in-memory *merge log* in the critical path. Using these merge logs, indexes are updated in the background. Merge logs must be examined to fetch the most recent metadata while routing get requests; Skye uses bloom filters to lower this cost. With this design choice, Skye incurs additional overheads (merge log lookups) while routing get requests and achieves high throughput for puts (avoids index updates in the critical path).

Checkpoints. Since the data layer also persists metadata using nvLOG, Skye can use metadata logs to recover from a power failure or a crash. However, recovering purely from metadata logs requires reading complete logs. To avoid reading full logs and reconstructing entire indexes during recovery, Skye checkpoints indexes. To prevent checkpointing traffic from overloading NVDIMMs, Skye leverages disks and periodically checkpoints to disks.

4.2.6 Life of a request

We discuss the life cycle of a put and get request in Skye. Figure-4.3 illustrates the life of a put (in red) and a get request (in green).

Put Request

- The put request arrives at the router.
- The router assigns a serial number to the put request and routes it uniformly to workers across NUMA node.

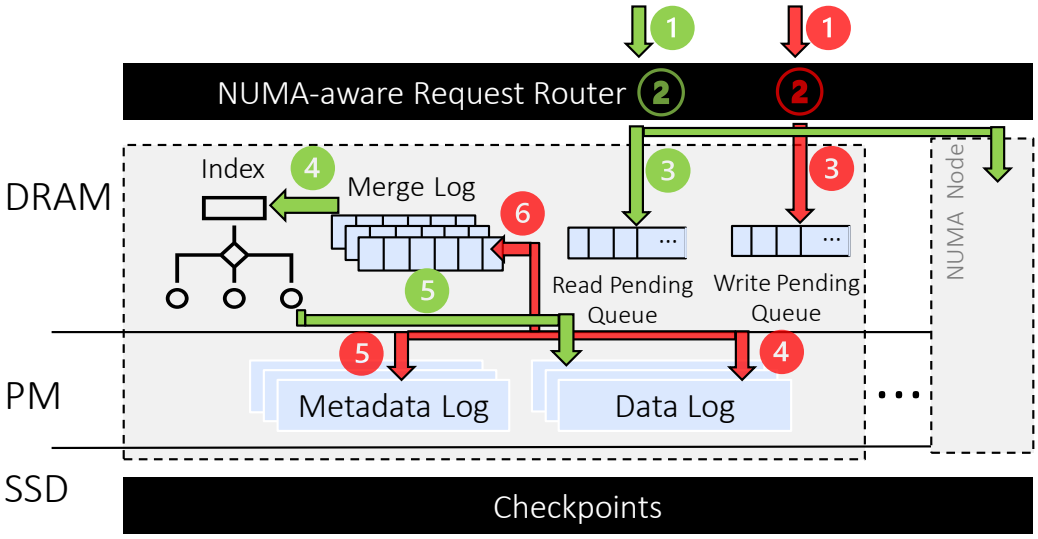


Figure 4.3: **Life of a request in Skye**. This figure shows the life of a put request (in red color) and get request (in green). The worker's data structures are in blue, per-node partitions in grey, and components shared across NUMA nodes in black.

- Application threads enqueue the request to the worker's write pending queue
- The worker dequeues a put request, appends value to the data log, and appends metadata to the metadata logs
- Finally, worker appends an index update record to its in-memory merge log and marks the request complete.
- Under low load workers wait until a few requests are enqueued to avoid contention on the queues and to reduce CPU utilization; this enables NVLOG to batch writes to NVDIMMs

In summary, in the critical path, workers dequeue requests, persist data and metadata to NVDIMMs using NVLOG and append index update records to their corresponding in-memory merge logs. A background merger thread reads the update records from the merge logs and asynchronously updates the index in the NUMA node. Note that application threads can cache the metadata from put requests and

send it with along with get requests (fast path).

Get Request

- The get request arrives at the router.
- The router directs the read request to the worker that manages the data log with the latest value of the key
- In the fast path, clients provide the latest metadata the key. Otherwise, router uses the metadata from all NUMA nodes (index and merge logs), finds the most up-to-date metadata; application threads enqueues the get request to that specific data log's worker
- The worker dequeues the get request, reads the value from the data logs on NVDIMMs, and marks the request complete

In summary, in the critical path, workers dequeue requests and perform a single PM read to fetch the value from the data logs. Note that application threads need not cache and pre-populate the metadata of get requests as the router looks up the metadata from all NUMA nodes to route a get request; routing is performed outside the (throughput) critical path of processing a get request.

4.2.7 Crash Consistency

Skye ensures that it can recover to a consistent point after a crash or a power failure. Skye orders updates and uses checksums to ensure crash consistency. Once the metadata is appended to the metadata log only then a key-value is considered committed.

If a crash happens before the metadata write is done, the data is lost; the data log entry will be overwritten by the next write. Partially written metadata entries will also be zeroed out. The data log and metadata log appends can get re-ordered at PM devices as Skye does not order data and metadata appends using fences (*e.g., sfence* [183]). Skye recovers consistently even in cases when metadata log is persisted

but the data log is not persisted. The metadata log contains a checksum of the data (value) so that Skye can detect partial writes, bit flips, and other forms of corruption.

nvLOG interface batches writes and issues fences after persisting a batch of data and metadata entries. Once data and metadata are persisted, a crash does not result in data loss. The in-memory metadata indexes will be reconstructed from checkpoints and the metadata logs.

4.2.8 Garbage Collection

To support deletes, Skye writes a new data entry with a higher serial number (provided by the router) indicating that the key is deleted. Skye garbage collects deleted keys when the key-value store starts up, by moving live keys to new data logs, and updating the metadata logs. Garbage collection is also triggered when there are no free log segments available.

4.2.9 PM Discussion

A natural question that arises is the relevance of Skye given that Intel is discontinuing its line of PM products.

The main contribution of this paper is our approach of empirically analyzing a new media, understanding how to effectively utilize it, and designing a high-throughput key-value store. We have demonstrated this for PM; PM requires fine-grained control over I/O to NVDIMMs to achieve high write-bandwidth utilization and Skye controls all PM accesses to achieve high and scalable throughput. The indirect-access architecture of Skye is a natural fit for products like Samsung's memory-semantic SSD [82]. The memory-semantic SSD has very similar characteristics to Optane PM. It has a built-in DRAM cache (like XPBuffer in PM (§2.1)) and is byte-addressable but has an underlying coarse-grained SSD media access. To show that Skye extends to newer media with minimal effort, we support traditional block-based SSD with a few lines of code changes within the nvLOG interface. In

the future, we envision Skye supporting newer byte-addressable persistent media, by extending its support for optimal logging. Our approach is also valid for PM expansion with CXL. The write bandwidth of real CXL [200] hardware is sensitive to I/O sizes, access patterns, and the number of concurrent accesses to the media. We demonstrate that Skye extends to CXL PM without any modifications (§4.5).

4.3 Implementation

We implement Skye in 10k lines of C++ code. By default, nvLOG maintains 1 GB-sized logs for data and metadata, uses non-temporal stores for log appends and AVX-512 [105] memcopies for log reads, and zero-pads appends to make them 8B-aligned. Skye maintains a fixed (configurable) number of workers per NVDIMM; each worker has its own read and write pending queues (RPQs); we use lock-free concurrent queues [51]. Each worker maintains its own RPQs to avoid concurrency overheads in the critical path. The workers in Skye process the pending requests in their RPQs and go back to sleep until the application threads notify the worker. Under low load, if request batching is enabled, the worker waits until a fixed (configurable) number of write requests arrive in its RPQs before it starts processing writes. Batching enables high throughput and better PM write bandwidth utilization. Workers avoid polling on the request queues to avoid contention on the queues, instead they get notified after requests are enqueued in their RPQs. We observe that RPQs have poor scalability with increasing number of application threads and we need scalable, high-throughput RPQs that are lock-free and wait-free to realize the benefits of indirect-access in practice. Skye uses FASTER [58] as its metadata index, implements in-memory merge logs to avoid index updates in critical path, and uses bloom filters [210] to optimize lookups on merge logs.

4.4 Limitations and Trade-offs

Skye provides indirect applications with indirect access to PM. With dedicated worker threads Skye increases its CPU utilization in exchange for high PM bandwidth utilization. Skye also allows trading off low latency for high throughput with configurable request batching. Skye aims at obtaining high PM write bandwidth utilization since saturating PM write bandwidth is easier than read bandwidth; PM has higher read latency than write latency.

4.5 Evaluation

We answer the following questions in our evaluation:

- What is the throughput of Skye? Does it scale to multiple NVDIMMs and NUMA nodes? (§4.5.1)
- What is the latency for Skye requests? (§4.5.2)
- How does Skye compare to state-of-the-art PM stores on workloads resembling real-world applications? (§4.5.3)
- What are the overheads and trade-offs in Skye? (§5.7.4)
- How do we tune the number of workers in Skye? (§4.5.5)

Experimental setup. We use a four-socket machine with 224 cores, 790 GB DRAM, and 3 TB of Intel Optane DC Persistent Memory, with Ubuntu 20.04 and Linux 5.16 kernel for our experiments. This machine has six 128 GB NVDIMMs per node, has Intel(R) Xeon(R) Platinum 8276 processor. To emulate CXL PM, we use a dual-socket machine with 16 cores, 125 GB DRAM, Intel(R) Xeon(R) Silver 4314 processor, and with 1 TB PM (four 128 GB NVDIMMs per node). To emulate CXL PM, we pin threads to one node and access NVDIMMs on the remote node (§4.5.1).

Comparison points. We compare Skye against three state-of-the-art PM stores: FlatStore [61], Viper [48], and ChameleonDB [238]. Since FlatStore and ChameleonDB

are not publicly available, we use their implementations from the authors of Pacman [214]. We use FlatStore-H (with a hashtable index) implementation and observe that the measured performance aligns with the performance reported in their papers. We use Viper's publicly-available code [49]. We also evaluate Skye against the following baselines: The peak PM bandwidth of a single NVDIMM of 2.2 GB/s for sequential writes and 5.85 GB/s for random reads. With six NVDIMMs in a NUMA node, the peak PM bandwidth of 12.7 GB/s for sequential writes and 31.5 GB/s for random reads.

Workloads and configurations. We use the YCSB benchmark suite [66] which consists of workloads that represent real-world applications. We generate YCSB workloads with 8B keys and 256B values (total size: 12.29 GB). We use YCSB's LoadA (write-only) with 100% writes, RunA (write-heavy) with 50% reads and writes, RunB (read-heavy) with 95% reads and 5% writers, RunC (read-only) with 100% reads, RunD (read-latest) and RunF (read-modify-write) workloads. We generate YCSB workloads with uniform distribution and with moderate request skew using the default YCSB Zipfian coefficient of 0.99. Skye and PM stores we compare against do not support the RunE (range-scans) workload. We compare Skye against PM stores on a single NVDIMM and with 6 NVDIMMs on a single NUMA node. By default, Skye uses 8 workers on a single NVDIMM. With six NVDIMMs on a single node, Skye uses 16 workers per NVDIMM. In our experiments, we use 8 application threads per NVDIMM by default.

4.5.1 Throughput and Scalability

We evaluate Skye on a single NVDIMM and discuss its scalability to multiple NVDIMMs across NUMA nodes. We also evaluate Skye on emulated CXL PM. We use YCSB LoadA and RunC workloads with uniform distribution of keys for these experiments.

Performance on a single NVDIMM. First, we configure the number of workers

# NVDIMMs	Mops/s	Scaling	% bw utilization
1	7	1x	89%
2	14	1.9x	84%
4	28	3.9x	86%

Table 4.2: **NUMA scalability**. The write throughput of Skye scales across multiple NUMA nodes. Skye achieves 3.9× higher throughput on 4 nodes compared to one node from its fine-grained control over all PM accesses.

Configuration	CXL PM	CXL PM (2×)
% write bw utilization	88%	86%
write (Mops/s)	7 (1×)	14 (1.9×)
write (GB/s)	1.9 (1×)	3.8 (1.9×)

Table 4.3: **CXL PM**. Skye throughput scales by 2× with two emulated CXL NVDIMMs; Skye utilizes up-to 88% of the write bandwidth of emulated CXL PM.

in Skye for a single NVDIMM. With 16 workers and 8 application threads, Skye obtains up to 88% of PM write bandwidth. Skye processes 7 Mops/s and has average request latency of 53us. With higher number of workers, NVDIMMs get overloaded which result in lower performance. In this setting, Skye can handle more than 48 application threads without drop in its throughput (Figure-4.1). In the same setting, Skye processes reads at about 6 Mops/s and has 30us latency, and obtains 25% of PM read bandwidth. Skye uses 16 workers, 1 background merger thread, and 16 application threads. This experiment uses 34 out of 54 available CPU cores on a single NUMA node on the server. Thus, the write throughput of Skye is bandwidth-bound on a single NVDIMM and is CPU-bound within a NUMA node. We now discuss the scalability of Skye performance to NVDIMMs across multiple NUMA nodes.

Performance across multiple NUMA nodes. Skye has scalable write throughput with increasing number of NVDIMMs across NUMA nodes. We scale NVDIMMs by 4× across four NUMA nodes, and measure the write throughput of Skye when it is configured to use 16 workers per NVDIMM and 8 application threads per node. In-

stead of a single NVDIMM in node-0, when Skye is configured to use two NVDIMMs across two NUMA nodes, Skye processes 14 Mops/s and increases throughput by 94%. On configuring Skye to use 4 NVDIMMs across 4 NUMA nodes, Skye processes around 28 Mops/s and achieves 3.9× higher throughput, as shown in Table-4.2. On a single NVDIMM, Skye processes reads at 9 Mops/s; this read throughput does not scale with increasing number of NVDIMMs. On scaling to multiple NUMA nodes, workers on NVDIMMs remain underutilized due to load imbalance of read requests. Note that with more workers, Skye increases its read bandwidth utilization.

CXL Discussion. To evaluate the impact of PM capacity expansion on Skye, we emulate CXL PM following CXL memory emulation from prior work [148, 29]. Note that recent CXL research [200] reports that sequential accesses (using load and non-temporal store instructions) to remote NUMA memory of a dual-socket server have higher performance and lower latency than real CXL memory; we assume that this observation translates to PM. We measure the peak write bandwidth of emulated CXL PM using the Intel Memory Latency Checker [172] and our microbenchmarks from PM analysis (§4.1). We observe that a single CXL-attached NVDIMM (CXL-PM) has a maximum write bandwidth of 2.23 GB/s (4 writers and 512B writes); CXL PM has similar sequential write bandwidth as PM.

Skye saturates 88% of the available write bandwidth with CXL-PM on YCSB LoadA workload. With CXL PM, Skye processes ≈7 Mops/s and writes at 1.9 GB/s. Further, with 2× the capacity of CXL PM, Skye saturates 86% of available write bandwidth. In this setting, Skye processes ≈14 Mops/s and writes at 3.8 GB/s. Therefore, by increasing CXL-attached PM capacity by 2×, Skye performance scales by 1.9× (Table 4.3). Skye supports CXL-PM without any modifications and shows how the benefits of Skye translate to persistent memory media with lower write bandwidth than PM. Since CXL performance is not similar to remote NUMA performance for other access patterns like reads [200], we only evaluate the performance of sequential writes (using non-temporal stores).

Mean Latency	FlatStore	Viper	ChameleonDB	TyrantKV	
				No Batching	Batching
Put (us)	3	7 (1×)	4	3	35 (5×)
Get (us)	3	3 (1×)	3	3	32 (10×)

Table 4.4: **Average request latency**. This table reports the average put and get request latency of PM key-value stores.

Ablation study. We discuss end-to-end impact of the design choices of Skye. Skye has up to 14% lower throughput across all YCSB workloads when run on a single interleaved (six NVDIMMs on a node) PM device instead of managing individual NVDIMMs. By managing non-interleaved NVDIMMs, Skye has higher benefits from hardware prefetching. Next, Skye uses fixed number of workers per NVDIMMs, implements indirect-access, and avoids overloading PM. However, without indirect-access, too many applications threads can simultaneously access an NVDIMM and overload PM. Skye experiences 17% lower performance on increasing the number of workers form 16 to 32 threads. Further, using threads of a remote NUMA node drops Skye performance by up to 4×. We note that nvLOG and batching also provide significant performance improvements.

4.5.2 Latency

We now discuss the latency characteristics of Skye. We use default YCSB LoadA and RunC workloads for these experiments.

Low latency settings. Workers process requests in batches by default; workers wait till a configurable number of requests arrive at their request pending queues before processing them. While batching increases the average latency of requests, it enables high throughput; Skye achieves a peak write throughput of 7.2 Mops/s (88% of PM write bandwidth) on a single NVDIMM (§4.5.1). Further, batching reduces CPU consumption and limits the contention on the request pending queues. If applications disable batching then Skye takes around 3us to process a put or a get request and

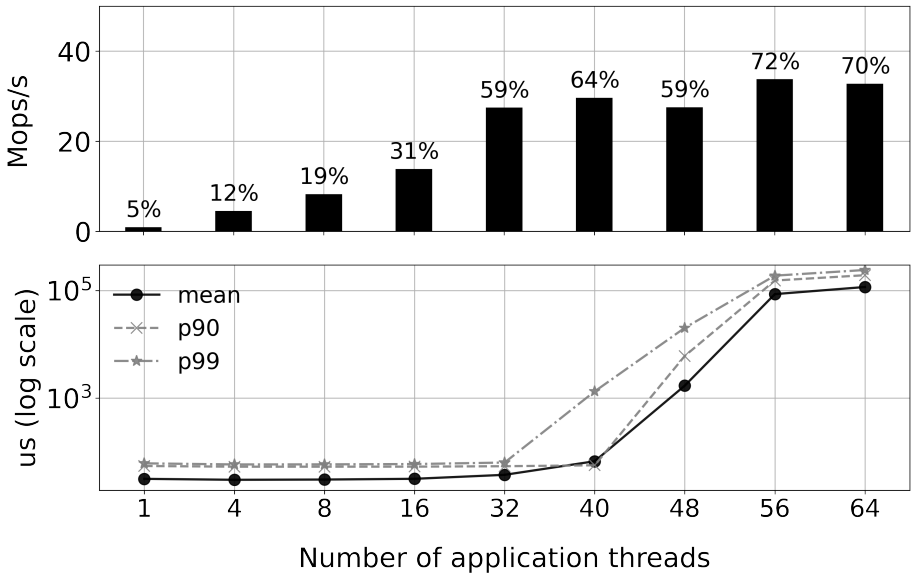


Figure 4.4: **Throughput and latency**. The throughput of Skye increases with the number of application threads, it saturates with >48 threads; latency continues to increase as request wait longer in the queues. Labels show PM bw utilization; Skye uses 48 workers (8 per NVDIMM).

has comparable latencies with the state-of-the-art PM stores, as shown in Table-4.4. Without batching, Skye has 30% lower peak write throughput; it processes 5.4 Mops/s on a single NVDIMM (66% of PM write bandwidth).

Put latency. With 32 application threads (high load) and in default settings, Skye takes ≈35us to process a put request. Application threads take 16% of this time to receive a serial number and to get routed to a worker's request queue. They spend ≈44% of the time to enqueue the requests, due to overheads from the concurrent queues. The workers in Skye take ≈39% of the time to dequeue to the request, append data and metadata to the logs on NVDIMMs, and update the in-memory merge log.

Get latency. Similarly, with high load, Skye takes ≈32us to process a get request. Skye spends >90% of this time to enqueue the request and to notify the worker which dequeues the request. The worker takes around 4us (<10% of the time) to

read the value from data logs on PM and to set the return value and its expected checksum. In the fast path, application threads cache the metadata from put requests and forward it along with the get requests. However, if application threads do not provide metadata (*e.g.,* serial number of the key), then Skye searches the merge logs and indexes for the latest value of the key and routes the application threads to workers; Skye takes ≈60us to process the request in its slow path.

Tradeoffs. There is a fundamental trade off between providing low latency to applications and obtaining high throughput. Indirect-access is crucial for achieving high throughput (limits the number of concurrent accesses to NVDIMMs), however, it requires additional work (like dequeuing requests) in the critical path and increases the latency of operations.

Throughput and latency discussion. Skye is designed for applications that require high throughput and can tolerate microsecond-scale latencies. For applications that can tolerate even higher latencies, Skye provides a throughput and latency trade off, as shown in Figure-4.4. Applications can scale their threads based on their latency SLAs. On a single NUMA node with six NVDIMMs and with 32 application threads, Skye processes ≈28 Mops/s with 8 workers per NVDIMM. Skye saturates ≈59% of PM write bandwidth while taking ≈37us to process a put request on average. In this configuration, Skye has a 99% tail latency of 63us. For applications that benefit from higher throughput and can tolerate higher latencies, Skye allows applications to scale their threads and provides higher performance. With 56 application threads, Skye processes 34 Mops/s with an average latency of 85ms per request and a 99% latency of 0.2s respectively, and obtains ≈72% of PM write bandwidth.

4.5.3 Yahoo Cloud Serving Benchmark

We compare Skye to prior PM stores. For each experiment, we load the PM stores with 50M key-value pairs and run YCSB workloads with 50M operations. The state-of-the-art PM stores run on a single interleaved PM device across six NVDIMMs

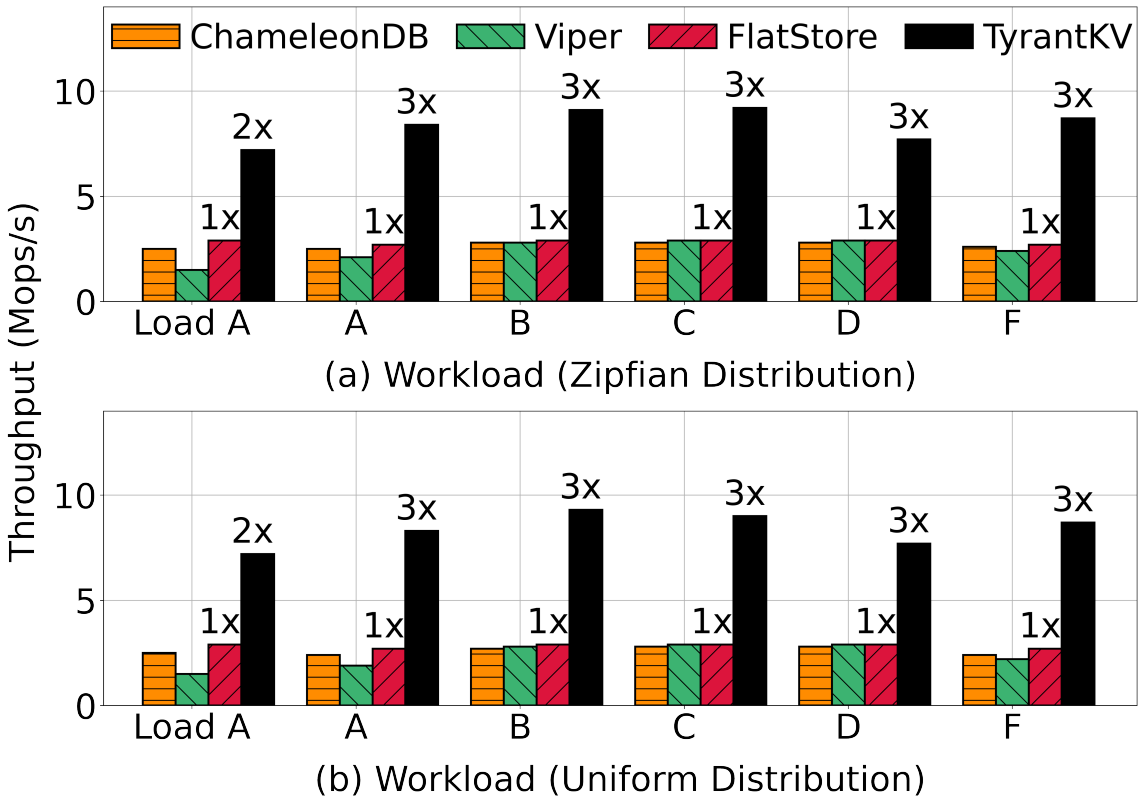


Figure 4.5: **Comparison against PM stores on a single NVDIMM**. Skye outperforms FlatStore for all YCSB workloads by up to 3×; this is from providing indirect-access for application threads and using logs from nvLOG.

in a NUMA node. Since Skye is configured to use 16 worker threads, other PM stores use 16 application threads; Skye uses 8 application threads to provide 30–35us latencies. We use the same number of threads to access PM across all PM stores; with higher number of application threads Skye has higher throughput.

Single NVDIMM. Skye outperforms state-of-the-art PM key-value stores by 3–4× on all YCSB workloads and achieves a throughput of at least 8.4 Mops/s for workloads with Zipfian distribution (Figure 4.5(a)) and about 7.2 Mops/s for workloads with Uniform distribution (Figure 4.5(b)). We observe that Skye has higher performance for Zipfian distributions relative to Uniform for read-dominant workloads; the request skew helps workers utilize higher PM read bandwidth.

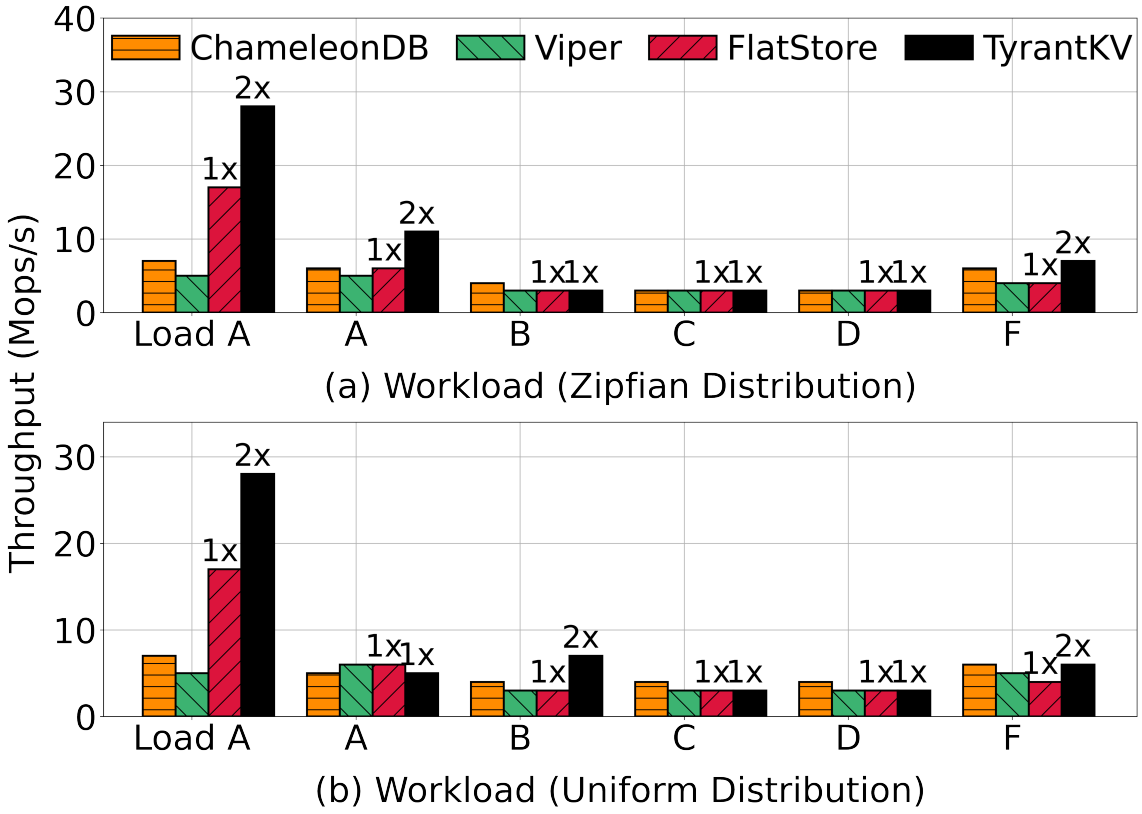


Figure 4.6: **Comparison against PM stores on a single node**. Skye outperforms FlatStore by 2× on LoadA and performs comparably for read-dominant workloads with six NVDIMMs on one node; the workers in Skye are underutilized due to low load.

LoadA. Skye outperforms current PM stores by 2× on write-only workloads. Skye trades off low latency for high and scalable PM write bandwidth utilization. Skye obtains about 88% of PM write bandwidth in this configuration and scales to multiple application threads without overloading PM due to its novel indirect-access architecture.

RunA. Skye achieves 8.4 Mops/s on RunA workloads with Zipfian distribution and about 8.3 Mops/s for workloads with uniform distribution. FlatStore, Viper, and ChameleonDB obtain 1.9–2.7Mops/s. Skye outperforms these PM stores by 3× on the write-dominant RunA workloads; Skye achieves better bandwidth utilization for

writes with nvLOG and its indirect-access architecture.

RunB. Skye processes 9.1 and 9.3 Mops/s for RunB workloads with Zipfian and Uniform distributions. Skye outperforms FlatStore by 3× that achieves 2.9 Mops/s. Skye outperforms ChameleonDB and Viper that processes 2.8 Mops/s by 3.3×. With 16 workers on a single NVDIMM, Skye utilizes the available read bandwidth better than existing PM stores (using dedicated workers) and hence achieves better read throughput.

RunC. Skye processes the read-only RunC workload at 9 Mops/s. It outperforms FlatStore, Viper, and ChameleonDB by 3× as they process 2.8–2.9 Mops/s. On a single NVDIMM with 16 workers and 8 application threads, Skye performance for reads is latency bound, however, it uses available PM read bandwidth better than existing PM stores (by using dedicated workers). Skye obtains about 25% of PM read bandwidth in this workload. With higher number of workers, we observe higher read throughput for Skye and better PM read-bandwidth utilization. However increasing the number of workers trades off write performance and increases CPU utilization.

RunD. Skye processes workloads with read-latest distribution at 7.7 Mops/s. Other PM stores obtain around 2.8–2.9 Mops/s. Skye outperforms them by 3×. With skewed reads, Skye achieves better read bandwidth utilization and hence outperforms PM stores.

RunF. Skye obtains a throughput of 8.7 Mops/s with RunF workload. FlatStore, Viper, and ChameleonDB obtain 2.2–2.7 Mops/s. Skye outperforms these PM stores by 3×. Skye does not support special read-modify-write operations. Hence, the performance of Skye for RunF workload is similar to workloads with similar put and get request distributions.

Summary. The state-of-the-art PM stores allow application threads to directly access PM and suffer from poor throughput and scalability. Skye uses dedicated threads to access the NVDIMM and uses efficient nvLOG to obtain high read and write

throughput. Skye supports higher number of application threads without a drop in throughput. We show that designing for high PM bandwidth utilization provides better read and write throughput.

Single NUMA node We evaluate Skye on a single NUMA node with six NVDIMMs. We observe that Skye has at least 2× higher performance compared to FlatStore on LoadA and has comparable performance on run workloads, with Zipfian (Figure 4.6(a)) and Uniform (Figure 4.6(b)) distributions.

Write-dominant workloads. Skye outperforms state-of-the-art PM stores like FlatStore by 2×, Viper by 4×, and ChameleonDB by 5×, on the write-only LoadA workload. For write-heavy RunA workloads, Skye outperforms other PM stores by 2×.

Mixed workloads. With mixed workloads, PM stores and Skye have much lower throughput relative to their throughput on a single NVDIMM. The PM read bandwidth is twice that of the write bandwidth while the latency is 2-3× that of the write latency. Therefore, accessing PM with 16 threads does not saturate its bandwidth; hence read throughput for all PM stores becomes latency-bound. In this setting, Skye has comparable performance to other PM stores. However, with higher number of workers, Skye achieves better read throughput since read bandwidth is better utilized with more threads. Thus, with multi-NVDIMM per NUMA node settings Skye read throughput is CPU-bound.

4.5.4 Trade-offs and Overheads

Memory utilization. Skye uses ≈8 GB of DRAM for 768 GB of PM per NUMA node (1%). Skye uses FASTER for its in-memory index per NUMA node and maintains 1GB-sized data and metadata logs on PM. We observe that, a majority of Skye's total memory footprint is from request queues; Skye uses per-worker read and write pending queues and merge logs. Skye uses per-worker structures and uti-

lizes more memory to avoid concurrency overheads in the critical path; scaling them further based on the number of application threads reduces these overheads at the cost of more memory.

CPU utilization. Skye builds on the insight that concurrent I/O to NVDIMMs provides similar throughput with fewer threads relative to avoiding them (contrast to prior recommendations). If Skye were to achieve high PM random-read bandwidth, it would require 4× higher CPU consumption; Skye trades off read performance for lower CPU utilization and higher write throughput. Skye performs concurrent reads and writes on NVDIMMs to achieve higher throughput with fewer number of workers per NVDIMM. Further, workers in Skye sleep till a fixed (configurable) number of requests arrive at their queues; this helps lower utilize CPU utilization and increases write throughput.

Recovery time. During recovery, Skye first reads records from data and metadata logs. Then its parses the records which takes 2.7× more time than sequentially reading from PM logs, and then reconstructs the in-memory structures which takes 15× more time. Overall, Skye recovers at 1.7 GB/s per NUMA node and is limited by the performance of in-memory metadata indexes.

4.5.5 Performance Impact of Tunable Parameters

Skye uses dedicated workers to provide indirect-access and allows applications to configure the number of workers. We discuss tuning the number of workers and its impact on the throughput and latency of Skye while increasing number of application threads. We use the default YCSB LoadA and RunC workloads and run Skye on a single NUMA node with six NVDIMMs.

Write performance. With a single application thread and when configured to use one worker per NVDIMM, Skye has a write throughput of ≈2 Mops/s. On scaling applications to six threads, write throughput of Skye gradually increases to ≈6 Mops/s. With 1–6 application threads, Skye observes an average put latency of

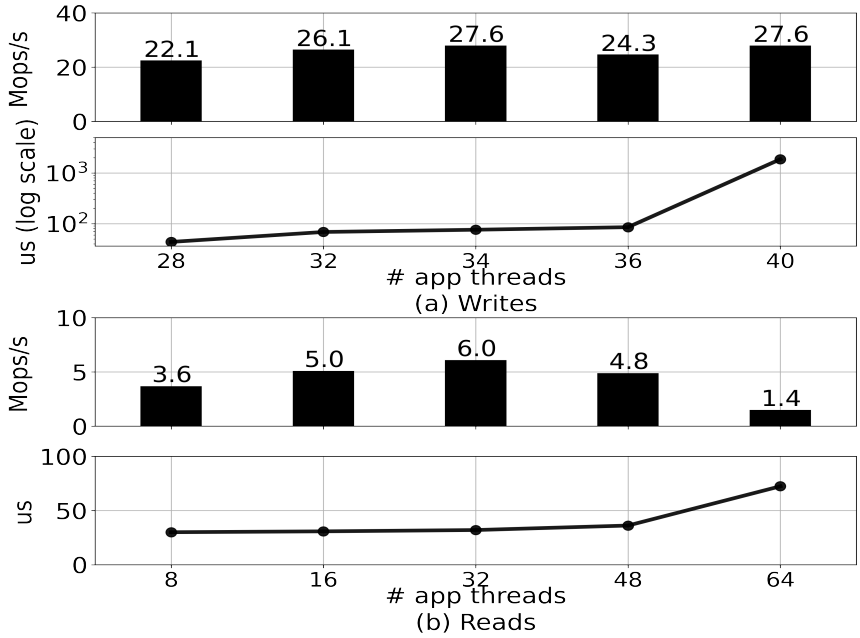


Figure 4.7: **Skye with 8 workers per NVDIMM**. The throughput and mean latency for put (a) and get (b) requests in Skye with increasing #app. threads. The write throughput of Skye is bound by PM bandwidth which causes writes to wait longer in queues with >40 app. threads (note the log scale); the read throughput of Skye is CPU-bound and latency increases gradually.

Workload	Puts					Gets			
#App. threads	1	2	4	6	8	1	4	16	32
Mops/s	2	3	5	6	6	2	4	3	2
Avg Latency (us)	33	34	31	43	75k	29	29	36	47

Table 4.5: **One worker per NVDIMM**. The throughput and mean latency of put and get ops in Skye with one worker per NVDIMM. The worker waits to batch under low load, and requests wait for longer in the queues with >6 app. threads.

31–43us, as shown in the Table-4.5. However, with 8 application threads, while write throughput of Skye increases to 6.4 Mops/s, the average latency of requests shoots up to 75ms. We observe that the sudden spike in latency is due to queueing delays; requests wait in the worker's queues for longer. Thus, Skye needs more workers to handle beyond six application threads.

Scaling workers. Increasing the number of workers improves the write throughput and latency of Skye. With two workers per NVDIMM and 8 application threads, Skye processes ≈9 Mops/s and takes 32us per request. With two workers per NVDIMM, Skye can handle <10 application threads before requests experience high queueing delays. On increasing the number of workers to four per NVDIMM, Skye handles up to 24 application threads. With 8 workers per NVDIMM, Skye process 27.6 Mops/s and scales till 34 application threads, as shown in Figure-4.7(a). We observe that increasing the number of application threads further does not improve the write throughput of Skye. Even with 40 application threads, write throughput remains stable at 27.6 Mops/s while the average request latency spikes as requests wait in the queues for longer (Figure-4.7(b); please note the log scale).

Read performance. With a single application thread and when configured to use one worker per NVDIMM, Skye processes reads at ≈2 Mops/s and takes ≈30us to process a get request on average (Table-4.5). Scaling application (till 16 threads) improves the read throughput of Skye. With higher number of workers (8 per NVDIMM) and with 32 application threads, Skye achieves a read throughput of 6 Mops/s and takes 32us per request. Beyond 32 application threads, Skye's read throughput decreases; note that the average request latency grows slowly, as shown in Figure-4.7(b). This behavior is primarily due to the poor scalability of queues. We observe that with 32 concurrent threads, the lock-free queue [51] has an average enqueue and dequeue latency of 23us. Even with a single thread, queues add an extra 2us latency while enqueueing and dequeueing requests (note: PM latency is ≈100–300ns). Thus, we need low-latency high-throughput concurrent queues to fully

realize the benefits of indirect-access architectures.

Write-optimization and trade-offs. The read throughput of Skye grows gradually relative to its write throughput with increasing number of workers; we observe similar trends in all PM stores (§4.5.3). This is fundamentally due to the higher read latency of PM; PM read latency is almost 2–3× higher than its write latency. Our analysis shows that PM requires up to 4–6× higher number of threads to effectively utilize PM read bandwidth relative to its write bandwidth (§2.1). However, increasing the number of workers to increase read throughput trades off write throughput (§4.1). Hence, workers in Skye are configured to achieve high write throughput by default; this trades off read throughput and reduces CPU utilization.

Chapter 5: Cascades

In this chapter, we empirically evaluate the I/O bottlenecks in distributed databases and outline their impact on transaction throughput, scalability, and their recoverability (§5.1). Next, we introduce Cascades, a novel distributed transactional store (§5.2), and Lattice, a logging infrastructure that simplifies recovery and its API for building recoverable applications (§5.3). Note that our contribution Cascades builds on Lattice, a logging framework that already existed at Microsoft Research. Then, we summarize the life of a transaction in Cascades (§5.4). Finally, we discuss the implementation of Cascades (§5.5) and evaluate its throughput and scalability for highly contented workload (§5.7). We show that Cascades achieves 25× higher throughput with Lattice while employing (replicated) ultra SSDs and for workloads with 100% contention. Lattice provides 2× higher performance benefits with premium SSDs that are slower and less expensive than ultra SSDs.

5.1 I/O bottlenecks from recovery logs

In this section, we first review the throughput and scalability limitations of distributed databases (§5.1.1). Next, we trace these limitations to I/O bottlenecks from writing to recovery logs in the critical path of processing transactions (§5.1.2). Finally, we revisit proposals that seek to mitigate these I/O bottlenecks and describe their challenges (§5.1.3) and motivate our contributions.

5.1.1 Performance limitations of distributed databases

We review the performance and scalability bottlenecks in modern distributed databases. Main memory databases suffer from poor performance for highly contented transactional workloads; they have hotspots (data records that are frequently accessed by a large number of concurrent transactions) that limit their parallelism and thereby

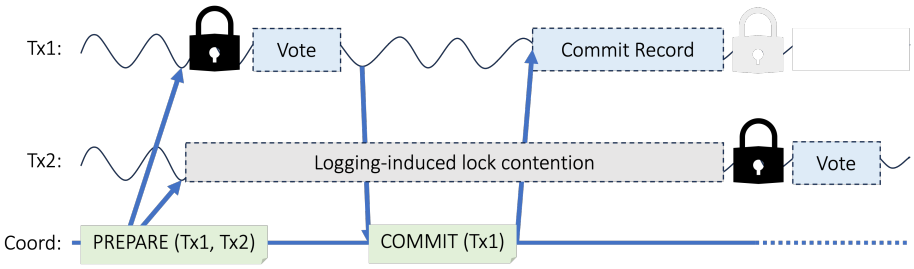


Figure 5.1: **Tx commit path in 2PC**. This figure illustrates a timeline for two transactions in modern databases; it shows I/O-related delays, and logging-induced lock contention; Tx1 and Tx2 are conflicting transactions, and grey stands for waiting, green for network communication, and blue for logging.

reduce their performance and scalability.

Highly contended workloads. Prior research on transaction processing and concurrency control has made significant progress on improving the performance of main memory multicore OLTP systems for low contention. However, these systems suffer from poor performance and scalability for workloads with high contention [35, 234].

Contending transactions perform conflicting operations on a few popular data records. Since executing conflicting operations in parallel while providing strong consistency proves challenging, OLTP engines serialize conflicting transactions across multiple cores, as shown in Figure-5.1. Thus, achieving linear scalability under highly-contented workloads is impractical. Prior research shows that adding more cores for highly-contented workloads results in a throughput drop proportional to the level of contention [170, 179, 216]. Thus, there is ongoing research on further reducing transaction conflicts by exploring novel dynamic data partitioning schemes to lower the number of transactions that span across multiple data partitions [170].

Dominant overheads: storage and network. Most studies analyzing the scalability of main memory databases to multiple cores study synchronization overheads, however, leave out the overheads from storage and network. Logging a commit record of transactions is the longest part of a transaction [109]. Further, with real-world applications requiring cross-zone replication to guarantee durability and availability,

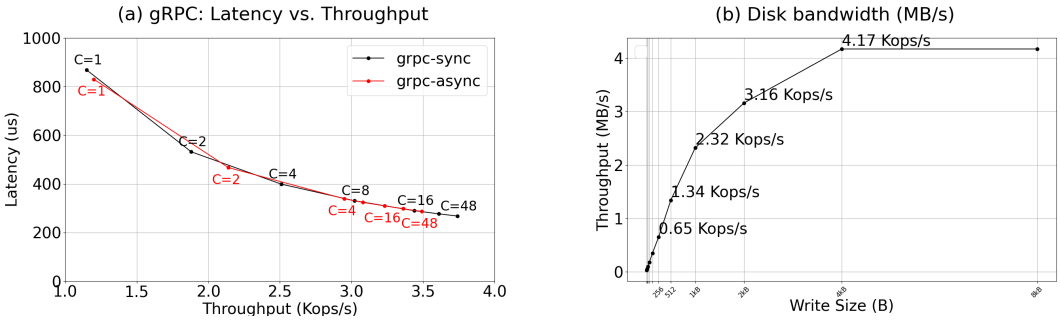


Figure 5.2: **Storage vs. network performance**. This figure higlights the network bottleneck with general-purpose communication systems like gRPC. Figure(a) shows the latency and throughput of gRPC with increasing number of client connections ($\#c <= 48$); gRPC supports $\approx$ 4kops/s. Figure (b) shows single-threaded logging throughput of $>$ 4kops/s on local SSDs.

the latency of a durable write to a replicated log is in the order of a few milliseconds. Moreover, with databases maintaining multiple replicas per data partition to enable high availability, they tend replicate the commit record of a transaction across replicas using consensus protocols like Paxos. Together, these I/O delays limit the performance and scalability of databases; we will discuss these I/O bottlenecks in more detail in this chapter. Note that modern databases adopt partitioned architectures for higher scalability, however, suffer from poor performance for distributed transactions as they incur higher delays from network communications.

Modern networks shift the bottleneck to storage. With slow gRPC, network communication has been the primary bottleneck as shown in Figure-5.2. This figure shows that the maximum throughput with gRPC is still much lower than the throughput of logging to local disks using a single thread. However, fast datacenter network systems like eRPC achieve orders of magnitude higher throughput relative to gRPC and have 2× lower latencies, as shown in Figure-5.3. eRPC shifts the bottlenecks to storage. Thus, I/O bottlenecks fundamentally limit the throughput and scalability of distributed databases today.

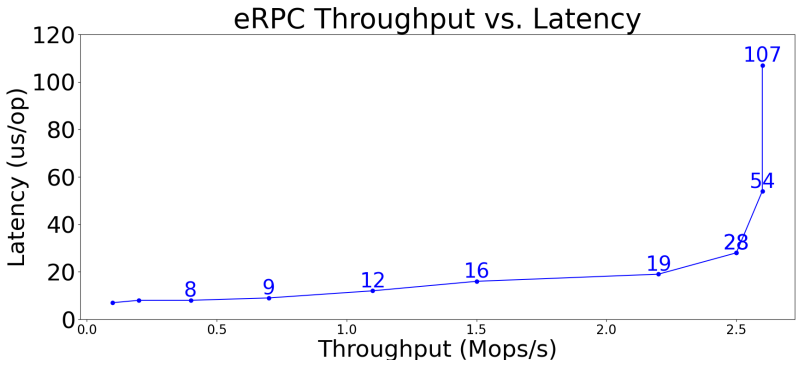


Figure 5.3: **eRPC performance**. This figure shows the throughput and scalability of eRPC. At peak throughput, eRPC supports 2.6 Mops/s (two orders of magnitude higher throughput than gRPC) and has 2× lower latency than gRPC. At low load, eRPC can provide ≈8μs round-trip latencies.

5.1.2 Synchronous durability of commit records

Most databases employ ARIES-style [155] write-ahead logs to persist the commit records of transactions for ensuring consistent recovery in the event of failures. These recovery logs introduce I/O overheads as shown in Figure-5.1; prior work [109, 81] breaks these down and quantifies these I/O bottlenecks.

Durable writes for recovery. It takes milliseconds to log a commit record to a zone-replicated or network disk and this latency is termed flush latency. Further, performing a large number of small-sized I/O results in higher latency. Note that databases often rely on logging across paxos-replicated servers in the critical path.

Log-induced lock contention. Most databases retain all locks until the commit record of the transaction is durably persisted and replicated. This causes a significant increase in lock contention and introduces additional performance bottlenecks. These bottlenecks result in serializing contending transactions for large durations of time, reducing the utilization of compute resources, and lowering the end-to-end throughput and scalability of databases.

Optimizations to recovery logs. These I/O bottlenecks result in poor utilization

Standard transaction	Early Lock Release Transaction
Take locks	Take locks
Perform work	Perform work
Commit transaction in memory	Commit transaction in memory
Flush transaction to log file	Flush transaction to log file (async)
Release locks	Release locks
Notify clients of the tx completion	Notify clients of the tx completion after the log flush competes

Table 5.1: **Early lock release (ELR)**. This table compares the transaction execution with standard execution and highlights the differences with ELR [80].

of compute resources [109] and limit the performance and scalability of databases. Most databases employ group commit [101, 173] and batch commit records to avoid overwhelming the disk. Further, they rely on asynchronous commits [159, 168] to reduce scheduling overheads. Moreover, databases adopt shared-nothing or partitioned architectures to improve their scalability and avoid overheads from centralized logs. However, they continue to face log-induced lock contention overheads that serialize the execution of contending transactions for long durations of time.

Early lock release to minimize log-induced lock contention. The early lock release (ELR) [74] proposes allows processes to release locks once a logging request for the commit record is made (to the centralized log), as shown in Table-5.1. Notifying the clients only after the commit record is durable (asynchronous operation completes) ensuring the same isolation guarantees for applications.

Correctness and consistency. With ELR, the next transaction cannot asynchronously persist a commit record before the commit record of its previous transaction. Once a request to log the commit record has been made, the transaction cannot be aborted due to other errors like disk capacity errors etc.

Fault tolerance. If a transaction aborts after requesting an asynchronous commit of its log record, then any transaction that has started after the failed transaction released

its locks must be aborted.

5.1.3 Potential solutions

We discuss two potential solutions to removing I/O overheads from the critical path of a transaction.

Early lock release (ELR). Early lock release [74] proposes an approach to completely eliminate long-induced contention overheads [89]. Follow-up research [194] discussed the correctness and proved the consistent recoverability of ELR in the event of different failures. Recent work [37] implements early lock release, with a single centralized log, and highlights promising performance benefits. However, this work does not discuss its generality to systems with multiple distributed logs (shared-nothing or partitioned architectures) and thus introduces scalability bottlenecks. For distributed logs and partitioned databases, this work does not discuss the fault tolerance (rollback mechanisms) and the durability of writes to ensure consistent recovery.

Trading off durability with higher availability. Prior research [199] discusses a unique approach of relying on data availability to avoid logging and associated overheads. It proposes replicating transaction commit records to multiple database instances (in-memory state machine replicas) to avoid logging for durability. Servers failover to hot standbys that ensure the availability of these commit records. However, this approach relies on additional compute and memory resources which are relatively more expensive than storage to provide durable transactions. Therefore, this dissertation deviates from the proposed approach by continuing to rely on logging for durability without facing the associated I/O bottlenecks.

Concerns with ELR. With databases writing to zone-replicated disks or networked disks for fault-tolerant logs, I/O requests could be lost, delayed, or appear to fail while they actually complete. These scenarios make recovery quite complex and impractical. For example, ELR requires additional support like memory-only rollback of committed transactions which most databases are not equipped to do [80].

Summary. Thus, we need a transactional database that supports efficient logging and replication, without trading of the simplicity of recovery or introducing the I/O overheads of durability. Further, we need a system that simultaneously achieves scalable performance via partitioning (multi-log systems), employs ELR (speculates on the durability of writes to avoid log-induced lock contention), and simplifies recovery and distributed rollbacks. This dissertation introduces Cascades to meet these goals and requirements. In summary, Cascades extends prior research on group or batch commit [101, 173], asynchronous commit [159, 168], and ELR [74, 194], to remove I/O bottlenecks while executing transactions.

5.2 Cascades: Design

Cascades is a distributed transactional store that is co-designed along with Lattice, its logging infrastructure, to enable scalable, high-throughput transactions and to simultaneously simplify recovery and replication.

5.2.1 Goals and guarantees

Cascades aims to achieve the following goals simultaneously:

- High throughput transactions that effectively utilize the available storage and network bandwidth
- Atomic transactions with strong consistency. The servers in Cascades observe a linearizable transactions and return the latest values on reads; other relatively-weaker consistency levels can be supported
- Simple and efficient recovery and replication without performing synchronous, blocking I/O in the critical path of processing transactions

Target applications. Many cloud applications like distributed databases [75, 241] maintain important information in replicated storage which is crucial for recovering

to a consistent state in the event of a failure. These applications tightly couple transaction processing and log appends (often alternate between the two) to limit the possible states of on-disk logs to recover from in the event of failures. Typically, applications implement their own recovery logs. Moreover, for workloads with high conflicts, these databases perform synchronous I/O to append to their recover logs in the critical path of process transactions. They are designed to interact with other entities only after the data in their recovery logs and replicas are durably stored, ensuring that other parties act on fully recoverable information, simplying recovery.

Performance and recovery tradeoff. Today, the latency of appending to a replicated log is in the order of milliseconds (up to ≈2ms) while the network latencies are in the order of few microseconds (≈10us). Thus, these applications tradeoff performance (about two orders in magnitude) for simplifying recovery. With Lattice, applications can effectively recover to a consistent system state in the event of failures without trading off throughput or scalability.

5.2.2 Architecture and system components

Cascades supports multiple primary servers that each manage and maintain a subset (partition) of application data. Each primary has a few, configurable number of replicas that they fall back on in the event of power, software, or hardware failures. Cascades assumes fail-stop failures.

Clients send transactions to a primary replica or coordinator. Cascades supports multiple coordinators for different transactions since having a single coordinator introduces scalability bottlenecks. In case of failures, primary replicas and the client connections to these primaries are reset. Thus, clients detect failures, query the progress of incomplete transactions, and resend transactions to coordinators.

In-memory system state. In Cascades, primaries and secondaries maintain an in-memory key-value store for application data. Each primary owns and maintains an application log that is used for recovery; secondary replicas share this application log.

CASCADES maintains these application logs to track the progress of transactions and to periodically checkpoint the key-value data to the log; these logs help reconstruct the latest system state during recovery.

Two-phase commit (2PC). CASCADES provides strong, linearizable consistency using two-phase commit [64]. Note that several alternatives protocols can also be used to achieve atomic and linearizable transactions [64, 129, 154, 130]. While there is significant research on minimizing I/O bottlenecks in scenarios with no conflicts, like batch commit and using asynchronous I/O [109], these databases fall back to performing blocking synchronous I/O to resolve conflicts and to provide strong consistency [234]. They rely on variants of the 2PC protocol to make forward progress without compromising on consistency.

CASCADES provides linearizable consistency using 2PC, highlights the performance impact of network communication relative to logging overheads, and shows the performance improvements from using the logging infrastructure LATTICE.

Lattice. CASCADES relies on LATTICE to manage logs and replicas, and for fault tolerance and recovery. The primary replicas in CASCADES log PREPARE, COMMIT, and ABORT records using LATTICE to ensure recovery. Note that CASCADES servers wait until the data is fully committed and recoverable before making it visible to clients providing the same consistency guarantees.

LATTICE supports a novel application programming interface *i.e.,* the recoverable application and recoverable processes interfaces. CASCADES implements these interfaces, simplifies recovery, and automates replication, without trading off performance or compromising on consistency.

eRPC and batching. CASCADES relies on eRPC for all network communication; eRPC is efficient and provides ≈10us latency for a network round trip. Further, all communications use batching to effectively utilize available network bandwidth; clients send a few batches of transactions and wait for their completion notifications

```
class IRecoverableApplication
{
        // New main application entry point, called on the
        // usual main thread
        virtual void Main(int argc, char *argv[],
             ApplicationLog *appendLogSessionManager,
                        LogScanner *logScanner) = 0;

        // Called by Lattice whenever a log record becomes
        // committed (guaranteed to be provided during recovery).
        // Called on a Lattice system thread, and will be called
        // while Main is executing on the main process thread.
        virtual void OnCommit(LSN committedLSN) = 0;

        // Called prior to Main to feed log records to the application
        // when it is running as a hot standby. Note that this is called
        // synchronously on the main process thread.
        virtual void RecoverLogRecords(LogScanner *newLogRecords) = 0;
};
```

Figure 5.4: **RecoverableApplication Interface**. CASCADES implements this interface to employ LATTICE and achieve high performance and simplify recovery.

from CASCADES. In CASCADES, coordinators perform 2PC on a batch of transactions.

5.2.3 Recoverable Application and Recoverable Processes

CASCADES implements the following interface to employ LATTICE, the logging framework. LATTICE has three application entry points, as shown in Figure-5.4; the Main, RecoverLogRecords, and OnCommit functions.

Main. CASCADES implements the transaction execution engine in its Main function. The servers in CASCADES establish connections with other servers and start listening for client transactions in this function. Once a request is received, the corresponding request handler is invoked.

RecoverLogRecords. CASCADES implements the RecoverLogRecords where it reads the PREPARE, COMMIT, and ABORT, records and updates its in-memory key-value

```
class RecoverableProcess {
  public:
    RecoverableProcess(IRecoverableApplication &applicationToRun,
        ILogWriterManager* logWriterManager,
        ILogReaderManager* logReaderManager,
        wstring instanceName,
        wstring instanceLocation,
        uint64_t logAdvanceTriggerSizeMB = 0);

    // Starts executing the recoverable process.
    // Start steals the caller thread which is used
    // to call the application main.
    void Start();
};
```

Figure 5.5: **RecoverableProcesses**. CASCADES launches its distributed servers each as a LATTICE recoverable process. A recoverable process registers a recoverable application and is launched by calling Start().

store accordingly. The RecoverLogRecords in CASCADES gets called in three cases: when CASCADES restarts from an existing execution trail, or when CASCADES incurs a failure and when LATTICE makes a secondary replica a primary, and once in Main to recover any pending log records from the application log. The secondaries keep recovering log records until they become primary and start executing their Main. Thus, secondaries can recover to a consistent state and continue processing transactions.

OnCommit. LATTICE calls into the OnCommit callback with an commitLSN to notify the client that the data until commitLSN is fully durable and will never be rolled back during recovery. CASCADES notifies clients of request completion in the OnCommit callback.

Recoverable application. A recoverable application implements the three virtual functions of the RecoverableApplication interface of LATTICE. The recoverable application is launched in a recoverable processes.

Recoverable process. Each server maintaining a data partition in CASCADES is

running a recoverable process with the recoverable application. A server and its replicas register the recoverable process with the same instance name and arguments; LATTICE automatically detects and manages these replicas. The recoverable process is registered with the recoverableApplicationToRun, as shown in Figure-5.5. The Start function of the RecoverableProcesses calls the Main of the RecoverableApplication, which first RecoversLogRecords and then listens for and handles requests from clients. Only the primary has write-access to the application log, while primaries RecoverLogRecords until they get exclusive write access to the application log and become the primary.

5.2.4 Speculation and recovery

Speculative execution. With LATTICE, CASCADES can make forward progress and process transactions before the appends to application logs become recoverable. Note that, even in scenarios with high conflicts, CASCADES servers can assume that the log records will be durable, and continue processing transactions. Thus, servers speculate on the durability of log records, and process transactions at network bandwidths, while relying on LATTICE to make the log records durable and recoverable.

Observable consistency. However, CASCADES notifies the completion of transaction to clients only after LATTICE ensures the recoverability of the log records, in the OnCommit function. Thus, CASCADES maintains the same observable consistency and claims orders of magnitude improvement in end-to-end throughput relative to other databases which rely on synchronous logging in the critical path.

Recovery: hot standbys. In the event of the failure of a primary replica, LATTICE automatically allows a secondary replica running a RecoverableProcess to take over as a primary. It automatically will rollback other primaries to a consistent point in time that undoes speculation on the transactions that are lost due to the failure.

Failures: clients. Any failure of a primary replica resets its clients. Overall, CAS-

CADES requires clients to handle such failure by querying the progress of the ongoing transactions and resending transactions that have not completed. Note that the results of speculation in CASCADES are not visible to the clients until LATTICE notifies CASCADES of the log records being recoverable via its OnCommit interface.

Secondary replicas. In CASCADES, secondary replicas are crucial for fault tolerance. However, they are not used for handling read requests. The requests from clients are handled only by the primary servers. CASCADES relies on LATTICE to update its secondary replicas with the recoverable log records.

5.3 Lattice: Design

This section outlines the design of LATTICE and its interface that supports recoverable applications. It describes how lattice enables data durability at high throughput while ensuring fast and simple recovery.

5.3.1 Goals and guarantees

LATTICE achieves the following goals and consistency guarantees.

- Recoverable applications: LATTICE is a logging library with an API that helps applications manage replication, handle failures, and simplifies recovering to a consistent state without trading off performance.
- High throughput: LATTICE is designed to effectively utilize the available storage bandwidth by employing asynchronous I/O. LATTICE prioritizes high throughput rather than low latency without complicating recovery.
- Consistency: LATTICE guarantees that a fully committed record would never be rolled back during recovery.

Infrastructure. LATTICE handles application servers of two kinds; primary and secondary servers. A primary server is responsible for managing and maintaining a

```
class ApplicationLog :
{
  // Creates a new concurrent session for appending records to the log
  LogSession* CreateSession();

  // Defines a new recovery point, which once fully committed
  // (cannot ever be rolled back), becomes active. Threadsafe.
  bool RegisterRecoveryPoint(LSN newLSNToRecoverFrom);
}

class LogSession
{
  LSN Append(ByteArray payload,
      ByteArray timestamp, ByteArray &outTimestamp);
}
```

Figure 5.6: **Lattice ApplicationLogs**. This figure shows the the application programming interface (API) of LATTICE application logs.

subset of application data. The primaries are responsible for persisting or appending new log records to LATTICE logs. A secondary replica is a standby server that is used when a primary server crashes or fails. LATTICE automatically registers secondaries when multiple servers are configured to process, append, and recover from the same unique LATTICE application log.

Failure assumptions. LATTICE assumes fail-stop failures of primary and secondary application servers (*e.g.,* power, hardware, or software failures). Failures cause the in-memory components to be lost while the data persisted on disks is not corrupted; RAID [59] helps ensure data integrity.

5.3.2 Lattice API

LATTICE applications have primary servers create an *ApplicationLog*. Its processes can create new and concurrent *LogSession* and can *Append* new data or *log records* to LATTICE logs. Figure-5.6 summarizes the application programming interface (API) of LATTICE.

Metadata. LATTICE creates an ApplicationLog at a specified datapath; it creates log files with read-write permissions and uses *LogScanners* to read from these log files. LATTICE allows applications to create new and concurrent sessions (*LogSession*) to *Append* new log records. With LATTICE, applications call *CreateSession* to create new and concurrent LogSessions and can *Append* new log records in parallel.

Log records and LSNs. In LATTICE, log records are byte arrays of data. LATTICE assigns a unique log sequence number (LSN) to every log record. Typically, applications consider log record to be units of recoverable information.

Appending new records. In traditional logging infrastructures, applications *Append* log records or byte arrays of data and receive an LSN that points to the location of the log record in the physical logs. Similarly, LATTICE returns a *Log Sequence Number (LSN)* for every appended log record.

Recovery points. LATTICE allows applications to use the *RegisterRecoveryPoint* call to mark a log record with the data required to reconstruct a consistent system state without having to scan prior log records. The beginning of LATTICE logs is always a recovery point. This call helps keep the recovery times of CASCADES in check; applications can periodically snapshot their system state and register new recovery points. Once applications append a log record that summarizes their current system state, they receive an LSN of this recovery log record (say newLSNToRecoverFrom), and can RegisterRecoveryPoint at this newLSNToRecoverFrom, as shown in Figure-5.6. Consequently, LATTICE reads the latest recovery point and scans the subsequent log records during recovery.

Timestamps of log records. LATTICE assigns a vector of timestamps to log records and allows applications to specify the dependencies across log records via these timestamps. The *outTimestamp* in the Append call returns the timestamp associated with the new log record; outTimestamp can be used to take dependencies on the log record being written.

Capturing the dependencies of log records. LATTICE allows applications to specify the dependencies of a log record using their vector timestamps. The append call takes in a new log record, a timestamp vector that specify its dependencies, and an outTimestamp that reads the timestamp of the new log record. LATTICE asynchronously persists these log records, ensures that the system state is consistent, and that the log record dependencies are met during recovery.

5.3.3 Transitive durability: Fully committed records

LATTICE asynchronously persists log records. LATTICE considers a log record committed if the log record and all its dependent log records are durably persisted; we term this *transitive durability*. In LATTICE, a transitively durable log record or a fully committed log record is never rolled back during recovery.

Vector timestamps. LATTICE allows applications to capture the distributed dependencies of a log record with a vector of LSNs that represents the bounds on the causally consistent rollback of all other servers. For instance, in a distributed system, this vector would consist of k LSNs if there are k ApplicationLogs, primary servers, or data partitions.

Multiple dependencies. If a newly written log record is dependent on multiple incoming messages from different processes, then the log record dependents on the maximum LSN from each of the k different partitions to fully capture all its transitive dependencies.

Computing fully committed log records. In traditional systems, if a log record or a unit of recovery is durable, then it used be committed or recoverable. However, with lattice, it is important for the log record and all its dependencies to be durable, for it to be fully committed or recoverable. To compute when transitive durability is met, LATTICE relies on notifications from other primaries. The latest log record that is transitively durable at each primary is notified to other entities; this can be implemented via a centralized server that is periodically updated. It tracks the

progress of transitive durability of other servers and computes the committed portion of its ApplicationLog.

Consistent rollbacks. LATTICE can rollback application logs to a causally consistent point in time in the event of a failure. For each log record, LATTICE helps bound the extent of rollback of each of the other primary servers using vector timestamps.

5.3.4 Recovery: Detecting and handling failures

Detecting failures. In LATTICE, timestamps are associated with epochs; an epoch identifies a round of recovery. If one of the application servers crashes and recovers, then it increments its epoch. Other servers (taking dependencies on the log records of the server with the originating failure) receiving the updated timestamps, detect the crash from its epoch number. LATTICE ensures that the failures are detected only by the subset of servers that take dependencies on the records that are part of the originating failure. Such nodes will fail themselves and initiate recovery.

Handling failures. LATTICE enables fast and simple recovery by leveraging its active secondaries or hot standbys. The secondaries are always updated only with transitively durable log records. Once a primary server crashes, this is detected by other dependent primaries and causes them to crash and recover as well. To recover, they fallback on their active secondaries which are up-to-date with the latest fully committed log records. Thus, LATTICE keeps recovery fast and simple.

Recovery and observable consistency. Each fallback server that is now the primary, notifies the last fully committed point in their ApplicationLogs to other servers. Other servers scrub or rollback the log records that have taken dependencies on records with higher LSNs. LATTICE notifies applications of the fully committed and recoverable log records by calling into OnCommit; applications can then make these computations visible to external entities (like clients).

5.3.5 Discussion

Storage media. LATTICE supports storing logs in several places like the local file system, distributed file system, or even in the cloud blob storage. LATTICE allows writing to zone-replicated ultra SSDs or premium SSDs, where ultra SSDs provide similar IOPs as premium SSDs at much lower latencies. However, ultra SSDs are more expensive than premium SSDs. Traditionally, applications had to use ultra SSDs for high throughput. CASCADES achieves high throughput at low costs by effectively utilizing premium SSDs with lower price and higher latencies.

Garbage collection To clean up portions of logs that are no longer required for recovery *i.e.,*, portions of logs that are behind the latest registered recovery point, LATTICE truncates its log files into a configurable size and starts new logs periodically; CASCADES configures logAdvanceTriggerSizeMB while registering as a RecoverableProcess.

5.4 Life of a distributed transaction

This section describes the life of transaction that spans across partitions in CASCADES that achieves high performance and simplifies recovery.

CASCADES uses simple two-phase commit protocol to implement atomic, consistent, and durable transactions. CASCADES servers use LATTICE to persist prepare and commit records that are crucial for recovery; they speculate on the durability of these records and claim high performance without modifying the observable consistency at clients.

Transaction Coordinator

1. Receives the transaction from the client.
2. Prepares the transaction by acquiring read/write locks for keys in the read/write sets respectively. Computes the number of participants (say all n) and sends

the transaction to the n participant servers

3. Waits till it receive a vote and vector timestamp (VT) from each server.

4. On receiving their votes and VTs, the coordinator *Appends* a PREPARE record that depends on the VTs from n participants and receives a vector timestamp from LATTICE; if all participants vote YES the coordinator COMMITs (otherwise ABORTs) the transaction.

5. Sends a COMMIT/ABORT message to the n servers and wait for their responses

6. On receiving the VTs of their COMMIT/ABORT records, the coordinator completes executing the transaction, also *Appends* a COMMIT/ABORT record that depends on the received VTs, receives an LSN, and marks this as the LSN of the COMMIT/ABORT record of the transaction.

7. The coordinator speculates on the durability of these records, releases the locks (makings its results visible to the internal n servers), and continues processing other transactions.

8. In the *OnCommit (LSN commitLSN)* callback, the coordinator notifies the clients of the completion of transactions with COMMIT/ABORT records that have LSNs <= the commitLSN.

Participants

1. Receives PREPARE request for a transaction from the coordinator.

2. Prepares the transaction by acquiring read/write locks for keys in the read/write sets respectively.

3. If successful, the participant votes YES. First, it *Appends* a PREPARE record and its vote in LATTICE, receives a vector timestamp (VT) as outTimestamp from LATTICE, and returns its vote and VT to the coordinator.

4. Receives COMMIT/ABORT request for a transaction from the coordinator.

5. Then the coordinator executes the transaction, *Appends* a COMMIT/ABORT record in lattice, receives a VT as outTimestamp and returns the VT to the coordinator.

6. The participant speculates on the log records becoming durable, releases all its locks (making the updates from the transaction visible to the n servers), and continues processing other transactions.

Handling failures

Designing CASCADES as a speculative recoverable application also allows it to recover consistently in the event of coordinator or participant failures. For instance, consider the scenario where the coordinator fails during the first phase. Note that the collection of n nodes in CASCADES that participate in the internal speculative execution include the coordinator and the participants (database partitions). Recall that, when one node fails and recovers, LATTICE will also fail and recover nodes which have taken dependencies on parts of the computation which were lost as a result of the originating failure. CASCADES notifies clients of transaction completion only after LATTICE commits the final COMMIT/ABORT record at the coordinator.

In this scenario, the originating coordinator failure cascades into all the participants and all of them recover to a durably consistent state. In this case, if only the part of the computation till phase-I is made fully durable (including all its dependencies), even if the computation had actually proceeded further, with some partitions even maybe logging commit messages, all participants will be rolled back to this point, and execution continues as if we were operating non-speculatively, and the coordinator and partitions had failed and recovered to this point. Since a commit or abort was never leaked to the client, this situation is no different, from the client's perspective, from a failure at the recovered point in the computation.

Conflicts and high contention

Although the n servers may be speculating on log records becoming durable, the coordinator waits in the OnCommit callback for the log records to be fully committed or transitively durable before the final notification is sent out to the clients; this prevents side effects from being visible to clients or external entities outside the collection of n servers. In highly contented settings, transactions are executed almost sequentially, one after the another. In such scenarios, without Lattice, coordinator has to wait to fully commit the PREPARE and COMMIT/ABORT log records before executing other transactions and notifying the clients. With Lattice, although the clients are notified after the records are fully committed (no change in observable consistency), transactions can speculate on the durability of PREPARE and ABORT/COMMIT records, and achieve high throughput. For highly contented transactions, Cascades achieves higher performance improvements.

5.5 Implementation

We implement Cascades in C++. Cascades, a transactional store, is a prototype application that is built using Lattice; Cascades builds on Lattice which was implemented as a logging library. Currently, Cascades and Lattice are built to run on Windows 2019 servers. Cascades relies on Lattice for logging, managing replicas, and for fault tolerance. Lattice uses AVX bit vectors to assign timestamps and track dependencies of log records while computing fully committed log records. Cascades employs the eRPC networking library for all network communications. Cascades servers use the C++ port of FASTER as their in-memory key-value store to manage their data partition. eRPC provides fast, reliable, general-purpose RPC interfaces. It supports DMA capable message buffers and zero-copy packet I/O to provide performance comparable to low-level interfaces such as DPDK and RDMA. We plan to make the prototype of Cascades, 10K lines of C++ code,

publicly available on GitHub[1].

5.6 Limitations and Trade-offs

Cascades achieves high throughput and trades off transaction low latency. Cascades processes batches of transactions and delays their commit notification to the clients without blocking other transactions from making forward progress. Thus, Cascades also achieves high scalability and strong consistency. In the event of a crash of a primary replica, Cascades achieves efficient recovery by inducing cascading failures to other primaries that have taken dependencies on failed transactions. Thus, Cascades aborts more transactions to handle efficient recovery; we leave testing or verifying the correctness of recovery in Cascades and dependency tracking in Lattice as part of the future work.

5.7 Evaluation

In this section, we evaluate the performance of Lattice and Cascades. We seek answers to the following questions.

1. What is the available network and storage bandwidth, and what is the throughput of appending log records in Lattice? (§5.7.2)
2. What is the end-to-end throughput and latency of Cascades for workloads with super hot keys and conflicting transactions? (§5.7.3)
3. How significant are the overheads from Lattice? (§5.7.4)

5.7.1 Experimental setup

Cascades is evaluated on the following testbed with three servers and one client per server. Each server in Cascades is run on a virtual machine, which runs

[1]https://github.com/microsoft/RecoverableProcesses

the Windows 2019 Server. These VMs are run on a bare metal instance with the following hardware configuration. Each server has an AMD EPYC 7402P 24-Core Processor with 48 threads, 128 GB of DRAM, and 1.6 TiB of NVMe SSD. It has two dual-port Mellanox ConnectX-5 (CX5) NICs that support 25Gbps and 100Gbps links. The Windows VMs have direct access to the CX5 NICs via host PCIe passthrough.

Workloads We measure the performance of Cascades against two kinds of workloads. First is a workload with simple transactions where the coordinators can process the transactions using the data within their partitions. Second is a workload with complex transactions where the coordinating server relies on up to two other primary servers for executing them. Complex transactions are processed using the 2PC protocol and show the performance improvements from Lattice when there is significant network overheads from 2PC and highest amount of conflicts; each transaction conflicts with ongoing transactions. These workloads have ≈7–8 key-value updates per transaction on average. The updates to the key-value pairs in each partition follow a uniform random distribution.

Comparion points. We measure Cascades against several configurations for recovery and replication. Cascades-NoLog represents no logging. Cascades-Sync represents having synchronous logging while processing batches of transactions. Cascades-ASync represents having asynchronous logging and batching, however, there is no simple way of recovering from failures. Cascades relies on Lattice for asynchronous logging and simplifying recovery.

We also compare Cascades against different levels of replication. We measure the throughput of Cascades on logging to premium and ultra SSDs which are replicated across availability zones. They are Zone-Redundant Storage (ZRS) services provided by Azure.

	Local SSD	Premium SSD	Ultra SSD	eRPC
Latency	50us	450us	2ms	7us
Throughput	3 kops/s	30 kops/s	30 kops/s	2.6 Mops/s

Table 5.2: **Microbenchmarks**. The latency and throughput of writing to different storage media with a single thread and the eRPC networking layer.

5.7.2 Microbenchmarks and Lattice

Logging. The latency of writing 8B records to premium, and ultra SSDs with a single thread are 50us, 450us, and 2ms respectively. While the local SSD supports around 3 kops/s premium and ultra SSDs support 30 kops/s, as shown in Table-5.2. With ZRS replication, Azure synchronously replicates data to three availability zones within the same region; each availability zone is a separate physical location with independent power, cooling, and networking.

Networking. With eRPC, servers are able to communicate with each other with a round-trip time of 7–9 us, and a maximum throughput of 2.6 Mops/s (each op is ≈8B), saturating the 25 Gbps network bandwidth.

Lattice performance. We observe that Lattice saturates the write bandwidth of the underlying storage media while appending log records. We measure that Lattice can *Append* ≈10 M records/s on Premium and Ultra SSDs. We show that the latency of recoverably committing these log records is dependent on the latency of the underlying media and the batching of Lattice. We observe that Lattice takes 450us, and 2ms to commit a record on premium, and ultra SSDs respectively.

5.7.3 Cascades: End-to-end performance

We show that Cascades achieves 25× higher throughput with ultra SSDs relative to its synchronous logging variant; synchronous logging variant represents the behavior of state-of-the-art databases; relative to its synchronous variant, Cascades obtains 3.5× lower latency. Cascades obtains simple recovery and automated fault

Media	Configuration	Throughput Mops/s	Avg ms	p50 ms	p90 ms	p95 ms	p99 ms
	No logging	0.134	15	15	15	15	16
UltraSSD	Synchronous	0.004	553	546	562	562	578
	Cascades	0.099	156	156	156	156	163
PremiumSSD	Synchronous	0.001	2541	2546	2547	2548	2562
	Cascades	0.099	641	640	641	641	656

Table 5.3: **End-to-end performance of Cascades**. With ultra SSDs, Cascades achieves 3.5× lower latency and 25× higher throughput relative to its synchronous logging configuration. With premium SSD, Cascades achieves 4× lower latency and 99× higher throughput relative to its synchronous logging configuration. Overall, Cascades obtains 74% of throughput achieved when logging is disabled.

tolerance with 10× higher latency relative to its no logging variant. However, with premium SSDs, a cost-efficient storage media, Cascades obtains 99× higher throughput relative to its synchronous variant; with premium SSDs, Cascades has 5× lower latency relative to its synchronous variant. On premium SSDs, Cascades manages replication with an additional 42× higher latency. Overall, Cascades obtains 74% of the throughput obtained when logging is disabled, as shown in Table-5.3.

Cascades achieves 25–99× (up to two orders of magnitude higher throughput) and 3–5× lower latency relative to performing synchronous I/O in the critical path of processing transactions. Cascades presents a novel way of achieving high throughput even for workloads with high conflicts, without trading off consistency or complicating recovery.

5.7.4 Overheads

Cascades achieves up to two-orders of magnitude higher throughput. We measure that Lattice provides high throughput while simultaneously simplifying logging and recovery. We evaluate that with Lattice, Cascades has high throughput which is comparable to the throughput achieved without any logging. Further,

the latency of transactions in CASCADES is 45% higher relative to the latency achieved with performing synchronous logging per batch of transactions. Due to overheads of LATTICE from tracking dependencies and computing fully recoverable points in application logs, CASCADES incurs 30% higher latency compared to the state-of-the-art; p99 latency in CASCADES increases by up to 70% relative to the state-of-the-art.

Chapter 6: RainBlock

In this chapter, we empirically evaluate the I/O bottlenecks in Ethereum [16] and outline their impact on transaction throughput and scalability (§6.1). Then, we introduce RainBlock[1], a novel architecture for public blockchains (§6.2), and the novel Distributed, Sharded Merkle Tree that enables efficient and scalable data authentication (§6.3) at miners. Later, we discuss the trust assumptions, security, and limitations, of RainBlock (§6.4). Finally, we discuss the implementation of RainBlock and DSM-Tree (§6.5) and evaluate their throughput and scalability across various workloads, including workloads that resemble transactions in Ethereum (§6.7). We show that a single RainBlock miner processes 27.4 Ktxs per second (27× higher than a single Ethereum miner). In a geo-distributed setting with four regions spread across three continents, RainBlock miners process 20 Ktxs per second.

6.1 I/O bottlenecks from authenticated storage

In this section, we first discuss the poor performance and scalability of Ethereum (§6.1.1). Next, we perform an empirical study to highlight the root cause of these performance limitations (§6.1.2). Then, we trace these performance limitations to I/O bottlenecks from to the design of the authenticated storage system in Ethereum (§6.1.3). Finally, we discuss a few strawman solutions and their drawbacks (§6.1.4) and motivate our work, RainBlock and DSM-Tree.

6.1.1 Performance limitations of Ethereum

Ethereum has an end-to-end throughput of 10-12 transactions per second [16], which is orders of magnitude lower than centralized transaction processing engines like

[1]This chapter describes the following work that is published at ATC'21 [167]: *RainBlock: Faster Transaction Processing in Public Blockchains.*

Visa [11]. Further, increasing the number of servers processing transactions does not increase its overall throughput. The poor performance and scalability of Ethereum is due to the low rate of *processing transactions*.

Processing a transaction. Processing a transaction involves reading and updating multiple values in the system state. In public blockchains, once a server receives a new block, it processes the transactions in that block and also verifies if its Merkle root matches the Merkle root in that block. We study the I/O bottlenecks in the critical path of processing transactions in Ethereum.

How does Ethereum process transactions? Servers that add blocks to the Ethereum blockchain are termed *miners*. Miners receive transactions submitted by users, execute these transactions, and group them into a block. We term executing transactions and grouping into a block as processing transactions. Several miners compete to add a block to the blockchain. Each miner tries to solve a Proof-of-Work (PoW) puzzle; if it solves the puzzle, it attaches the solution to the block and broadcasts it to other miners. Other miners verify the solution and begin to build on top of the received block. A transaction is *confirmed* or *finalized* once ten blocks are built on top of the block containing the transaction.

At a high level, a miner has two threads as shown in Figure 6.1. One worker thread is processing transactions and grouping them into a block. The other sealer thread obtains the transactions from the worker thread, and then tries to solve the PoW puzzle. The PoW consensus has miners emitting a block every 10–12 seconds, for example. While the sealer thread is working on one block, the worker thread tries to get the next block ready. Thus, the worker sealer has about 10–12 seconds to process transactions; if it can process transactions faster, it can pack more transactions into the block it passes to the sealer thread.

Low transaction throughput. Ethereum and other public blockchains suffer from low throughput: only tens of transactions added to the blockchain per second. The

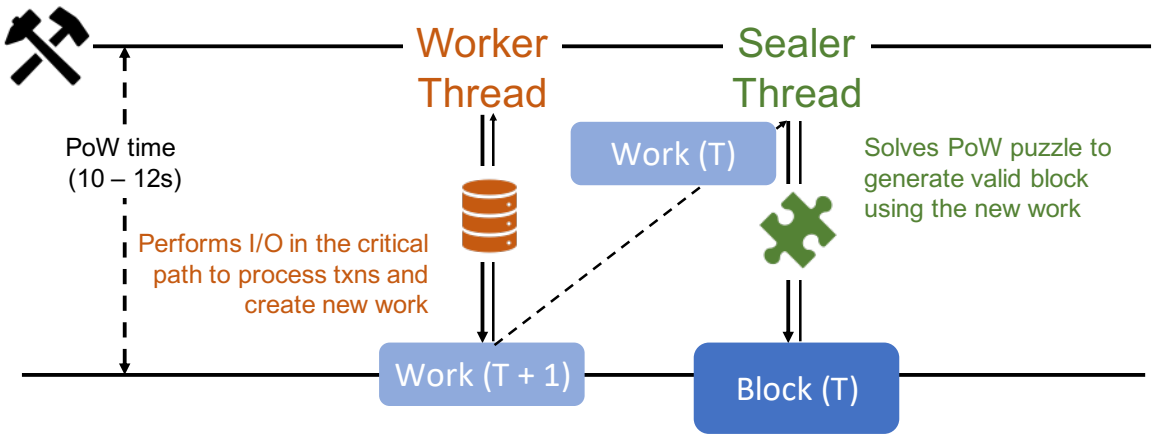


Figure 6.1: **How Ethereum miners work**. The worker thread processes transactions, packages them into a block, and hands them to the sealer thread. The sealer thread takes 10–12 seconds to solve the PoW puzzle; the worker thread must process transactions in this time-frame. I/O bottlenecks result in worker threads packing fewer transactions into each block.

low throughput comes from two factors. First, the PoW consensus limits the block creation rate to one block every 10–12 seconds so that a majority of miners can receive and process a block before a new one is released; this ensures that every miner is building on the same previous block, preventing forks in the blockchain. Note that while PoW consensus limits the block creation rate, it doesn't directly limit the size of the block. The second factor lowering throughput is transaction processing time. In each miner, the worker thread has to group transactions into a block within 10–12 seconds; the rate at which the worker thread can process transactions limits the maximum size of the block.

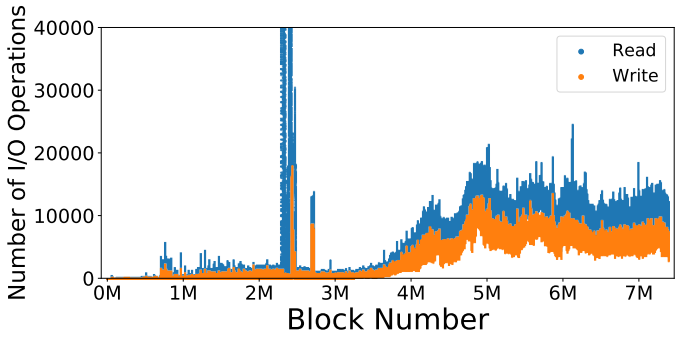
Authenticated storage. Processing a transaction involves executing the transaction and modifying system state such as account values. Since miners do not trust each other in a public blockchain, miners *authenticate data* and can prove that the data they provide is correct. This is done by maintaining a Merkle tree [150] over the data and publishing the latest Merkle root in the blockchain; another miner is able to independently verify that the data is correct, using the data and a vertical path in

Metric	No state	State: 10M	Ratio
Time taken to mine txs (s)	1047	6340	6× ↑
# of txs per block	2150	833	2.5× ↓
tx throughput (tx/s)	28.6	4.7	6× ↓

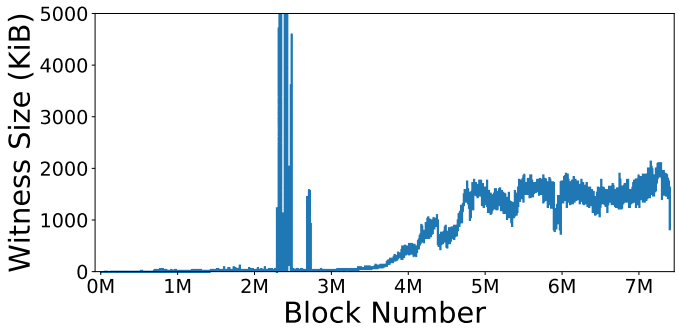
Table 6.1: **Impact of authenticated storage on e2e throughput**. The table shows the throughput of Ethereum with proof-of-work consensus in two scenarios when 30K transactions are mined using three miners. In the first scenario, there are no accounts on the blockchain. In the second scenario, 10M accounts have been added. Despite no other difference, transaction throughput is 6× lower in the second scenario; we trace this to the I/O involved in processing transactions.

the Merkle tree (termed a *witness*).

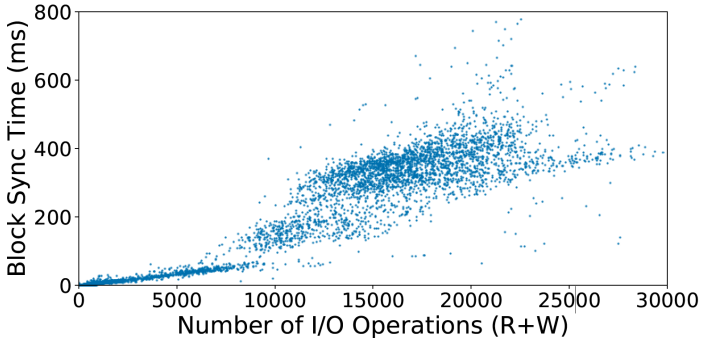
Scalability limitations. Unfortunately, *accessing authenticated data becomes more expensive as the total size of the authenticated data increases* [222]. As a result, transaction processing increasingly becomes bottlenecked on I/O. We demonstrate this with an experiment. We create two private Ethereum networks using the Geth client [26]; each network has three miners, 30K transactions to mine, and the same proof-of-work (PoW) configurations. While the first network has only 3 miner accounts (total size: 220 MB), the second has 10M additional accounts (total size: 4 GB). Note that Ethereum currently has ≈130M accounts (total size: > 400 GB [25, 27]) in its blockchain state. Overall, the second network takes 6× more time and 2.5× more blocks to mine all transactions using PoW consensus (Table 6.1). Using profilers, we see that in the second scenario, the time spent solving the PoW puzzle has not increased disproportionately; however, the worker thread takes 6× more time to process transactions, and 69% of the time is spent in accessing and updating the system state. Thus, the miner's transaction processing rate depends on the system state; this impacts the block size and overall throughput.



(a) Number of I/O operations



(b) Witness sizes



(c) Block processing latency

Figure 6.2: **Overheads of the MPT**. This figure highlights the I/O bottleneck due to Merkle trees. (a) First, it shows the number of IO operations performed per block. (b) Then, measures the size of witnesses (represents the amount of data read) per block. (c) Finally, it shows the increase in block processing time with the increasing number of I/O operations (I/O bottleneck).

6.1.2 Empirical study

We use Parity 2.2.11 [3], a popular Ethereum blockchain client, for studying the I/O bottleneck and I/O amplification from Merkle trees. We use Parity to initialize a new server that joins the Ethereum network, and replays the blockchain (until 7.3 Million blocks) to measure various costs. We analyze the performance impact of the Merkle tree in terms of the number of I/O operations performed and witness sizes. We also study the data structure's effect on the block processing time.

Number of I/O operations. For processing a single block with around 100 transactions, Ethereum requires performing more than 10K random I/O operations (two orders of difference). Most of these I/O operations are performed for reading and updating the Merkle tree. Figure-6.2(a) shows the number of I/O operations incurred per block while processing the first 7.3 Million block of transactions. This result shows the combined overheads from the Merkle tree and its on-disk layout using RocksDB.

Witness sizes. Witness sizes represent the amount of data read and modified per block while reading and updating paths in the Merkle tree. In Ethereum, which uses secure 256-bit cryptographic hashes, the witness size of a single 100 byte user account (or value) can be above 4 KB, showing a 40-60× overhead. The witness size also increases as the total data in the Ethereum state increases, as shown in Figure-6.2(b).

Block processing time. We measure the time taken to process each block, *i.e.,* executing the transactions in that block, and verifying if the resultant local Merkle root matches with the Merkle root in that block. Thus, the block processing time is affected by the I/O overheads from Merkle trees. For example, processing an Ethereum block with about 100 transactions takes hundreds of milliseconds even on a datacenter-grade NVMe SSD. Overall, Figure 6.2 (c) shows the direct correlation between the time taken to sync or process Ethereum blocks against the number of IO operations required, indicating the effect of I/O bottlenecks on block processing time.

6.1.3 Poor design of authenticated storage

The I/O bottleneck in Ethereum arises from two sources. First, reading or writing nodes of the Merkle tree generates many random I/Os in a pointer-chasing fashion (that prevents pre-fetching). Second, the Ethereum Merkle tree is stored on-disk using the RocksDB key-value [24] that has inherent I/O amplification [175, 176].

I/O bottlenecks. Merkle trees and their on-disk key-value layout introduces significant I/O overheads that directly impact the block processing time. Notice that our results are optimistic estimates, as we use a datacenter-grade NVMe SSD which is probably much better hardware than that available at an average untrusted server in the network. In Figure-6.2 (a) and (b), the spikes are the result of a DDOS attack [222] on the Ethereum's state, which creates dummy user accounts to increase the values in the Merkle tree and thereby increases the number of I/O operations and witness sizes. Finally, these overheads will increase as the system state increases and, as of April 1, 2019 [25], the Ethereum state is already above 200 GB. Although we analyze Ethereum and MPT in this study, our analysis is generally applicable to other authenticated data structures and blockchain systems. The I/O bottlenecks from authenticated dynamic dictionaries [180] are also seen in multi-token blockchain systems such as the Nxt cryptocurrency [nxt] [45].

6.1.4 Strawman solutions

Storing state in memory. Can every server store the entire state in memory to eliminate the I/O bottlenecks? This would not work as public blockchains seek to allow commodity servers to join their networks for increasing decentralization. If servers need to have 100s of GB of DRAM to join the public blockchain network, it decreases decentralization. Such a solution cannot be adopted.

Increase block size. Can we keep the current block creation rate and simply increase the number of transactions in each block? Increasing the block size is the goal of this

work; however, doing this in a naive fashion would not work. If we simply increased the number of transactions in each block, miners receiving the block would need more time to process the block and build on top of it; as a result, the block creation rate would have to be lowered to ensure that previous block is processed by a majority of the miners before a new block is released. Overall, transaction throughput would not increase though the block size increased. Ethereum has carefully increased the block size multiple times in the past [28]; it is prevented from raising the block size further due to I/O bottlenecks; with large blocks, new servers may take a long time to sync and participate in mining. This weakens decentralization. Thus, tackling the I/O bottleneck is crucial to increasing block size.

Alternative consensus protocols. Tackling the I/O bottlenecks in transaction processing is orthogonal to the underlying consensus protocol. Faster consensus protocols would result in blocks being released quicker, increasing overall throughput. Researchers have noticed that even faster consensus protocols ultimately run into the I/O bottlenecks in transaction processing [229]. Even with faster consensus protocols that release new blocks of transactions at a much higher rate, I/O bottlenecks will continue to limit the number of transactions in each block and thereby reduce overall throughput [229].

Summary. Thus, we need a mechanism to reduce the I/O bottlenecks in transaction processing. RainBlock achieves this goal with a new architecture and a novel authenticated data structure DSM-Tree. With faster transaction processing, RainBlock enables larger blocks *without* changing the block creation rate. Thus, RainBlock increases the throughput of public blockchains without compromising their security or liveness and without diluting their decentralization.

6.2 RainBlock

RainBlock aims to achieve the following goals simultaneously:

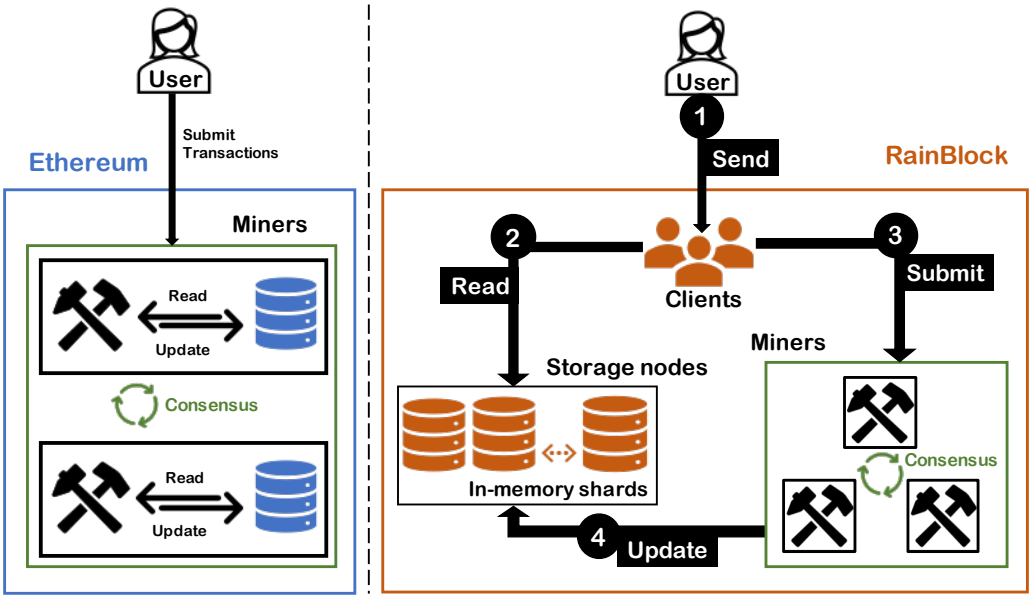


Figure 6.3: **Ethereum and RainBlock architecture**. Miners in Ethereum perform local disk I/O in the critical path. In RainBlock, clients read data from remote in-memory storage nodes (out of the critical path) on behalf of its miners. Miners execute txns without extra I/O and update storage nodes.

- High transaction throughput
- Scalability with increasing system state and number of users
- Support for Turing-complete smart contracts [202]
- The same degree of decentralization and security as Ethereum

Achieving these goals simultaneously in public blockchains is challenging. For example, achieving higher throughput by assuming large amounts of DRAM at every participating server reduces decentralization as only specific servers can participate.

RainBlock proposes a new architecture for public blockchains that achieves all these goals. RainBlock delivers high throughput by tackling the I/O bottleneck on two fronts. RainBlock avoids I/O in the critical path using *prefetching clients* and reduces I/O amplification using the novel DSM-Tree.

Overview. RainBlock achieves its goals by introducing a new architecture. The architecture allows RainBlock to both reduce I/O amplification and to remove I/O bottlenecks in the critical path, as shown in Figure-6.3.

RainBlock introduces three kinds of participating entities: *clients*, *miners*, and *storage nodes*. Users submit transactions to clients. Clients pre-execute transactions, and fetch data and witnesses from storage nodes. Clients submit the fetched data and witnesses to miners, who use this to execute transactions. Miners do not perform I/O in the common case. The miners create a new block, gossip it to other miners, and update the storage nodes. Storage nodes shard the system state and each shard stores the partitioned system state in memory. Figure-6.3 shows how the RainBlock architecture differs from that of Ethereum.

We now illustrate the RainBlock architecture and how it achieves scalability and supports smart contracts, without compromising on decentralization or security.

6.2.1 High-Level Design

In this section, we build up the design of RainBlock. We start with the problems that our study on Ethereum highlights. We discuss how RainBlock solves these problems and the resulting challenges.

6.2.1.1 Problem-I: I/O amplification from storing Merkle trees in key-value stores

Ethereum stores system state in a Merkle tree [19], and persists it using the RocksDB [24] key-value store. Traversing such a Merkle tree requires looking up nodes using their hashes. Hashing is computationally expensive and results in the nodes of the tree being distributed to random locations on storage. As a result, traversing the Merkle tree to read a leaf value requires several random read operations. The log-structured merge tree [158] that underlies RocksDB results in additional I/O amplification [176, 175].

Solution: store state in an optimized in-memory representation. RainBlock introduces an in-memory version of the Merkle tree. Persisting the data is done via a write-ahead log and checkpoints. Traversing the Merkle tree is *decoupled* from hashing; obtaining the next node in the tree is a simple pointer dereference. RainBlock introduces a technique termed *lazy hash resolution*: when a leaf node is updated in a Merkle tree, all the nodes from the leaf to the root need to be re-hashed; RainBlock defers the re-hashing until the nodes are actually read. Lazy hash resolution is effective since hashing requires serializing the node contents [23]; thus, lazily hashing the nodes saves both hashing and serialization operations. Note that simply running RocksDB in memory would not be effective: the hashing and serialization would still add significant overhead.

6.2.1.2 Resulting challenge: Tackling Scalability and Decentralization

Simply keeping the Merkle tree in memory does not achieve the goals of RainBlock. As the blockchain grows, the amount of state in the Merkle tree will increase; soon, a single server's DRAM will not be sufficient. Furthermore, for maintaining decentralization, we cannot require servers to have significant amount of DRAM.

Solution: decouple storage from servers and shard the state. RainBlock solves this problem using separate storage nodes. RainBlock shards the Merkle tree into subtrees such that each subtree fits in the memory of a storage node. As the amount of data in the blockchain increases, RainBlock increases the number of shards. In this manner, RainBlock scales with commodity servers and storage nodes without diluting the decentralization.

6.2.1.3 Problem-II: Miners perform I/O in the critical path

On receiving a new block, miners in Ethereum process its transactions by traversing and updating the Merkle tree in RocksDB (causing random I/O on storage) and verifying if their Merkle root matches the Merkle root in the block. Only then

can the miner process the next block of transactions. Thus, transaction processing includes performing slow I/O operations in the critical path; and the transactions are processed one at a time.

Solution: decouple I/O and transaction execution. RainBlock solves this problem by removing the burden of doing I/O from the miners. RainBlock introduces *prefetching clients* (clients) that prefetch data and witnesses from the storage nodes and submit them to the miners. Miners use this information to execute transactions without performing I/O and asynchronously update the storage nodes. Since transaction processing now becomes a pure CPU operation, it is significantly faster. This architecture also increases parallelism as multiple clients can be prefetching data for different transactions at the same time.

6.2.1.4 Resulting challenge: Prefetching I/O for smart contracts

One challenge with clients prefetching data for transactions is that some transactions invoke smart contracts. Smart contracts are Turing-complete programs that may execute arbitrary code. Thus, how does the client know what data to prefetch?

Solution: pre-execute transactions to get their read and write sets. RainBlock solves this problem by having the clients pre-execute the transactions. As part of this execution, the clients read data and witnesses from the storage nodes. One challenge is that the pre-execution may have different results than when the miner executes the transactions (*e.g.,* the smart contract may execute different code based on the block which it appears in). We will describe how clients handle smart contracts correctly despite stale data from pre-execution (§6.2.4).

6.2.1.5 Resulting challenge: Consistency in the face of concurrency

Another challenge that arises due to the RainBlock architecture is consistency. Multiple clients are reading from the storage nodes, and multiple miners are updating them in parallel. Using locks or other similar mechanisms will reduce con-

currency and throughput.

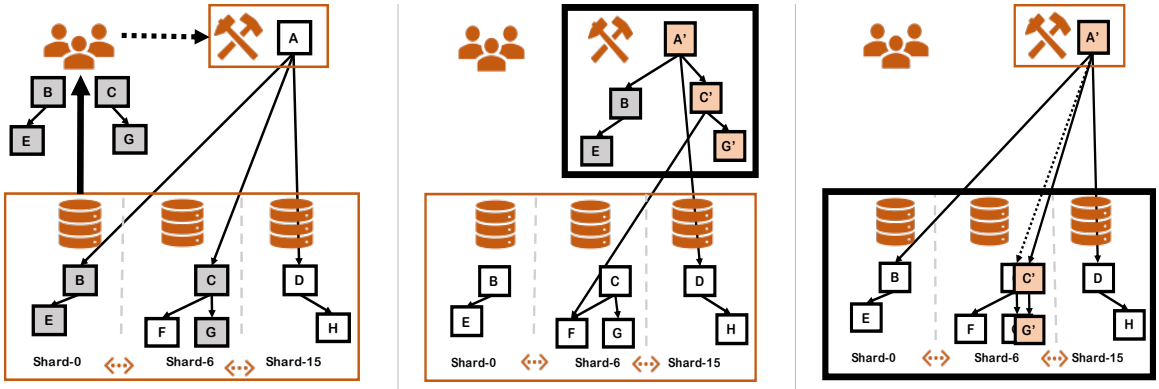
Solution: store system state in the two-layer, multi-versioned DSM-Tree. The DSM-Tree is an in-memory, sharded, multi-versioned Merkle tree. RainBlock uses DSM-Tree to store the system state. The DSM-Tree has two layers:

- The *bottom layer* is sharded across the storage nodes and contains multiple versions. Every write causes a new version to be created in a copy-on-write manner; there are no in-place updates. As a result, concurrent updates from miners simply create new versions and do not conflict with each other. When a fork of the blockchain is discarded, the bottom layer garbage collects the associated versions.
- The *top layer* represents a consistent version of the tree. Each miner has a top layer that is private to the miner. The miner executes all transactions against the data and witnesses in its top layer. New versions being created in the bottom layer do not affect the version in the top layer, ensuring consistency.

6.2.1.6 Resulting challenge: RainBlock has higher network traffic

Finally, the architecture of RainBlock trades local disk I/O for remote network I/O. As a result, RainBlock results in more network utilization, and the network bandwidth may become the bottleneck.

Solution: RainBlock reduces network I/O via deduplication and the synergy between the DSM-Tree layers. RainBlock uses multiple optimizations to reduce network I/O. First, the bottom and top layer of DSM-Tree collaborate with each other; when the bottom layer sends witnesses to the top layer, it will skip sending nodes of the Merkle tree that are known to be present at the top layer. We term this *witness compaction*. Second, when any component of RainBlock sends witnesses over the network, it will batch witnesses and perform deduplication to ensure only a single copy of each Merkle tree node is sent. We term this *node bagging*. Finally,



(A) Clients prefetch compact witnesses

(B) Miners execute txns without I/O

(C) Shards update asynchronously

Figure 6.4: **RainBlock architecture**. RainBlock processing a Txn that reads and updates accounts in two shards that are along the paths ABE and ACG. (A) Clients prefetch compact witnesses BE and CG from storage nodes and submit to miners. (B) Miners verify and use these witnesses to execute Txn against their top layer, and later update storage nodes. (C) Storage nodes verify updates from miners and asynchronously update their bottom layer, creating a new version for the modified account $A'C'G'$.

miners send logical updates to storage nodes rather than physical updates as logical updates are smaller in size.

6.2.2 Architecture

We now describe the RainBlock architecture in detail.

Overview. RainBlock introduces three kinds of participating entities: *prefetching clients* (clients), *miners*, and *storage nodes*. Users send transactions to clients. Clients pre-execute these transactions and prefetch data and witnesses from storage nodes. Clients submit transactions and the prefetched information to miners, Figure-6.4(a). Miners are responsible for creating new blocks of transactions and extending the blockchain; each miner maintains a private copy of the top layer of the DSM-Tree. Miners use the submitted information to execute these transactions against their top layer, Figure-6.4(b); and do not perform I/O in the common case. Finally,

miners create a new block, gossip it to other miners, and update the storage nodes. Storage nodes are responsible for maintaining and serving the system state. They use the multi-versioned bottom layer of the DSM-Tree to provide consistent data to clients while handling concurrent updates from miners. Storage nodes asynchronously update the bottom layer of the DSM-Tree, Figure-6.4(c).

6.2.3 The Life of a Transaction in RainBlock

. We now describe the various actions that take place from the time a transaction is submitted, to when it becomes part of the blockchain.

1. A user submits the transaction (tx) to a client.
2. The client will pre-execute the transaction, reading data and witnesses from storage nodes
3. The client will optimize the witnesses before sending them over the network using node bagging
4. The client will submit the transaction, data, and optimized (compact) witnesses as node bags, to the miner
5. The miner will verify these node bags and advertise them to other miners
6. The miner will execute tx using witnesses and the top layer of the DSM-Tree. The miner does not need to perform any I/O in the common case
7. The miner creates the new block, and sends it to other miners
8. The miner sends new block and the updates to storage nodes as logical operations (*e.g.,* A− >A') with new Merkle root
9. The storage nodes verify if block is valid (proof of work check), log updates, return successful to miner

10. The storage nodes apply the updates and verify them (based on provided Merkle root)

11. The other miners verify the block using their top layer and node bags without any I/O, and gossip to other miners

12. The tx is added to the blockchain when its block is processed by majority of miners

6.2.4 Speculative Pre-Execution by Clients

In RainBlock, clients read all the witnesses required for executing a transaction from storage nodes. These transactions can be simple or can call smart contracts. Since smart contracts are Turing-complete, it is not known apriori what locations they will access. RainBlock clients handle this by *speculatively* pre-executing the smart contract to obtain the data that is read or modified by the smart contract.

Speculative pre-execution. Smart contracts can use the timestamp, or block number of the block in which they appear, during their execution at the miner. These values are not known yet during their pre-execution at the clients. As a result, clients speculatively return a guess while pre-executing the contract. Our analysis of Ethereum contracts shows that despite providing estimated values, clients still successfully prefetch the correct witnesses and node bags. For example, the CryptoKitties `mixGenes` function as shown in Figure 6.5 repeatedly references the current block number and its hash. Since these numbers are only used to generate randomness of *written* values in the function, substituting inaccurate values does not affect the witnesses that are pre-fetched.

Stale data. We make a similar observation that clients can pre-execute with *stale* data and still prefetch the correct node bags. For example, the CryptoKitties `giveBirth` function is a *fixed-address* contract, where the addresses read (loads from the `kitties` array) only depend on the inputs from the message call. To deal with rare variable-

```
1: function MIXGENES(mGenes, sGenes, curBlock)
2:     uint256 randomN ← curBlock.blockHash
3:     randomN ← KeccakHash(randomN, curBlock)
4:     MemoryAry babyGenes ← mix(mGenes, sGenes, randomN)
5:     return babyGenes
```

Figure 6.5: **Indeterminate contract values do not affect pre-fetching**. Psuedocode of CryptoKitties *mixGenes* function. It makes repeated calls to *curBlock*. Although client substitutes it with a speculative value, it doesn't affect witness prefetching because these numbers only affect *written* values.

address contracts, the miner may asynchronously read from a storage node after the transaction is submitted. Even in these cases, the client will have retrieved few of the correct witnesses required for the transaction (*e.g.,* the `to` and `from` accounts).

Benefits from pre-executing clients. One of the main advantages of the speculatively pre-executing client is that it can filter out transactions that miners would abort. Contrast this with the Ethereum blockchain, in which aborted transactions are no-ops, but still take up valuable space in the public ledger. In RainBlock, clients can prevent miners from spending valuable cycles executing transactions that will be eventually aborted. The trade-off is that staleness might cause clients to abort transactions conservatively. If users believe that a client incorrectly aborted their transaction, they can send it to other clients or prefetch the witnesses themselves.

6.2.5 Benefits

In summary, RainBlock achieves high throughput by reducing I/O amplification (using DSM-Tree to scalably store the system state) and eliminating I/O bottlenecks (using clients to decouple I/O from transaction execution). Thus, RainBlock increases the transaction processing rate at miners without assuming any hardware limits on them. With faster transaction processing, RainBlock allows miners to pack more transactions per block without impacting the underlying PoW consensus (RainBlock does not impact block creation rate). Thus, RainBlock achieves high throughput without compromising the security or decentralization of

public blockchains.

The RainBlock architecture has many additional benefits:

- Every component is typically visited once for processing a transaction, presenting an efficient architecture for public blockchains.
- RainBlock can scale with increasing transaction load (by increasing number of clients) and with increasing system state (by increasing number of storage shards).
- Clients can execute read-only transactions bypassing miners.
- RainBlock supports transactions on the sharded system state without requiring locking or additional coordination among clients and miners.
- RainBlock does not assume trust between any of the components.

6.3 DSM-Tree

The Distributed, Sharded Merkle Tree (DSM-Tree) is an in-memory, multi-versioned, sharded variant of the Merkle tree. The DSM-Tree has two layers; we first present the common in-memory representation, then describe each layer in turn, and then discuss how the layers collaborate and their trade-offs in different configurations.

6.3.1 In-Memory Representation

DSM-Tree uses an efficient in-memory representation of the Merkle tree. Tree traversal is decoupled from hashing: traversing the Merkle tree is done by dereferencing pointers; in contrast, Ethereum's Merkle tree has to perform expensive cryptographic hashing to find the next node during traversal. DSM-Tree uses periodic checkpoints for persisting the data. The checkpoints are only used to reconstruct the in-memory data structure in case of failures; reads are always served from memory.

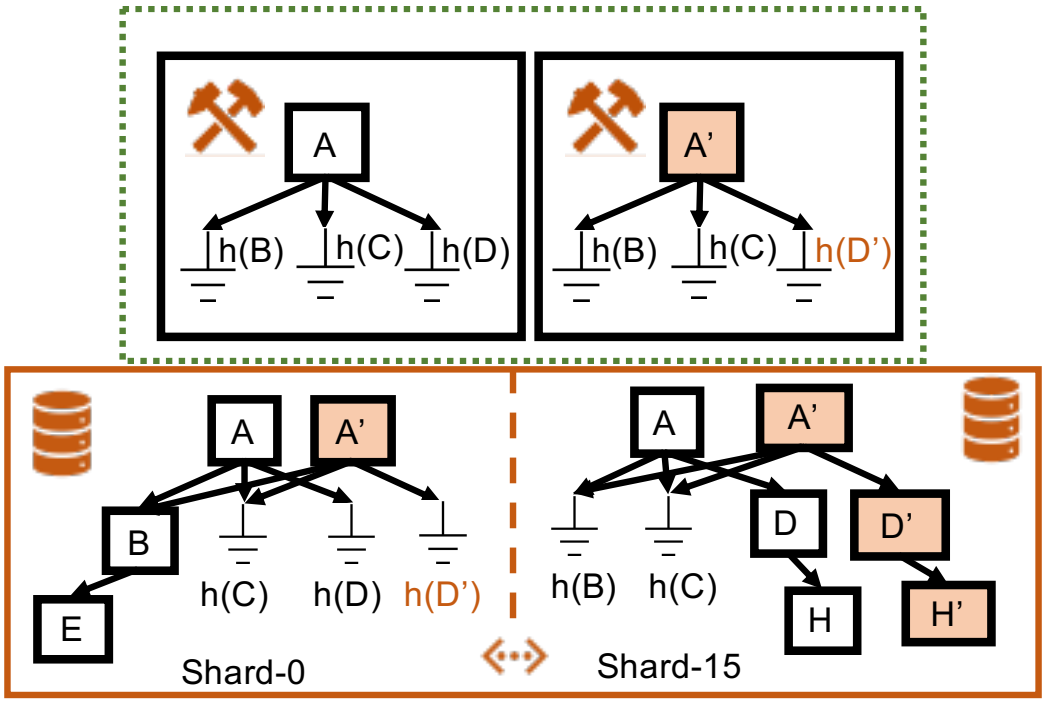


Figure 6.6: **DSM-Tree design in RainBlock**. This figure shows the two-layered DSM-Tree where miners have their private copy of the top layer for consistency and the bottom layer is sharded for scalability.

Lazy Hash Resolution. When a leaf node in a Merkle tree is updated, hashes of nodes from the leaf to the root need to be recomputed. DSM-Tree defers doing this recomputation until a node is actually read. This makes writes efficient as only the leaf node has to be updated in the critical path. Recomputing hashes is expensive as nodes have to be serialized before being hashed; as a result, lazy hash resolution improves performance significantly by saving expensive hashing and serialization operations [23].

6.3.2 Bottom Layer

The bottom layer of the DSM-Tree consists of a number of shards. Each shard is a vertical subtree of the Merkle tree, stored in DRAM. The bottom layer supports multiple versions to allow concurrent updates to the DSM-Tree, as shown in Figure-6.6. The bottom layer has a write-ahead log to persist logical updates.

Multi-versioning. Each write to the bottom layer creates a new version of the tree. There are no in-place updates. This versioning is required as miners may submit multiple blocks concurrently that potentially conflict with each other; the bottom layer creates a new version for each write. It creates versions only for the modified data in a copy-on-write manner. Thus, writes never conflict with each other, and DSM-Tree does not require locking or additional coordination among miners or clients.

Garbage collection. Garbage collection of versions is driven by the higher-level blockchain semantics. When multiple miners are working on competing forks of the blockchain, multiple versions are maintained. Eventually, one of the forks is accepted as the mainline fork, and the others are discarded (and their associated versions are garbage collected by the bottom layer of the DSM-Tree).

6.3.3 Top Layer

Given that the bottom layer maintains multiple versions across multiple shards, we need a way for miners to access data in a consistent fashion. The top layer provides this mechanism.

Each miner has a private top layer. The top layer contains the first few levels of the Merkle tree, till a configurable retention level (r). The top layer has the Merkle root node that summarizes the entire system state, and presents a consistent snapshot of the system state. Miner executes transactions against this snapshot; all reads return values from this snapshot of the system.

As the miner executes transactions, their top layer is updated, switching to a different consistent view of the system state, as shown in Figure-6.6. The changes in the top layer's Merkle tree will also be reflected in the bottom layer's storage shards after the miner sends logical updates to the storage nodes.

Caching and Pruning. The top layer acts as an in-memory cache of witnesses for the miners. By design, the top layer stores the recently used and the frequently changing parts of the Merkle tree. The top layer receives witnesses (or paths of the Merkle tree) from the prefetching clients. The top layer uses the node bags from the clients to reconstruct a *partial* Merkle tree that allows miners to execute transactions, typically without performing I/O from the bottom layer. The top layer also supports pruning the partial Merkle tree to help miners reclaim memory. Pruning replaces the nodes at the retention level $(r + 1)$ with *Hash nodes*. Hash nodes also help miners to identify the DSM-Tree shard which has pruned nodes.

Witness Revision. In RainBlock, the bottom layers of the DSM-Tree update asynchronously. Therefore, the top layer (miner) may receive stale witnesses from the bottom layer (prefetching clients). DSM-Tree introduces a new technique termed *witness revision* to tolerate stale witnesses. A witness is determined to be stale or incorrect because the Merkle root in the witness doesn't match the top layer's Merkle root. However, this could happen because of an *unrelated update* to another part of the Merkle tree. The top layer detects when this happens, and *revises* the witness to make it current. If the Merkle root matches now, then the witness is accepted. Witness revision is similar to doing `git push` (trying to upload your changes), finding out something else in the repository has changed, doing a `git pull` (obtaining the changes in the repository) to merge changes, and then doing a `git push`. With witness revision, the top layer tolerates stale data from the bottom layer and allows miners to execute non-conflicting transactions that would otherwise get rejected. Note that, witness revision cannot revise every potential stale witnesses. If the top layers are pruned aggressively, the top layer may have insufficient information to detect if

the changes are from an unrelated part of the Merkle tree.

6.3.4 Synergy among the layers

The top and bottom layers collaborate to reduce network traffic. We also briefly discuss the potential DSM-Tree configurations with r (retention at the top layer) and c (compaction level at the bottom layer), and the trade-offs involved.

Witness compaction. As the top layer of the DSM-Tree stores the top levels of the Merkle tree, the storage nodes do not need to send a full witness. Like the configurable retention level at the top layer r, the bottom layer has a configurable compaction level, c. Only nodes below the compaction level (compact witnesses) are sent in node bags, after deduplicating nodes across witnesses, reducing the network burden of transmitting witnesses.

Configurations. The DSM-Tree is configurable to operate entirely from local memory without any network overhead, or just from remote memory with high network utilization. For example, If the top layer of the DSM-Tree has $r = \infty$, then the top layer caches the entire Merkle tree and is fully served from local memory. Similarly, if the bottom layer has $c = 0$, then un-compacted witnesses are sent over the network and accessed entirely from remote memory. Thus, DSM-Tree provides a unique (and flexible) point in the design spectrum of distributed, in-memory, authenticated data structures.

Trade-offs. In a Merkle tree that has n levels, any DSM-Tree configuration that satisfies $c >= (n-r)$ allows the top layer to use compacted witnesses from the bottom layer. Note that having a higher r results in a lower number of transaction aborts, as the top layer has more information to detect non-conflicting updates and perform witness revision. Therefore, in RainBlock, ideally, the top layers should set r based on the amount of memory available. Pruning the top layer should only be done under memory pressure.

6.3.5 Summary

In summary, the DSM-Tree is a novel variant of the Merkle tree modified for faster transaction processing in public blockchains. It uses an efficient in-memory representation to reduce I/O amplification, multi-versioning to handle concurrent updates, and the Merkle root in the top layer to provide a consistent view of the system state. DSM-Tree presents a new point in the design space of authenticated data structures. The top layer exploits the cache-friendliness of the Merkle tree, while the sharded bottom layer relies on the fact that witness creation only requires a vertical slice of the tree. While the DSM-Tree is exclusively used with RainBlock in this paper, it can be easily modified to work with other blockchains.

6.4 Discussion

We now discuss the trust assumptions, incentives, security, and limitations of the RainBlock architecture.

Trust Assumptions. In keeping with existing public blockchains, RainBlock **does not require trust** between any of its components. Miners operate without trusting the clients or the storage nodes by re-executing transactions and verifying the data they receive. Clients operate without trusting the storage nodes and miners, as clients verify their reads from storage nodes, and can verify the block produced by a miner. Finally, storage nodes also operate without trusting miners, as they can verify updates from miners.

Incentives. We sketch a possible incentive model here. A more rigorous analysis would require game-theoretic and economic models, which are beyond the scope of this paper. Users can prefetch data from the storage nodes themselves or pay clients. Users, clients, and miners pay the storage nodes for reading authenticated data. Miners get paid for mining a new block of transactions through block rewards. Every RainBlock component can detect misbehaving entities and blacklist them,

incentivizing correct behavior.

Security. RainBlock provides the same security guarantees as Ethereum, as it does not change the consensus protocol (Proof of Work) or trust assumptions between participating servers. Further, RainBlock does not impact the block creation rate by packing more transactions into each block.

6.5 Implementation

We implement RainBlock and DSM-Tree in Typescript, targeting node.js. The miners and storage nodes use the DSM-Tree as a library. The performance critical portions of the code, such as `secp256kp1` key functions for signing transactions and generating `keccak` hashes, are written as C++ node.js bindings. To execute smart contracts, we implement bindings for the Ethereum Virtual Machine Connector interface (EVMC) and use Hera (v0.2.2). Hera can run contracts implemented using Ethereum flavored WebAssembly (ewasm) or EVMC1 bytecode through transcompilation. Our speculative pre-executing client is implemented in C++. The DSM-Tree and RainBlock implementations, together 15K lines of code, is open source and available on GitHub[2]. Our current implementation of storage nodes assumes a 16-way sharded Merkle Patricia tree by default. It supports a configurable number of shards.

6.6 Limitations and Trade-offs

RainBlock trades local storage I/O for network I/O, so the network may become a bottleneck. RainBlock recognizes this risk and uses multiple techniques such as witness compaction and node bagging to reduce network traffic. The RainBlock architecture introduces additional storage nodes and shows how a small percentage of additional hardware can provide orders of magnitude improvement in end-to-end

[2]https://github.com/RainBlock

transaction throughput in public blockchains.

6.7 Evaluation

In this section, we evaluate the performance of DSM-Trees and RainBlock. We seek to answer the following questions:

1. What is the performance of the DSM-Tree for various operations on a single node? (Section-6.7.2)
2. How does the size of the DSM-Tree cache impact the witness sizes, memory overhead, and rate of transaction aborts in RainBlock? (Section-6.7.3)
3. What is the performance of RainBlock on various end-to-end workloads that characterize the Ethereum public blockchain? (Section-6.7.4)

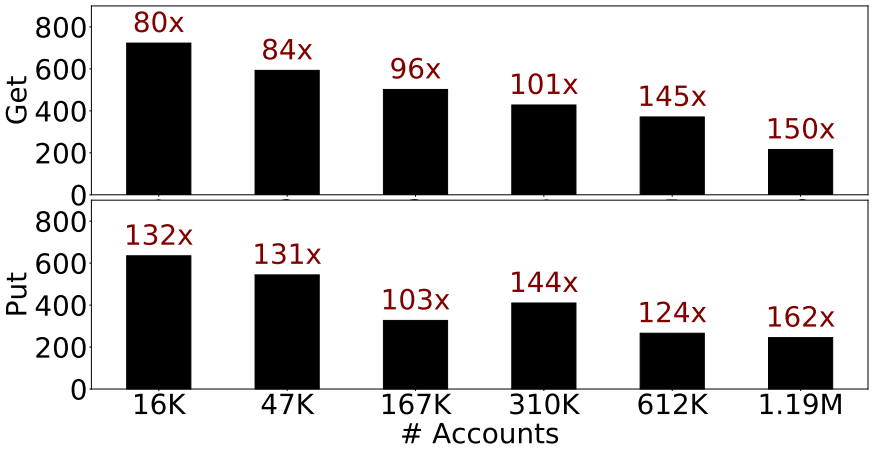
6.7.1 Experimental Setup

We run the experiments in a cloud environment on instances which are similar to the `m4.2xlarge` instance available on Amazon EC2 with 32GB of RAM and 48 threads per node. We use Ubuntu 18.04.02 LTS, and node.js v11.14.0. For the end-to-end benchmarks, each storage node, miner, and client is deployed on its own instance.

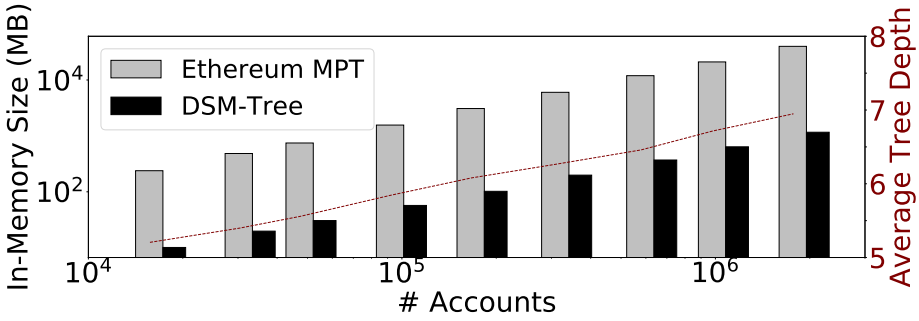
6.7.2 Evaluating DSM-Tree on a single node

First, we evaluate the DSM-Tree running on a single node. This tests the performance of the optimized in-memory representation of DSM-Tree. We measure the throughput of point `put` and `get` operations for a variety of tree sizes against the state-of-the-art Ethereum MPT. Point `put` operations create or update a key-value pair and `get` operation returns the value and witness for a key.

To make a fair comparison, we compare DSM-Tree with the in-memory imple-



(a) DSM-TREE Put and Get Throughput (K ops/s)



(b) Size of In-Memory Merkle tree

Figure 6.7: **Performance of DSM-Tree on a single node**. (a) The figure shows the absolute put and get throughput of DSM-Trees. Throughput relative to the Ethereum MPT is shown on the bars. As the number of accounts increase, DSM-Tree throughput increases relative to Ethereum MPT. (b) This figure shows the memory used by DSM-Tree and Ethereum MPT across varying number of accounts. The trend line captures the height of the MPT. DSM-Trees are orders of magnitude more memory-efficient than Ethereum MPT. Note the log scale on the axes.

mentation of Ethereum MPT [15]. The in-memory Ethereum MPT uses `memdown` [22], an in-memory key-value store built on a red-black tree. We are comparing an in-memory MPT that uses the key-value representation to the in-memory DSM-Tree. The difference in performance comes from the in-memory design and optimizations in DSM-Trees, and not due to different storage media.

We dump the Ethereum world state every 100K blocks until 4M blocks and

use it to micro-benchmark DSM-Trees; every key in these benchmarks is a 160-bit Ethereum address and values are RLP-encoded Ethereum accounts [23].

Gets. DSM-Trees with 1.19M accounts, obtain a `get` throughput of **≈216K ops/s**, that is **150**× the throughput of Ethereum MPT. The main reason for the DSM-Tree's better performance is the use of in-memory pointers. To fetch a node, the DSM-Tree simply needs to follow a path of in-memory pointers to the leaf node. On the other hand, walking down a tree path means looking up the value (node) at a particular hash for each node in the Ethereum MPT. Even though this database is in-memory, looking up values in an in-memory key-value map is still more expensive than a few pointer lookups. Furthermore, the larger the world state, the better DSM-Tree's in-memory Merkle tree performs over the Ethereum MPT. This is simply because the larger the state, the taller the tree, so the more nodes on the path to a leaf, see Figure 6.7 (a).

Puts. DSM-Trees with 1.19M accounts obtain a `put` throughput of **≈245K ops/s**, that is **160x** the throughput of Ethereum MPT. Due to lazy hash resolution, a `put` does not need to adjust any values in the path from the leaf to the root; in contrast, every node in the path has to be updated in the Ethereum MPT. `put` throughput in the DSM-Tree is more than two orders of magnitude higher than in the Ethereum MPT.

Tree Size. Figure 6.7 (b) shows that DSM-Trees are significantly smaller than Ethereum MPTs when the same number of accounts are stored. With 1.19M accounts, the Ethereum MPT consumes ≈26021MB and DSM-Trees consume ≈775MB, using **34**× lesser memory. The primary reason for this is the efficient in-memory representation of DSM-Trees. Ethereum MPT is not-memory efficient as it uses 32-byte hashes as pointers and relies on `memdown` [22] to store the flatenned MPT as key-value pairs. The significantly reduced size of the DSM-Tree, along with sharding, enables DSM-Trees to be stored entirely in memory, eliminating the IO bottleneck.

Lazy hash resolution. We run an experiment where we trigger a root hash calculation after every N write (`put` or `delete`) operations. As N increases, the performance of DSM-Tree write operations also increases. At $N = 1000$ (the root hash is read every 1000 writes), DSM-Tree is 4–5× faster than Ethereum MPT. Since the root hash calculation is expensive (requiring RLP serialization of nodes), performing it even once every 1000 writes reduces DSM-Tree performance from 150× Ethereum MPT performance to 5×.

6.7.3 Impact of cache size

Next, we evaluate the performance of the distributed version of the DSM-Tree when the cache size (retention level of the top layer) is changed. Pruning the cache reduces memory consumption but results in larger witnesses being transmitted, and more transactions being aborted due to insufficient witness caching. We evaluate these effects.

Memory consumption. We evaluate the reduction in the application memory utilized, from pruning the DSM-Tree cache, across varying cache sizes r. Figure 6.8 (a) shows that lower r will result in higher memory savings, with a tree of depth five consuming only 40% of the memory consumed by the full tree. However, this means that either 1) DSM-Tree shards will have to provide larger witnesses or 2) the application will experience a higher abort rate due to insufficient witness caching.

Witness Compaction. DSM-Trees transmit compact witnesses which include only the un-cached parts of the witness. DSM-Trees employ node-bagging where they combine multiple witnesses and eliminate duplicate nodes. Figure 6.8 (b) shows the reduction in witness size due to node bagging and witness compaction, based on the height of the cached tree r. Witness compaction and node bagging together reduce witness sizes by up-to **95%** of their original size.

Transactions. Pruning the cache discards cached witnesses. Since transactions abort if the witnesses are not cached, this increases the abort rate. To study the

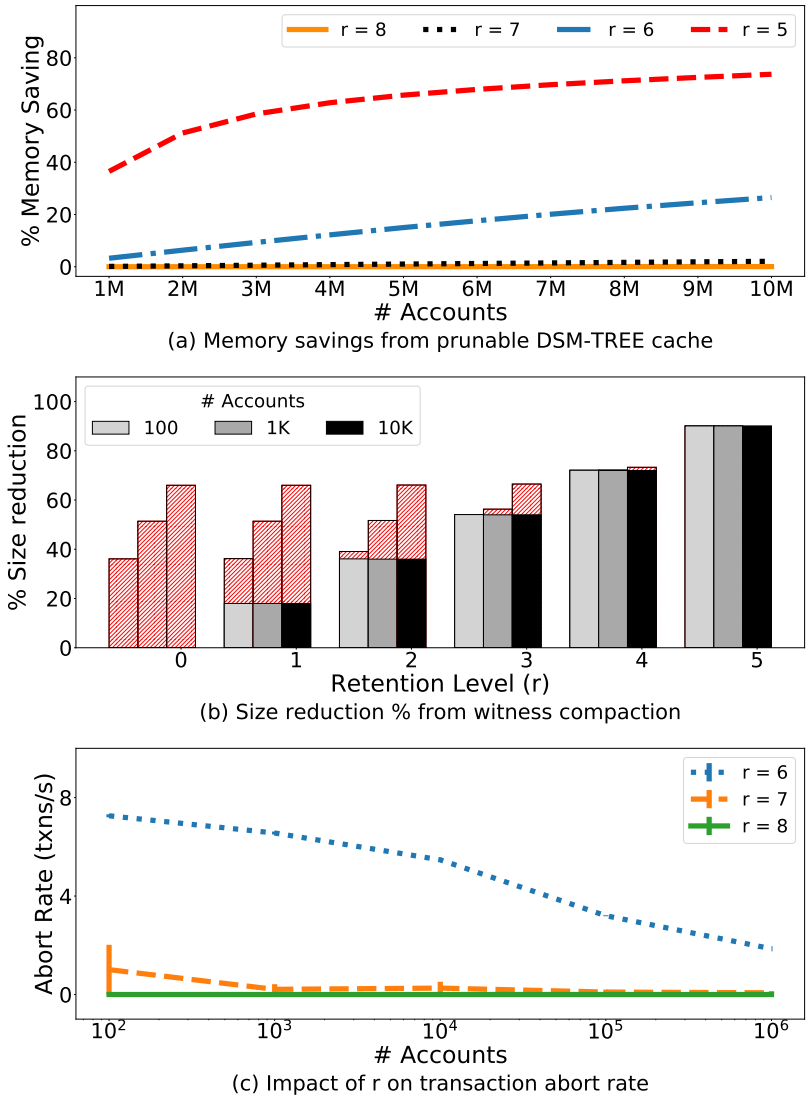


Figure 6.8: **Tuning DSM-Tree Cache Retention** r. *(a) This figure shows that the caching fewer levels in the cache leads to higher memory savings (compared to storing the full tree in memory). We do not report the memory savings of higher values of r as they were negligible. (b) The figure shows the reduction in witness size due to combining witnesses and eliminating duplicates (red striped bar) and due to witness compaction (solid bar). (c) The figure shows the impact of r (height of cached tree) on transaction abort rate. Higher r results in lower number of transaction aborts. Abort rate decreases with fixed r as the total number of accounts N increases, because this reduces the probability that transaction will involve accounts that conflict at the pruned levels.*

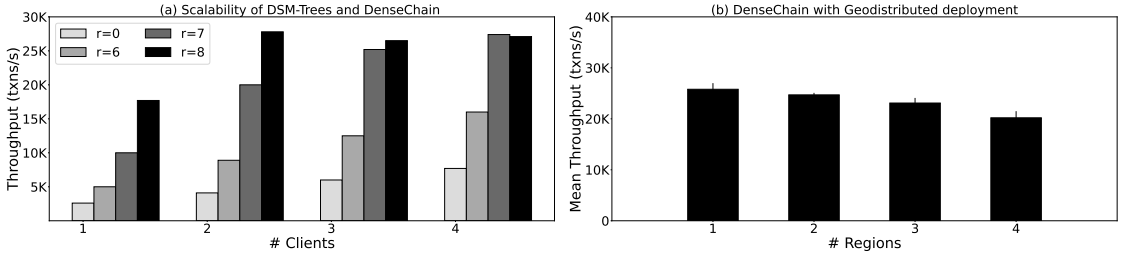


Figure 6.9: **End-to-End Throughput**. (a) The figure shows DSM-Tree scalability in RainBlock with increasing number of clients and varying cache retention levels (r). The workload used in the experiment is representative of the account distributions in Ethereum transactions. Miners in RainBlock can process about **30K tps** with 4 clients each, when configured at $r = 7$. (b) This figure shows the overall throughput of RainBlock in a geo-distributed deployment. Miners at $r = 8$ can process about **20K tps** using 4 clients each, when communicating with the DSM-Tree across WAN.

effect of varying the cache size on transaction abort rate, we use RainBlock with 16 storage nodes, 1 miner, and enough clients to saturate the miner. Transactions are generated by selecting two random accounts from a set of N accounts. Figure 6.8 (c) shows that the transaction abort rate is dependent on two factors: the DSM-Tree cache retention level, and the number of accounts. In particular, increasing r reduces the transaction abort rate. More importantly, with large number of accounts N, the contention on Merkle tree nodes reduces, reducing the abort rate for fixed a r, making DSM-Trees practical for application with low available memory.

6.7.4 End-to-End Blockchain Workloads

Finally, we evaluate the end-to-end performance of RainBlock against synthetically generated workloads that mirror transactions on the Ethereum public *mainnet* blockchain.

Challenges. Since Ethereum transactions are signed, the public transactions are not conducive to experiments: we cannot change transaction data or the source accounts, because we do not have the `secp256k1` private key. Since RainBlock runs transactions at a much higher rate than Ethereum, we quickly run into state mismatch

errors, and eventually, exhaust the available transactions.

To tackle this challenge, we analyze the public blockchain to extract salient features, and develop a *synthetic workload generator* which generates accounts with private keys we control, so our clients can run and submit signed transactions.

Synthetic Workload Generator. We analyze the transactions in the Ethereum `mainnet` blockchain to build a synthetic workload generator. We analyzed 100K recent (since block 7M) and 100K older blocks (between blocks 4M and 5M) in the Ethereum blockchain to determine: 1) the distribution of accounts involved in transactions, 2) what fraction of all transactions are smart contract calls. We observe that 10-15% of Ethereum transactions are contract calls and the rest are simple transactions. This is true of both recent blocks and older blocks. It is also the case that a small percentage of accounts are involved in most of the transactions. Based the analyzed data, we generate workloads where 90% of accounts are called 10% of the time, and 10% of the accounts are called 90% of the time. Smart contracts are invoked 15% of the time.

Throughput. Figure 6.9 (a) shows the transaction throughput results. First, this figure shows that the RainBlock can achieve an end-to-end verification throughput of 30,000 transactions per sec. It also demonstrates the scalability of the DSM-Tree and RainBlock, which scales as more clients are added. By varying the DSM-Tree retention level at the miners from 0 to 6, the DSM-Tree shard throughput increases by **7x**, from 1.3K ops/s to 9.4K ops/s, increasing the scalable creation and transmission of witnesses.

Geo-distributed Experiment. We also ran a geo-distributed experiment, with varying numbers of regions across 3 continents. Each region has 4 clients, 1 miner, and 16 storage nodes, caching eight levels of the DSM-Tree tree ($r = 8$). Figure 6.9 (b) reports the throughput experienced by the RainBlock. RainBlock in a single region achieves a throughput of ≈25K transactions/sec; when we scale to four regions, the throughput drops to ≈20K transactions/sec, thus retaining 80% of the performance in a geo-distributed setting.

Contract Calls. We also ran a workload where accounts repeatedly call the `OmiseGO Token`, which is an ERC-20 token contract [14]. Four clients repeatedly called the token contract against a single RainBlock miner with DSM-Tree cache configured at $r = 8$, achieving a throughput of $17.9K \pm 796$ contract calls per second. This demonstrates that even for pure contract calls, RainBlock can provide orders of magnitude higher transaction throughput than other blockchains.

Chapter 7: Related Work

In this section, we discuss state-of-the-art PM key-value stores, distributed databases, and public blockchains, and place the contributions of this dissertation in the context of the relevant research in these domains.

7.1 Skye

We place our contributions Skye in the context of related prior work on PM key-value stores and PM file systems.

PM key-value stores. Key-value stores designed from scratch for PM [48, 61, 214] exploit the byte-addressability and the low latency of PM. The work closest to Skye is Viper [48]. Viper evenly distributes threads across NVDIMMs and designs NVDIMM-aligned storage layouts to benefit from the parallelism of interleaved PM and avoid cross-NVDIMM contention. However, prior work does [61, 238, 48] not take into account the limited concurrent access allowed by PM, or that managing individual NVDIMMs yields higher throughput. Finally, prior work [238, 48] is not designed to be NUMA-aware, exhibiting poor performance (or lack of support) on multiple NUMA nodes. In contrast, Skye takes all these aspects into consideration and implements the indirect-access for applications, and obtains high and scalable write throughput.

OdinFS. A concurrent work that is close to Skye in the file-system domain is OdinFS [240]. Skye and OdinFS share several goals: scalability, NUMA-awareness, and PM bandwidth utilization. OdinFS improves performance for syscall-based applications. Since it relies on striping to scale across multiple NUMA nodes, it trades off PM contiguity and hurts the performance of memory-map applications. Skye supports a simpler key-value API, uses memory-map interface, has a modular design, and obtains high PM bandwidth utilization. Skye exploits the benefits of managing all PM accesses by providing indirect-access (in contrast to current PM stores) with-

out paying the overheads from POSIX API unlike OdinFS. Note that OdinFS [240] and all existing PM file systems [115, 226] manage a single interleaved PM device (in contrast to managing individual NVDIMMs) and suffer from poor PM bandwidth utilization.

7.2 Cascades

In this section, we place our contribution Cascades in the of context state-of-the-art distributed databases and prior research on mitigating the I/O bottlenecks and network overheads in databases.

Distributed databases. The state-of-the-art distributed NoSQL databases [33, 205, 192], SQL databases [209, 204, 4], and NewSQL databases [103, 68, 241, 206] rely on logging across multiple replicas to ensure consistent recovery in the event of failures; state-of-the-art distributed databases and their choice of concurrency control protocols and isolation guarantees are summarized in Table-2.1. However, every database performs synchronous I/O to durably commit transactions; they wait for the durability of writes across a set of replicas before committing a transaction. Further, with highly contented workloads, compute cores wait on the durability of writes before releasing locks [109]. Note that these I/O bottlenecks result in the poor utilization of the available compute (multiple core parallelism) and network resources. Further, replicas are chosen across multiple availability zones within a datacenter [8, 5]. Thus, synchronous durability of writes introduces latencies in the order of millisecond and reduces the overall throughput by multiple orders of magnitude. Cascades with Lattice tackle these I/O bottlenecks in the critical path and claim the orders of magnitude improvement in throughput and scalability. We believe that existing research on reducing other overheads in databases *e.g.,* synchronization [234, 35, 170], complement our work; modern databases can be augmented to employ Lattice and thereby achieve high throughput and scalability even for highly contented workloads.

Network layer. Several distributed systems are co-designed along with the underly-

ing network to achieve low-latency and high-throughput communication [60, 79, 237, 220]. In contrast to co-designing with RDMAs [223, 110, 141, 121], eRPC [119, 117] provides performance close-to hardware limits in the common case without having to modify application. Thus, CASCADES relies on eRPC, which supports the full generality of RPCs, for all network communications.

Expressive systems APIs. Recent research has introduced novel APIs to increase the expressivity of RDMA interface [53]. These works aim to achieve high utility and to translate unique performance benefits to applications. Similarly, CASCADES implements LATTICE's interface for recoverable processes to achieve high performance and to simply recovery.

Causally consistent logs and fault tolerance. Prior research on novel fault-tolerant mechanisms includes multiplexing the failed process [221] reducing the amount of data written to logs, batching to avoid small-sized I/O *etc.*. CASCADES leverages LATTICE that tracks the causal dependencies of log records using vector timestamps following prior research [133, 163, 86]. However, instead of resolving dependencies during recovery, CASCADES relaxes the synchronous durability of writes across all replica and allows failures to spread within speculating boundaries.

7.3 RainBlock and DSM-Tree

In this section, we place our contributions RAINBLOCK and DSM-TREE, in the context of prior research on blockchain systems, authenticated data structures, and transactional stores.

Stateless Clients. The Stateless Clients [55] proposal seeks to insert witnesses into blocks, enabling Ethereum miners to process a block without performing I/O. Despite active discussions [21, 20, 18], Stateless Clients have not been implemented due to concerns about witness sizes [203]. Witnesses for a single, simple Ethereum transaction can be $\approx$ 4-6KB, resulting in 40-60$\times$ the network overhead. In contrast,

DSM-Tree reduces witness sizes (by $\approx 95\%$), and RainBlock uses prefetching clients to remove the I/O burden on miners.

Permissioned blockchains. Hyperledger Fabric [31] proposes a novel execute-order-validate architecture for permissioned (private) blockchains. Fabric optimistically executes transactions and relies on the signatures from trusted nodes for verifying transactions. In contrast, RainBlock improves transaction throughput in public blockchains without trusting any of the participating servers.

Sharding and off-chain computation. Recent work increase blockchain throughput by sharding the blockchain into independent parallel chains that operate on subsets of state [112, 212, 128, 236, 144, 213]. However, sharding requires synchronizing these independent chains for consistency, requires complex protocols for cross-shard transactions, and is less resilient to failures or attacks [196, 174, 235]. In contrast, RainBlock does not shard the global blockchain; the storage is sharded, but all miners add to a single chain. RainBlock does not require locking or additional communication for executing transactions that span across multiple storage shards. Payment channels [142, 123, 96, 152, 12, 111] that offload work to side chains while ensuring a total order of transactions are complementary to our work.

Consensus: Ethereum and Bitcoin employ Nakamoto consensus based on Proof-of-Work (PoW) [108, 195]. There is active research on designing novel consensus for blockchains [85, 151, 232, 56, 43, 145] including Proof-of-Stake [91, 124, 17] and Proof-of-Elapsed-Time [104] protocols, primarily because PoW limits block creation rates, trades off wasted work for security [222, 211], and is intolerant to the 51% attack [84]. While new protocols can replace PoW in Ethereum, low transaction processing rates will remain a concern [230, 78, 38, 229]; since alternative consensus protocols aim at releasing blocks at a higher rate; but perform I/O in the critical path for creating new blocks. Therefore, orthogonal to the consensus protocols, RainBlock alleviates I/O bottlenecks in transaction processing to increase the throughput of public blockchains.

Discussion: Recent work on reducing network overheads in blockchains [161, 67, 77], and including forks into the main chain [138, 137], are orthogonal to our work. These techniques can be applied to RainBlock to further increase its overall throughput. Other work using trusted hardware for reducing storage overheads [140, 71] or consensus protocols [46, 153] is orthogonal to our work.

Dynamic accumulators. Merkle trees belong to a general family of cryptographic techniques called dynamic accumulators [57, 47]. Merkle trees, known for their fast processing, have proofs that grow with the underlying state. Constant-size dynamic accumulators based on RSA signatures [57, 47] have fixed size proofs. However, constant-size accumulators have low processing rates, and improving their performance is an ongoing effort [52]. DSM-Tree provides a practical solution to achieve high processing rates and small witness sizes while supporting transactions.

Authenticated data structures. Recent work has proposed a number of new authenticated data structures [120, 233, 177, 39, 180, 225, 63, 208]. In contrast to these work, DSM-Tree scales Ethereum's Merkle Patricia trie [19] without changing its core structure, or how proofs are generated.

Transaction execution. RainBlock adopts a design similar to Solar [243] and vCorfu [219], where transactions are executed based on data from sharded storage. RainBlock modifies the design for decentralized applications and authenticated data structures. This allows RainBlock to execute transactions on sharded state without requiring locking or additional coordination among miners. Similar to RAMCloud [160], the DSM-Tree design argues that large random-access data structures can get higher throughput and scalability when served from memory over the network.

Chapter 8: Discussion and Conclusion

In this chapter, we discuss two directions in which systems' infrastructures are headed (§8.1); resource disaggregation in datacenters and decentralization of trust in distributed systems. We outline our vision for increasing the resource utilization in fully disaggregated datacenters and implementing trust in storage systems for the wider-adoption of decentralization. Finally, we conclude this chapter by summarizing the contributions of this dissertation (§8.2).

8.1 Vision for the future

This section outlines a potential vision for extending our work in the future, considering the emerging hardware technologies and infrastructure trends. It discusses two distinct goals of improving the resource utilization in next-generation datacenters and moving towards fully decentralizing trust in storage systems.

8.1.1 Improving resource utilization in disaggregated datacenters

Data centers and cloud infrastructures are disaggregating hardware resources like memory, compute, and storage [65, 136, 139, 113, 148, 190, 94]. The emerging network fabrics [125] and interconnects like Compute Express Link (CXL) [65] enable fully disaggregated data centers [189]. Resource disaggregation supports on-demand resource provisioning and addresses the issue of under utilized resources which amounts to millions of dollars in costs [239, 139]. Moreover, this transition reduces energy consumption allows better scalability and utilization of resources [190, 188].

Elastic systems for handling composable resources. Resource disaggregation allows (de-)provisioning individual resources as per demand. For instance, it supports allocating more memory without increasing the number of CPU cores. Recent research on memory and storage harvesting VMs [239, 30, 88, 178, 239], address the

visibility of these allocations to applications at low latencies. However, underlying systems infrastructures and applications need to be able to detect such dynamic resource elasticity, and scale their performance accordingly. Thus, disaggregation calls for scalable systems that can elastically compose memory, storage, and compute.

Dynamic-tiered systems to harness heterogeneous resources. CXL enables specialized accelerators like GPUs to seamlessly collaborate with general-purpose CPUs, leading to mixed computing capabilities [189]. Furthermore, CXL introduces a new array of memory and storage performance profiles, including CXL memory, CXL PM, and CXL-attached SSDs, in addition to bus-attached memory and PM, and PCIe-attached SSDs. Recent research on real CXL hardware shows that CXL performance is hindered by the use of too many threads and is sensitive to access patterns and data distribution across CXL devices [200]. Thus, resource disaggregation calls for building tiered systems rooted in an understanding of the unique behavior of CXL devices which is crucial for ensuring high resource utilization [228, 107, 70]. These next-generation tiered systems should support both horizontal scaling across resources with similar performance and vertical scaling across resources with distinct performance. It will be crucial for these systems to optimize for high performance per dollar, addressing the diversity in the performance and cost of available resources.

Retrofitting existing systems. It can be challenging to achieve scalable performance by retrofitting existing PM key-value stores for CXL-attached PM or utilizing existing distributed databases and caches with CXL memory pools. Note that it will be important to evaluate the trade-off between reusing existing infrastructure and fundamentally redesigning systems for disaggregated infrastructures (§4). We believe that the learnings from this work will prove useful for building efficient systems for fully disaggregated datacenters that aim for high utilization and scalable performance.

8.1.2 Achieving software-defined trust in storage for decentralization

There is an increasing demand for systems that can prove their behavior without assuming their users' trust since the ownership over data is no longer with the entities storing, processing, and analyzing, that data [69]. In the recent years, decentralized storage systems have resurfaced; these systems can tolerate Byzantine behavior [201], guarantee the correctness of data [166], *etc.*

What trust do systems assume? Distributed databases for instance, assume trust when storing user data (data integrity), executing transactions (tolerate only fail-stop failures), for preserving the privacy of their data (privacy), and for disallowing an external entity to access their data (access control). Several policies that safeguard the personal data of users are just forming [106]. There is ongoing research on enforcing a few of these without assuming trust from applications. Data integrity via authenticated data structures, availability via consensus protocols, and confidentiality via privacy-preserving algorithms, can be enforced in systems [69]. Thus, it is crucial to build systems that can simultaneously enforce these policies and achieve high performance and scalability.

Challenges and opportunities. Today, there is a lack of systems that are flexible with enforcing data integrity when required. We have systems that always ensure data integrity and have poor performance [36] or that which do not guarantee it and provide better performance [58]. Thus, it is important to build systems that can fill the gap between these two extreme design choices by providing additional guarantees at higher performance costs on demand and whenever necessary.

Logs for decentralization. A shared totally-ordered log is essential for blockchains and distributed partially-ordered logs for distributed databases, etc. Therefore, with a novel logging infrastructure that allows systems and applications to choose the data consistency and integrity guarantees they need, systems can slowly move toward providing decentralized services. Similarly, minimizing the performance penalty for

applications in the common case without Byzantine failures and without requests for verifying data integrity is a primary research challenge. Further, an expressive API that meets the needs of applications and simplifies the design and architecture of the logging infrastructure is important.

8.2 Conclusion

With increase in I/O-intensive applications, existing systems infrastructures suffer from I/O bottlenecks which are inherent to their design and architecture. In this dissertation, we have discussed how I/O bottlenecks surface in different systems, and showcased their impact on end-to-end performance and scalability. We have outlined a few core techniques that minimize the performance impact of I/O bottlenecks and combined these ideas to architect three novel, scalable, high-performance systems.

First, we present Skye, a novel PM key-value store that retains fine-grained control over all PM accesses; Skye provide indirect-access to applications and manages individual NVDIMMs to obtain high and scalable PM bandwidth utilization. We demonstrate that Skye outperforms state-of-the-art PM stores by 2.5–5× on the standard Yahoo Cloud Serving Benchmark (YCSB) on a single NVDIMM. With four NVDIMMs across four NUMA nodes, Skye obtains ≈86% of PM write bandwidth, and its write throughput scales by 3.9×. The Skye prototype will be available at https://github.com/utsaslab/skye. In the future, we envision Skye supporting newer PM media.

Next, we present Cascades a distributed transactional store built over the logging infrastructure Lattice. Cascades speculates on the durability of log records that are important for consistent recovery, avoids I/O bottlenecks in the critical path of processing transactions, and relies on Lattice to manage recovery and replication in the event of failures. Thus, Cascades achieves 25–99× higher throughput with speculation and Lattice when logging to storage media with different performance characteristics. The RecoverableApplication and RecoverableProcess API

of Lattice and the prototype implementation of Cascades will be available at https://github.com/microsoft/RecoverableProcesses.

Finally, we present RainBlock, a public blockchain architecture that increases transaction throughput without modifying the consensus protocol. RainBlock achieves this by tackling the I/O bottleneck in transaction processing, allowing miners to pack more transactions into each block. RainBlock introduces a novel architecture that removes I/O from the critical path, and the DSM-Tree, a new authenticated data structure that provides cheap access to system state. A single RainBlock miner processes 27.4K transactions per second, or 27× more transactions than a single Ethereum miner. In geo-distributed settings RainBlock miners process 20K transactions per second. The RainBlock prototype is publicly available at https://github.com/RainBlock and we welcome working with the community on its adoption.

In conclusion, this dissertation evaluates the approach of fundamentally re-architecting systems to minimize I/O bottlenecks. It highlights the observed performance improvements and discusses the associated trade-offs. It highlights the benefits of specializing systems to the underlying hardware and their target applications.

References

[1] Cloudscene: Data generation. https://explodingtopics.com/blog/data-\generated-per-day.

[2] MongoDB: A cross-platform document-oriented database. https://www.mongodb.com/.

[3] Parity ethereum 2.2.11-stable. 2019. https://github.com/paritytech/parity-ethereum/releases/tag/v2.2.11.

[4] Redis: An Advanced Open Source Relational Database. https://www.postgresql.org/.

[5] Microsoft Azure Premium SSDs v2. https://learn.microsoft.com/en-us/azure/virtual-machines/disks-deploy-premium-v2?tabs=azure-cli.

[6] Remote Direct Memory Access (RDMA) Consortium. https://www.rdmaconsortium.org/.

[7] Redis: An open source, in-memory data store. https://redis.io/.

[8] Microsoft Azure Ultra Disks. https://learn.microsoft.com/en-us/azure/virtual-machines/disks-enable-ultra-ssd?tabs=azure-portal.

[9] Overview of Amazon Web Services. https://media.amazonwebservices.com/AWS_Overview.pdf, 2014.

[10] Microsoft Azure. https://azure.microsoft.com/en-us/, 2014.

[11] Visa acceptance for retailers. https://usa.visa.com/run-your-business/small-business-tools/retail.html, 2018.

[12] Fast, cheap, scalable token transfers for ethereum. https://raiden.network/, 2019.

[13] Bitcoin. https://bitcoin.org/en/, 2019.

[14] ERC20 Token Standard. https://theethereum.wiki/w/index.php/ERC20_Token_Standard, 2019.

[15] Implementation of the modified merkle patricia tree as specified in the Ethereum's yellow paper. https://github.com/ethereumjs/merkle-patricia-tree, 2019.

[16] Ethereum. https://github.com/ethereum/, 2019.

[17] Hybrid casper ffg. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1011.md, 2019.

[18] Ethereum improvement proposals repository. https://github.com/ethereum/EIPs, 2019.

[19] The modified Merkle Patricia tree. https://github.com/ethereum/wiki/wiki/Patricia-Tree, 2019.

[20] Detailed analysis of stateless client witness size, and gains from batching and multi-state roots. https://www.ethresear.ch/t/detailed-analysis-of-\-stateless-client-witness-size-and-gains-from-batching-and-multi-\-state-roots/862, 2019.

[21] A possible solution to stateless clients. https://ethresear.ch/t/a-possible-\solution-to-stateless-clients/4094, 2019.

[22] In-memory abstract-leveldown store for Node.js and browsers. https://github.com/Level/memdown, 2019.

[23] Recursive Length Prefix Encoding. https://github.com/ethereum/wiki/wiki/RLP, 2019.

[24] RocksDB — A persistent key-value store. http://rocksdb.org, 2019.

[25] Full Node sync with Default Settings. https://etherscan.io/chartsync/chaindefault, 2020.

[26] Geth ethereum 1.9.25-stable. https://github.com/ethereum/go-ethereum/tree/v1.9.25, 2020.

[27] Number of unique addresses in ethereum. https://etherscan.io/chart/address, 2021.

[28] Ethereum average gas limit chart. https://etherscan.io/chart/gaslimit, 2021.

[29] Minseon Ahn, Andrew Chang, Donghun Lee, Jongmin Gim, Jungmin Kim, Jaemin Jung, Oliver Rebholz, Vincent Pham, Krishna Malladi, and Yang Seok Ki. Enabling cxl memory expansion for in-memory database management systems. In *Data Management on New Hardware*, 2022. ISBN 9781450393782.

[30] Pradeep Ambati, Inigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini. Providing SLOs for Resource-Harvesting VMs in cloud platforms. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 735–751. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/ambati.

[31] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, page 30. ACM, 2018.

[32] Apache. Search results apache flink: Scalable stream and batch data processing. https://flink.apache.org, 2017.

[33] Cassandra Apache. Apache cassandra. *Apache cassandra*, 2013.

[34] Dmytro Apalkov, Alexey Khvalkovskiy, Steven Watts, Vladimir Nikitin, Xueti Tang, Daniel Lottis, Kiseok Moon, Xiao Luo, Eugene Chen, Adrian Ong, et al. Spin-transfer torque magnetic random access memory (stt-mram). *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 9(2):1–35, 2013.

[35] Raja Appuswamy, Angelos C Anadiotis, Danica Porobic, Mustafa K Iman, and Anastasia Ailamaki. Analyzing the impact of system architecture on the scalability of oltp engines for high-contention workloads. *Proceedings of the VLDB Endowment*, 11(2):121–134, 2017.

[36] Arvind Arasu, Badrish Chandramouli, Johannes Gehrke, Esha Ghosh, Donald Kossmann, Jonathan Protzenko, Ravi Ramamurthy, Tahina Ramananandro, Aseem Rastogi, Srinath Setty, et al. Fastver: Making data integrity a commodity. In *Proceedings of the 2021 International Conference on Management of Data*, pages 89–101, 2021.

[37] Manos Athanassoulis, Ryan Johnson, Anastasia Ailamaki, and Radu Stoica. Improving oltp concurrency through early lock release. 2009.

[38] Vivek Bagaria, Sreeram Kannan, David Tse, Giulia Fanti, and Pramod Viswanath. Prism: Deconstructing the blockchain to approach physical limits. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 585–602, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367479. doi: 10.1145/3319535.3363213. URL https://doi.org/10.1145/3319535.3363213.

[39] Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. {SPEICHER}: Securing lsm-based key-value

stores using shielded execution. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, pages 173–190, 2019.

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

[41] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed data structures over a shared log. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, pages 325–340, 2013.

[42] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, et al. Virtual consensus in delos. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 617–632, 2020.

[43] Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. Consensus in the age of blockchains. *arXiv preprint arXiv:1711.03936*, 2017.

[44] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS operating systems review*, 37(5):164–177, 2003.

[45] BCNext. The nxt cryptocurrency. https://nxt.org, November, 2013.

[46] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. Hybrids on steroids: Sgx-based high performance bft. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 222–237, 2017.

[47] Josh Benaloh and Michael De Mare. One-way accumulators: A decentralized alternative to digital signatures. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 274–285. Springer, 1993.

[48] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. Viper: an efficient hybrid pmem-dram key-value store. *Proceedings of the VLDB Endowment*, 14 (9):1544–1556, 2021.

[49] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. Viper: An efficient hybrid pmem-dram key-value store. https://github.com/hpides/viper, 2021.

[50] Philip A Bernstein, Vassos Hadzilacos, Nathan Goodman, et al. *Concurrency control and recovery in database systems*, volume 370. Addison-wesley Reading, 1987.

[51] Tim Blechmann. Boost: Lockfree concurrent queue. https://www.boost.org/doc/libs/1_66_0/doc/html/lockfree.html, 2011.

[52] Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching techniques for accumulators with applications to iops and stateless blockchains. In *Annual International Cryptology Conference*, pages 561–586. Springer, 2019.

[53] Matthew Burke, Sowmya Dharanipragada, Shannon Joyner, Adriana Szekeres, Jacob Nelson, Irene Zhang, and Dan RK Ports. Prism: Rethinking the rdma interface for distributed systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 228–242, 2021.

[54] Geoffrey W Burr, Matthew J Breitwisch, Michele Franceschini, Davide Garetto, Kailash Gopalakrishnan, Bryan Jackson, Bülent Kurdi, Chung Lam, Luis A Lastras, Alvaro Padilla, et al. Phase change memory technology. *Journal of Vacuum Science & Technology B, Nanotechnology and Microelectronics: Materials, Processing, Measurement, and Phenomena*, 28(2):223–262, 2010.

[55] Vitalik Buterin. The Stateless Clients Concept. https://ethresear.ch/t/the-stateless-client-concept/172, 2017.

[56] Christian Cachin and Marko Vukolić. Blockchain consensus protocols in the wild. *arXiv preprint arXiv:1707.01873*, 2017.

[57] Jan Camenisch and Anna Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *Annual International Cryptology Conference*, pages 61–76. Springer, 2002.

[58] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. Faster: A concurrent key-value store with in-place updates. In *Proceedings of the 2018 International Conference on Management of Data*, pages 275–290, 2018.

[59] Peter M Chen, Edward K Lee, Garth A Gibson, Randy H Katz, and David A Patterson. Raid: High-performance, reliable secondary storage. *ACM Computing Surveys (CSUR)*, 26(2):145–185, 1994.

[60] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and general distributed transactions using rdma and htm. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–17, 2016.

[61] 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 *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1077–1091, 2020.

[62] Zhangyu Chen, Yu Hua, Bo Ding, and Pengfei Zuo. Lock-free concurrent level hashing for persistent memory. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, 2020. ISBN 978-1-939133-14-4.

[63] Alexander Chepurnoy, Charalampos Papamanthou, and Yupeng Zhang. Edrax: A cryptocurrency with stateless transaction validation. 2018.

[64] Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[65] CXL Consortium. Compute express link specification revision 3.0. https://www.computeexpresslink.org/download-the-specification, 2020.

[66] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[67] Matt Corallo. Compact block relay in bitcoin core. https://github.com/bitcoin/bips/blob/master/bip-0152.mediawiki, 2019.

[68] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[69] Natacha Crooks. Efficient data sharing across trust domains. *SIGMOD Rec.*, 52(2):36–37, aug 2023. ISSN 0163-5808. doi: 10.1145/3615952.3615962. URL https://doi.org/10.1145/3615952.3615962.

[70] Björn Daase, Lars Jonas Bollmeier, Lawrence Benson, and Tilmann Rabl. Maximizing persistent memory bandwidth utilization for olap workloads. In *Proceedings of the 2021 International Conference on Management of Data*, pages 339–351, 2021.

[71] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. Towards scaling blockchain systems via sharding. In

Proceedings of the 2019 International Conference on Management of Data, SIGMOD '19, page 123–140, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450356435. doi: 10.1145/3299869.3319889. URL https://doi.org/10.1145/3299869.3319889.

[72] Jeff Dean. Designs, lessons and advice from building large distributed systems. *Keynote from LADIS*, 1, 2009.

[73] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[74] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A Wood. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD international conference on management of data*, pages 1–8, 1984.

[75] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1243–1254, 2013.

[76] Cong Ding, David Chu, Evan Zhao, Xiang Li, Lorenzo Alvisi, and Robbert Van Renesse. Scalog: Seamless reconfiguration and total order in a scalable shared log. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 325–338, 2020.

[77] Donghui Ding, Xin Jiang, Jiaping Wang, Hao Wang, Xiaobing Zhang, and Yi Sun. Txilm: Lossy block compression with salted short hashing. *arXiv preprint arXiv:1906.06500*, 2019.

[78] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. Blockbench: A framework for analyzing private blockchains.

In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1085–1100, 2017.

[79] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. {FaRM}: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, 2014.

[80] Oren Eini. Early lock release: Transactions and errors. https://dzone.com/articles/early-lock-release, Jan 16, 2014.

[81] Tamer Eldeeb, Xincheng Xie, Philip A Bernstein, Asaf Cidon, and Junfeng Yang. Chardonnay: Fast and general datacenter transactions for {On-Disk} databases. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 343–360, 2023.

[82] Jim Elliott and Jin-Hyeok Choi. Flash memory summit keynote 6: Memory innovations navigating the big data era. In *Flash Memory Summit*, Santa Clara, CA, August 2022. URL https://www.flashmemorysummit.com/English/Conference/Keynotes_2022.html. Accessed: 2022-12-13.

[83] Dawson R Engler, M Frans Kaashoek, and James O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. *ACM SIGOPS Operating Systems Review*, 29(5):251–266, 1995.

[84] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *International conference on financial cryptography and data security*, pages 436–454. Springer, 2014.

[85] Ittay Eyal, Adem Efe Gencer, Emin Gun Sirer, and Robbert Van Renesse. Bitcoin-ng: A scalable blockchain protocol. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 45–59, Santa Clara, CA, 2016. USENIX Association. ISBN 978-1-931971-29-4. URL https:

//www.usenix.org/conference/nsdi16/technical-sessions/presentation/eyal.

[86] Michael J Fischer and Alan Michael. Sacrificing serializability to attain high availability of data in an unreliable network. In *Proceedings of the 1st ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 70–75, 1982.

[87] Matt Freels. Faunadb: An architectural overview, 2018.

[88] Alexander Fuerst, Stanko Novaković, Íñigo Goiri, Gohar Irfan Chaudhry, Prateek Sharma, Kapil Arya, Kevin Broas, Eugene Bak, Mehmet Iyigun, and Ricardo Bianchini. Memory-harvesting vms in cloud platforms. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 583–594, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392051. doi: 10.1145/3503222.3507725. URL https://doi.org/10.1145/3503222.3507725.

[89] Dieter Gawlick and David Kinkade. Varieties of concurrency control in ims/vs fast path. *IEEE Database Eng. Bull.*, 8(2):3–10, 1985.

[90] Sanjay Ghemawat and Jeff Dean. Leveldb: a fast and lightweight key/value database library by google. https://github.com/google/leveldb, 2011.

[91] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 51–68, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5085-3. doi: 10.1145/3132747.3132757. URL http://doi.acm.org/10.1145/3132747.3132757.

[92] Google. Google cloud storage: Object storage for companies of all sizes. https://cloud.google.com/storage, 2008.

[93] Google. Leveldb. https://github.com/google/leveldb, 2019.

[94] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. Direct access,{High-Performance} memory disaggregation with {DirectCXL}. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 287–294, 2022.

[95] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD international Conference on Management of Data*, pages 173–182, 1996.

[96] Matthew Green and Ian Miers. Bolt: Anonymous payment channels for decentralized currencies. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 473–489, 2017.

[97] Bulbul Gupta, Pooja Mittal, and Tabish Mufti. A review on amazon web service (aws), microsoft azure & google cloud platform (gcp) services. In *Proceedings of the 2nd International Conference on ICT for Digital, Smart, and Sustainable Development, ICIDSSD 2020, 27-28 February 2020, Jamia Hamdard, New Delhi, India*, 2021.

[98] Shukai Han, Dejun Jiang, and Jin Xiong. Lightkv: A cross media key value store with persistent memory to cut long tail latency. In *Proceedings of the 36th International Conference on Massive Storage Systems and Technology (MSST'20)*, 2020.

[99] Steven Hand, Andrew Warfield, Keir Fraser, Evangelos Kotsovinos, and Daniel J Magenheimer. Are virtual machine monitors microkernels done right? In *HotOS*, 2005.

[100] Jeff Heaton. Ian goodfellow, yoshua bengio, and aaron courville: Deep learning: The mit press, 2016, 800 pp, isbn: 0262035618. *Genetic programming and evolvable machines*, 19(1-2):305–307, 2018.

[101] 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. Springer, 1987.

[102] Daokun Hu, Zhiwen Chen, Jianbing Wu, Jianhua Sun, and Hao Chen. Persistent memory hash indexes: An experimental evaluation. volume 14, page 785–798. VLDB Endowment, jan 2021.

[103] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. Tidb: a raft-based htap database. *Proceedings of the VLDB Endowment*, 13(12):3072–3084, 2020.

[104] Hyperledger. Hyperledger sawtooth. https://www.hyperledger.org/projects/sawtooth, 2019.

[105] Intel. Avdanced vector extension (avx) instructions. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html, 2017.

[106] Zsolt István, Soujanya Ponnapalli, and Vijay Chidambaram. Software-defined data protection: Low overhead policy compliance at the storage layer is within reach! *Proc. VLDB Endow.*, 14(7):1167–1174, mar 2021. ISSN 2150-8097. doi: 10.14778/3450980.3450986. URL https://doi.org/10.14778/3450980.3450986.

[107] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. Basic performance measurements of the intel optane dc persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.

[108] Markus Jakobsson and Ari Juels. Proofs of work and bread pudding protocols. In *Communications and Multimedia Security*, 1999.

[109] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. Aether: a scalable approach to logging. *Proceedings of the VLDB Endowment*, 3(1-2):681–692, 2010.

[110] Jithin Jose, Hari Subramoni, Krishna Kandalla, Md Wasi-ur Rahman, Hao Wang, Sundeep Narravula, and Dhabaleswar K Panda. Scalable memcached design for infiniband clusters using hybrid transports. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 236–243. IEEE, 2012.

[111] Thaddeus Dryja Joseph Poon. The bitcoin lightning network: Scalable off-chain instant payments. https://lightning.network/lightning-network-paper.pdf, 2019.

[112] Vitalik Buterin Joseph Poon. Plasma: Scalable autonomous smart contracts. https://plasma.io/plasma.pdf, 2019.

[113] Myoungsoo Jung. Hello bytes, bye blocks: Pcie storage meets compute express link for memory expansion (cxl-ssd). In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, pages 45–51, 2022.

[114] 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 *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 494–508, 2019.

[115] Rohan Kadekodi, Saurabh Kadekodi, Soujanya Ponnapalli, Harshad Shirwadkar, Gregory R Ganger, Aasheesh Kolli, and Vijay Chidambaram. Winefs: a hugepage-aware file system for persistent memory that ages gracefully. In

Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, pages 804–818, 2021.

[116] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H Noh, and Young-ri Choi. {SLM-DB}:{Single-Level}{Key-Value} store with persistent memory. In *17th USENIX Conference on File and Storage Technologies*, pages 191–205, 2019.

[117] Anuj Kalia. Efficient remote procedure calls for datacenters.

[118] Anuj Kalia, Michael Kaminsky, and David G Andersen. {FaSST}: Fast, scalable and simple distributed transactions with {Two-Sided}({{{{{RDMA}}}}}) datagram {RPCs}. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 185–201, 2016.

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

[120] Janakirama Kalidhindi, Alex Kazorian, Aneesh Khera, and Cibi Pari. Angela: A sparse, distributed, and highly concurrent merkle tree. 2018.

[121] Sudarsun Kannan, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, Yuangang Wang, Jun Xu, and Gopinath Palani. Designing a true {Direct-Access} file system with {DevFS}. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 241–256, 2018.

[122] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning {LSMs} for nonvolatile memory with {NoveLSM}. In *2018 USENIX Annual Technical Conference*, pages 993–1005, 2018.

[123] Rami Khalil and Arthur Gervais. Revive: Rebalancing off-blockchain payment networks. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 439–453, 2017.

[124] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017*, pages 357–388, Cham, 2017. Springer International Publishing. ISBN 978-3-319-63688-7.

[125] Ted H Kim. Brief history of infiniband: Hype to pragmatism. *Musing of a Random Dude (blog)*, 2004.

[126] Wonbae Kim, Chanyeol Park, Dongui Kim, Hyeongjun Park, Young ri Choi, Alan Sussman, and Beomseok Nam. ListDB: Union of Write-Ahead logs and persistent SkipLists for incremental checkpointing on persistent memory. In *16th USENIX Symposium on Operating Systems Design and Implementation*, pages 161–177, 2022. ISBN 978-1-939133-28-1.

[127] Wook Hee Kim, R. Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. Pactree: A high performance persistent range index using pac guidelines. pages 424–439, 10 2021. ISBN 9781450387095.

[128] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 583–598, May 2018. doi: 10.1109/SP.2018.000-5.

[129] Heine Kolltveit and Svein-Olaf Hvasshovd. Main memory commit protocols for multiple backups. In *Seventh International Conference on Networking (icn 2008)*, pages 479–484. IEEE, 2008.

[130] Hsiang-Tsung Kung and John T Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[131] Cockroach Labs. Cockroachdb. https://github.com/cockroachdb/cockroach, 2017.

[132] Leslie Lamport. The part-time parliament. *FAST*, 3:15–30, 2004.

[133] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*, pages 179–196. 2019.

[134] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[135] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe: Converting concurrent dram indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 462–477, 2019.

[136] Sekwon Lee, Soujanya Ponnapalli, Sharad Singhal, Marcos K. Aguilera, Kimberly Keeton, and Vijay Chidambaram. Dinomo: An elastic, scalable, high-performance key-value store for disaggregated persistent memory. *Proc. VLDB Endow.*, 15(13):4023 – 4037, 2022. ISSN 2150-8097. doi: 10.14778/3565838.3565854. URL https://doi.org/10.14778/3565838.3565854.

[137] Yoad Lewenberg, Yonatan Sompolinsky, and Aviv Zohar. Inclusive block chain protocols. In *International Conference on Financial Cryptography and Data Security*, pages 528–547. Springer, 2015.

[138] Chenxing Li, Peilun Li, Dong Zhou, Wei Xu, Fan Long, and Andrew Yao. Scaling nakamoto consensus to thousands of transactions per second. *arXiv preprint arXiv:1805.03870*, 2018.

[139] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 574–587, 2023.

[140] Joshua Lind, Oded Naor, Ittay Eyal, Florian Kelbert, Peter Pietzuch, and Emin Gün Sirer. Teechain: Reducing storage costs on the blockchain with offline payment channels. In *Proceedings of the 11th ACM International Systems and Storage Conference*, pages 125–125, 2018.

[141] Jiuxing Liu, Jiesheng Wu, Sushmitha P Kini, Pete Wyckoff, and Dhabaleswar K Panda. High performance rdma-based mpi implementation over infiniband. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 295–304, 2003.

[142] Marta Lokhava, Giuliano Losa, David Mazières, Graydon Hoare, Nicolas Barry, Eli Gafni, Jonathan Jove, Rafał Malinowsky, and Jed McCaleb. Fast and secure global payments with stellar. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 80–96, 2019.

[143] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. Dash: Scalable hashing on persistent memory. *Proc. VLDB Endow.*, 13(8):1147–1161, 2020. ISSN 2150-8097.

[144] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 17–30, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4139-4. doi: 10.1145/2976749.2978389. URL http://doi.acm.org/10.1145/2976749.2978389.

[145] Loi Luu, Yaron Velner, Jason Teutsch, and Prateek Saxena. Smartpool: Practical decentralized pooled mining. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1409–1426, Vancouver, BC, 2017. USENIX Association. ISBN 978-1-931971-40-9. URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/luu.

[146] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. ROART: Range-query optimized persistent ART. In *19th USENIX Conference on File and Storage Technologies*, pages 1–16, 2021. ISBN 978-1-939133-20-5.

[147] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, et al. Jitsu:{Just-In-Time} summoning of unikernels. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 559–573, 2015.

[148] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. Tpp: Transparent page placement for cxl-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, page 742–755, 2023. ISBN 9781450399180.

[149] Memcached. Free & open source, high-performance, distributed memory object caching system. https://memcached.org/, 2023.

[150] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*, pages 369–378. Springer, 1987.

[151] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 31–42, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4139-4. doi: 10.1145/2976749.2978399. URL http://doi.acm.org/10.1145/2976749.2978399.

[152] Andrew Miller, Iddo Bentov, Ranjit Kumaresan, and Patrick McCorry. Sprites: Payment channels that go faster than lightning. *CoRR abs/1702.05812*, 306, 2017.

[153] Mitar Milutinovic, Warren He, Howard Wu, and Maxinder Kanwal. Proof of luck: An efficient blockchain consensus protocol. In *Proceedings of the 1st Workshop on System Software for Trusted Execution*, SysTEX '16, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450346702. doi: 10.1145/3007788.3007790. URL https://doi.org/10.1145/3007788.3007790.

[154] C Mohan and Bruce Lindsay. Efficient commit protocols for the tree of processes model of distributed transactions. *ACM SIGOPS Operating Systems Review*, 19(2):40–52, 1985.

[155] Chandrasekaran Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)*, 17(1):94–162, 1992.

[156] Moohyeon Nam, Hokeun Cha, Young ri Choi, Sam H. Noh, and Beomseok Nam. Write-Optimized dynamic hashing for persistent memory. In *17th USENIX Conference on File and Storage Technologies*, pages 31–44, Boston, MA, 2019. ISBN 978-1-939133-09-0.

[157] Nvdia. A comparison of mellanox network adapters. https://www.enterprise\-support.nvidia.com/s/article/mellanox-adapters---comparison-table, 2022.

[158] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[159] Oracle. Oracle asynchronous commit: Oracle database advanced application developer's guide. http://download.oracle.com/docs/cd/B19306_01/appdev.102/b14251/adfns_sqlproc.htm.

[160] John K. Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen M. Rumble, Ryan Stutsman, and Stephen Yang. The ramcloud storage system. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, 2015. doi: 10.1145/2806887. URL https://doi.org/10.1145/2806887.

[161] A. Pinar Ozisik, Gavin Andresen, Brian N. Levine, Darren Tapp, George Bissias, and Sunny Katkuri. Graphene: Efficient interactive set reconciliation applied to blockchain propagation. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 303–317, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450359566. doi: 10.1145/3341302.3342082. URL https://doi.org/10.1145/3341302.3342082.

[162] Jianli Pan and James McElhannon. Future edge cloud and edge computing for internet of things applications. *IEEE Internet of Things Journal*, 5(1):439–449, 2017.

[163] D Stott Parker, Gerald J Popek, Gerard Rudisin, Allen Stoughton, Bruce J Walker, Evelyn Walton, Johanna M Chow, David Edwards, Stephen Kiser, and Charles Kline. Detection of mutual inconsistency in distributed systems. *IEEE transactions on Software Engineering*, (3):240–247, 1983.

[164] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems (TOCS)*, 33(4):1–30, 2015.

[165] Pmem-Redis. Redis v4.0.0 that supports intel optane dcpmm (data center persistent memory. https://github.com/pmem/pmem-redis, 2019.

[166] Soujanya Ponnapalli, Aashaka Shah, Amy Tai, Souvik Banerjee, Vijay Chidambaram, Dahlia Malkhi, and Michael Wei. Rainblock: Faster transaction processing in public blockchains, 2020.

[167] Soujanya Ponnapalli, Aashaka Shah, Souvik Banerjee, Dahlia Malkhi, Amy Tai, Vijay Chidambaram, and Michael Wei. {RainBlock}: Faster transaction processing in public blockchains. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 333–347, 2021.

[168] PostgreSQL. Postgresql asynchronous commit: Postgresql documentation. http://www.postgresql.org/files/documentation/pdf/8.4/postgresql-8.4.2-\A4.pdf.

[169] Vijayan Prabhakaran, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Analysis and evolution of journaling file systems. In *USENIX Annual Technical Conference, General Track*, volume 194, pages 196–215, 2005.

[170] Guna Prasaad, Alvin Cheung, and Dan Suciu. Handling highly contended oltp workloads using fast dynamic partitioning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 527–542, 2020.

[171] Intel Corporation Products. Intel optane dc persistent memory module. https://www.intel.com/content/www/us/en/products/details/memory-storage/optane-dc-persistent-memory.html, 2019.

[172] Intel Corporation Products. Intel memory latency checker v3.9a. https://www.intel.com/content/www/us/en/developer/articles/tool\/\intelr-memory-latency-checker.html, 2019.

[173] Abbas Rafii and Donald DuBois. Performance tradeoffs of group commit logging. In *Int. CMG Conference*, pages 164–176, 1989.

[174] Tayebeh Rajab, Mohammad Hossein Manshaei, Mohammad Dakhilalian, Murtuza Jadliwala, and Mohammad Ashiqur Rahman. On the feasibility of sybil attacks in shard-based permissionless blockchains. *arXiv preprint arXiv:2002.06531*, 2020.

[175] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*, Shanghai, China, October 2017.

[176] Pandian Raju, Soujanya Ponnapalli, Evan Kaminsky, Gilad Oved, Zachary Keener, Vijay Chidambaram, and Ittai Abraham. mLSM: Making Authenticated Storage Faster in Ethereum. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, Boston, MA, 2018. USENIX Association.

[177] Pandian Raju, Soujanya Ponnapalli, Evan Kaminsky, Gilad Oved, Zachary Keener, Vijay Chidambaram, and Ittai Abraham. mLSM: Making Authenticated Storage Faster in Ethereum. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, Boston, MA, 2018. USENIX Association.

[178] Benjamin Reidys, Jinghan Sun, Anirudh Badam, Shadi Noghabi, and Jian Huang. BlockFlex: Enabling storage harvesting with Software-Defined flash

in modern cloud platforms. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 17–33, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/reidys.

[179] Kun Ren, Jose M Faleiro, and Daniel J Abadi. Design principles for scaling multi-core oltp under high contention. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1583–1598, 2016.

[180] Leonid Reyzin, Dmitry Meshkov, Alexander Chepurnoy, and Sasha Ivanov. Improving authenticated dynamic dictionaries, with applications to cryptocurrencies. In *International Conference on Financial Cryptography and Data Security*, pages 376–392. Springer, 2017.

[181] Dennis M Ritchie and Ken Thompson. The unix time-sharing system. *Communications of the ACM*, 17(7):365–375, 1974.

[182] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, feb 1992. ISSN 0734-2071. doi: 10.1145/146941.146943. URL https://doi.org/10.1145/146941.146943.

[183] Andy Rudoff. Persistent memory programming. *Login: The Usenix Magazine*, 42(2):34–40, 2017.

[184] Jerome H Saltzer, David P Reed, and David D Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS)*, 2(4): 277–288, 1984.

[185] Steve Scargall. pmemkv: A persistent in-memory key-value store. In *Programming Persistent Memory*, pages 141–153. Springer, 2020.

[186] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[187] Amazon Web Services. Amazon s3: Object storage built to retrieve any amount of data from anywhere. https://aws.amazon.com/s3/?nc2=h_ql_prod_fs_s3, 2006.

[188] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. {LegoOS}: A disseminated, distributed {OS} for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 69–87, 2018.

[189] Yizhou Shan, Will Lin, Zhiyuan Guo, and Yiying Zhang. Towards a fully disaggregated and programmable data center. In *Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '22, page 18–28, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450394413. doi: 10.1145/3546591.3547527. URL https://doi.org/10.1145/3546591.3547527.

[190] Yizhou Shan, Will Lin, Zhiyuan Guo, and Yiying Zhang. Towards a fully disaggregated and programmable data center. In *Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 18–28, 2022.

[191] Dharma Shukla. Azure cosmos db: Pushing the frontier of globally distributed databases, 2018.

[192] Swaminathan Sivasubramanian. Amazon dynamodb: a seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 729–730, 2012.

[193] Dale Skeen. Nonblocking commit protocols. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 133–142, 1981.

[194] Eljas Soisalon-Soininen and Tatu Ylönen. Partial strictness in two-phase locking. In *International Conference on Database Theory*, pages 139–147. Springer, 1995.

[195] Yonatan Sompolinsky and Aviv Zohar. Secure high-rate transaction processing in bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 507–527. Springer, 2015.

[196] Alberto Sonnino, Shehar Bano, Mustafa Al-Bassam, and George Danezis. Replay attacks and defenses against cross-shard consensus in sharded distributed ledgers. *arXiv preprint arXiv:1901.11218*, 2019.

[197] Thomas Sterling, Maciej Brodowicz, and Matthew Anderson. *High performance computing: modern systems and practices*. Morgan Kaufmann, 2017.

[198] Michael Stonebraker and Ariel Weisberg. The voltdb main memory dbms. *IEEE Data Eng. Bull.*, 36(2):21–27, 2013.

[199] Michael Stonebraker, Samuel Madden, Daniel J Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: It's time for a complete rewrite. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*, pages 463–489. 2018.

[200] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Ipoom Jeong, Ren Wang, and Nam Sung Kim. Demystifying cxl memory with genuine cxl-ready systems and devices, 2023.

[201] Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Lorenzo Alvisi, and Natacha Crooks. Basil: Breaking up bft with acid (transactions). In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 1–17, 2021.

[202] Nick Szabo. Smart contracts. *Unpublished manuscript*, 1994.

[203] Peter Szilagyi. Are stateless clients a dead end? https://www.reddit.com/r/ethereum/comments/e8ujfy/are_stateless_clients_a_dead_end/, December, 10, 2019.

[204] NuoDB Team. Nuodb: Distributed relational sql database. https://www.3ds.com/nuodb-distributed-sql-database/.

[205] Scylla Team. Scylladb.[mar. 20, 2018], 2015.

[206] YugabyteDB Team. Yugabytedb: cloud native distributed sql database for mission-critical applications, 2021.

[207] Douglas B Terry, Alan J Demers, Karin Petersen, Mike J Spreitzer, Marvin M Theimer, and Brent B Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*, pages 140–149. IEEE, 1994.

[208] Alin Tomescu, Ittai Abraham, Vitalik Buterin, Justin Drake, Dankrad Feist, and Dmitry Khovratovich. Aggregatable subvector commitments for stateless cryptocurrencies. *IACR Cryptol. ePrint Arch.*, 2020:527, 2020.

[209] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1041–1052, 2017.

[210] Jyri J. Virkki. Libbloom - a simple and small bloom filter implementation in plain c. http://www.virkki.com/libbloom/, 2012.

[211] Vitalik Buterin. Toward a 12-second Block Time. https://blog.ethereum.org/2014/07/11/toward-a-12-second-block-time/, 2014.

[212] Marko Vukolić. The quest for scalable blockchain fabric: Proof-of-work vs. bft replication. In *International workshop on open problems in network security*, pages 112–125. Springer, 2015.

[213] Jiaping Wang and Hao Wang. Monoxide: Scale out blockchains with asynchronous consensus zones. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 95–112, Boston, MA, February 2019. USENIX Association. ISBN 978-1-931971-49-2. URL https://www.usenix.org/conference/nsdi19/presentation/wang-jiaping.

[214] Jing Wang, Youyou Lu, Qing Wang, Minhui Xie, Keji Huang, and Jiwu Shu. Pacman: An efficient compaction approach for {Log-Structured}{Key-Value} store on persistent memory. In *2022 USENIX Annual Technical Conference*, pages 773–788, 2022.

[215] Qing Wang, Youyou Lu, Junru Li, and Jiwu Shu. Nap: A Black-Box approach to NUMA-Aware persistent memory indexes. In *15th USENIX Symposium on Operating Systems Design and Implementation*, pages 93–111, 2021. ISBN 978-1-939133-22-9.

[216] Tianzheng Wang and Hideaki Kimura. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *Proceedings of the VLDB Endowment*, 10(2):49–60, 2016.

[217] James Warren and Nathan Marz. *Big Data: Principles and best practices of scalable realtime data systems*. Simon and Schuster, 2015.

[218] Michael Wei, Amy Tai, Christopher J. Rossbach, Ittai Abraham, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Udi Wieder, Scott Fritchie, Steven Swanson, Michael J. Freedman, and Dahlia Malkhi. Vcorfu: A cloud-scale object store on a shared log. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, page 35–49, USA, 2017. USENIX Association. ISBN 9781931971379.

[219] Michael Wei, Amy Tai, Christopher J Rossbach, Ittai Abraham, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Udi Wieder, Scott Fritchie, Steven Swanson, et al. vcorfu: A cloud-scale object store on a shared log. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 35–49, 2017.

[220] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 87–104, 2015.

[221] John H Wensley, Leslie Lamport, Jack Goldberg, Milton W Green, Karl N Levitt, Po Mo Melliar-Smith, Robert E Shostak, and Charles B Weinstock. Sift: Design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE*, 66(10):1240–1255, 1978.

[222] Jeffrey Wilcke. The Ethereum network is currently undergoing a DoS attack. https://www.ethereum.github.io/blog/2016/09/22/ethereum-network\-\currently-undergoing-dos-attack/, 2016.

[223] Jiesheng Wu, Pete Wyckoff, and Dhabaleswar Panda. Pvfs over infiniband: Design and performance evaluation. In *2003 International Conference on Parallel Processing, 2003. Proceedings.*, pages 125–132. IEEE, 2003.

[224] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. {HiKV}: A hybrid index {Key-Value} store for {DRAM-NVM} memory systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 349–362, 2017.

[225] Cheng Xu, Ce Zhang, and Jianliang Xu. vchain: Enabling verifiable boolean range queries over blockchain databases. In *Proceedings of the 2019 international conference on management of data*, pages 141–158, 2019.

[226] Jian Xu and Steven Swanson. {NOVA}: A log-structured file system for hybrid {Volatile/Non-volatile} main memories. In *14th USENIX Conference on File and Storage Technologies*, pages 323–338, 2016.

[227] J Joshua Yang and R Stanley Williams. Memristive devices in computing system: Promises and challenges. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 9(2):1–20, 2013.

[228] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies*, pages 169–182, 2020.

[229] Lei Yang, Vivek Bagaria, Gerui Wang, Mohammad Alizadeh, David Tse, Giulia Fanti, and Pramod Viswanath. Prism: Scaling bitcoin by 10,000 x. *arXiv preprint arXiv:1909.11261*, 2019.

[230] Renlord Yang, Toby Murray, Paul Rimba, and Udaya Parampalli. Empirically analyzing ethereum's gas mechanism. In *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 310–319. IEEE, 2019.

[231] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. MatrixKV: Reducing write stalls and write amplification in LSM-tree based KV stores with matrix container in NVM. In *2020 USENIX Annual Technical Conference*, pages 17–31, 2020. ISBN 978-1-939133-14-4.

[232] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus in the lens of blockchain. *arXiv preprint arXiv:1803.05069*, 2018.

[233] Mingchao Yu, Saeid Sahraei, Songze Li, Salman Avestimehr, Sreeram Kannan, and Pramod Viswanath. Coded merkle tree: Solving data availability attacks in blockchains. *arXiv preprint arXiv:1910.01247*, 2019.

[234] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. 2014.

[235] Jusik Yun, Yunyeong Goh, and Jong-Moon Chung. Trust-based shard distribution scheme for fault-tolerant shard blockchain networks. *IEEE Access*, 7: 135164–135175, 2019.

[236] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 931–948, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5693-0. doi: 10.1145/3243734.3243853. URL http://doi.acm.org/10.1145/3243734.3243853.

[237] Erfan Zamanian, Carsten Binnig, Tim Kraska, and Tim Harris. The end of a myth: Distributed transactions can scale. *arXiv preprint arXiv:1607.00655*, 2016.

[238] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. Chameleondb: a key-value store for optane persistent memory. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 194–209, 2021.

[239] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 724–739, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450387095. doi: 10.1145/3477132.3483580. URL https://doi.org/10.1145/3477132.3483580.

[240] Diyu Zhou, Yuchen Qian, Vishal Gupta, Zhifei Yang, Changwoo Min, and Sanidhya Kashyap. {ODINFS}: Scaling {PM} performance with opportunistic delegation. In *16th USENIX Symposium on Operating Systems Design and Implementation*, pages 179–193, 2022.

[241] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J Beamon, Rusty Sears, John Leach, et al. Foundationdb: A distributed unbundled transactional key value store. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2653–2666, 2021.

[242] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. Dptree: Differential indexing for persistent memory. volume 13, pages 421–434. VLDB Endowment, 12 2019.

[243] Tao Zhu, Zhuoyue Zhao, Feifei Li, Weining Qian, Aoying Zhou, Dong Xie, Ryan Stutsman, Haining Li, and Huiqi Hu. Solar: towards a shared-everything database on distributed log-structured storage. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 795–807, 2018.

[244] Pengfei Zuo, Yu Hua, and Jie Wu. Write-Optimized and High-Performance hashing index scheme for persistent memory. In *13th USENIX Symposium on Operating Systems Design and Implementation*, pages 461–476, 2018. ISBN 978-1-939133-08-3.