The Dissertation Committee for Sekwon Lee
certifies that this is the approved version of the following dissertation:

Designing Key-Value Stores for Emerging Memory and Disaggregation Technologies

Committee:

Vijay Chidambaram, Supervisor

Christopher J. Rossbach

James Bornholt

Kimberly Keeton

Marcos K. Aguilera

Designing Key-Value Stores for Emerging Memory and Disaggregation Technologies

by

Sekwon Lee

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

Dedicated to my wonderful family:

My constant source of inspiration and motivation.

This wouldn't have been possible without your love and support.

Acknowledgments

The completion of this dissertation has been a remarkable journey, filled with challenges, triumphs, and moments of profound personal growth. I am deeply indebted to the countless individuals who have enriched my academic experience and provided unwavering support throughout this endeavor.

First and foremost, I would like to express my heartfelt gratitude to my beloved family, whose unwavering love and encouragement have been the bedrock of my success. Their unconditional belief in my abilities has been a constant source of inspiration and motivation, propelling me forward even during the most difficult times.

I am immensely grateful to my advisor, Vijay Chidambaram, whose guidance, wisdom, and mentorship have been instrumental in shaping my academic development. Vijay has consistently challenged me to think critically, expand my intellectual horizons, and strive for excellence in all that I do. He is a true leader who united our lab members. He always put the lives of his students first and did not spare his support, even during the toughest of times like COVID pandemic periods. Without his support, guidance, and help, I would not be where I am today. Vijay, you were an amazing advisor. I am deeply grateful for the opportunity to have had you in my life. I will never forget the time I spent under your guidance.

I would also like to extend my deepest appreciation to my mentors, Kimberly Keeton, Sharad Singhal, and Marcos K. Aguilera, who have played a pivotal role in my professional and personal growth. Their insights, expertise, and unwavering support have been invaluable as I navigated the complexities of my research. Kimberly, I first met you at the FAST conference in 2017. From then on, your interest in my research and your encouragement were a great source of motivation and inspiration for me. I am so fortunate to have had the opportunity to work with you as my mentor for the past seven years. You taught me that kindness is the most valuable virtue, and your kindness always gave me the strength to believe in myself and achieve my goals.

Sharad, I am so grateful for your continued dedication to me since my internship at Hewlett Packard Labs in 2019. Your perspectives and insights on practical research has inspired me to think beyond the academic world and to consider the real-world applications of my research. Your sincere research and career advice have been a great inspiration to me, and they have been very helpful in shaping my post-doctoral career path. Marcos, meeting you through Kimberly and Sharad during my internship in 2019 was a truly lucky break for me. I am so grateful for your guidance and support throughout the DINOMO project, even though the project often progressed slowly or faced communication difficulties. Your patience, understanding, and expertise were invaluable to me.

I am deeply grateful to my dissertation committee members, Christopher J. Rossbach and James Bornholt, for their invaluable guidance and feedback to my dissertation. Their insightful questions and comments have undoubtedly enhanced the quality of this dissertation. I would also like to extend my heartfelt appreciation to Katie Traughber Dahm, Gabrielle Bouzigard, Lydia Griffith, Eva Fox, and the entire staff at UTCS for their kindness and support during my time at UT. Their prompt assistance with administrative matters has been instrumental in my progress.

I was unbelievably fortunate to have wonderful labmates, Supreeth Shastri, Jayashree Mohan, Rohan Kadekodi, Soujanya Ponnapalli, Aashaka Shai, Aastha Tripathi, Hayley LeBlanc, and Yeonju Ro. The UT SaSLab members have been a constant source of camaraderie, intellectual stimulation, and friendship. Their collaborative spirit, infectious enthusiasm, and willingness to share their knowledge have made my time in the lab a truly enriching experience. With heartfelt gratitude, I extend my special thanks to my cherished friends, Rohan, Jayashree, Soujanya, and Aashaka, who have been my unwavering pillars of support throughout the rollercoaster ride of graduate life. I never could have imagined forming such deep friendships and family-like bonds with people from different corners of the world. Your warm and genuine support has been a constant source of comfort and encouragement throughout my time at UT. The memories we have created together, both joyful and bittersweet,

will forever be etched in my heart. I hope you always find happiness and success in whatever you do, wherever you are.

I would like to express my sincere appreciation to my friends studying together at UT Austin, Hochan Lee, Deukyeon Hwang, Dae Yeol Lee, Jay Whang, Taeklim Kim, Jeho Oh, Wonjoon Goo, and Yingchen Wang, who have provided me with unwavering support and friendship throughout my doctoral journey. Their presence in my life has been a constant source of joy and comfort. With profound gratitude, I extend my heartfelt thanks to Hochan's family including Jarim Seo for their constant consideration, kindness, and friendship throughout my time at Austin. I have learned from you the joy of sharing and the importance of a positive attitude in life. I never thought I would find such compatible best friends in a foreign country. Starting my life in Austin with you was truly a blessing.

I would also like to express my gratitude to my former advisor and colleagues, Sam H. Noh, Beomseok Nam, Changhee Jung, Taesoo Kim, Hyunsub Song, and Sanidhya Kashyap for their invaluable support and encouragement. Their help has made it possible for me to pursue my PhD and achieve many other goals.

I want to acknowledge the generous financial support I have received throughout my PhD journey. I am thankful to Microsoft Research, National Science Foundation, and ACM Special Interest Group in Operating Systems for generously supporting my PhD through the 2021 Microsoft Research PhD fellowship, SOSP 2019 student travel scholarship, and VLDB 2023 NSF travel fellowship. Their support has played a pivotal role in enabling me to pursue my academic aspirations and achieve my research goals.

Finally, I would like to extend my deepest gratitude to my girlfriend, Xinyi He, whose love, support, and understanding have been the foundation upon which I have built my success. Xinyi He has been my unwavering confidante, my source of strength, and my constant source of inspiration. I am eternally grateful for her presence in my life.

Abstract

Designing Key-Value Stores for Emerging Memory and Disaggregation Technologies

Sekwon Lee, PhD
The University of Texas at Austin, 2023

SUPERVISOR: Vijay Chidambaram

With the increasing convergence of applications to the cloud, cloud-based key-value stores (KVSs) should offer high performance, scalability, elasticity, utilization, and crash resilience. However, conventional storage technologies and monolithic server models make it challenging to achieve these goals. The transition to the new emerging memory and disaggregation technologies, such as PM (Persistent Memory), RDMA (Remote Direct Memory Access), and CXL (Compute Express Link), can readily offer opportunities to achieve these goals. However, these new technologies have distinct characteristics from the conventional technologies. Thus, to efficiently and reliably utilize them, KVSs must be carefully designed to avoid sub-optimal design choices without compromising their inherent hardware-guaranteed benefits.

In this dissertation, we seek to answer the following question: how can we achieve a high-performance, scalable, elastic, and crash-recoverable KVS for disaggregated persistent memory (DPM)? In particular, we explore solutions to achieve these goals by introducing new indexing, caching, and partitioning techniques. We design new indexing data structures for a high-performance, scalable, and crash-recoverable data storage at PM, employ caching strategies for high performance by reducing expensive accesses to DPM, and tailor partitioning techniques to achieve

elastic, scalable resource deployment.

This dissertation first presents RECIPE, a principled approach for converting concurrent DRAM indexes to crash-consistent indexes for PM. The main insight behind RECIPE is that isolation provided by a certain class of concurrent DRAM indexes can be translated to crash consistency when the same index is used in PM. We present a set of conditions that enable the identification of this class of DRAM indexes, and the actions to be taken to convert each index to be persistent. Next, we presents DINOMO, the first key-value store for DPM based on RDMA interconnects that simultaneously achieves high common-case performance, scalability, and elasticity. DINOMO uses a novel combination of techniques such as ownership partitioning, disaggregated adaptive caching, selective replication, and lock-free and log-free PM indexing to achieve these goals. Finally, we present SHIFT, a cache-conscious KVS designs for CXL disaggregated memory. SHIFT sheds new light on the existing PM indexes and partitioning schemes originally proposed for the different system domains to achieve a high-performance, scalable, elastic, crash-recoverable KVS for CXL disaggregated memory. Furthermore, SHIFT employs lock intention log to improve the PM indexes to be partial-failure-resilient and non-hierarchical processing to take both advantages of KN cache and direct accesses to CXL disaggregated memory.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Key-Value Stores (KVSs) are critical pieces of software infrastructure. KVSs provides a high-performance and scalable data storage by employing a simple key-value abstraction (*e.g.,* get(key), put(key, value)) [51]. These systems have been employed for diverse data storage and large-scale Internet services, such as web object caching [155], backend storage engines for DBMS [83], state storage for machine learning [126], web pages/analytics, best seller lists, shopping carts, customer preferences, session management, and product catalog [51].

As many applications recently converges to the cloud infrastructures, the cloud-based KVSs need to handle a vast amount of data with dynamic working sets/sizes, and non-uniform workloads with varying skew [156, 171, 204, 212]. Thus, the efficient and reliable data storage and I/O are extremely important to handle the vast amount of data effectively. Furthermore, an elastic resource deployment is necessary for meeting the Service Level Objectives (SLOs) from the large variations in workloads [28, 219]. Finally, the KVSs need to keep resource utilization high for a cost-efficient service deployment.

Conventional slow storage devices (*e.g.,* HDDs, SSDs) and monolithic server models, however, make it challenging to achieve these goals. New emerging memory and disaggregation technologies, such as Persistent Memory (PM), RDMA (Remote Direct Memory Access), and CXL (Compute eXpress Link), have evolved over the last decade. Transition to these new technologies can readily offer opportunities to improve the data storage and I/O costs, elasticity, and resource utilization. However, these new technologies have unique performance specifications, correctness semantics, and architectural goals/benefits that differ from conventional ones. Thus, to efficiently and reliably utilize them, KVSs must be carefully designed to avoid sub-optimal design choices without compromising their inherent hardware-guaranteed benefits.

In this dissertation, we seek to answer the following question: **how can we achieve a high-performance, scalable, elastic, and crash-recoverable KVS for the new emerging memory and disaggregation technologies?** In particular, we explore solutions to achieve these goals by introducing new indexing, caching, and partitioning techniques. We design new indexing data structures for a high-performance, scalable, and crash-recoverable data storage, employ caching strategies to enhance the performance of KVSs, and tailor partitioning techniques to achieve elastic, scalable resource deployment.

The remainder of this chapter presents a concise overview of our target technologies, the challenges encountered in building the KVSs for the emerging technologies, and our proposed solutions.

1.1 Emerging memory and disaggregation technologies

PM is an emerging class of memory technology such as phase-change memory [203], spin-transfer torque MRAM [10], Intel Optane DC PM [86], and Samsung Memory-Semantic CXL SSD [56]. PM can be attached to the memory bus and accessed like DRAM via processor loads and stores. It is non-volatile and high-capacity like traditional storage devices, but has high performance close to DRAM. The low latency and durability of PM make it an attractive medium for building KVSs. However, since PM has a much higher cost per GB than conventional storage devices [8], it is critical to achieve high utilization in deploying PM.

One promising way to increase resource utilization is resource disaggregation [9, 64, 97, 128]. Resource disaggregation, where resources like CPU, memory, and storage are pooled and shared over a high-speed interconnect, can significantly enhance resource utilization by enabling independent-resource scaling [98]; for example, memory can be added without the need to also add CPU or storage. This approach also enables a separation of failure domains [29], ensuring that a failure in one resource does not affect others. Disaggregation has been successfully applied

to storage (NAS [68], SAN [15]), and we extend this concept to Disaggregated PM (DPM) for further utilization gains.

DPM is still under active research and development, and hence there are different kinds of DPM to build upon. This dissertation assumes that DPM is available as a centralized, reliable pool accessible via the high-performance interconnects [98]. We further assume the KVSs built on DPM consists of a number of KVS nodes (KNs) that are equipped with general-purpose processors, a relatively small amount of local DRAM, and use the high-performance interconnects to access DPM efficiently [196]. Depending on the interconnect technologies, the DPM settings can be classified into RDMA or CXL disaggregated memory. Note that we collectively call both disaggregated DRAM and DPM as disaggregated memory unless otherwise specified.

Efficient memory disaggregation for DPM hinges on the high-performance interconnects. Most prior studies on disaggregated memory target RDMA (Remote Direct Memory Access) network interconnects (*e.g.,*, Infiniband) [3, 6, 9, 116, 128, 137, 177, 190, 198, 217, 222, 225, 228]. RDMA-based disaggregated memory can be efficiently implemented in a CPU-less manner using one-sided RDMA operations, enabling direct remote memory accesses at sub-microsecond latency without involving remote CPUs. Another emerging technology for memory disaggregation is CXL (Compute eXpress Link) [182], a cache-coherent interconnect based on the PCIe interface. CXL enables load-store accessible coherent disaggregated memory pooling and sharing at hundreds of nanoseconds access latency.

1.2 Challenges in building key-value stores for DPM

An ideal KVS for DPM would have a number of properties: high common-case performance, scalability, crash recoverability, and quick reconfiguration. However, building such KVS for DPM is challenging. First, KNs incur expensive overheads to access data and metadata at DPM through the interconnects. Despite these overheads, the KVS must provide high performance. Second, the KVS must provide

scalable performance without bottlenecks due to load imbalance at KNs or from non-uniform workload patterns. Finally, to benefit from independent scaling and separate failure domains of KNs and PM, the KVS must be elastic and crash-recoverable, supporting lightweight reconfiguration of resources.

To address these challenges, in the first part of this dissertation, we design new indexing structures for a high-performance, scalable, crash-recoverable data storage at PM. In the second part, we present new partitioning and caching techniques to achieve an elastic, scalable, high-performance DPM KVS. Finally, in light of the emerging disaggregation technology, CXL, we reconsider existing techniques for RDMA-based memory disaggregation and propose novel cache-conscious indexing, caching, and partitioning approaches for CXL disaggregated memory.

1.3 Indexing data structures for PM

Indexes are key to achieving efficient and reliable data storage at DPM, thus are a crucial component of DPM KVSs. Researchers have designed several new PM indexes to better utilize PM [36, 82, 111, 149, 161, 211, 227]. However, designing these indexes from scratch is challenging; the indexes must provide high performance and concurrency while ensuring that the index recovers correctly in the event of a power loss or a system crash. This complexity leads to subtle bugs [62, 63, 112, 151].

While research on building concurrent, crash-consistent PM indexes has been gathering traction recently, there have been decades of research on building concurrent DRAM indexes. Modern DRAM indexes are carefully designed keeping in mind cache efficiency, pre-fetching, concurrency, and parallelism. Concurrent DRAM indexes are widely used in industry and academia; for example, latch-free BwTree in the Hekaton OLTP engine [53], Adaptive Radix Tree (ART) in the HyPer database [100], the Timeline Index in SAP HANA [95], and Masstree in the Silo database [192]. In this work, we seek to leverage the research on concurrent DRAM indexes to build crash-consistent PM indexes.

This dissertation presents **Recipe** [112], a principled approach for converting concurrent DRAM indexes into crash-consistent indexes for PM. The main insight behind RECIPE is that *isolation* provided by a certain class of concurrent DRAM indexes can be translated with small changes to *crash-consistency* when the same index is used in PM. We present a set of conditions that enable the identification of this class of DRAM indexes, and the actions to be taken to convert each index to be persistent. Based on these conditions and conversion actions, we modify five different DRAM indexes based on a hash table (CLHT [48]), a trie (HOT [19]), a B+ tree (BwTree [123]), a radix tree (ART [120]), and a hybrid index (Masstree [141]) to their crash consistent PM counterparts. The effort involved in this conversion is minimal, requiring 30–200 lines of code (1–9% of the codebase).

Building a PM index using the RECIPE approach offers several benefits. First, it drastically lowers the complexity of building a PM index; the developer simply chooses an appropriate DRAM index and modifies it as indicated by our approach. The developer does not have to worry about crash recovery, even in the presence of concurrent writes. Second, if the developer converts a DRAM index that has high performance and scalability, the converted PM index also offers good performance without any further optimization.

To test the performance of our converted PM indexes, we use the YCSB benchmark [45] to perform multi-threaded insertions, point queries, and range queries on Intel DC Persistent Memory. We compare the converted PM indexes against state-of-the-art manually-designed PM indexes. We find that our converted PM indexes outperform the state-of-the-art by up-to 5.2× in multi-threaded YCSB workloads. The main performance gain for DINOMO-converted indexes comes from the fact that the DRAM indexes we convert are already optimized for concurrency and cache efficiency; the high read latency of PM makes cache efficiency even more important.

1.4 Partitioning and caching for RDMA-based DPM

Based on the RECIPE indexes, we extend our focus to system-wide design issues for building a persistent KVS for DPM. Prior DPM KVSs [137, 190] apply traditional distributed system architectures for DPM that are derived from the monolithic server model. This approach makes design trade-offs sacrificing one of high common-case performance, scalability, or lightweight reconfiguration for the other two. For example, shared-nothing architectures can achieve high-performance and scalability by enabling high cache locality at KNs, but compromise elastic reconfiguration due to expensive data reorganization. Similarly, shared-everything architectures can support high elasticity without expensive data reorganization due to partitioning. However, they suffer from low performance and scalability as a result of poor cache locality and consistency overheads due to sharing in the common case [168].

To address these limitations, this dissertation presents **Dinomo** [116], the first DPM KVS that simultaneously achieves high common-case performance, scalability, and lightweight online reconfiguration. DINOMO also provides linearizable reads and writes. DINOMO uses a novel combination of techniques such as Disaggregated Adaptive Caching (DAC) and Ownership Partitioning (OP), selective ownership replication, and lock-free and log-free PM indexing to achieve these goals.

Similar to other disaggregated systems, DINOMO reduces network round trips (RTs) to DPM by caching data and metadata in the local DRAM of each KN. Data is cached by storing the key-value pair, and metadata is cached by storing a pointer to the data on DPM (termed *shortcuts* [190]). To determine how best to divide the cache space between data and metadata, DINOMO uses DAC, a novel adaptive caching policy that actively maintains the right balance between caching values and shortcuts based on the workload patterns and available memory at KNs. DAC allows DINOMO to make efficient use of the DRAM at KNs without making any assumptions about the workload.

While caching at the KNs can reduce network RTs, it can incur significant

consistency overheads when KNs can share the same data. To handle this concern, Dinomo partitions the *ownership* of data across KNs, while data and metadata are shared via DPM. This provides three benefits. First, it allows KNs to cache the data they own, thus providing high cache locality without consistency overheads. Second, by sharing the data and metadata, OP supports changing the number of KNs or rebalancing their load by repartitioning only the ownership of data among KNs, without expensive data reorganization at DPM. Finally, since each key is only accessed by one KN at any given point, combined with our principled reconfiguration protocol, Dinomo achieves linearizable reads and writes. With OP, Dinomo achieves high performance/scalability from locality-preserving KN-side caching without consistency overheads and high elasticity from lightweight reconfiguration.

We implement Dinomo in 10K lines of C++ code. We compare the end-to-end performance and scalability of Dinomo with Clover [190], a state-of-the-art DPM KVS. Our experiments show that Dinomo achieves both better common-case performance and scalability than Clover. Dinomo's throughput scales to 16 KNs, while Clover's throughput does not scale beyond 4 KNs. With 16 KNs, Dinomo outperforms Clover by at least 3.8× on all workloads we evaluate. We also show that Dinomo elastically scales out KNs, balances the load across KNs, and handles KN failures quickly.

1.5 Indexing, caching, and partitioning for CDM

The new CXL interconnect enables load-store accessible and cache-coherent disaggregated memory with low latency at hundreds of nanoseconds. These unique characteristics introduce new challenges in designing KVSs for CXL disaggregated memory. As CXL disaggregated memory is now integrated into the cache-coherent hierarchy, cache-conscious designs regain importance for CDM KVSs to attain high performance, scalability, and partial failure tolerance. Moreover, software overheads associated with the KNs' local processing can substantially impact system performance,

considering that the cost of accessing CXL disaggregated memory ($170 - 300ns$) approaches that of local memory ($80 - 140ns$) at KNs and can even be masked by CPU caches upon hits.

Our findings confirm that existing design approaches for RDMA disaggregated memory may be sub-optimal or not reliably applicable to CXL disaggregated memory. Indexes designed for RDMA disaggregated memory can exhibit lower performance and scalability compared to their cache-conscious counterparts due to their CPU-cache oblivious, network I/O-oriented designs. Moreover, we demonstrate that conventional hierarchical caching at KNs does not consistently outperform non-caching counterparts for CXL disaggregated memory. This is attributed to software overheads induced by KN cache misses, which can introduce non-negligible overheads to the total runtime due to the CXL's low latency. Furthermore, the additional algorithmic complexity in caching mechanisms (*e.g.,* eviction policy) can deteriorate overall performance and scalability by causing more CPU cache misses than a standalone index design. Lastly, the state-of-the-art partitioning technique for RDMA disaggregated memory can still generate significant cache coherence traffic in CXL settings for common write-heavy workloads due to its shared metadata structure.

To overcome these limitations, this dissertation presents **Shift**, a novel cache-conscious KVS designs for CXL disaggregated memory. Shift introduces new indexing, caching, and partitioning approaches for CDM. Shift reuses existing PM indexes for CXL disaggregated memory that are already designed to be CPU-cache efficient, concurrent, and crash consistent. For the correct PM-index deployment for CXL, we retrofit lock intention log to the PM indexes so that permanent locks caused the partial failures at KNs can be safely and elastically released to prevent deadlock. Moreover, Shift presents NHP (Non-Hierarchical Processing), a new request processing scheme across two hierarchies of KN and CXL disaggregated memory. NHP dynamically balances processing ratio between KN-side cache and direct accesses to CDM using in-situ A/B testing by monitoring the processing latency from

each layer. Finally, SHIFT propose reusing ownership partitioning for CXL, but optimize its shared metadata to minimize potential cache-coherence overheads by using a NUMA-aware lock.

We implement SHIFT in 10K lines of C++ code and evaluate each technique on our emulated CXL disaggregated memory using the remote NUMA-node memory. Through various performance studies, we show our PM indexes ported to CXL settings outperform Sherman, the state-of-the-art RDMA-based B+tree, up-to 6× at scale and show the performance tradeoff of the lock intention log on the PM indexes. Furthermore, we show NHP can match the best performance among static policies (hierarchical KN cache and CDM direct accesses) in various scenarios.

1.6 Contributions

We describe the main contributions of this dissertation.

- A principled approach to convert concurrent DRAM indexes into crash consistent PM indexes, RECIPE that translates non-blocking isolation to crash consistency for PM (§3)
- The novel combination of Disaggregated Adaptive Caching (DAC) and Ownership Partitioning (OP) that enables DINOMO to simultaneously achieve high performance, scalability, and elasticity for DPM (§4)
- The design and implementation of the SHIFT approaches that suggest paradigm shift from I/O-oriented designs to cache-conscious designs for CXL disaggregated memory (§5)
- Open source implementations of RECIPE and DINOMO that contain all the information needed to reproduce their main results

1.7 Overview

The rest of this dissertation is organized as follows. Chapter 2 reviews background technologies and relevant concepts for this dissertation; Persistent Memory, Disaggregated Persistent Memory, RDMA, CXL, and key-value stores, indexing data structures for these technologies. Chapter 3 discusses the challenges in building correct, concurrent, crash-consistent PM indexes, and presents RECIPE, a principled approach to building such PM indexes with low effort by reusing existing concurrent DRAM indexes. Chapter 4 discusses the architectural limitations of existing DPM KVSs and presents DINOMO that achieves an elastic, high-performance, scalable DPM KVS using novel partitioning and caching techniques. Chapter 5 comprehensively discusses the design techniques originally proposed for RDMA disaggregated memory in the context of CXL along with their limitations, and presents SHIFT, a novel cache-conscious KVS designs for CXL disaggregated memory. Chapter 7 presents the summary of this dissertation and concluding remarks.

Chapter 2: Background

This chapter provides background on key-value stores (KVSs), persistent memory (PM) and various disaggregation technologies that enable disaggregated PM. We then discuss PM indexing data structures, partitioning and caching techniques used in prior KVSs for disaggregated PM (DPM).

2.1 Key-value stores

Key-value stores, often referred to as key-value databases, are a type of non-relational database that stores data in a simple key-value format. This data model associates each data item with a unique key, allowing for efficient retrieval and manipulation of data. KVSs are highly scalable and adaptable to various deployment environments, making them a popular choice for modern applications.

KVSs find their application in a wide range of scenarios, including web object caching [155], backend storage engines for DBMS [83], state storage for machine learning [126], web pages/analytics, best seller lists, shopping carts, customer preferences, session management, and product catalog [51].

KVSs typically provide a simple API interface for storing, retrieving, and deleting data. Common operations include:

put(key, value): Inserts or updates the value associated with the specified key.

get(key): Retrieves the value associated with the specified key.

delete(key): Removes the key-value pair associated with the specified key.

KVSs have been designed at different scales, depending on their target applications and deployments. Many key-value stores are designed for a large-scale distributed architecture to achieve high aggregate throughput and scalability while

utilizing data parallelism [30, 51, 205]; others are optimized for high-performance single-machine embedded settings [124, 176]. In recent years, these distinct target scales have been converging in cloud infrastructures [21, 35, 204].

As applications increasingly converge onto cloud infrastructures, cloud-based KVSs face the challenge of handling vast amounts of data with dynamic working sets/sizes and non-uniform workloads with varying skew [156, 171, 204, 212]. This necessitates efficient and reliable data storage and I/O mechanisms to effectively manage the ever-growing data volumes. Furthermore, elastic resource deployment becomes crucial for meeting Service Level Objectives (SLOs) amidst significant workload fluctuations [28, 219]. Finally, KVSs must maintain high resource utilization to ensure cost-efficient service deployment.

To address these challenges, KVSs need to employ sophisticated data management strategies that can adapt to dynamic workloads and varying data sizes. Techniques such as indexing, caching, and partitioning can help improve I/O performance and distribute data efficiently across storage nodes. Additionally, KVSs should leverage elastic resource provisioning mechanisms (*e.g.,*, autoscaling) to automatically scale resources up or down based on workload demands while maximizing their utilization. This ensures that sufficient resources are available to meet SLOs while minimizing unnecessary costs.

Conventional slow storage devices and monolithic server models, however, still make it challenging to achieve these goals. Traditional block-based storage devices, such as HDDs and SSDs, offer persistence and large capacity but suffer from high latency and low bandwidth, hindering the performance of KVSs [14, 167, 172]. Additionally, monolithic server architectures, where resources are tightly coupled within a single machine, restrict elastic resource deployment and utilization [181]. Despite optimization efforts, data-store services offered by many cloud providers are inelastic [4, 204], and cloud data centers often exhibit low resource utilization [76, 134, 189], with memory utilization as low as 60% [38, 189, 193].

2.2 Persistent Memory

PM is an emerging class of memory technology such as phase-change memory [203], spin-transfer torque MRAM [10], Intel Optane DC PM [86], and Samsung Memory-Semantic CXL SSD [56]. PM has unique characteristics different from conventional memory and storage devices [89, 210]. PM is byte-addressable like DRAM, that can be directly connected to the memory bus and accessed via load/store instructions. It is non-volatile and high-capacity like conventional storage devices (*e.g.,* HDD, SSD), but provides high performance close to DRAM. For example, Intel's Optane PM product has read latency 3.7× that of DRAM, while read and write bandwidth are $1/3^{rd} - 1/6^{th}$ that of DRAM [89], and its capacity supports up to 512GiB per NVDIMM [86]. The low latency and durability of PM make it an attractive medium for building KVSs, enabling efficient data storage and I/O in nature.

PM, however, exhibits distinct correctness semantics (atomicity and ordering) compared to conventional block-based storage devices. Writes to the PM are issued in 8-byte failure-atomic units, which are first written to the volatile CPU cache. These cache lines can be written back to the Persistent Memory Controller in an arbitrary order. Intel x86 architecture provides fence instructions (*e.g.,* `mfence`, `sfence`) to prevent such memory reordering [84]; if a store instruction is followed by a `mfence`, then it is guaranteed to be visible before any other stores that follow the `mfence`. Additionally, to explicitly flush a cache line to the persistent controller, x86 architecture provides `clflush`, `clwb` and `clflushopt` instructions. A temporal store followed by `clflush` and `mfence` guarantees the cache line written back to the volatile CPU cache to be persisted at PM after the fence instruction [178]. Another feature that can be used for this persistence is non-temporal store instructions (*e.g.,* `movnt`). These stores bypass the CPU cache, thus provides the persistence without requiring the explicit flushes [178]. `mfence` is still required after the non-temporal stores to ensure the stores have reached persistence domains. Our work uses `clwb`, `mfence`, and `non-temporal store` to guarantee the persistence.

	DDR4 DRAM	PM	SSD	HDD
$ per GB	9.77	3.83	0.32	0.05

Table 2.1: **Memory and storage price.** The table shows the market prices of modern memory and storage technologies in October 2020 [8].

2.3 Disaggregated Persistent Memory

As shown in Table 2.1, the per-GB cost of PM is much higher than high-end solid state drives, but less than DRAM. Therefore, it is important to keep PM utilization high for a cost-efficient PM deployment. To improve the PM utilization and cost efficiency, prior work proposes DPM [99, 132, 137, 190, 196, 217].

In Disaggregated PM (DPM) settings, PM is available as a central, reliable pool of memory accessible over a fast interconnect. Separate computing units from DPM (also called as hosts in CXL settings [182]) are used to access the data in DPM. We call the computing units assigned for specifically operating key-value stores as *KVS nodes* (KNs). KNs have limited DRAM with general-purpose processors and use the DRAM for their local processing or caching data in DPM. DPM allows independent scaling of PM and KNs and introduces separate failure domains, where KN failures do not cause DPM failures. Depending on the interconnect technologies, the DPM settings can be classified into RDMA-based or CXL-based disaggregated memory. Note that we collectively call both disaggregated DRAM and DPM as disaggregated memory unless otherwise specified.

2.3.1 RDMA-enabled disaggregated persistent memory

In conventional DPM settings, KNs use network primitives like RDMA to access the PM pool over a fast network interconnect such as InfiniBand [9], PMoF (Persistent Memory over Fabric) [69, 73], or Gen-Z (Generation Z) [66]. Recent RDMA NICs (*e.g.,* ConnectX6 adapters) are capable of delivering 200 Gbps of bandwidth with sub-microsecond latency (1 – 4 µs). The RDMA protocol provides two types of

communication primitives, one-sided and two-sided RDMA operations. With a one-sided operation (e.g., RDMA read, write, and atomic verbs), a KN executes directly on DPM without involving the DPM processor. One-sided operations have lower latency and higher bandwidth than two-sided operations (e.g., RDMA send and receive verbs) [54, 92, 148, 158, 202], but one-sided operations are limited in functionality [3].

Previously proposed RDMA-based DPM can be classified as active or passive. Active DPM has small processing units such as ARM SOCs, ASICs, or FPGAs, with high-bandwidth network ports. In active DPM, DPM compute capacity is used for local processing, including network, application-level, and data store processing [101, 137, 183]. Prior work has proposed data stores for active DPM that leverage this limited computational power [75, 132, 137, 190, 217]. In contrast, passive DPM has no computational abilities at the DPM pool. KNs can use only one-sided RDMA operations to access and modify the data in DPM. Data stores for passive DPM [190] have poor performance and scalability due to the limited functionality of the one-sided network primitives [3], showing that active DPM is a more practical deployment.

2.3.2 CXL disaggregated memory

CXL is a new class of an open, industry-supported, cache-coherent interconnect over PCIe interface [182]. It enables memory capacity/bandwidth expansion and heterogeneous memory for disaggregated computing platforms. CXL consists of three protocols; (1) *CXL.io* for device discovery, configuration, register access, and interrupt, (2) *CXL.cache* for device access to host processor memory, and (3) *CXL.memory* for host processor access to device attached memory. There are three types of CXL devices. Type 1 is a CXL device without host-managed device memory like SmartNIC using CXL.io and CXL.cache. Type 2 is a CXL device with host-managed device memory like GPU or FPGA using all three CXL protocols. Type 3 is a CXL device only with host-managed device memory using CXL.memory. A typical application of type 3 is disaggregated memory expansion. CXL.memory is not restricted in a specific memory type, supporting both DRAM and PM as the attachable CXL memory

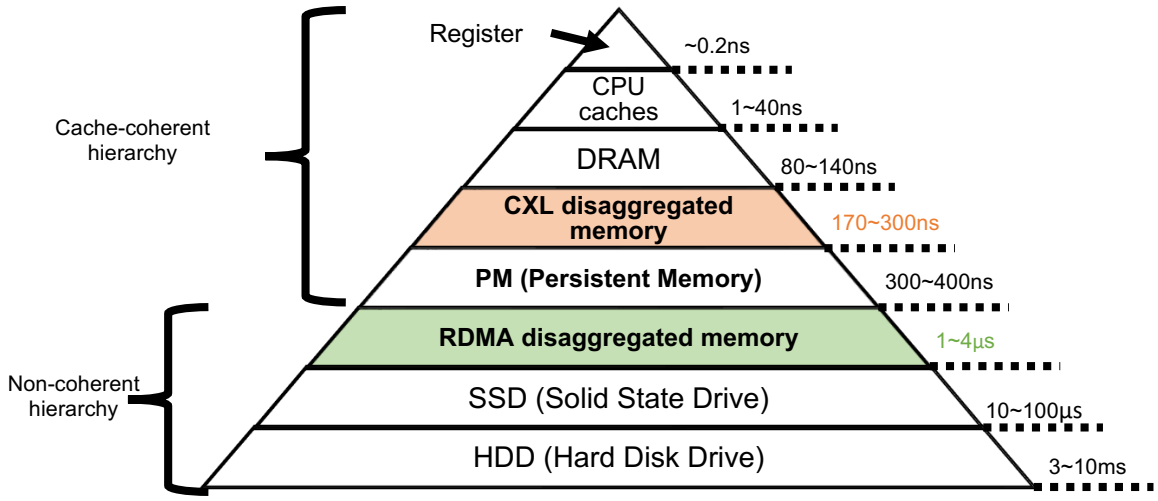


Figure 2.1: **Emerging memory and storage hierarchy.** Diverse memory/storage devices have different characteristics with varying access latency. CXL adds another disaggregated memory tier that is accessible at nanosecond-scale latency and cache-coherent similar to main memory.

device.

CXL-based memory disaggregation has unique characteristics. Similar to existing RDMA-based disaggregated memory, each host processor contains small-sized local memory and can share the disaggregated memory pool. The capacity and bandwidth of the disaggregated memory pool are expendable on demand without increasing the host-processor resources together. However, different from the RDMA-based disaggregation, the CXL disaggregated memory pool is load-store accessible by host processors without data copies and provides nanosecond-scale access latency (Figure 2.1). Furthermore, the coherent accesses to the shared memory pool are guaranteed across hosts by hardware. Due to the support of low-latency cache-coherent load-store accesses, CXL receives attention as a promising alternative for building disaggregated memory.

2.4 Indexing data structures

Indexing plays a crucial role in KVSs, enabling efficient data storage and retrieval. Indexes are data structures that map keys to their corresponding values, allowing for faster lookups. Well-designed indexes can significantly improve the performance of KVSs, especially for large datasets. This section begins by describing DRAM indexes and their interfaces, how indexes achieve concurrency and scalability, and persistent indexes.

2.4.1 DRAM Indexes

DRAM indexes are used to efficiently lookup data items in databases, file systems, and other storage systems. Their interface involves five main operations:

insert(key, value) inserts the pair of `key` and `value` into the index. `value` is usually the location in the storage system where `key` can be found.

update(key, value) update `key` with `value` in the index. Some key-value stores use `insert` for both insertions and updates, while other key-value stores will fail insertions if the key already exists.

lookup(key) returns the `value` associated with `key` in the index.

range_query(key1, key2) returns all key-value pairs where the keys are within the specified range. Range queries are sometimes implemented using an iterator: a cursor that can be incremented to the next key in the sequence.

delete(key) removes the specified key from the index.

Structural Modification Operations (SMOs). SMOs are operations internal to the data structure, that are required either to ensure that the invariants of the data structure holds, or to improve performance. For instance, when the nodes in a B-tree overflow (during insertion) or underflow (during deletion), node splits or merges are required to re-establish the invariants of a B-tree. In other data structures like

hash tables, SMOs like re-hashing are necessary to keep constant average cost per operation.

Performance. DRAM indexes take special care to have high lookup and insertion performance, as these are often performed in the critical path. Lookup and insertion performance depend on the number of processor loads and store required, along with aspects like whether the layout is cache-friendly and prefetcher-friendly.

Correctness. A DRAM index should return the latest inserted `value` for any given `key`. Unless the key is explicitly deleted, an inserted key should never be lost.

2.4.2 Concurrency and Isolation

DRAM indexes use multiple threads to increase throughput on multi-core machines. However, since all threads operate on the same shared index, additional mechanisms are required to ensure correctness. Concurrent DRAM indexes need to provide *isolation*: ensuring that even if multiple writers are modifying the index at the same time, the final index state corresponds to the insertions or updates happening in some sequential order. The index also needs to ensure that reads do not reflect the result of a partial or incomplete insertion or update operation.

Blocking operations. The easiest way to ensure correctness in a concurrent index is to obtain a lock on the index, and only allow threads with lock to read or write. This serializes all operations and decreases throughput to that of a single thread. To increase performance, reader-writer locks are often used [146, 179, 227]; readers can get a shared lock, all writers have to contend on a single lock, and there is mutual exclusion between readers and the writer.

Non-blocking operations. Non-blocking operations [191] are employed to fully exploit the parallelism offered by modern hardware. Non-blocking operations guarantee progress of some or all remaining threads regardless of the suspension, termination, or crash failure of one of the threads [61, 78]. They provide consistency and correctness

by carefully ordering load and store instructions using memory fence (`mfence`) [1, 79], while avoiding the use of mutual exclusion and expensive synchronization primitives.

Non-blocking operations can be categorized into lock-free and wait-free, based on their progress guarantee. Lock-free operations allow multiple threads to simultaneously access a shared object, while guaranteeing that at least one of these operations finish after a finite number of steps [78]. Wait-free operations are a subset of lock-free operations, with the additional condition that every thread finishes the operation in a finite number of steps [78].

Non-blocking operations are built using hardware-atomic primitives such as compare and swap (`CAS`) or test and set. If every update is performed via a single atomic store, correctness is implicitly guaranteed. If updates consist of a sequence of atomic stores, then the readers can either make progress by reasoning about the deterministic order of stores, or can use additional techniques such as version-based retry [33, 59].

While non-blocking operations are known to provide high performance and scalability, high contention to the shared resource reduces performance and could lead to starvation [58]. For example, if a lock-free write is interrupted by the scheduler, it might need to retry the operation after being rescheduled if the shared state has changed. To protect against starvation, many indexes use non-blocking reads and blocking, lock-protected writes [19, 48, 120, 141].

2.4.3 Crash-Consistent PM Indexes

Building PM indexes is attractive for two reasons. First, the larger capacity of PM at close-to-DRAM latencies allows using larger indexes than possible with just DRAM. Second, DRAM indexes need to be reconstructed after a crash; for large indexes, reconstruction could take several minutes or hours. In contrast, a PM index is instantly available. This has motivated a number of researchers to design efficient indexes on PM; we count fifteen PM indexes published in top systems and database

conferences since 2015. The PM indexes include variants of B+ trees [11, 36, 39, 82, 102, 161, 200, 207, 211], radix trees [111], and hash tables [149, 180, 206, 226, 227].

Crash Recovery. One of the main differences between a DRAM index and a PM index is that the PM index has to ensure that it can correctly recover in the case of power loss or kernel crash. This requires carefully ordering stores to PM using `mfence` instructions and then flushing the dirty data from volatile caches to persistent media using cache line flush instructions (`clflush`, `clwb`, or `clflushopt`) or bypassing the caches using non-temporal stores [163]. If the write is larger than eight bytes, a crash could lead to a torn write where the data is partially updated; techniques such as logging [77] and copy-on-write [80] are used to provide atomicity.

2.5 Partitioning/sharing and caching

Caching is an essential technique in KVSs, aimed at reducing latency and improving performance. Caching involves storing frequently accessed data in a temporary, high-speed memory or storage area, reducing the need to access slower primary storage. Partitioning and sharing data represent two fundamental approaches to data management in distributed KVSs, each with its own set of advantages and disadvantages.

2.5.1 System architectures for distributed KVSs

Distributed KVSs can be categorized into two architectural approaches: shared nothing (partitioning) and shared everything (sharing). Each architecture offers distinct advantages and disadvantages, influencing the performance, scalability, load balancing, and elasticity of the distributed KVSs.

Shared nothing. In a shared nothing architecture [51, 94], each node in the distributed KVS maintains its own independent memory and storage resources. Data is partitioned and distributed across these nodes, with each node responsible for man-

aging its own portion of the data.

This approach provides several advantages:

- High data Locality: Data locality is improved as data is stored on the same node that processes it, reducing network traffic and improving performance.
- High scalability: Shared nothing architectures are highly scalable, eliminating contention for shared resources and allowing for horizontal scaling by adding more nodes to accommodate increasing data volumes.

However, the shared nothing approach also presents challenges:

- Low elasticity: Elasticity can be complex to manage in shared nothing architectures. Adding or removing nodes requires careful consideration of data redistribution across partitions to maintain balanced performance.
- Load imbalance: Load imbalance can occur in shared nothing architectures if data or workloads are not evenly distributed across partitions. This imbalance can lead to some nodes becoming overloaded while others remain underutilized.

Shared everything. In a shared everything architecture [23, 70, 223], all nodes in the distributed KVS share a common pool of memory and storage resources. Data is accessible to all nodes, and any node can process any request.

This approach offers some benefits:

- High elasticity: Elasticity is relatively straightforward in shared everything architectures. Adding or removing nodes simply expands or contracts the pool of available resources, simplifying resource allocation and adaptation to changing workloads.

- Load Balancing: Shared everything architectures inherently provide load balancing as all nodes have access to the entire dataset and can process any request. This approach distributes the workload evenly across the system, preventing individual nodes from becoming overloaded and ensuring efficient resource utilization.

However, shared everything also poses limitations:

- Low Data Locality: Shared everything architectures can suffer from lower performance due to reduced data locality. As data may not be stored on the same node that processes it, network traffic increases, leading to latency and potential bottlenecks.

- Low Scalability: Shared everything architectures can face scalability limitations due to potential contention for shared resources. As the system grows, the shared resources, such as memory, storage, or processing units, may become a bottleneck, limiting scalability.

2.5.2 Caching for distributed KVSs

The choice between the two architectural approaches has significant implications for caching strategies and their effectiveness. Caching strategies for the shared nothing architecture focus on leveraging data locality and scalability. Each node maintains its own local cache to store frequently accessed data from its partition and updates the cached data locally. This approach improves data locality, reducing network traffic and latency, and simplifies cache consistency between local memory caches. However, load imbalance in local caches can arise due to uneven data access patterns and workload distribution across nodes [157]. This imbalance can lead to performance degradation and inefficient resource utilization.

In the shared everything architecture, a shared distributed cache is used by all nodes, providing a unified view of cached data [187]. Cache nodes in a shared dis-

tributed cache are typically abstracted as a pool of resources, allowing load balancing algorithms to treat them as interchangeable units [65]. This abstraction simplifies load balancing decisions and enables dynamic allocation of requests based on node availability and load levels. However, This approach introduces cache consistency challenges between local memory caches. Updates to the shared cache need to be propagated to all nodes to maintain consistency, requiring cache invalidation or update propagation mechanisms.

2.5.3 Partitioning/sharing and caching strategies for DPM KVSs

Two previous studies propose distributed key-value stores for RDMA-based DPM: Clover [190] and AsymNVM [137]. They are designed with unique system architectures and caching strategies.

Clover. Clover adopts a shared-everything architecture on passive DPM; every KN can access any data. The KNs in Clover cache pointers to the DPM key-value data (termed shortcuts). Clover keeps the metadata index (a hash table) in dedicated KNs (used as metadata servers) along with the most up-to-date shortcuts, while entire data resides in DPM[1]. The KNs fetch actual data from DPM at cache hits by directly using one-sided READ operations via the shortcuts, bypassing metadata traversals; for cache misses, the KNs need to contact the metadata servers to obtain the associated shortcuts.

Caching shortcuts instead of data copies allows for storing more keys in KNs local memory, but it introduces consistency overheads. KNs perform updates out-of-place to avoid concurrent contention over the same data, but it requires KNs to traverse lock-free linked lists storing the multiple versions of values to find the latest value. These consistency overheads limit performance and scalability.

[1]Note that although Clover assumes passive DPM, it uses extra dedicated metadata servers for managing the DPM metadata, highlighting the practicality of active DPM.

AsymNVM. AsymNVM assumes active DPM and supports a hash table, queue, stack, tree, or skiplist as a metadata index. It selectively employs a shared-nothing or shared-everything architecture depending on the type of the metadata index. For the hash table, queue, and stack, AsymNVM deploys a separate DPM node per KN after partitioning DPM data across the DPM nodes individually, while sharing the same DPM node for a tree or skiplist. KNs cache the DPM data copies and batch redo logs for the updates to the DPM data in their local memory. KNs flush these logs to DPM while DPM processors apply the log operations asynchronously to the metadata index.

KNs can keep their local memory copies without consistency overheads owing to the no sharing approach for the hash table, queue, and stack. However, AsymNVM cannot elastically scale KNs independent of DPM, and requires expensive data reorganizations during cluster reconfigurations. With a tree or skiplist, KNs share the same DPM node and hence must synchronize their caches and redo logs with other KNs using distributed locks, introducing performance and scalability bottlenecks.

Chapter 3: Recipe - Converting Concurrent DRAM Indexes to PM Indexes

This chapter presents RECIPE [112], a principled approach for converting a certain class of DRAM indexes to their crash-consistent PM counterparts. We first motivate the needs of a principled approach to build concurrent, crash-consistent PM indexes. Next, we present the RECIPE approach, a set of conditions that enable the identification of this class of DRAM indexes and the actions to be taken to convert each index to be persistent. Then, we provide case studies to show how to convert existing DRAM indexes to PM indexes through RECIPE in practice.

3.1 Motivation

Hand-crafted PM indexes employ non-blocking operations to increase scalability [82, 149]. However, while non-blocking operations offer high performance and scalability, their complexity makes it challenging to develop, test, and debug indexes with non-blocking operations. Persistence makes the problem even harder, since developers have to ensure that crash recovery and concurrency mechanisms interact correctly. We analyze two state-of-the-art PM indexes: the FAST & FAIR B+ tree [82], and the CCEH hash table [149].

FAST & FAIR. FAST & FAIR is a PM B+ tree that provides lock-free reads. The reads detect and tolerate inconsistencies such as duplicated elements in a sorted list. Writers hold a lock for mutual exclusion. The writes detect inconsistencies such as duplicated elements, and try to fix them. However, we found that concurrent writes could lead to loss of a successfully written key.

Consider the following scenario. Two threads try to insert keys to the same internal node concurrently; one thread gets a lock and performs a node split. When the other thread gets the lock, it does not realize the node has been changed, and

inserts the key into the wrong node. The insert is successful, but a reader would never be able to find the inserted value. We confirmed this design-level bug with the FAST & FAIR authors. The solution is to add metadata about the high-key to B+ tree nodes, as done by prior works [33, 123, 141]. Please refer to our bug report for more details [114].

We also found an implementation bug. According to its design, FAST & FAIR recovers correctly from crashes at any point, not losing any inserted keys. However, when we crashed FAST & FAIR consecutively in the middle of split and merge operations on two nodes, keys present in the right node were lost. This is a testament to the complexity of these indexes; a correct design is not always translated properly to a correct implementation.

Finally, we found that incorrect crash recovery can result in poor performance. If FAST & FAIR crashes in the middle of splits, although the recovered structure is correct, it is not efficient. A series of such crashes transforms the B+ tree into a linked list, leading to poor read and write performance.

In summary, our investigation of FAST & FAIR revealed a design-level bug that lost data, an implementation-level bug that lost data, and that crashes can lead to poor performance. The design-level bug resulted from not leveraging prior research on concurrency, where the high-key problem and its solution is well-known.

CCEH. We discovered that the CCEH PM hash table [149] has two bugs: one in its directory doubling code, and one in crash recovery code. Directory doubling is similar to rehashing the hash table. There are three pieces of metadata that CCEH has to atomically update in correct order during directory doubling: the pointer to the directory, the directory width, and the global depth. If a crash happens before the global depth is updated, insertion operations loop infinitely. If a crash happens after the pointer to the directory is swapped, the crash recovery algorithm goes into an infinite loop. The authors of CCEH have acknowledged both bugs.

Summary. We find the ad-hoc design of concurrent, crash-consistent PM indexes makes it hard to reason about behavior during concurrent writes and crashes, leading to bugs. There is a need both for principled design of PM indexes, and testing whether PM indexes correctly recover from crashes.

3.2 The Recipe Approach

We present RECIPE, a principled approach for converting a specific class of DRAM indexes to their crash-consistent PM counterparts. The converted PM index inherits correctness and scalability from the DRAM index. The RECIPE approach guarantees that the converted PM index will recover from crashes correctly. Thus, if the developer uses the RECIPE approach to convert an appropriate DRAM index, the resulting PM index will be correct, concurrent, and crash-consistent.

RECIPE identifies three categories of DRAM indexes to guide this conversion. Each category is accompanied by a condition and conversion action of the form: "*if the DRAM index satisfies these conditions, then convert it to a PM index using these conversion actions*". We first present the intuition behind the RECIPE approach, and then describe each category.

3.2.1 Overall Intuition

We observe that some DRAM indexes use non-blocking reads (such as lock-free reads) to improve performance. These non-blocking reads may observe inconsistent states since writes may be underway at the time of read; the read operations can then *tolerate* such inconsistencies, returning a consistent answer to the user. For example, the read operation may see duplicate records and only return a single record to user [67, 82]. Similarly, write operations may also see an inconsistent state and *fix* the inconsistency; write operations in BwTree perform such fixes [123]. Prior theoretical work has termed this a *helping mechanism*, where an operation started by one thread which fails is later completed by another thread [32].

The RECIPE approach is based on the following insight: if reads can tolerate inconsistencies, and writes can fix them, a separate crash-recovery algorithm is not required. DRAM data structures that have such read and write operations are *inherently crash-consistent*. If such data structures are stored on PM instead of DRAM, they would be crash-consistent with minimal modifications; the developer would only need to ensure that all data dirtied by store operations are persisted to PM in the right order. We refine this observation through three conditions with corresponding conversion actions that help a developer convert a DRAM index into a crash-consistent, concurrent PM index.

3.2.2 Assumptions

RECIPE assumes that the locks used in the index are non-persistent, and that the locks are re-initialized after a crash (to prevent deadlock). RECIPE also assumes that unreachable PM objects will be garbage collected, as a failed update operation may result in an allocated but unreachable object. Finally, RECIPE also assumes that the DRAM index operates correctly in the face of concurrent writes.

3.2.3 Condition #1: Updates via single atomic store

Reads must be non-blocking, while writes may be blocking or non-blocking. The index makes write operations visible to other threads using a single hardware-atomic store.

Conversion Action. Insert cache line flush and memory fence instructions after each `store`.

Fig 3.1 illustrates the scenario covered by Condition #1. The index moves in a single atomic step from its initial state to its final state. A crash at any point leaves the index consistent, so crash recovery is not required.

Converting an index that fits these conditions is straight-forward; each store instruction must be followed by a cache line flush and a memory fence instruction.

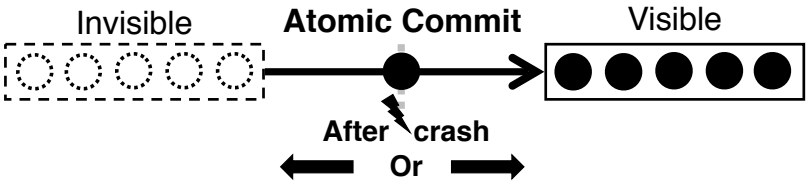


Figure 3.1: **Recipe Condition 1.** When a crash occurs in the middle of an update operation that completes in a single hardware atomic step, there is no recovery required. The state after the crash is either the initial state or the final state.

This ensures that all dirty data is flushed to PM, and that the order in which the writes happen in CPU cache is the same order in which they are persisted to PM. Performance can be increased by allowing stores preceding the final critical store to be reordered [163]. Instead of putting a fence after each store, we would need fences only surrounding the final atomic store.

Examples. We converted two indexes, the Height Optimized Trie (HOT), and the Cache-Line Hash Table (CLHT) based on Condition #1. These indexes employ copy-on-write for updates and failure-atomically make them visible to other threads via atomic pointer swap. Thus, their conversions just require adding cache line flushes and memory fences after each store.

3.2.4 Condition #2: Writers fix inconsistencies

Reads and writes must be both non-blocking. The index performs write operations using a sequence of ordered hardware-atomic stores. If the reads observe an inconsistent state, they detect and *tolerate* the inconsistency without retrying. If writes detect an inconsistency, they have a *helping mechanism* which allows them to fix the inconsistency.

Conversion Action. Insert cache line flush and memory fence instructions after each `store` and specific `load` instructions.

Figure 3.2 illustrates the scenario for Condition #2. A crash leaves Thread 1's write partially completed. Thread 2 is able to detect this; since the write operation

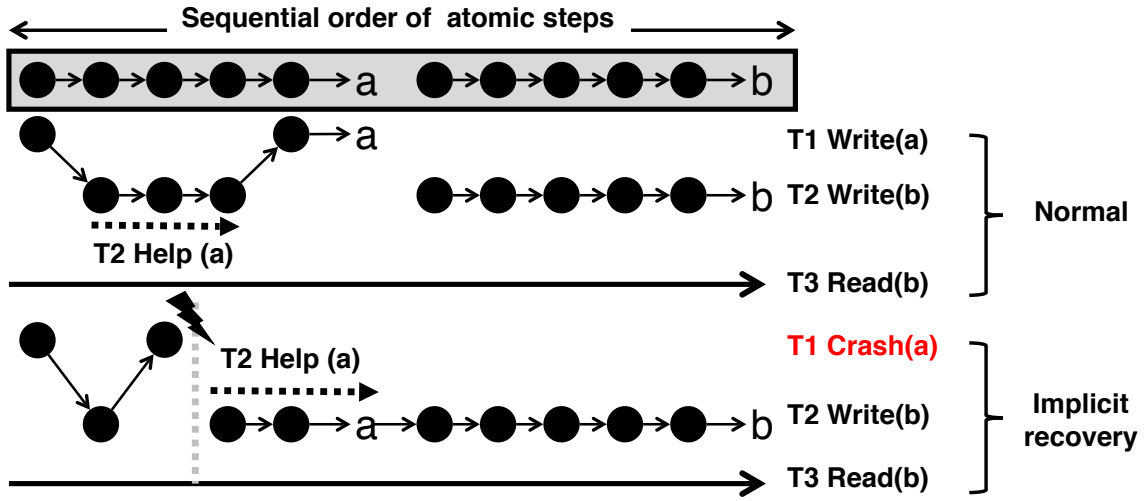


Figure 3.2: **Recipe Condition 2.** A crash occurs in the middle of Thread 1's write operation. Thread 2 detects this, completes Thread 1's write operation using its helper mechanism, and then proceeds with its own write operation.

comprises of a small sequence of deterministic steps, Thread 2 can identify where the crash happened. Thread 2 then proceeds to complete the operation, and then proceed with its own write. This restores the index back to its consistent state. Any read observing these actions is able to tolerate the inconsistency and return a consistent value back to the user.

Note that in general, it is hard after a crash to identify what happened before the crash if extra information is not logged. Indexes meeting Condition #2 are able to do this because write operations in such indexes are comprised of a small number (typically fewer than five) of ordered store operations which mutate the index in a deterministic fashion. Thus, after a crash, the write operation can always deduce what happened before a crash.

Indexes matching Condition #2 do not need any explicit crash-recovery code because *implicit crash recovery* is already part of the read and write operations. The first writer that tries to update the index after a crash and detects the inconsistency is responsible for the recovery of the part of the index the writer deals with. Load instructions used to detect the inconsistency (or to detect which step of recovery has

been already performed) must be followed by a cache line flush to ensure the thread is not acting on stale or un-persisted information [88, 200]. As with Condition #1, stores must be followed by a cache line flush and a memory fence instruction. Please refer our tech report [113] for more details.

Example. The BwTree has non-blocking read and write operations. It uses a sequence of ordered atomic stores to perform Structural Modification Operations (SMO) like node splits and merges. BwTree write operations have helper mechanisms which complete and commit any intermediate SMO state encountered, before proceeding with their own write. Thus, BwTree fits into Condition #2, and we converted it to its persistent version simply by adding cache line flushes and memory fences.

3.2.5 Condition #3: Writers don't fix inconsistencies

Reads must be non-blocking, while writes must be blocking. Write operations involve a sequence of the ordered atomic steps similar to Condition #2, but they are protected by write exclusion (locks). Reads can detect and tolerate inconsistencies. Writes can detect inconsistencies; however, they lack the helper mechanisms needed to fix the inconsistency.

Conversion Action. Add mechanism to allow writes to detect permanent inconsistencies. Add helper mechanism to allow writes to fix inconsistencies. Insert cache line flush and memory fence instructions after each `store`.

Indexes conforming to Condition #3 are the hardest to convert, as they require multiple steps. The root of the problem is that Condition #3 indexes do not have helper mechanisms in their write operations. Therefore while reads and writes tolerate inconsistencies, the permanent inconsistency will never get fixed.

First, the write operation must distinguish between a transient inconsistency due to another on-going write or a permanent inconsistency due to a crash. It differentiates these scenarios by trying to acquire the write lock; if it is successful, there

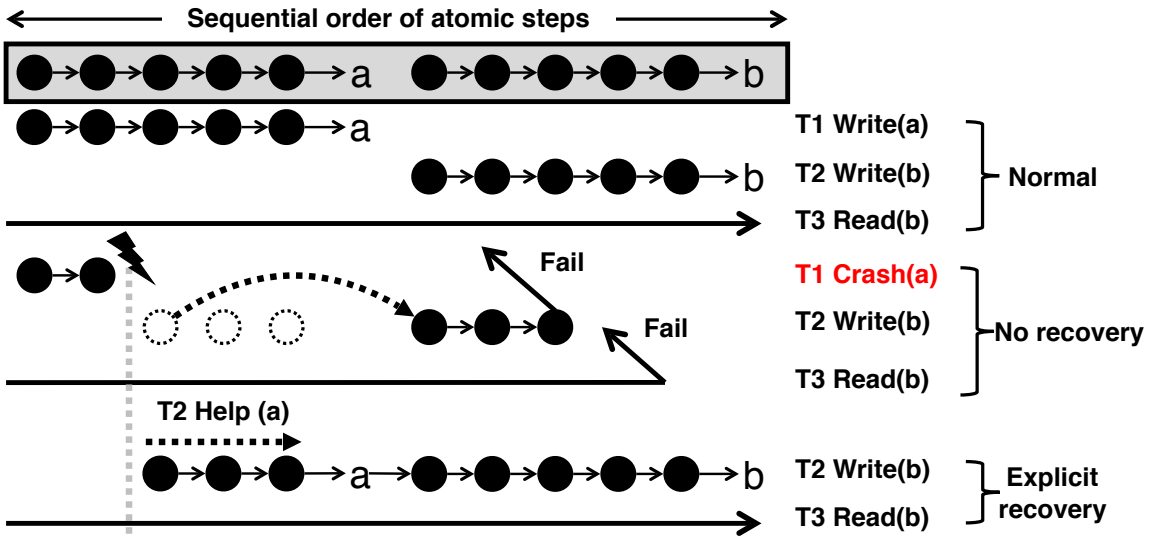


Figure 3.3: **Recipe Condition 3.** Condition #3 indexes lack the helper mechanism which allows them to resume an interrupted write operation. We explicitly add the helper mechanism which identifies that Thread 1's write operation was interrupted, and finishes the write operation before proceeding with Thread 2's write operation.

are no other writes happening concurrently, so an inconsistent state must be due to a crash.

Second, a helper mechanism must be added to finish an interrupted write operation. We find that helper mechanism can be built using code from the write path. The helper mechanism must first identify what was happening at the point of the crash (similar to Condition #2); it must then complete the interrupted write operation. Figure 3.3 illustrates that explicit recovery code must be added into the writer for Condition #3 indexes.

Adding the helper mechanism to write operation is correct since it re-uses code from the write path; reads can already tolerate the inconsistencies due to on-going writes. Adding the helper mechanism converts a Condition #3 index into a Condition #2 index. At this point, only adding cache line flushes and memory fences after each store are required to produce a crash-consistent, concurrent PM index.

Example. The Adaptive Radix Tree (ART) falls into the category of Condition

#3. The writes in ART do not have the helper mechanism, so they just tolerate inconsistencies, when encountering an intermediate state of Structural Modification Operations (SMO). Fortunately, ART's SMO consist of exactly two ordered steps; after a crash, the helper mechanism only needs to identify if step one or two has occurred. We modified ART to introduce permanent inconsistency detection and helper mechanisms, along with adding cache line flushes and memory fences.

3.3 Testing Crash Recovery of PM Indexes

We introduce a novel method to test whether PM indexes recover correctly after crashes. Testing crash recovery involves testing two things: whether the PM index recovers to a consistent state, and whether the PM index loses any data successfully persisted before the crash. Consistency for a PM index involves reads and range queries of all previously inserted keys returning the correct values, and further writes completing successfully.

The main challenge in testing crash recovery is deciding where to crash in each workload. A crash could happen after each 8-byte atomic store in a workload; this makes the total space of crashes in a reasonable workload prohibitively large. We address this challenge by observing that most operations in PM indexes are comprised of a small number of atomic stores; it is enough to simulate a crash after each atomic store. For each operation in a PM index, we simulate a crash after all its atomic stores. This is feasible since PM indexes have few operations, and each operation has few atomic steps. Structure modifications operations and insertions have less than five atomic steps in all the PM indexes we tested. Thus, crashing only after atomic stores drastically reduces the search space. While there are existing tools like PM-Inspector [85], `pmreorder` [166], and `yat` [110] to simulate crashes, these tools still pick crash points in a random or exhaustive manner; our targeted crashing strategy is powerful, revealing bugs with limited testing.

Testing consistency. We test for consistency using three steps. First, we run a

DRAM	Synchronization		Conditions	
Index	Reader	Writer	Non-SMO	SMO
CLHT	Non-blocking	Blocking	#1	#1
HOT	Non-blocking	Blocking	#1	#1
BwTree	Non-blocking	Non-blocking	#1	#2
ART	Non-blocking	Blocking	#1	#3
Masstree	Non-blocking	Blocking	#1	#3

Table 3.1: **Categorizing convertion actions.** The table lists the converted DRAM indexes with their category and synchronization properties.

write-heavy workload, and probabilistically simulate a crash after an atomic store in either insertion or a structure modification operation like a node split. A crash is simulated by returning from the operation without any clean-up activities, leaving a partially modified state. Next, we explicitly call the recovery function if the PM index has one. We perform a number of read and write operations using multiple threads, keeping track of all successfully inserted keys. Finally, we read back all successfully inserted keys and check that they have the right values. Note that this approach does not require actual PM; we are able to emulate crashes using DRAM.

Testing durability. Testing durability involves checking that all cache lines which were dirtied during the workload are flushed to PM. This ensures that data written to the PM index is not lost if there is a crash. To test durability, we use the Pin [135] tool to trace all allocations made using `malloc`, `posix_memalign`, and `new`. We then trace all store instructions to these allocated regions, and verify that all dirtied cache lines are safely flushed to PM. We perform this testing using two phases: a load phase and a test phase. We first load the index with enough keys such that future insertions will trigger node splits and other structure modification operations. In the test phase, we perform the insertion while tracing allocation, stores, and cache line flushes. For each insertion, we verify that all dirtied cache lines were persisted.

3.4 Case Studies

We describe how we modified five concurrent DRAM indexes to their PM counterparts. For each index, we discuss and modify the main write operations of the indexes in accordance with the proposed conversion actions. The operations we modify are classified into Structural Modification Operations (SMOs) and Non-SMOs (Inserts and Deletes). Non-SMOs affect a single node (in tree based indexes) or a single bucket (in hash tables), whereas SMOs require changes to multiple nodes or buckets. Table 3.1 lists the converted indexes along with their categories and properties.

Lock initialization. Some of the converted indexes use locks for write exclusion. These locks are embedded into the node or bucket structure and are persisted along with the node. However, locks are required only to provide concurrency; persisting them can result in deadlocks if a system crash occurs. We re-initialize locks on startup for all indexes converted using RECIPE. We statically allocate a lock table that holds pointers to each node's lock. This lock table is initialized when the PM index is restarted after crash.

Crash detection. When a converted PM index detects an inconsistency during path traversal, it tries to acquire the lock for the node using `try lock`. If it fails to acquire the lock, either the inconsistency is transient due to a concurrent write, or another write operation is in the process of fixing the inconsistency. If the write operation acquires the lock, it fixes the inconsistency using the helper mechanism.

3.4.1 Trie: Height Optimized Trie (HOT)

The Height Optimized Trie (HOT) is a lookup and space-optimized variant of a trie, where the children of each node in the search tree share a prefix of the key. HOT achieves cache efficiency, dynamically varying the number of prefix bits mapped by a node to maintain consistent high fanout. The layout is designed for compactness and fast lookup using SIMD instructions.

Non-SMOs. HOT uses copy-on-write and commits an insert or delete operation by atomically swapping the single parent pointer per operation. It uses non-blocking read and exclusive write to prevent the updates from getting lost due to competing pointer swap operations.

SMOs. SMOs in HOT occur when prefix bits are mismatched. If SMOs are required during insertion and deletion, HOT first identifies the set of nodes to be modified, locks them bottom up to avoid deadlock, performs the update using copy-on-write and then unlocks them top down.

Conversion to PM. HOT abides by Condition #1 because every update to the index is installed through an atomic pointer swap. Therefore, as long as the `store` instructions are correctly ordered and flushed, crashes will not result in inconsistencies. Conversion to P-HOT required adding 38 LOC (<2% of the 2K LOC in HOT core).

3.4.2 Hash Table: Cache-Line Hash Table (CLHT)

CLHT is a cache-friendly hash table that restricts each bucket to be of the size of a cache line (64 bytes). At most three key-value pairs, whose keys and values are 8 byte each, fits into one bucket. The design aims at addressing the cache-coherence problem by ensuring that each update to the hash table requires one cache line access in the common case. To ensure that a non-blocking reader finds the correct value, CLHT uses atomic snapshots of key-value pairs [48, 49]

Non-SMOs. CLHT installs any update to the hashtable by locking the appropriate bucket, performing the update in-place and then unlocking it. CLHT installs the insert and delete operation using a single atomic commit point, ordered by memory fences: writing the correct value first prior to updating 8 byte key (for insertion) and writing 0 to the key (for deletion).

SMOs. If the inserts extend the number of buckets per hash beyond a threshold,

CLHT performs re-hashing using copy-on-write. The old hash table is first locked for write. The entries in each bucket are then copied over to the new hash table, and finally, the old hash table is atomically swapped with the new one.

Conversion to PM. CLHT abides by Condition #1 because the inserts, deletes, and re-hashing are effected via a single atomic store. Similar to HOT, we insert cache line flushes and memory fences after appropriate `store` instructions to build P-CLHT. Common-case non-SMOs (inserts and deletes), except for re-hashing, require only one cache line flush per update. Conversion involved 30 LOC (CLHT lock-based implementation is 2.8K LOC).

3.4.3 B+ TREE: BwTree

BwTree is a variant of B+ tree that provides non-blocking reads and writes. It increases concurrency by prepending delta records (describing the update) to nodes. It uses a mapping table that enables atomically installing delta updates using a single Compare-And-Swap (`CAS`) operation. Subsequent reads or writes to this node replay these delta records to obtain the current state of the node.

Non-SMOs. Insert and delete operations prepend the delta record to the appropriate node, and update the mapping table using `CAS`. If a `CAS` to the mapping table fails because of another concurrent update, the thread simply aborts its operation and restarts from the root.

SMOs. When the base node in BwTree overflows (or underflows), a node split (or merge) is necessary. BwTree uses a helper mechanism [123] to co-operatively perform concurrent updates in the presence of structural modifications due to node splits and merges. Any subsequent writer thread that observes an ongoing split or merge operation first tries to complete it, before going forward with its own operation.

Splits and merges first post a special delta record to the node to indicate that a modification is in progress. It then uses the two-step atomic split mechanism of

B-link trees [118] to create a new sibling node in the first step and later update the split key in the parent node. For node merges, the left sibling of the node to be merged is updated with a physical pointer to this node and then the merge key in the parent is removed.

Conversion to PM. BwTree's non-SMOs are completed by preprending new delta node with a single `CAS`, so they fit into Condition #1. We perform a cache line flush if the `CAS` succeeds. BwTree's node split and merge mechanisms expose intermediate states to other readers and writers. While readers never restart in the original design of the BwTree, the open-source implementation of BwTree allows reads to restart if a node merge is in progress [201]. We address this issue by modifying the reader to avoid retry using the inconsistency detection and fix algorithm already present in the write path of BwTree.

Using their helper mechanism, the writers in BwTree detect and fix any partially completed operation. As a result, SMOs of BwTree (after modifications to the read operation) fits into Condition #2. We build P-BwTree by adding cache line flushes and memory fences after every `store` and `load` operation to the nodes and mapping table. Building P-BwTree involves modifying 85 LOC, as compared to 5.2K LOC in the core BwTree index.

3.4.4 Radix Tree: Adaptive Radix Tree (ART)

ART is a radix tree variant that reduces space consumption by adaptively varying node sizes based on the valid key entries. 8-bit prefix (one byte) is indexed by each node. The 8-byte header of each node in ART compresses some part of common prefix and the length of it. The level field in each node represents the full length of common prefix shared at this node and is never modified after its creation. As in HOT, synchronization is provided using non-blocking read and exclusive write [119, 120].

Non-SMOs. For an insertion, a new key-value pair is appended into the end of the entries in a node and is atomically made visible by increasing counter value. Deletion

is completed via a single atomic store, simply invalidating a key by setting the value entry to be NULL. If the node overflows (or underflows), the node is copied to a new larger (or smaller) node and then the parent pointer is atomically swapped.

SMOs. If two keys share the same prefix, ART compresses the native radix tree structure by simply storing the common prefix in a single node (instead of allocating a node per character in the key). As key distribution varies, the compressed prefix could be expanded or compressed, resulting in split or merge of existing nodes. Unlike non-SMOs, these structural changes are installed in multiple atomic steps. If the insertion of a key requires a path compression split, a new node pointing to the key is first installed, and then the header is updated to contain the correct prefix.

Conversion to PM. Since non-SMOs are always committed atomically, they abide by Condition #1. However, the path compression mechanism in ART exposes intermediate states which reads can tolerate. A read counts the depth of the native decompressed radix tree while traversing tree, and compares level field with the sum of the depth and the prefix length stored in a node; if there is a mismatch, the read simply ignores a part of the prefix at this node to access the correct key. To ensure correctness, reads verify if the retrieved key is same as the search key before returning. Writes similarly detect inconsistencies, but do not fix them.

To build P-ART, we modify the write path to include crash detection and recovery. When the node traversal in the write path detects an inconsistency, it first checks for a crash using a `try lock`. If it successfully acquires the lock, the write calculates and persists the correct prefix. Implementing these changes, along with insertion of cache line flushes and memory fences required adding 52 LOC to the 1.5K LOC of ART.

3.4.5 Hybrid Index: Masstree

Masstree is a cache-efficient, highly concurrent trie-like concatenation of B+ tree nodes [141]. Masstree provides synchronization using write exclusion and lock-

free readers retry when inconsistencies are detected by using version numbers.

Non-SMOs. Similar to ART, the non-SMOs of Masstree start with non-blocking tree traversal to return correct leaf node. Inserts to the leaf nodes in Masstree are performed by appending a new key-value pair to the node with unsorted order and atomically switching to an updated copy of the 8-byte permutation table, specifying the sorted orders of keys and empty entries. For deletes, it is sufficient to atomically update the permutation table to invalidate the entry.

SMOs. The internal nodes in Masstree maintain keys in sorted order using a non-atomic key-shifting algorithm which exposes inconsistent data to readers [36]. Reads therefore retry until the ongoing operation completes. Node splits and merges lock the corresponding nodes and update version counters upon completion. Meanwhile, all concurrent reads and writes to these nodes would simply retry from the root.

Conversion to PM. The non-SMOs of Masstree abide by Condition #1, since insertions and deletions are atomically reflected by updating a permutation table. However, Masstree SMOs do not directly fit into our conditions as the readers do not tolerate inconsistency without restarts. While the structure of leaf nodes allows a 2-step atomic split mechanism, the internal nodes do not. Therefore, we modify the internal nodes to resemble the leaf nodes, modifying the data structure to resemble the B-link Tree. This modification allows a 2-step atomic split mechanism across all levels. For example, if the insertion requires node split, half of the entries in split node are copied into the new sibling node, and then the sibling pointer of split node is atomically installed to the new sibling. Finally, the entries copied into new sibling node are atomically invalidated from split node by updating 8-byte permutation table. Furthermore, this eliminates restarts at the read path. All the intermediate states exposed by SMOs are tolerated by moving towards next sibling node, utilizing the B-link Tree's sibling link and high key [33]. Reads therefore always return consistent data and writes can reach the correct leaf node without retry.

Workload	Description	Application pattern
Load A	100% writes	Bulk database insert
A	Read/Write, 50/50	A session store
B	Read/Write, 95/5	Photo tagging
C	100% reads	User profile cache
E	Scan/Write, 95/5	Threaded conversations

Table 3.2: **YCSB workload patterns.** The table describes different workload patterns from the YCSB test suite.

With this change, SMOs of Masstree fits into Condition #3, where reads return consistent values, but writers have no mechanism to fix inconsistent states. We implement write path recovery by simply replaying the node split algorithm whenever a crash is detected using a `try lock`. If the intermediate state observed was due to a node split, then this action would complete the split operation. If the observed crash state was due to a node merge, replaying the split operation will undo the merge, bringing the index back to a consistent state.

3.5 Evaluation

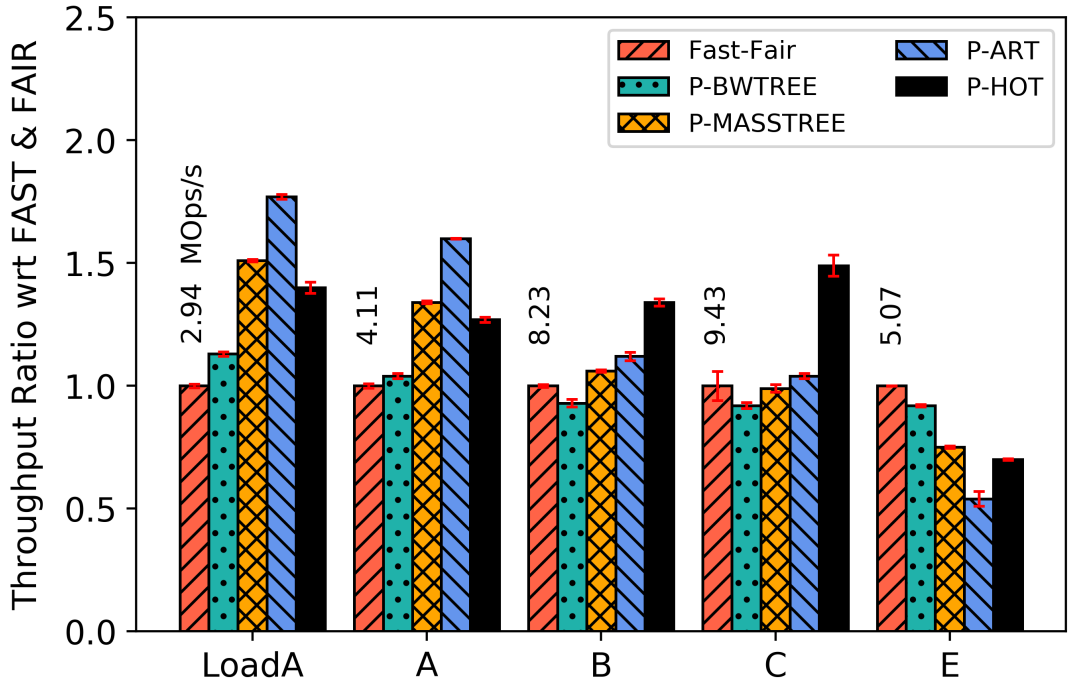
We evaluate the performance of indexes converted using the RECIPE approach against state-of-the-art hand-crafted PM indexes on Intel Optane DC Persistent Memory Module (PMM). The experiments are performed on a 2-socket, 96-core machine with 768 GB PMM, 375 GB DRAM, and 32 MB Last Level Cache (LLC). We use the ext4-DAX file system running kernel 4.17 on the Fedora distribution. All our experiments are performed in the *App Direct* mode of Optane DC which exposes a separate persistent memory device [89]. All experiments are performed on a single socket by pinning threads to a local NUMA node. Since the machine supports `clwb` instruction which is more efficient than `clflush`, we use `clwb` for cache line flushes in our experiments.

We split our evaluation based on the data structure into ordered indexes and unordered indexes. An ordered index aims to support both point and range queries,

but an unordered index only provides point queries. FAST & FAIR, P-Bw tree, P-Masstree, P-ART, and P-HOT are the ordered indexes, while CCEH, Level Hashing, and P-CLHT are the unordered indexes. We use the `libvmmalloc` library from PMDK that transparently converts traditional dynamic allocation interfaces to work on a volatile memory pool built on a memory-mapped file on PMEM [165]. We further collect low-level performance counters such as the number of `clwb` and `mfence` instructions along with the number of LLC misses per operation using the `perf` tool [164].

Workloads. We use the Yahoo! Cloud Serving Benchmark (YCSB) [45], the industry standard for evaluating key-value indexes. We use the index micro-benchmark to generate workload files for YCSB and statically split them across multiple threads [216]. Each generated workload mimics a real application pattern as shown in Table 3.2. We exclude workloads D and F as they involve updates and some indexes (FAST & FAIR, CCEH, CLHT) do not support key updates. For each workload, we test two key types - **randint** (8 byte random integer keys) and **string** (24 byte YCSB string keys), all uniformly distributed.

To evaluate the ordered indexes, we use both random integer and string type keys. As the open-source implementation of FAST & FAIR does not support string type keys, we implement string type support for FAST & FAIR by replacing integer key entries with pointers to the address of the actual string key, which is simplest way to support variable-sized string-type keys in B+tree in a crash-safe manner [36]. In both cases, we first populate the index with 64M keys using Load A, and then run the respective workloads that insert or read a total of 64M keys. For unordered indexes, we only use integer key types. We present the results from multi-threaded workloads using 16 threads and omit single threaded results as the performance trends are comparable to the multi-threaded workload. We use the default node size for each of the tree-based indexes, and a starting hash table size of 48KB. The reported numbers are averaged over several runs (with an average variance of 0.1%).



(a) Integer keys : Multi threaded YCSB

PM Index	Instructions		Last Level Cache Miss				
	`clwb`	`mfence`	LoadA	A	B	C	E
FAST & FAIR	7	8	11	10	8	7	8
P-Bw Tree	7	4	17	15	10	9	26
P-Masstree	3	5	7	7	6	5	8
P-ART	3	3	4	4	4	4	12
P-HOT	7	5	4	4	2	2	10

(b) Integer keys : Performance counters

Figure 3.4: **YCSB workload, integer keys for tree indexes.** The plot compares the performance of various tree based PM indexes using YCSB workloads for random integer keys (higher is better). On all workloads except for scan, the indexes converted using RECIPE outperform FAST & FAIR by up to 1.8×. The fine grained performance counters per operation help explain the observed trends (lower is better).

3.5.1 Ordered indexes

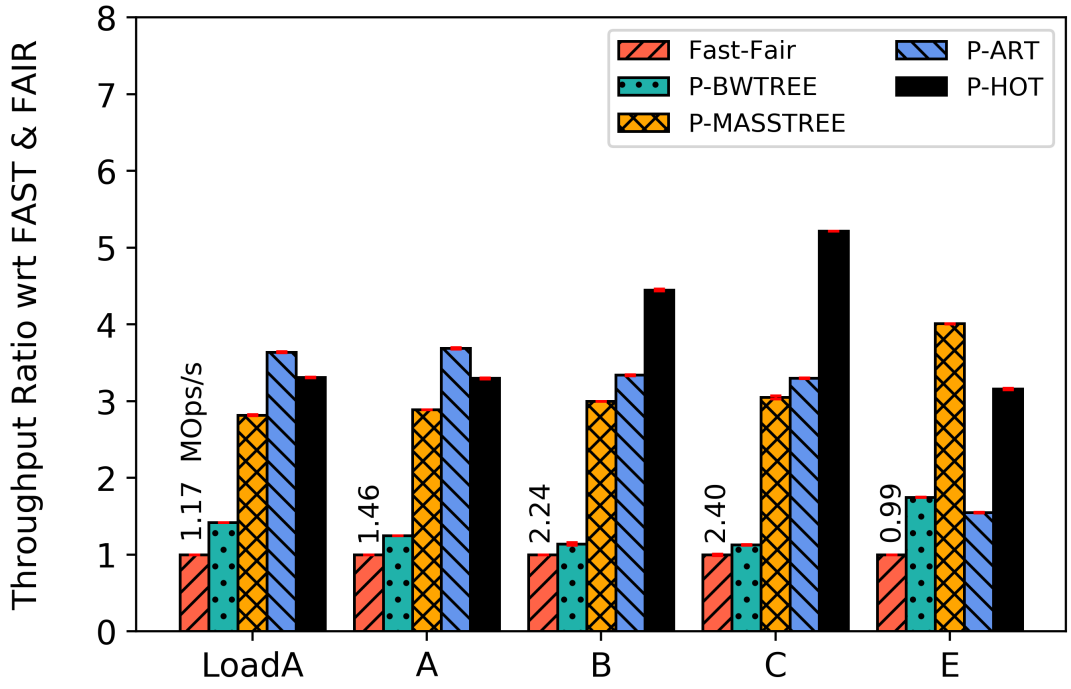
We evaluate converted indexes P-ART, P-HOT, P-Masstree, and P-BwTree against the only concurrent and open-source state-of-the-art PM B+ tree, FAST & FAIR.

Integer type keys. Figure 3.4 shows the performance comparison of various tree-based PM indexes using integer type keys. P-ART outperforms FAST & FAIR by up to 1.6× on write-heavy workloads as shown in Figure 3.4 a. The FAST algorithm sorts inserted keys in-place, which results in higher number of cache line flushes as compared to P-ART (Figure 3.4 b). This explains the lower performance of FAST & FAIR in write-intensive workloads. Trie-based indexes like P-HOT eliminate key comparisons in their search path as they do not store full keys in internal nodes. Therefore, point reads are more cache-efficient (P-HOT incurs 3× lower LLC misses compared to FAST & FAIR shown in Figure 3.4 b), thereby outperforming FAST & FAIR by 1.5× on read intensive workloads.

The performance of FAST & FAIR and P-BwTree is similar. P-BwTree performance is low because its operations require pointer chasing; for example, an insert can be only be performed after applying prior deltas. This leads to many LLC misses. As a result, P-BwTree performance is not significantly better than B+ trees with in-place updates.

FAST & FAIR outperforms all other indexes in range scans. There are two primary reasons for this. First, the keys are more compactly packed into nodes in B+ trees unlike tries, which makes it cache efficient in range scans. Second, the leaf nodes do not have sibling pointers in prefix tries, thereby requiring extensive traversals for range queries.

String type keys. Figure 3.5 shows the performance comparison of various tree-based PM indexes using string type keys. The absolute value of throughput decreases for string key types as compared to randint keys for all indexes. However, the mag-



(a) String keys : Multi threaded YCSB

PM Index	Instructions		Last Level Cache Miss				
	`clwb`	`mfence`	LoadA	A	B	C	E
FAST & FAIR	8	10	36	47	40	39	76
P-Bw Tree	8	6	40	48	39	37	62
P-Masstree	4	7	9	10	8	7	11
P-ART	3	4	4	5	5	5	22
P-HOT	7	5	5	5	3	3	12

(b) String keys : Performance counters

Figure 3.5: **YCSB workload, string keys for tree indexes.** The plot compares the performance of various tree based PM indexes using YCSB workloads with string keys (higher is better). All the indexes converted using RECIPE outperform FAST & FAIR by up to 5×. The fine grained performance counters per operation help explain the observed trends (lower is better).

nitude of performance drop is the highest for FAST & FAIR and native B+ trees, due to the high cost of string key comparison and pointer dereference to access the string key. This results in 8× more LLC misses in average as compared to prefix tries.

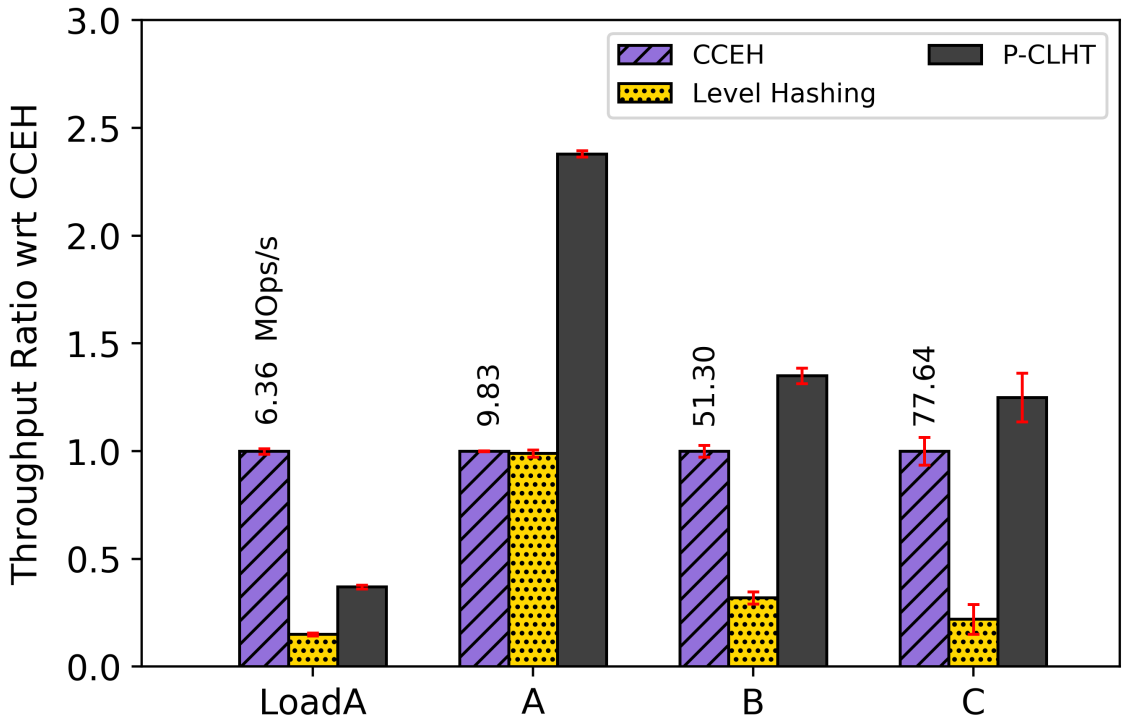


Figure 3.6: **YCSB workload with integer keys for hash tables.** The figure compares the performance of various hash based PM indexes using YCSB workloads (higher is better). P-CLHT, the index converted using RECIPE outperforms CCEH, the state-of-the-art hash table, by up to 2.4×.

Comparing absolute throughput, we see that FAST & FAIR performs $2.5-5\times$ worse for all YCSB workloads using string type keys as compared to integer keys. Whereas, prefix tries are only about 20% slower when switched to the string keys.

As shown in Figure 3.5 b, B+ tree's cache inefficiency results in $3.2-5.2\times$ worse performance compared to P-HOT. We observe that although Masstree uses a data structure that is a combination of B+ trees and prefix tries, its trie-based structure enables native key comparison by storing 8-byte partial keys to each B+tree's layer. Furthermore, it uses a collection of cache-friendly techniques such as prefetching, reduced tree depth, and careful layout of data across cachelines. These design choices makes Masstree better than its B+ tree counterparts across all workloads.

3.5.2 Unordered indexes

We evaluate P-CLHT against two state-of-the-art persistent hash tables, CCEH and Level hashing. Figure 3.6 shows that P-CLHT outperforms CCEH by up to 2.5× on the multi-threaded YCSB workload. Starting from a hash table size of 48KB, we insert 64M keys into the hash table in Load A, which triggers multiple re-hashing operations in both indexes. P-CLHT is 2× worse than CCEH for concurrent insert only workload, due to the globally-locked rehashing scheme that throttles concurrency. We confirm this by evaluating the two indexes using a single thread, where P-CLHT is only 12% slower than CCEH even in the presence of rehashing.

Table 3.3 shows that CCEH has lower throughput than P-CLHT though both similar number of cache misses and `clwb` instructions. This is due to the segment split mechanism of CCEH. When the hash table is sufficiently large, P-CLHT performs no rehashing (in workload A and B), thereby requiring only one `clwb` per insert. On the other hand, even when similarly sized, CCEH performs frequent segment splits that require multiple cache line flushes and expensive copy-on-write of new segments (117K segment splits occurred on inserting 10M keys into a sufficiently large hash table). CCEH requires additional pointer reads due to indirections introduced using directory and segment, which results in lower read performance over P-CLHT. Level hashing incurs a higher number of cache misses due to its two level architecture that results in non-contiguous cache line accesses [149] and lower throughput.

3.5.3 Comparison to WOART

WOART [111] is a single-threaded, hand-crafted, write-optimal PM variant of ART. WOART introduces a new recovery mechanism and modifies the node structure to be failure-atomic. The authors suggest modifying WOART to be multi-threaded using a global lock; since this leads to low concurrency, P-ART outperforms WOART on multi-threaded YCSB workloads by by $2 - 20\times$.

PM Index	Instructions		Last Level Cache Miss			
	`clwb`	`mfence`	LoadA	A	B	C
CCEH	2.3	3.0	1.5	1.5	1.1	1.0
Level hashing	3.7	5.8	4.0	3.3	4.0	4.0
P-CLHT	1.5	2.5	2.4	1.3	1.1	1.1

Table 3.3: **Performance counters.** The table shows the average number of `clwb`, `mfence` instructions per insert operation, and the average number of LLC misses per operation during each workload for randint keys (lower is better).

3.5.4 Summary

RECIPE-converted indexes outperform state-of-the-art hand-crafted PM indexes by up-to 5.2× on multi-threaded YCSB workloads. RECIPE-converted indexes are optimized for cache- efficiency and concurrency as they are built from mature DRAM indexes. RECIPE-converted indexes encounter fewer cache misses as compare to hand-crafted PM indexes. The append-only nature of indexes like P-ART results in up-to 2× lower cache line flushes, compared to hand-crafted PM indexes like FAST & FAIR. All these factors contribute to the performance gain of RECIPE-based PM indexes.

3.5.5 Testing Crash Recovery

We test each index for 10K crash states. We load 10K entries into the index, allowing it to crash probabilistically. We then perform a mixed workload consisting of a total of 10K inserts and reads into the index using 4 concurrent threads. Finally, we read back all successfully inserted keys from the index. On average, the end-to-end time for generating a crash state and testing it is 20ms.

We tested the current state-of-the-art PM indexes, and our converted PM indexes using the approach outlined in Section 3.3. Our testing revealed crash-consistency bugs in FAST & FAIR and CCEH. In FAST & FAIR, when two consecutive crashes occur during a node split and a node merge, the node to be deleted by the merge algorithm is not cleaned up correctly, which makes its right sibling in-

accessible by a reader. This results in data loss. CCEH results in stalled operations if a crash occurs during directory doubling, as it does not update directory metadata atomically. All PM indexes converted using RECIPE passed the testing with no bugs. Additionally, our durability test reveals that the initial node allocation containing the root pointer is not persisted in FAST & FAIR and CCEH.

3.6 Limitations and Discussion

Limitations. The RECIPE approach has a number of limitations. RECIPE is not applicable to all DRAM indexes; it can only be applied to DRAM indexes that meet its specific conditions. For instance, RECIPE cannot be applied to indexes with blocking reads or non-blocking reads with version-based retry without the ordered deterministic steps. RECIPE assumes the original DRAM index is correct; if it has a bug, the converted PM index will also have a bug. Indexes converted by RECIPE provide a low level of isolation (Read Uncommitted) in which non-blocking readers can return non-persisted data. However, this is not a fundamental limitation; developers can achieve stronger isolation by simply replacing the final commit stores in the conversion actions with other primitives such as non-temporal stores [178], PSwCAS (a Persistent Single Word CAS) [200]. Finally, RECIPE focuses on correct and principled conversion of DRAM indexes into PM indexes; there are usually opportunities for further optimization.

Optimization. We can increase the performance of converted PM indexes by reducing the number of cache line flushes or memory fences using techniques like persist buffering and coalescing [163]. Persistent buffering reduces the excessive flush and fence overhead by allowing flushes between independent cache lines to be reordered. Persistent coalescing facilitates batching multiple cache line flushes to the same cache line [41, 42]. RECIPE-based conversion inserts a flush and fence operation after each store. We optimized this by buffering and coalescing the flushes wherever possible in our RECIPE-converted indexes presented in Section 3.4; however, such optimizations

turned out to be heavily dependent on the implementation of the index structure. As we could not generalize these optimizations into conditions, RECIPE leaves it to the developer to identify and apply them.

Automation. Converting indexes using Condition #1 and #2 only requires cache line flushes and memory fences after every `store` instruction. Although this sounds simple and easy to automate, the challenge in automating these conversions lies in the many different ways in which the same logical steps are implemented in different indexes. For example, an atomic `store` operation could be implemented using the C++ atomic library, or through a simple pointer assignment, followed by `mfence`.

3.7 Summary

This chapter presents RECIPE, a principled approach for converting concurrent DRAM indexes into crash-consistent indexes for PM. The main insight behind RECIPE is that *isolation* provided by a certain class of concurrent in-memory indexes can be translated with small changes to *crash-consistency* when the same index is used in PM. We present a set of conditions that enable the identification of this class of DRAM indexes, and the actions to be taken to convert each index to be persistent. Based on these conditions and conversion actions, we modify five different DRAM indexes based on B+ trees, tries, radix trees, and hash tables to their crash-consistent PM counterparts. The effort involved in this conversion is minimal, requiring 30–200 lines of code. We evaluated the converted PM indexes on Intel DC Persistent Memory, and found that they outperform state-of-the-art, hand-crafted PM indexes in multi-threaded workloads by up-to 5.2×. For example, we built P-CLHT, our PM implementation of the CLHT hash table by modifying only 30 LOC. When running YCSB workloads, P-CLHT performs up to 2.4× better than Cacheline-Conscious Extendible Hashing (CCEH), the state-of-the-art PM hash table.

Chapter 4: Dinomo - An Elastic, Scalable, High-Performance KVS for DPM

This chapter presents DINOMO [116], a new key-value store (KVS) for DPM. We first motivate the needs of new key-value store designs for DPM. Next, we describe the details of DINOMO including its API, target workloads, goals, and the guarantees it provides. Then, we explain how DINOMO achieves its goals (Table 4.2).

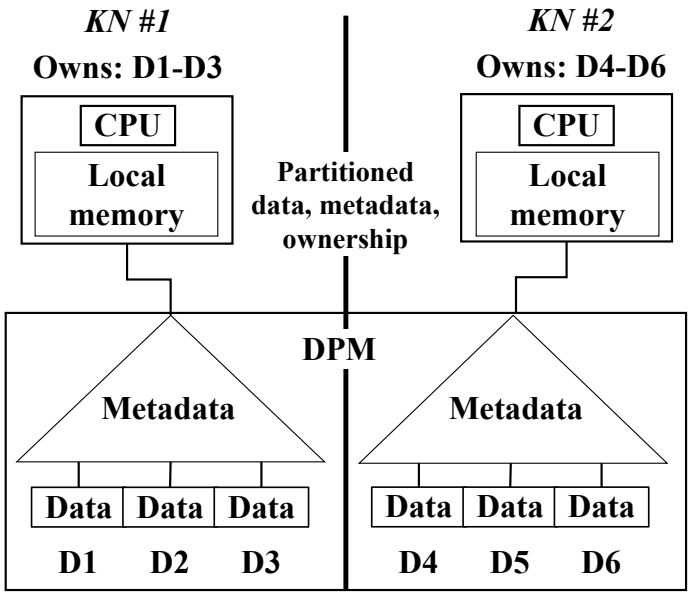
KVS property	Dinomo	Clover	AsymNVM
Data	shared	shared	partitioned
Metadata	shared	shared	partitioned
Ownership of data	partitioned	shared	partitioned
High performance	✓	×	✓
Scalability	✓	×	✓
Lightweight reconfiguration	✓	✓	×

Table 4.1: **Design choices and properties of different DPM KVSs.**

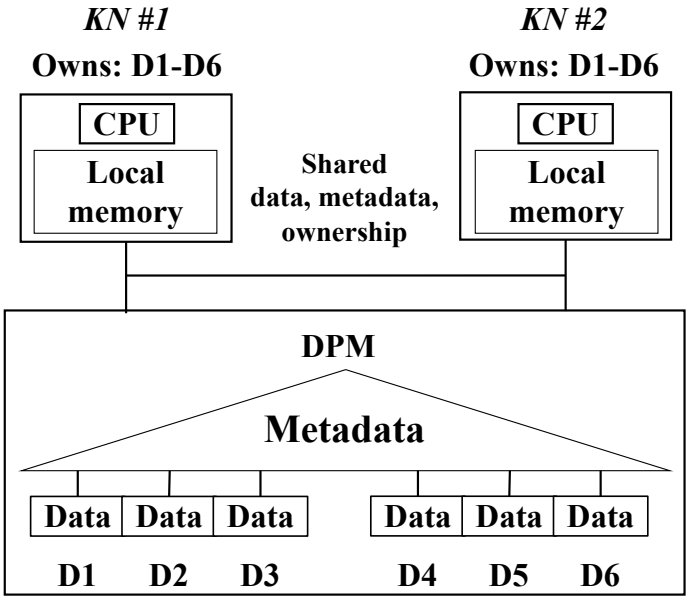
4.1 Motivation

Previously proposed DPM key-values stores differ based on how they handle data, metadata, and ownership. Metadata is information used to locate and access data (like an index). Ownership determines if a data item can be read or written.

AsymNVM. AsymNVM [137] adopts a shared-nothing architecture, as shown in Figure 4.1 (a). Data in DPM is partitioned, and each partition is exclusively accessed by a single KN. Every KN uses its local memory to cache data from its partition (Table 4.1); caching helps reduce expensive network round trips to DPM. As KNs have exclusive ownership over data, their caches can preserve locality and can be consistent without incurring additional consistency overheads. Thus, shared-nothing architectures provide high performance and scalability in the common case by effec-



(a) Shared nothing



(b) Shared everything

Figure 4.1: **System architectures for DPM KVSs.** In the shared-nothing architecture, data in DPM is partitioned, and each partition is exclusively accessed by a single KN. In the shared-everything architecture, all KNs share the ownership of data in DPM, and every KN can access and modify all data and metadata. Ownership is shown above each KN.

tively using KN caches to process requests. However, reconfiguring the number of KNs or balancing load across KNs requires physical reorganization of data and metadata [18, 76, 104, 137]. For example, adding a new KN may require the metadata of a partition to be split, resulting in expensive data copies at DPM. Thus, AsymNVM offers performance at the expense of elasticity and fast reconfiguration.

Clover. Clover [190] adopts a shared-everything architecture, as shown in Figure 4.1 (b). All KNs share the ownership of data in DPM, and every KN can access and modify all data and metadata (Table 4.1). KNs can use local memory to cache data. However, due to sharing, KNs have poor cache locality and need to keep their caches consistent, incurring significant consistency overheads that reduce the common-case performance and scalability [168]. Nevertheless, Clover can support lightweight reconfiguration without re-partitioning data or metadata and allow straightforward load balancing across KNs. Overall, Clover offers elasticity and lightweight reconfiguration at the expense of high common-case performance and scalability.

Goals	**Dinomo techniques**
High performance	Ownership partitioning, DAC
Lightweight reconfiguration and scalability	Ownership partitioning
Linearizable reads and writes	Shared DPM, Ownership partitioning

Table 4.2: **Dinomo goals and design techniques.**

4.2 Dinomo

We now present Dinomo, a key-value store (KVS) for DPM. We first describe its API, target workloads, goals, and the guarantees it provides. Then, we explain how Dinomo achieves its goals (Table 4.2).

API. Dinomo allows applications to perform `insert(key, value)`, `update(key, value)`, `lookup(key)`, or `delete(key)` on variable-sized key-value pairs. We refer to

the `lookup` operations as *reads*, and the `insert`, `update`, and `delete` operations as *writes*.

Target workloads. DINOMO targets applications with dynamic working sets and sizes, and non-uniform workloads with varying skew [156, 171, 204]. Large variations in workloads require DPM KVSs to allow the elastic deployment of resources (e.g., KNs) in response to those dynamics [28, 219].

Goals. DINOMO aims to achieve the following goals:

- High common-case performance, in the absence of failures or reconfiguration
- Scalability of performance when the number of KNs increases
- Lightweight online reconfiguration to effectively handle KN failures, bursty workloads, and load imbalance on available KNs
- Linearizable reads and writes

Guarantees. DINOMO guarantees that once committed, data will not be lost or corrupted regardless of KNs failures. It also ensures data remains available if at least one KN and DPM are available.

4.2.1 Architecture

Figure 4.2 shows the high-level architecture of DINOMO. DINOMO consists of clients, routing nodes (RNs), KVS nodes (KNs), DPM, and a monitoring/management node (M-node). We describe these components and how a request flows between them.

Applications and users interact with DINOMO through clients. RNs are the client-facing tier that provides cluster membership and isolates clients from the internal variation of the KVS cluster. A client first contacts an RN to obtain cluster membership and caches the mapping of key ranges to various KNs. The client contacts the appropriate KN, which will then perform the read or write operation on its behalf.

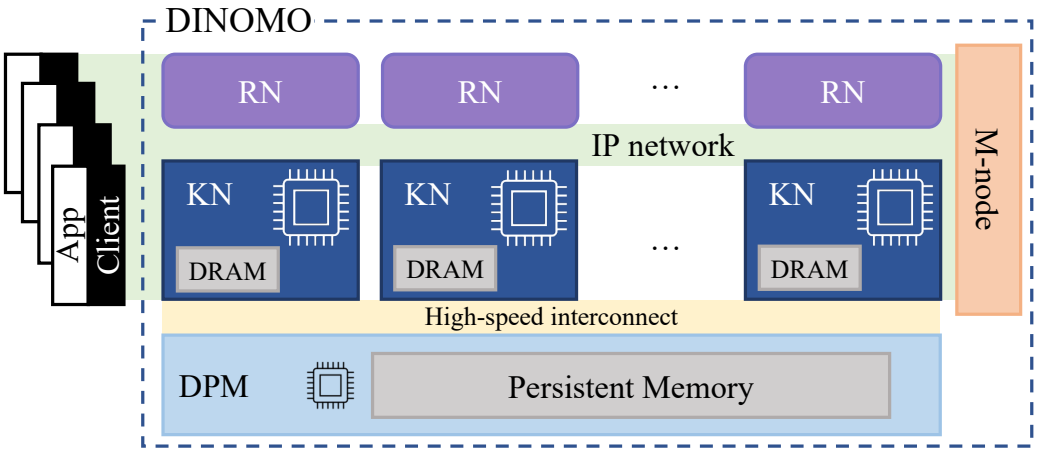


Figure 4.2: **Overview of the Dinomo cluster.**

Each KN is equipped with general-purpose processors and a small amount of DRAM relative to the DPM capacity. The KN uses one-sided or two-sided RDMA primitives to access DPM over the interconnect [9]; note that the one-sided RDMA primitive can read or write data without involving the DPM processors. DPM has the large shared PM pool and limited computational power relative to KNs [137, 190, 217]. This asymmetry is deliberate: KNs are intended to run complex operations in the critical path, whereas DPM is intended to execute lightweight tasks outside the critical path, while keeping the cost of provisioning DPM low. The KN caches the data it fetches from DPM in its local DRAM, and responds to client requests. The M-node observes KNs statuses and workload characteristics to detect KN failures, load imbalance, or workload skew, and triggers a suitable reconfiguration.

Note that we separately deploy the different functional components of Dinomo to enable us to independently scale them up and down as required. It is also possible to co-locate some components at the expense of reducing the efficiency of policy decisions when scaling resources.

Assumptions. We assume all components in the Dinomo cluster are inter-connected through a reliable local network (either over TCP/IP or RDMA RC). The interconnect bandwidth between KNs and DPM is lower than the memory bandwidth of the

PM itself, usually making network the bottleneck [8, 93]. KN failures are fail-stop and independent of DPM failures; when an KN fails, the KN abruptly terminates its execution and its local DRAM contents are lost. DPM has internal mechanisms or hardware support to ensure high availability [99, 117, 137, 190, 222] and hardware-level memory protection [154, 185, 195, 215]. The M-node is always alive; this can be ensured via consensus and replication [108, 109, 160]. As the M-node deals with infrequent lightweight tasks, using consensus does not introduce performance bottlenecks. Lastly, we assume M-node detects the KN failures correctly without false-positive or false-negative detection. stuidee perssone.

4.2.2 Data organization on DPM

Figure 4.3 shows the data-plane components in Dinomo. Dinomo stores data (key-value pairs) and metadata (indexing structures) in DPM for providing durability and as the source of ground truth.

Storing data in logs. In response to a write request, a KN writes data to an exclusive log on DPM. This write is performed with a single one-sided write operation in the critical path. The log is broken into a series of segments. Since each KN handles requests on exclusive logical data partitions (§4.2.4), two KNs will never log a write for the same key. The DPM processors asynchronously merge the write operations in a log segment in order into the metadata index. Logs of different KNs may be merged into the index simultaneously.

Metadata index. The metadata index in DPM must satisfy the following requirements. First, KNs should not hold locks while performing index traversals; locks cause cross-node synchronization overheads. Next, even if a KN fails while performing an index traversal, other KNs should be able to make progress. Finally, the index should support concurrent and consistent updates, allowing DPM threads to perform non-conflicting updates in parallel. Most state-of-the-art concurrent PM indexes satisfy these requirements [11, 37, 82, 112, 136]; these PM indexes provide lock-free reads and

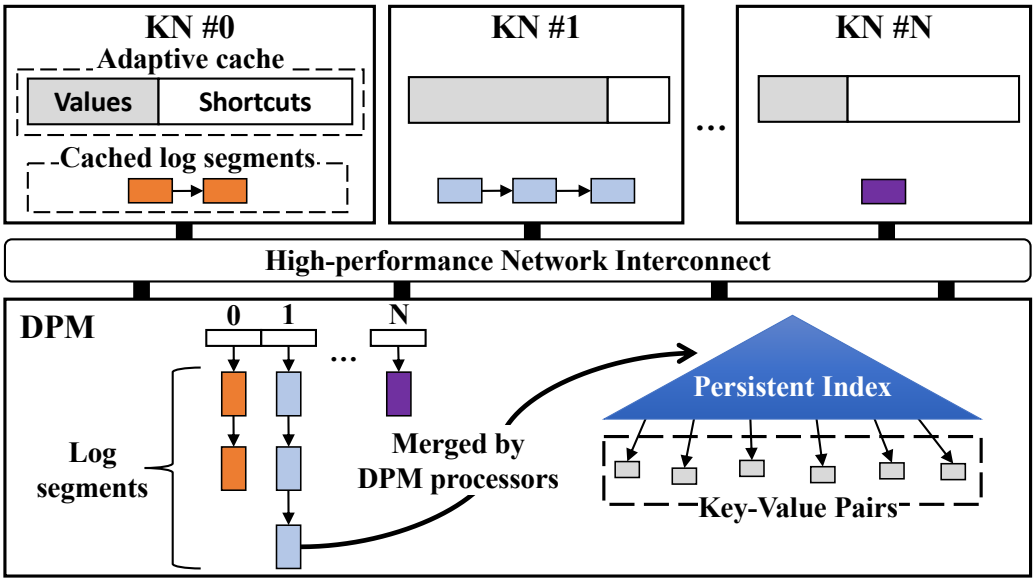


Figure 4.3: **Dinomo data plane.**

log-free in-place writes. Thus, with such PM indexes, DINOMO provides a globally consistent view of data in a scalable manner, independent of the number of KNs.

Consistency. DINOMO guarantees linearizability, the strongest consistency level for non-transactional stores [194]. DINOMO ensures that a successful write request commits the data atomically in DPM, and that subsequent reads return the latest committed value. To satisfy linearizability, DINOMO merges data logs in request order to the metadata index. Other core design decisions like ownership partitioning across KNs (§4.2.4), and using indirect pointers for selective replication (§4.2.4), help provide linearizability. Before reconfiguration or after failure, DINOMO merges all pending logs from the KNs involved before allowing the other KNs to serve reads.

4.2.3 Disaggregated Adaptive Caching

It would be prohibitively expensive for KNs to do network round trips (RTs) for every read operation. To avoid these overheads, KNs use local DRAM to cache data and metadata. Because KNs have limited memory, efficient caching is crucial

for high common-case performance. We introduce *Disaggregated Adaptive Caching* (DAC), a novel caching scheme to efficiently use DRAM at KNs.

Motivation. As DPM is directly accessible to KNs via one-sided RDMA operations with low latency owing to its byte addressability, KNs can cache not only data in the form of *values* but also metadata in the form of *shortcuts*. A *value* entry keeps the entire copy of a DPM value, so the KN can access everything locally. A *shortcut* entry keeps a fixed 64-bit pointer to the value in DPM; accessing the data incurs a one-sided operation to DPM. If neither value nor shortcut are cached, accessing the value incurs significant overhead: the KN needs to traverse a metadata structure in DPM to find the value's location and then access the value. Traversing metadata structures like trees, skip lists, or chaining lists in hash tables, requires multiple RTs to DPM or remote procedures in DPM, both of which have much higher overheads than a single one-sided operation [3, 198, 225, 228].

Caching values improves performance relative to caching shortcuts, but requires more cache space. This raises an interesting question: is it better to cache a few values with no overheads upon cache hits, or a larger number of shortcuts with fixed hit overheads?

The answer is simple in extreme cases: in highly skewed workloads, where a small number of hot key-value pairs can fit in the cache, storing values is better. When workloads are close to uniform distribution with total size larger than the cache, storing shortcuts is better. Unfortunately, most workloads fall between these two extremes and offer no clear answer. A simple static caching policy may reserve some fixed ratio of cache space for storing values and devote the rest to shortcuts. What should this ratio be? We observe that the efficient ratio is dependent on workload patterns and aggregate memory available for caching. In a disaggregated system like Dinomo that has autoscaling, neither workload patterns nor memory available is known ahead of time, ruling out static policies.

Adaptive Policy. We introduce DAC, a novel caching policy that dynamically

Disaggregated Adaptive Caching	
BEGIN	We start with an empty cache; start caching values
On a MISS	We cache the shortcut; if we need to make space for the shortcut, we DEMOTE a value (if present) or evict a least frequently used shortcut
On HIT	We check if we can PROMOTE this shortcut to value; we check if the benefits from caching the value instead of shortcut outweigh the benefits from evicting a suitable number of shortcuts
EVICT	Always evict the least frequently used shortcut
PROMOTE	Promote only if the benefits outweigh the costs
DEMOTE	Demote if we incur cache misses

Table 4.3: **Summary of the adaptive caching policy.**

selects the ratio of values to shortcut entries as needed. This policy automatically *adapts* to the changes in workload patterns and to the changes in the aggregate memory space for caching at KNs due to cluster reconfiguration, as shown in Figure 4.3.

Insight. DAC is based on the following insight. Performance is highly correlated with the number of network RTs, so we seek to minimize that. Caching a shortcut reduces RTs from M (where M is the cost of an index lookup) to one, while caching a value instead of a shortcut reduces RTs from one to zero. Thus, caching shortcuts provides the bigger gain. We treat value caching as an optimization on top of shortcut caching. Value caching is used when we have spare space in the cache, or when we observe that storing a value can serve more requests than storing an equivalent number of shortcuts. Table 4.3 details the policy.

In DAC, values can be demoted to shortcuts and shortcuts can be evicted. Shortcuts can also be promoted to values.

Demotions. Demotions occur on cache misses to make space for a new cache entry. To demote a value to a shortcut, we pick the least-recently-used key, leveraging temporal locality. To evict a shortcut, we pick the least-frequently-used key, in order to preserve frequently used keys in the cache and cater to skewed workloads.

Promotions. Promotions depend on whether the benefits from caching a value outweigh the benefits from caching a suitable number of shortcuts. To determine if a shortcut P needs to be promoted to a value, we use the following calculation. If at least N least-frequently-used shortcuts need to be evicted to make space for caching one value, then the shortcut P needs to satisfy the following relation to be promoted:

$$\begin{aligned} &Hits(P) \times \text{Avg. shortcut hit RTs} \geq \\ &\sum_{i=1}^{N} Hits(Shortcut_i) \times \text{Avg. cache miss RTs} \end{aligned} \tag{4.1}$$

This formula accounts for the two elements of the trade-off: the differences in the value and shortcut sizes, and the differences in the cost of a value miss and a shortcut miss. The left side of the inequality is the number of round-trips saved if we promote shortcut P to a value; the right side is the number of additional round-trips incurred if we evict N shortcuts to make space for the promotion of P. We promote if the savings are greater than the penalty. Note that the *Avg. shortcut hit RT* is always one, but the *Avg. cache miss RT* needs to be determined experimentally, which is done by keeping a moving average of past requests.

4.2.4 Ownership Partitioning

If multiple KNs cached the same value, this would incur consistency overheads (*e.g.,* cache invalidation) from ensuring linearizability. Dinomo sidesteps this via *ownership partitioning* (OP). Owing to the DPM architecture, where KNs are disaggregated from the shared PM pool, data access and ownership can be independent considerations: it is possible to partition ownership while sharing access to data. This insight motivates OP, which strikes a balance between shared everything and shared nothing. OP allows KNs to cache unique data, avoid consistency overheads, and thereby achieve high scalability. Although similar ideas have been previously used in other contexts [2, 15, 31, 197], we are the first to adapt it for DPM.

Central Idea. KNs have exclusive but temporary ownership of logical, disjoint

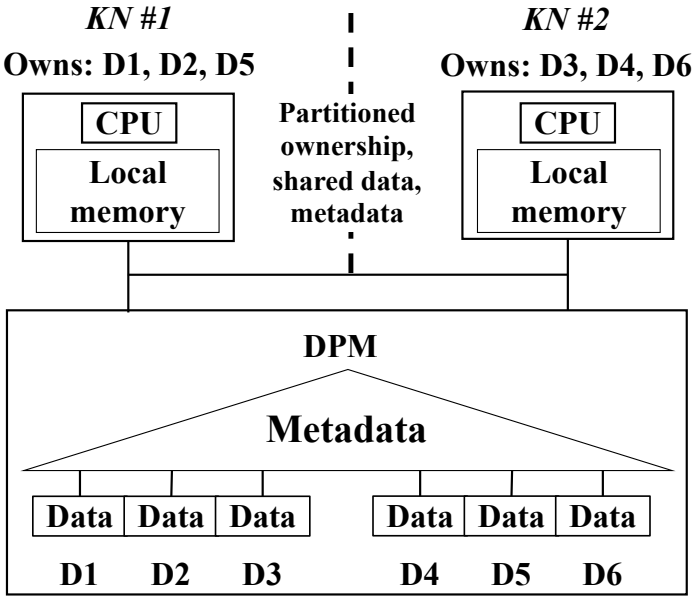


Figure 4.4: **Ownership partitioning for DPM** In ownership partitioning, the ownership of data is partitioned across KNs, while data and metadata are shared via DPM. Ownership is shown above each KN.

partitions of data, as shown in Figure 4.4. At any time, a partition is accessed by only one KN—its designated owner. OP allows KNs to scale without reorganizing data and metadata.

Partitioning the ownership. Routing nodes maintain the mapping of key ranges to their owner KNs. Clients' requests are routed to the appropriate owner KN. The owner KN can use its local DRAM to cache data and metadata with high cache locality and provide good read performance. DINOMO does not require cache coherence protocols at KNs, as KNs have exclusive access to their partitions. As scaling KNs increases the total DRAM available for caching, OP scales performance by utilizing the DRAM cache effectively (no redundant copies) and avoiding consistency overheads.

Ownership metadata. DINOMO uses consistent hashing to assign the primary owners for key ranges; DINOMO is compatible with other (*e.g.,* key-range or hash-based) partitioning algorithms. Within a KN, a key range is further partitioned

among its various threads. Both KNs and RNs maintain the partitioning metadata in a global hash ring, which stores key-to-KVS node-IP mappings, and a local hash ring, which stores key-to-thread mappings.

Whenever the mapping changes, RNs are updated together with KNs. Clients cache routing information; when the mapping changes, the KN they contact will direct them to a routing node to get the latest mapping information. Each KN always knows the key range it is supposed to handle, and will refuse requests for other key ranges.

Benefits. Ownership partitioning provides multiple benefits:

High performance. DINOMO achieves high performance in the common case by partitioning the ownership across KNs, allowing multiple KNs to cache unique data partitions with high cache locality.

Scalability. By avoiding the overhead for maintaining consistency at KN caches, DINOMO achieves scalability.

Lightweight reconfiguration. DINOMO can quickly change the number of KNs without physically reorganizing data or metadata; the current owner empties its cache, completes outstanding operations, hands ownership to the new KN, and the new owner begins serving requests. If a KN fails, partitions owned by the failed KN can be assigned to new owners that can immediately serve data.

Selective replication. Partition-based systems may suffer from load imbalance with highly skewed workloads. In these circumstances, adding more KNs does not distribute the load across available KNs. Even if a popular key's value is cached in a KN, performance is bottlenecked by that KN's processing or network capacity. DINOMO recognizes such scenarios and shares the ownership of highly popular keys across multiple KNs, effectively replicating such keys to provide scalability beyond a single node's abilities. The replication metadata is stored along with the mapping information at RNs and KNs and handled similarly. Clients cache and use this metadata to route requests to primary and secondary owners.

DINOMO uses *indirect pointers* to allow KNs to share ownership and read or write the shared key-value pairs consistently. An indirect pointer points to a location in DPM that stores a pointer to the value instead of the value itself, and the KNs access the shared value with one-sided CAS operations on the indirect pointers to ensure the linearizable access. Due to the sharing with indirect pointers, DINOMO incurs consistency overheads to balance the load across KNs. DINOMO limits these consistency overheads by using indirect pointers only for hot keys.

When a key becomes shared, DINOMO installs an indirect pointer to the key's value in DPM. When a KN updates a shared key, it writes the value at a new location and atomically updates the indirect pointer. A KN reading a shared key has to first read the indirect pointer and then read the value; thus, shared keys pay a cost that is avoided by default. Removing sharing from the key requires the KNs that own the shared key to invalidate it in their caches. Once the invalidation is done, the indirect pointer is removed in DPM.

4.2.5 Reconfiguration

The M-node triggers reconfigurations to improve performance when SLOs are violated, to release under-utilized resources, or to tolerate KN failures. We first present those policy details and then explain our principled reconfiguration protocol.

Policy engine. The policy engine in the M-node governs when and what kind of reconfigurations to trigger. Our policy engine follows prior autoscaling work [204], with simplifications for DINOMO; for example, memory consumption is not a consideration in scaling KNs since the memory in a KN is used as a cache without overflow. The policy engine allows the configuration of the following parameters: *average/tail latency SLOs*, *over-utilization lower bound*, *under-utilization upper bound*, *key hotness lower bound*, and *key coldness upper bound*. The M-node periodically collects latency information from clients, the average KN occupancy (*i.e.,* CPU working time per monitoring-epoch interval), and the average access frequency for keys from KNs. It

SLO	KN occupancy	Key access freq.	Action
Satisfied	Low	-	Remove KN
Violated	High	-	Add new KN
Violated	Normal	High	Replicate key
Satisfied	Normal	Low	De-replicate key

Table 4.4: **Policy violations and M-node action.**

then proactively detects the latency SLO violations and corrects them dynamically. Table 4.4 summarizes the reconfiguration scenarios.

Cluster membership changes. In DINOMO, cluster membership is changed under the following scenarios. First, the M-node may detect a KN failure and notify the alive nodes. Second, the M-node may detect a latency SLO violation (average or tail latency SLO) and find that all the KNs are over-utilized (the minimum occupancy of all KNs is larger than the *over-utilization lower bound*), which triggers the addition of a new KN. Third, the M-node may detect that there is an under-utilized KN (its occupancy is lower than the *under-utilization upper bound*); if the latency SLOs are not violated, this triggers that KN's removal. While ownership mapping is being redistributed due to the membership changes, clients' request latencies can briefly increase. To prevent the policy engine from over-reacting during the ownership redistribution, DINOMO adds or removes at most one node per decision epoch and applies a grace period to allow the system to stabilize before the next decision.

Ownership replication changes. If the M-node detects an SLO violation and notices that all KNs are not over-utilized, then the M-node identifies highly popular keys and increases their replication factor. In detail, the M-node considers a key to be highly popular if its average access frequency is greater than the *key hotness lower bound*. DINOMO increases the replication factor R (the number of secondary owners) of a hot key, based on the ratio between the average latency of the hot key and the *average latency SLO*. The M-node considers a key to be cold if its access frequency is below the *key coldness upper bound*. If the latency SLOs are met and none of the

KNs are under-utilized (the M-node cannot remove any KN), the M-node identifies cold keys with high replication factors ($R > 1$) and dereplicates them (R=1).

Fault tolerance. DPM is the source of ground truth in DINOMO; it persistently stores data (key-value pairs), metadata (indexing data structures), and other policy information (ownership/replication metadata). KNs and RNs store soft state that can be reconstructed if a node fails. When a KN or RN fails, it retrieves the up-to-date policy information from DPM and rebuilds the ownership mapping of key ranges before resuming. Unlike RNs, a KN failure changes the ownership mapping among the alive KNs. The M-node ensures that the ownership mapping is corrected before allowing the failed KN to resume. After detecting a KN failure, the M-node picks one of the alive KNs; this KN sends a request to DPM to complete the pending operations in the log segments from the failed KN. Upon completion, the M-node broadcasts the failure to all DINOMO components. On receiving a failure message, KNs and RNs repartition the ownership mapping by updating their hash rings.

Reconfiguration steps. We now describe how DINOMO performs reconfigurations. Broadly, the following steps occur:

1. KNs participating in the reconfiguration are identified (KNs for which the ownership mapping changes)
2. The KNs become unavailable
3. DPM synchronously merges the data in logs for these KNs
4. The KNs get their new mapping information
5. The KNs become available, and the cluster continues operation
6. The mapping information in the remaining KNs (not participating in the reconfiguration) is updated asynchronously
7. The RNs are asynchronously updated with the new mapping information

The cluster can continue operation at step five because KNs will reject requests for key ranges they do not own. Thus, other KNs can be updated without blocking the nodes undergoing reconfiguration. In certain special cases, DINOMO can perform reconfiguration without blocking any KNs. This can happen when a new partition is being added to DINOMO (no previous owner to race with) or when a KN fails and its partitions are being redistributed. Note that there is no expensive data copying or movement during reconfiguration. This is the key property that enables lightweight reconfiguration for DINOMO.

4.2.6 Optimizations

DINOMO includes optimizations in its data path to reduce CPU bottlenecks and network utilization from DPM.

One-sided & asynchronous post processing. To minimize the CPU bottlenecks and network utilization, DINOMO's data path uses *one-sided operations* with *asynchronous post processing*. With a one-sided operation (e.g., RDMA read, write, and atomic verbs), a KN executes directly on DPM without involving the DPM processor. One-sided operations have lower latency and higher bandwidth than two-sided operations (e.g., RDMA send and receive verbs) [54, 92, 148, 158, 202], but one-sided operations are limited in functionality [3]. For the best performance, DINOMO uses one-sided operations in the data path and delegates the post-processing of writes to the DPM processors asynchronously.

One-sided reads. For reads, an KN directly returns the value from its cache upon a value hit. On a shortcut hit, it performs a single one-sided operation to retrieve the value in DPM from the shortcut pointer. On a cache miss, the KN performs multiple one-sided operations to find the address of the value (index traversals), and uses another one-sided operation to fetch the value from that address.

Asynchronous post processing of writes. DINOMO batches multiple log entries into a log segment unit and writes them to DPM using a one-sided RDMA write operation.

With OP, DINOMO can batch the writes for the keys in the same partition without consistency concerns. The post processing to merge the writes into the metadata index is asynchronously handled by DPM processors off the critical path. DINOMO's KNs cache the committed log segments to aid the subsequent reads to be served locally at the KNs without expensive network RTs to read the large log segments remotely. These optimizations have two benefits. First, they reduce the latency as well as network costs per operation. Second, they amortize the merging operation across all the write operations in a log segment (typically several megabytes in size). Because the merging is done asynchronously, the DPM processors can have lower computing power without significantly affecting DINOMO performance.

4.3 Implementation

We implement DINOMO in 10K lines of C++ code. We use the standard C++ library and several open-source libraries including ZeroMQ [169], Google Protocol Buffers [24], libibverbs [46], and the PMDK library [103]. This section discusses DINOMO's DPM data structures, DAC implementation, and cluster management.

DPM metadata index. DINOMO uses RECIPE's P-CLHT (Persistent Cache Line Hash Table) [112], which supports lock-free reads and log-free in-place writes, as its metadata index in DPM. P-CLHT is a chaining hash table aimed at minimizing the CPU-cache coherence and persistence overheads on PM. Each bucket in P-CLHT has the size of a single cache line and holds three key-value pairs [48]. The design allows each access/update to the hash table to incur only a single cache-line access/flush in the common case. For lock-free reads, P-CLHT employs atomic snapshots of key-value pairs. We modify the index to use RDMA reads for lookups. On hash collisions, KNs may have to perform multiple one-sided RDMA reads to traverse the hash chain and read the value. The cacheline-conscious bucket design of P-CLHT, cache-coherent DMA [54, 92], and out-of-place value updates allow us to avoid memory-access races [148, 184] between the updates by DPM processors and

one-sided RDMA reads by KNs.

DPM log segments. DINOMO implements 8 MB log segments and handles variable length key-value pairs. KNs proactively preallocate log segments for their own use using two-sided operations. KNs log write operations into DPM log segments and cache them; upon cache misses in DAC, KNs have to search cached log segments to find the latest value. DINOMO implements Bloom filters atop cached log segments for quick membership queries. DINOMO maintains the following invariant: unmerged log segments are cached in the KNs that wrote them. Due to OP, other KNs will not access these log segments, thus eliminating the need for read operations to check the unmerged log segments on other nodes. KNs can add a new log segment to DPM without blocking until their unmerged log-segment length reaches a certain threshold (default is 2); when the threshold is reached, the critical write paths are blocked until the DPM processors complete merging below the threshold. DINOMO logs write operations with commit-markers (e.g., a seal byte at the end of the entry [54, 129]) to DPM log segments to ensure crash consistency and to aid recovery. The DPM index directly points to the values stored in the log entries. Since KNs know the address of the log segments they write (and therefore where values are stored), they can produce and locally cache shortcuts to values in DPM without an extra round trip. To garbage collect stale log segments, DINOMO maintains per-log-segment counters that reflect the number of valid and invalid values in each log segment. Once the number of invalid values matches the total number of values in a log segment, a DPM processor garbage collects the log segment.

DPM persistence. While merging log segments, DINOMO's DPM processing threads persist all the writes to the DPM index structure using `CLWB`, `sfence`, and non-temporal store instructions [178]. RDMA currently does not support durable RDMA writes. However, the proposed durable write in the IETF standards working document [186] behaves similar to a non-durable write, requiring one network round trip. Our implementation currently uses non-durable writes, and we plan to update these

to durable writes once they become available [101].

DAC. DAC is implemented using standard C++ libraries. DAC uses two unordered maps to store values and shortcuts. Least recently used values and least frequently used shortcuts are evicted. The key access frequency is tracked using a multimap. The shortcut entries contain a pointer to a DPM value, and the DPM value length. The value entries have two more extra fields, an access count and a copy of the DPM value. Demoted values are cached as shortcuts, and shortcuts being promoted inherit their access counts to preserve their access history.

Cluster management. DINOMO uses Kubernetes [71] for cluster orchestration. Pods are the smallest deployable units in Kubernetes. Each DINOMO component is instantiated in a separate Kubernetes pod with a corresponding Docker [60] container. DINOMO uses Kubernetes to add/remove KN pods and restart failed pods. The M-node pod is colocated with the Kubernetes master. The M-node's policy engine adds/removes KN pods by running simple bash scripts executing `kubectl` [159] commands to the Kubernetes master. The Kubernetes master keeps track of pod status using heartbeats, and the M-node uses this information to detect failures in KN pods.

4.4 Evaluation

We evaluate the performance of DINOMO and study the breakdown of the benefits from Ownership Partitioning (OP), Disaggregated Adaptive Caching (DAC), and selective replication. We design our experiments to answer the following questions:

- Does DAC help reduce network round trips? How does it fare against other caching policies?
- How much does the DPM compute capacity impact DINOMO's overall throughput?

- How does DINOMO fare against the state-of-the-art in terms of performance and scalability?
- What fraction of DINOMO's benefits can be attributed to the OP architecture and the DAC caching?
- How elastic and responsive is DINOMO while handling bursty workloads, load imbalance, and KN failures?

Comparison points. As our baseline, we use Clover [190], a state-of-the-art and open-source key-value store designed for DPM. Clover has a shared-everything architecture with a shortcut-only cache at its KNs. KNs perform out-of-place updates to the data in DPM, and incur additional overheads to provide strong consistency. For example, stale cached entries require KNs to walk through a chain of versions to find the most recent data in DPM.

Besides Clover, we compare DINOMO with two variants, DINOMO-S and DINOMO-N. DINOMO uses three techniques: DAC, OP, and selective replication. DINOMO-S caches only shortcuts; it is otherwise identical to DINOMO. As the source code of AsymNVM [137] is not publicly available, we implement DINOMO-N to compare DINOMO with a shared-nothing counterpart; it uses DAC but partitions data and metadata in DPM, where each partition is exclusively accessed by a single KN without selective replication.

Comparing DINOMO-S with Clover highlights the benefits of partitioning ownership in OP, and comparing DINOMO with DINOMO-S shows the benefits from DAC. We also investigate the trade-off from sharing data in OP by comparing DINOMO with DINOMO-N.

Experiment setup. We use Kubernetes pods to represent all of the node instances in the DINOMO cluster. We restrict the host resources assigned to the pods depending on the node types' features to emulate the asymmetric DPM architecture (i.e., KNs

have more-capable computation but smaller memory than DPM). Each individual pod is pinned to a separate server for resource isolation purposes.

We deploy Dinomo on the Chameleon Cloud [96], an experimental large-scale testbed for cloud research. We use InfiniBand-enabled (IB-enabled) servers as hosts for KNs and DPM; each two-socket server has Intel Xeon E5-2670v3 processors, 24 cores at 2.30 GHz in total, and 128 GB DRAM. The shared DPM uses a maximum of 4 threads and 110 GB of DRAM as a proxy for the PM, which is registered to be RDMA-accessible. Each KN uses a maximum of 8 threads and 1 GB of DRAM for caching (≈1% of the DPM size). DPM and the KNs are connected by Mellanox FDR ConnectX-3 adapters with 56 Gbps per port. We emulate PM using DRAM, as performance is constrained by the network rather than PM or DRAM: network latency (1–20 us) is at least 10× higher than DRAM or PM latencies (100s of ns); network bandwidth (7GB/s) is lower than PM bandwidth (32GB/s Read / 11.2GB/s Write) [8, 93].

The external servers that run application workloads, henceforth termed *client nodes*, and the routing service do not need a high-speed interconnect with the KNs or DPM. Hence, for client nodes and routing nodes (RNs), we use two-socket servers with AMD EPYC 7763 processors, 128 cores at 2.45 GHz in total, 256 GB of DRAM, and a 10 Gbps Ethernet NIC. Each client node uses 64 threads to run a closed-loop workload with one or more outstanding requests per thread. We use a single RN with 64 threads. The same routing layer is used across all KVS variants in our evaluation. In addition to the data plane components (KNs and DPM), Dinomo, Dinomo-N, and Dinomo-S use a control-plane instance for the M-node, which is deployed on a server (same server configuration as the RNs) with a single thread. For Clover, we use an extra IB-enabled server (same server configuration of the KNs) for its metadata server with 6 threads (4 workers, 1 epoch thread, 1 GC thread).

Workloads and configurations. We use YCSB-style workloads [44, 204] with five request patterns: read-only (100% reads), read-mostly (95% reads/5% updates and

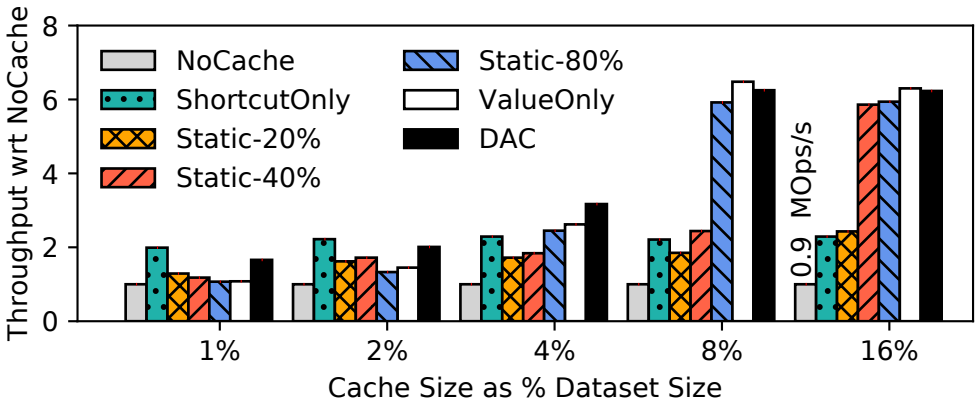


Figure 4.5: **Performance comparison of cache policies.** "Static-X%" means that X% of cache space is reserved for values. Apart from DAC, all other policies use LRU.

95% reads/5% inserts), and write-heavy (50% reads/50% updates and 50% reads/50% inserts). These workloads use 8B keys and 1KB values and the following Zipfian coefficients: 0.99 (the YCSB-default value) for moderate skew, 2 for high skew, and 0.5 for low skew (close to uniform). For each experiment, we first load 32 GB of data (key-value pairs) and then write up to 100GB of data during the workload including inserts. With 16 KNs, each equipped with a 1GB cache, the KNs can cache up to 50% of the loaded dataset. We generate the workload from the client nodes and measure system throughput and other profiling metrics, averaging them over a 10-second interval.

4.4.1 Microbenchmark

We use micro-benchmarks to investigate several issues. We first consider whether DAC is an effective caching strategy. Next, we explore how much compute capacity DPM requires to prevent the asynchronous merging of writes from becoming the bottleneck; based on the results, we also discuss how using DRAM to emulate PM affects our results.

DAC. The KN caches can be used to store values, shortcut pointers, or a mix of both.

Cache size as % dataset size	No Cache	Shortcut Only	Static 20%	Static 40%	Static 80%	Value Only	**DAC**
1%	5.0	1.5	2.0	2.5	3.6	4.2	**1.4**
2%	5.0	1.1	1.0	1.0	2.2	3.1	**0.9**
4%	5.0	1.1	1.0	0.8	0.5	1.4	**0.4**
8%	5.0	1.1	0.8	0.5	0.1	0.1	**0.1**
16%	5.0	1.1	0.5	0.1	0.1	0.1	**0.1**

Table 4.5: **RTs/operation of cache policies.** DAC has lowest RTs/op across different static caching strategies in all settings.

To evaluate DAC against different caching strategies, we use a single KN with 16 threads. We first load 30M key-value pairs into Dinomo with 8B keys and 64B values. We then run a read-only workload with a working set of 1.5M uniformly-distributed keys (5% of the dataset) to evaluate performance. We generate the workload locally and measure the peak throughput within the KN by varying the available DRAM for caching from 1%–16% of the dataset size. We configure Dinomo to use different caching policies (Figure 4.5). The static-X policies reserve X% of their cache size for storing values; the rest of the cache is used for shortcuts. All non-DAC policies use LRU to evict entries.

Figure 4.5 shows the read throughput obtained with the different cache policies. With an aggregate cache size of 2% of the dataset, a shortcut-only cache performs best, whereas with a cache size 4× as large, a value-only cache performs best. The aggregate cache size is dependent on the number of active KNs, which may dynamically change with cluster reconfiguration or KN failures. Therefore, a static caching policy is not a good fit. The right policy depends upon the workload patterns and aggregate cache size.

Despite not knowing the workload patterns or the aggregate cache size, DAC is within 16% of the best performing policy, in all settings. With a medium-sized cache that is 4% of the dataset size, DAC exceeds the performance of both shortcut-only and value-only caching policies by taking advantage of both. Further, as shown

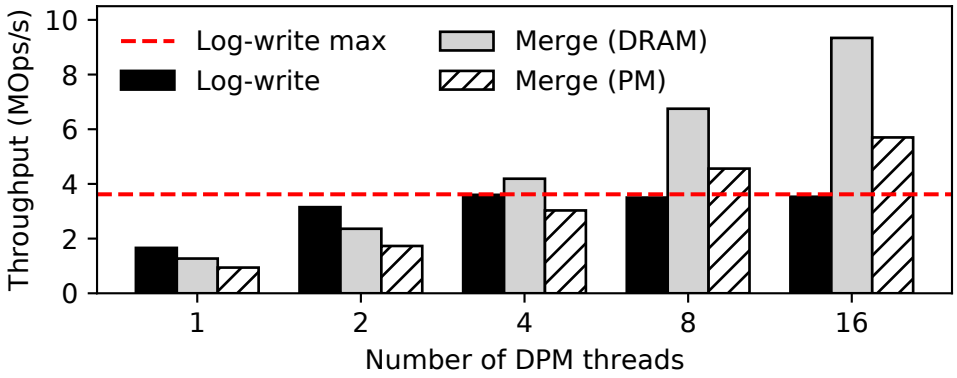


Figure 4.6: **Performance impact of DPM compute capacity.** The log-write throughput approaches the max with 4 threads on DRAM, while requiring more threads on PM.

in Table 4.5, DAC has the lowest number of round trips per operation compared to all static policies, reducing the network utilization and providing high performance.

Asynchronous post processing. A delay in merging log segments due to the limited compute capacity in DPM can block the critical path of KNs writing logs. To evaluate this impact from the worst-case scenario in our setup, we run an insert-only workload using 16 KNs and 8 client nodes; this is the most compute-intensive workload, as it incurs structural changes to the DPM index (*e.g.,* resizing hash table). We first load 32GB of data and then run the workload writing up to 100GB of data into DPM with 8B keys and 1KB values. We measure the peak throughput of log writing and merging for different DPM thread counts. For the log-write throughput, we collect the aggregate throughput across 16 KNs every 10 seconds for 30 seconds and average them; the log-write max is the maximum throughput the KNs can obtain if they never wait for DPM to merge logs. To measure the merge throughput, we pre-generate log segments locally on DPM for the dataset and then measure the performance of merging those log segments. As our testbed has no IB-enabled PM machines, we measure the merge throughput on PM using a local PM machine (Intel Xeon Silver 4314 CPU with 16 cores at 2.4GHz and 512GB Intel Optane DC PM on

4 NVDIMMs) to estimate the impact from using PM for DPM.

We make a number of observations based on the results in Figure 4.6. First, we observe that to write logs at the maximum rate, DPM should have enough computing capability to merge at the log-write max rate; four or more threads are required for our setup. Second, we observe that because of PM's higher access latency, PM merge throughput is lower than DRAM; when using four threads, the lower PM merge throughput can become the bottleneck. Third, we confirm despite in-DIMM write amplifications [93, 210], merge operations consume PM write bandwidth only up to 2GB/s (monitored by PCM [162]); 9.2GB/s out of the maximum (11.2GB/s) is still available to absorb incoming writes from the KNs over the network, making the network (7GB/s) the bottleneck rather than PM.

We conclude that, in some scenarios, using PM instead of DRAM requires a higher number of DPM threads to prevent the merging delay from becoming the bottleneck. However, even in this worst-case scenario, PM merge throughput with 4 threads was only 16% lower than log-write max; for more realistic scenarios with a mix of read and write operations (as used in our following end-to-end experiments), DPM should be able to operate with the same number of threads (4 threads or more) on both PM and DRAM for 16 KNs.

4.4.2 Performance and Scalability

We now compare the end-to-end performance and scalability of Dinomo, Dinomo-S (Dinomo with a shortcut-only cache), Dinomo-N (Dinomo with DAC and data/metadata partitioning), and Clover. We use workloads with moderate skew (Zipf 0.99) to observe the performance and scalability in the common case. We use 8 client nodes to run these workloads and measure the peak throughput by increasing the outstanding requests per client thread until the KNs' CPUs are saturated. After a 1-minute warm-up period, we collect the aggregate throughput across KNs every 10 seconds for 40 seconds and average them. In this experiment, the number of KNs

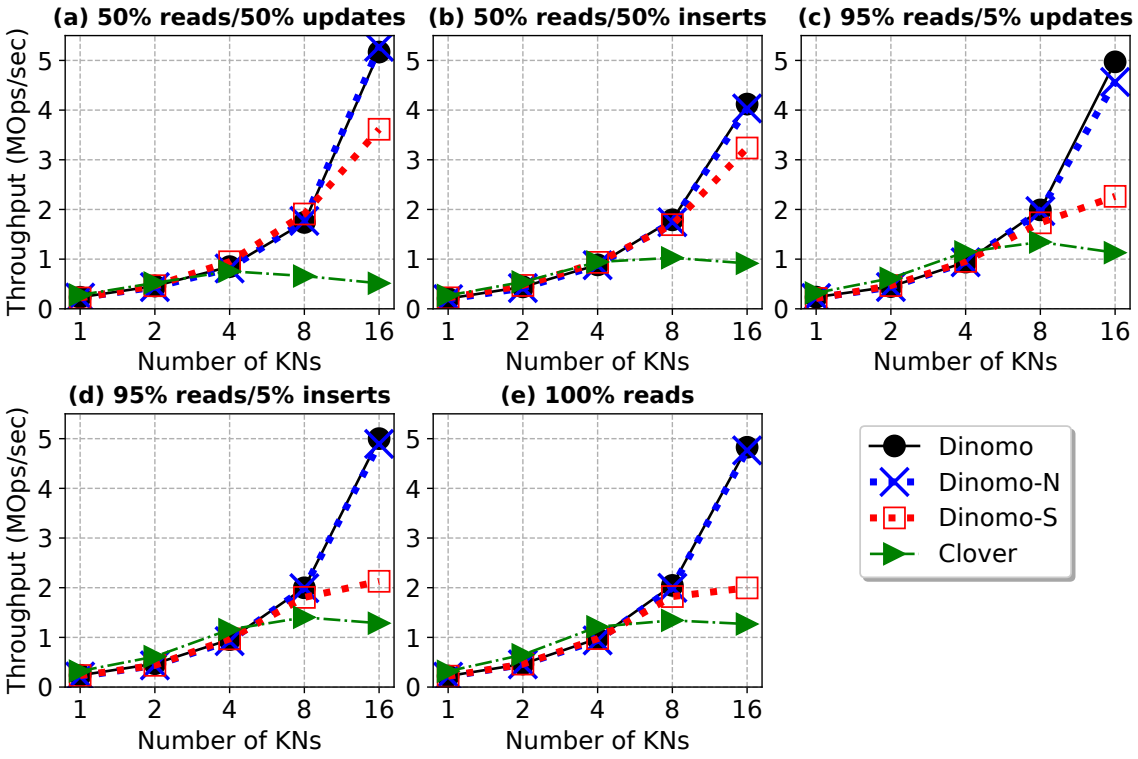


Figure 4.7: **Performance scalability.** DINOMO-S vs. Clover highlights the benefits from OP. DINOMO vs. DINOMO-S highlights the benefits from DAC. DINOMO vs. DINOMO-N shows the performance trade-offs between sharing data and OP.

is fixed, and hence there is no reconfiguration. However, the overhead to monitor system statistics (which are used to trigger reconfiguration) is reflected in the measurement of DINOMO and its variants. We profile the workload and collect metrics such as aggregate cache hit ratio and the average number of network round trips per operation (RTs/op) across all KNs. Due to space constraints, the full profiling numbers are omitted but can be found in our technical report [115].

As shown in Figure 4.7, DINOMO's throughput scales to 16 KNs. In contrast, Clover's throughput does not scale beyond 4 KNs due to either a network bottleneck or the CPU bottleneck from its metadata server. With 16 KNs, DINOMO outperforms Clover by at least 3.8× across all workloads. DINOMO-S does not scale beyond 8 KNs in read-dominated workloads because of network bottlenecks. The performance of DINOMO and DINOMO-N is almost on par (max difference is 11%). We observe that both DINOMO and DINOMO-N achieve high performance due to high cache locality at KNs resulting from partitioning. While partitioning data and metadata in DINOMO-N also reduces synchronization overheads, we did not notice significant benefit due to this in the tested workloads.

OP enables scalable performance. We observe that increasing the number of KNs from 1 to 16 reduces the cache hit ratio in Clover across all workloads. This performance drop is counterintuitive, as the DRAM available for caching increases with the number of KNs. However, in shared-everything architectures KNs can handle any request, so multiple KNs may incur cache misses on the same key. With more KNs, even with moderate skew, the redundant cache misses increase. In summary, shared-everything architectures do not provide good cache locality and prevent the efficient use of KN-side memory for caching. In contrast, OP partitions the ownership of keys across KNs, providing high cache locality for requests and eliminating redundant shortcuts at multiple KNs. Note that, for these workloads, DINOMO-S sees a 100% hit ratio across all KNs and with any number of KNs.

DAC boosts performance and scalability. DINOMO has a higher cache hit rate

(from values) with more KNs and takes fewer RTs/op, compared to both DINOMO-S and Clover. DINOMO-S has higher network costs: up to 10× more RTs/op than DINOMO. Clover is even worse: from 4× to 87× more RTs/op than DINOMO, due to shortcut-only caching and a lack of locality that results in consistency overheads and redundant caching. The aggregate memory available for caching increases with KNs for all systems. However, DAC helps KNs cache more values (as opposed to shortcuts), and thus incur fewer round trips to DPM per operation. In DINOMO, the cache hit % from values increases from 52% with 1 KN up to 88% with 16 KNs across all workloads. With 1 KN, DINOMO caches more shortcuts, incurring 1 RT at a cache hit, while with 16 KNs, DINOMO caches more values, and hence takes fewer RTs/op (0.1 RTs/op across all workloads). DINOMO has fewer RTs/op in write-heavy workloads on average than read-dominated workloads, as KNs persist multiple write operations in a batch with 1 RT on DPM. Overall, we see that DAC is effective in reducing RTs to DPM.

4.4.3 Elasticity

We now demonstrate DINOMO can elastically scale the number of KNs, balance loads across KNs, and tolerate failures. We use a workload with 50% reads and 50% updates with three different skew distributions. When a reconfiguration is triggered in this workload, any pending writes must be merged to DPM before the reconfiguration can proceed. We run a client node with one outstanding request per thread at a time.

Policy variables. We set the parameters of the policy engine (§4.2.5) and design the experiments to trigger various forms of reconfiguration. We use an *average latency SLO* of 1.2ms and a *tail latency SLO* (99-percentile latency) of 16ms. The *over-utilization lower bound* is configured to be 20% KN occupancy, and the *under-utilization upper bound* is set to 10% KN occupancy. Furthermore, we configure the *key-hotness lower bound* to 3 standard deviations above the mean key access frequency and the *key-coldness upper bound* to 1 standard deviation below the mean.

Note that the goal of the experiments is to study the elasticity of DINOMO under various scenarios; we chose these policy parameters as simple triggers for these scenarios, not as an indication of the best policies.

Auto scaling. We evaluate DINOMO with bursty, irregular workloads and compare its elasticity in scaling KNs with DINOMO-N. We were unable to run Clover for this experiment because Clover has no implementation for auto-scaling KNs. We produce scenarios where a new KN is required or an existing KN is no longer needed. Recall that DINOMO adds new KNs automatically only if a latency SLO is violated, the KNs are over-utilized, and an additional KN is available. DINOMO automatically evicts a KN only if the latency SLOs are met and the KN is underutilized. The grace period after each reconfiguration is configured to 90 seconds.

To produce a bursty workload, we start running the workload with low skew (Zipf 0.5) on DINOMO using 1 client node for 20 seconds. We then increase the load on DINOMO by 7× by adding 7 additional client nodes. We observe the performance of DINOMO for a few minutes until it stabilizes, and at the 230-second mark, we remove 7 client nodes to lower the load by 7× again. Figure 4.8 shows the behavior of DINOMO and DINOMO-N during this experiment.

DINOMO and DINOMO-N meet the latency SLOs until the load increases at 30 seconds, when the M-node detects a latency SLO violation: the tail latency SLO is exceeded. The M-node then observes that KNs are over-utilized (the minimum KN occupancy in DINOMO is about 35%), and hence corrects the situation by adding a new KN. Once the new KN comes online at 40-50 seconds, DINOMO shows a brief latency increase and throughput dip, as the nodes update their hash rings. However, DINOMO-N experiences a 40-second latency spike and throughput dip at 60 seconds, where the throughput drops to 0 due to the processing delay during data reorganization. After a 90-second grace period, although the average latency SLO is met, the tail latency SLO is still violated. DINOMO and DINOMO-N react to the situation by adding another KN. Again, DINOMO only sees a brief increase in latency, while

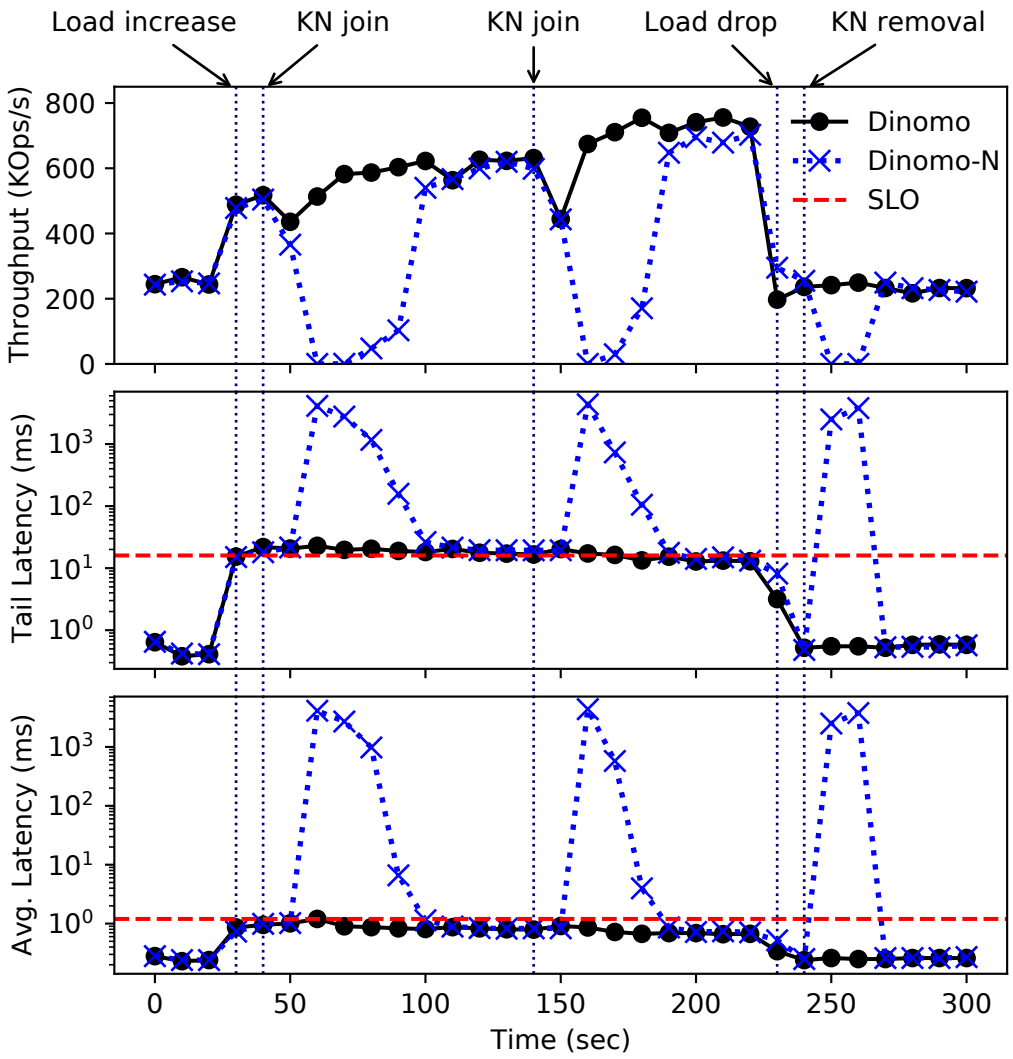


Figure 4.8: **Latency and throughput of Dinomo and Dinomo-N over time while changing load and number of KNs.** DINOMO is more responsive than DINOMO-N and can automatically scale KNs as required by changes in load.

DINOMO-N's latency increases for 30 seconds. After the grace period, as both latency SLOs are met, DINOMO and DINOMO-N do not take any further actions.

At 230 seconds, the load is suddenly reduced. In the next 10 seconds, the M-node detects an under-utilized KN with lower than 10% occupancy. As the latency SLOs are met, the policy engine triggers the KN eviction. While removing the under-utilized KN, DINOMO sees a brief rise in average and tail latency without violating SLOs. However, DINOMO-N shows a 20-second throughput dip and latency spike before stabilizing.

Overall, we see that DINOMO is more responsive with fewer throughput and latency disruptions than DINOMO-N and can automatically scale KNs as required by changes in load.

Load balancing. We now describe how DINOMO handles non-uniform load on its KNs and scales its throughput for hot spots, in comparison to DINOMO-N and Clover. To handle these scenarios, recall that DINOMO uses selective replication; this mechanism is triggered only if a latency SLO is violated due to a few hot keys and the KNs are not over-utilized.

For these experiments, we use a skewed workload with 8 client nodes and 16 KNs. We start the experiments with a low-skew workload (Zipf 0.5) and then switch to a highly-skewed workload (Zipf 2). DINOMO's policy engine checks that the KNs are not over-utilized (the minimum KN occupancy is lower than 10%) and identifies that the latency SLO is violated due to 4 hot keys. As a result, the policy engine triggers the selective replication of the 4 keys. Figure 4.9 shows the KVSs' behavior during the experiment.

Initially, all the KVSs meet the latency SLO and balance the load across KNs. At 20 seconds, the workload switches to the highly skewed pattern, resulting in latency SLO violations and an increase in load imbalance between KNs. DINOMO gradually increases the replication factor of the 4 keys between 30 and 90 seconds. During this period, DINOMO experiences brief tail latency spikes due to the additional delay

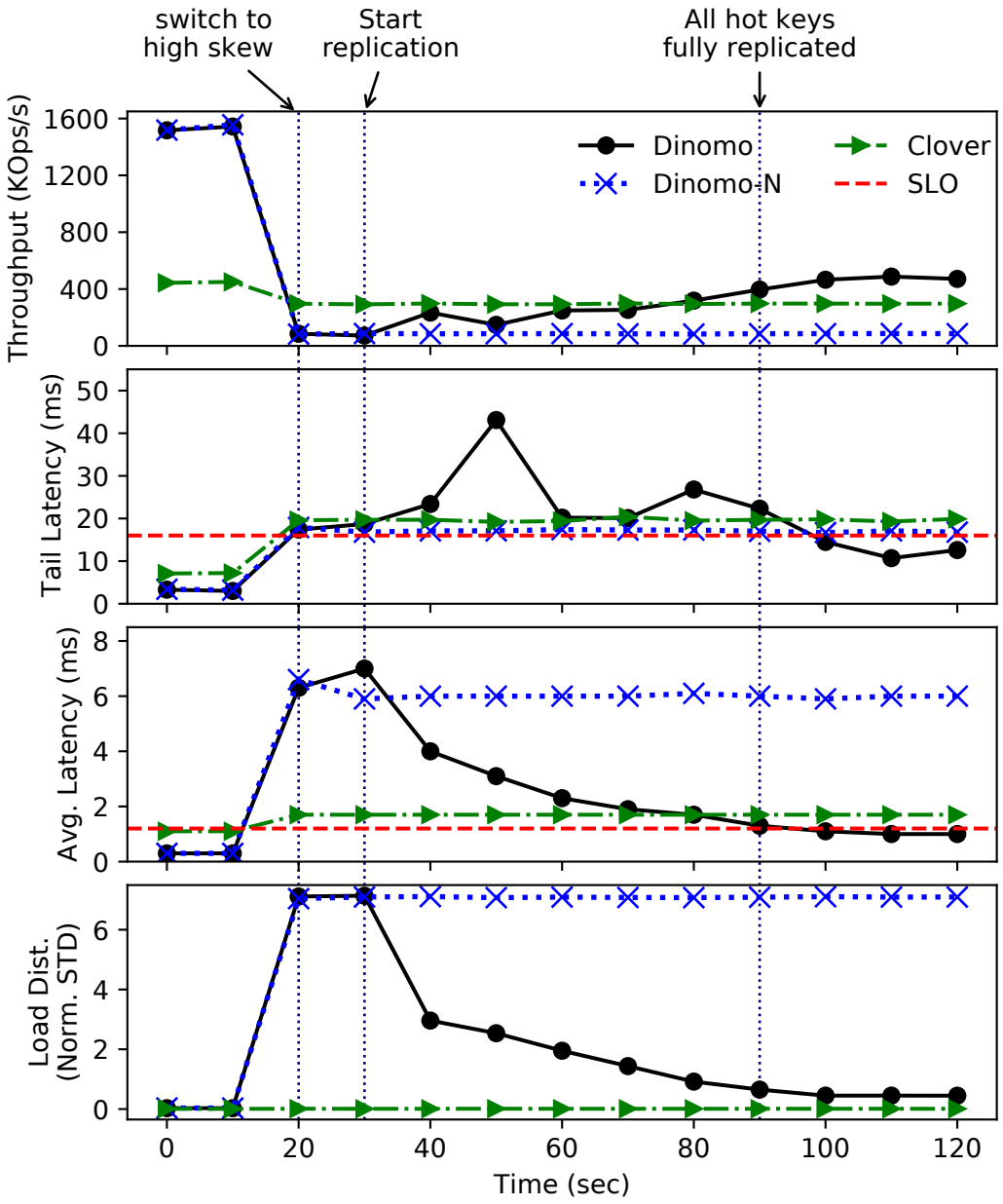


Figure 4.9: **Latency and throughput of Dinomo, Dinomo-N, and Clover over time while running the highly-skewed workload.** DINOMO selectively replicates hot keys to balance load and highlights the benefits of selective replication with OP for load balancing across KNs and for handling hot spots as a better alternative to shared-everything.

for clients to retrieve the up-to-date ownership mapping of replicated keys from the RN, but throughput gradually increases. At 90 seconds, DINOMO fully replicates the hot keys across all available KNs, and the throughput stabilizes. The latency SLOs are also met. DINOMO was the only system to satisfy the SLOs; both Clover and DINOMO-N constantly violate the SLOs for the highly-skewed workload.

Clover initially outperforms DINOMO without selective replication and DINOMO-N by almost 4× on the highly-skewed workload. However, once we enable selective replication in DINOMO, hot keys start becoming shared by multiple KNs at about 30-40 seconds; once all the hot keys are completely replicated, DINOMO's performance stabilizes in about 1 minute and it outperforms Clover by almost 1.6× and DINOMO-N up to 5.6×. Selectively replicating hot keys in DINOMO allows multiple KNs to access DPM for the hot keys, increasing the overall throughput. Our use of indirect pointers in accessing hot keys restricts KNs from caching values. Hence, DINOMO selectively replicates only the hottest keys while restricting KNs to cache only their shortcuts; KNs maintain exclusive ownership over non-hot keys and continue to cache their values adaptively.

Overall, our experiments highlight the benefits of selective replication with OP for load balancing across KNs and for handling hot spots as a better alternative to shared-everything.

Fault tolerance. Finally, we induce a KN failure to compare the resilience and elasticity of DINOMO, DINOMO-N and Clover. In a cluster with 16 KNs, we run a moderate skew (Zipf 0.99) workload for 2 minutes using 8 client nodes, and simulate a KN failure at around 40 seconds. We simulate the failure by eliminating a randomly selected KN. User requests are set to time out after 500ms. We observe that DINOMO quickly recovers from the KN failure (Figure 4.10). We notice that the throughput briefly drops by 45%, average latency increases by 1.2× (0.8 ms), and the tail latency increases by 1.5× (1.4 ms). Upon detecting the failure, DINOMO merges the pending log segments from the failed KN and redistributes ownership across other alive KNs.

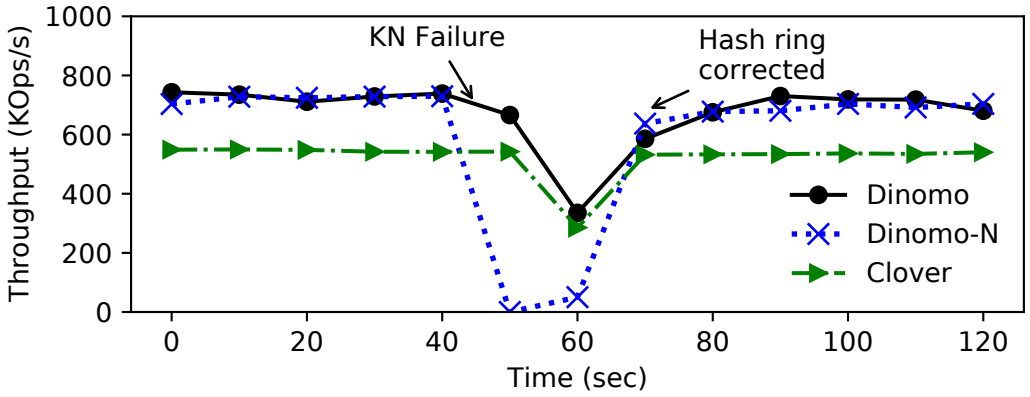


Figure 4.10: **Throughput of Dinomo, Dinomo-N, and Clover over time while handling KN failure.** Similar to Clover, DINOMO quickly tolerates a KN failure and stabilizes its performance.

These steps take less than 109 ms.

DINOMO-N, on the other hand, experiences a 20-second dip in performance at 50 seconds, where the throughput drops to 0 as it stops serving requests while reshuffling data. The time to reorganize data takes more than 11 seconds in DINOMO-N. Clover tolerates the KN failure elastically, showing a brief 55% drop in its throughput. Clover only needs to update the cluster membership of alive KNs in RNs after failures (without any data reorganization) to allow clients to retrieve the new membership after timeouts. The time to update RNs takes less than 68 ms.

Overall, compared to DINOMO-N, DINOMO recovers from KN failure faster since it is not required to reorganize data owing to the data sharing in OP. Similar to Clover, DINOMO stabilizes its performance quickly, and satisfies all SLOs.

4.5 Limitations and Discussion

Our work has a number of limitations. First, while we address the challenge of scaling KNs, we do not tackle how to make DPM reliable or scalable. Second, while our work provides mechanisms for scaling KNs, it does not tackle the policies for when

KNs should be scaled. Third, we assume that M-node accurately detects KN failures without false-positive or false-negative detection; such incorrect failure detection can have detrimental consequences. For instance, if M-node mistakenly identifies a KN failure when only a single KN exists, the system may malfunction. Furthermore, if a false-positively detected KN continues to respond to client requests, clients may receive incorrect data. However, evaluating the broader implications of the incorrect failure detection falls outside the scope of this work. Finally, Dinomo targets key-value store functionality for DPM systems. Many of the ideas presented in this work may be equally applicable for a broader range of DPM-based storage systems and disaggregated DRAM systems. For instance, OP could be valuable for other partitionable applications, such as partitioned databases [122, 188, 199, 221, 224] and graph processing systems [47, 50, 90, 139, 152, 214], to achieve both elasticity and scalability. Similarly, DAC could be employed in systems supporting KN-side caching [3, 6, 74, 75, 177] to consistently maintain high performance, regardless of workload dynamics, by minimizing I/Os to the slower memory/storage tier. However, the applicability of these techniques is currently limited to single-key or single-partition operations. To support multi-key, multi-partition operations, an additional coordinator may be necessary to ensure correct consistency across different keys and partitions. We consider these areas ripe for future work.

4.6 Summary

This chapter presents Dinomo, a novel key-value store for disaggregated persistent memory (DPM). Dinomo is the first key-value store for DPM that simultaneously achieves high common-case performance, scalability, and lightweight online reconfiguration. We observe that previously proposed key-value stores for DPM had architectural limitations that prevent them from achieving all three goals simultaneously. Dinomo uses a novel combination of techniques such as ownership partitioning, disaggregated adaptive caching, selective replication, and lock-free and log-free

indexing to achieve these goals. Compared to a state-of-the-art DPM key-value store, DINOMO achieves at least 3.8× better throughput at scale on various workloads and higher scalability, while providing fast reconfiguration.

Chapter 5: Shift - A Cache-Conscious Key-Value Store for CDM

This chapter presents SHIFT, a Cache-Conscious key-value store for disaggregated memory based on CXL (Compute eXpress Link). In this chapter, we explore solutions to achieve high performance, scalability, crash recoverability, and elastic reconfigurability for KVSs on CXL Disaggregated Memory (CDM). In particular, we rethink indexing, caching, and partitioning techniques that are previously proposed for RDMA disaggregated memory in the context of CXL. We first motivate the needs of new designs for CDM while analyzing existing solutions for RDMA disaggregated memory. Then, we present each technique in SHIFT in detail.

5.1 Motivation

We begin this section by explaining CXL in detail, an emerging cache-coherent interconnect technology for resource disaggregation. Next, we discuss solutions previously proposed to achieve the similar goals for RDMA disaggregated memory (we collectively call both disaggregated DRAM and DPM as disaggregated memory unless otherwise specified). Then, based on their limitations, we motivate the need to rethink existing indexing, caching, and partitioning techniques in the context of CXL.

5.1.1 CXL (Compute Express Link)

CXL [125, 143, 182] is an emerging disaggregation technology, which is a cache-coherent interconnect over PCIe. Among various CXL protocols, CXL.memory supports load-store accessible disaggregated memory expansion, pooling, and sharing. CXL.memory is not restricted in a specific memory type, supporting both DRAM and PM as the attachable CXL memory device. Like other disaggregation technologies, separate computing units, called as hosts in CXL settings, can access the dis-

aggregated memory over the CXL interconnects and they are equipped with DIMM-attached local memory. Depending on scale, CXL disaggregated memory can be directly attached to a single host as CPU-less local memory or can be interconnected to multiple hosts over a CXL switch(es)/fabric for memory pooling/sharing across the hosts. From the following sections, we call the hosts as KVS nodes (KNs) when they are specifically represented for KVS services.

Compared to RDMA-based memory disaggregation, CXL provides much lower access latency and higher maximum bandwidth. The access latency from hosts to CXL disaggregated memory is at hundreds of nanoseconds (*e.g.,* $170 - 300ns$) while RDMA network latency is 1 – 4µs [125, 143]. CXL 3.0, based on PCIe 6.0 technology, supports the transfer rate to 64GT/s, which allows for aggregate raw bandwidth of up to 256GB/s for x16 width link; ConnectX-6 RDMA NIC supports 200Gbps maximum total bandwidth. The low access latency allows us to efficiently disaggregate device memory while less compromising the performance of the applications compared to when their size fits into the local memory.

Moreover, CXL provides cache coherence between the host CPUs for the coherent load/store accesses to the same CDM addresses while the RDMA interconnect needs separate software-based protocols to guarantee this coherence. The hardware-guaranteed coherence can simplify software stacks by enabling load/store accessible coherent disaggregated memory without complex software-based coherence protocols. Furthermore, the applications over CXL disaggregated memory can enjoy performance benefits from CPU caches in accessing the disaggregated memory. However, the cache coherence traffic between hosts can easily become a scalability bottleneck in a system without careful designs. More importantly, incorporated into the cache-coherent hierarchy, CXL disaggregated memory incurs the write reordering from CPU caches at hosts to CDM, which causes more complexity for crash consistency.

Most modern processors reorder writes to memory at the cache-coherent hierarchy to optimize performance, including CXL disaggregated memory. This write

reordering can cause crash inconsistency when multiple hosts share the same CDM [52, 218]. Consider an example of an in-memory shared log built on CDM. Assume Host 1 writes data and commit records to the shared log (at different cache lines) in order, while Host 2 reads these records to check their validity. In this scenario, a power failure at Host 1 can result in the loss of the data record from CDM if the data record still exists in the CPU cache of Host 1 when the failure occurs, but only the commit record became evicted from the cache. If Host 2 reads the log records after this failure, it may assume that the log records were correctly written by Host 1, since the commit record exists in the shared CXL disaggregated memory.

To address this issue, GPF (Global Persistent Flush) support or programming using cache-line flushes (*e.g.,* `clwb`), non-temporal stores, memory fences (*e.g.,* `sfence`) is necessary for crash-consistent CDM sharing. GPF is originally proposed to support PM for CXL providing the same functionality as eADR with NVDIMMs [91]. It ensures the persistence of data written back to the volatile CPU cache of a host by flushing the entire CPU cache to the CXL DPM upon a host failure, such as a power failure. However, the usability of GPF is not restricted to the CXL DPM settings; it is also useful to guarantee crash consistency and to eliminate the expensive costs of manually flushing data during runtime for any CXL memory sharing scenarios regardless of the types of the memory devices. However, GPF is expected to be costly since it would require a large standby battery to back the entire cache hierarchy [5], thus GPF is likely to be provided as an optional feature like eADR. Therefore, applications developed for CDM sharing without GPF still need to be programmed using the flush, fence, and non-temporal store instructions.

5.1.2 KVS designs for RDMA disaggregated memory

All key-value stores priorly proposed for disaggregated memory are designed for RDMA-based memory disaggregation [116, 137, 190, 198, 225, 228]. However, as CXL provides the different characteristics from RDMA interconnects, we need to

rethink the existing solutions proposed for RDMA-based disaggregated memory in the context of the new CXL.

Indexing. Although one-sided RDMA operations have attractive features allowing low-latency remote memory accesses without involving remote CPUs, the usability of one-sided operations is highly restricted by their limited functionality. For example, due to the lack of pointer indirection, traversing remote data structures with one-sided RDMA operations requires multiple expensive round trips. To address this problem, many existing systems for RDMA-enabled memory servers as well as disaggregated memory redesign their index data structures to reduce the number of network round trips. For example, tree structures optimized for one-sided RDMA operations employ coarse-grained node size (> 1KB) to reduce an overall tree height [198, 225]; the smaller tree height results in fewer pointer-chasing counts using one-sided operations for tree traversals. Furthermore, hash tables are redesigned to be tree-like hierarchical structures to facilitate index caching [3] or to mitigate rehashing costs [228].

The existing designs, however, for RDMA-based memory disaggregation can be sub-optimal for the indexes in CXL disaggregated memory due to their I/O-oriented, cache-oblivious designs. Cache-conscious designs have been one of the key principles to build high-performance, scalable indexing data structures on byte-addressable memory/storage media in the cache-coherent hierarchy [48, 105, 112, 120, 141, 173, 174]. For example, historically, block-based tree structures for secondary storage medium (*e.g.,* HDD, SSD) use block-sized nodes optimized for storage I/Os [72], but cache-conscious in-memory tree structures [120, 141, 173, 174] employ smaller node sizes (*e.g.,* several cache lines: 128B, 256B, 512B) to minimize CPU cache misses and coherence traffic due to contentions [105]. As CXL disaggregated memory is in the cache-coherent hierarchy, improving cache efficiency is significantly important for high-performance, scalable CXL index structures.

Besides the performance issues, indexes for disaggregated memory must guarantee partial failure tolerance in an elastic manner; the indexed data in CDM is

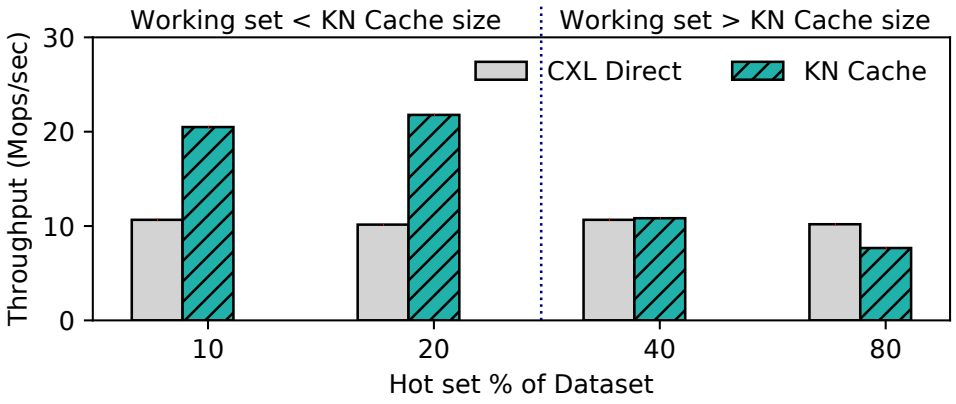
consistently available regardless of any crash failures in some KNs. There are three challenges in addressing the partial failures elastically. First, data written back to KN-attached CPU caches must be flushed to the shared CDM to guarantee crash consistency without data loss across KNs. Second, the partial updates to CDM due to a KN crash must be recovered (roll-back or roll-forward) elastically to allow other alive KNs to correctly make progress without blocking them. Third, locks held by the KN that crashes must be released in an online manner without a deadlock.

We observe, however, that existing index structures for RDMA disaggregated memory largely overlook the partial failure issues [198, 228]. Their write operations employ update-in-place designs without any recovery mechanisms, which can cause various issues like partial updates, data loss, or deadlock if a KN fails in the middle of the write operations. Note that it is non-trivial to convert crash-vulnerable indexes to be crash-resilient while keeping high performance and scalability without fundamental changes in their designs [112]; simply adding general crash recovery mechanisms like logging or CoW (Copy on Write) to the indexes can not only prevent elastic recovery, but also largely disrupt their original performance and scalability [82, 111, 112, 227].

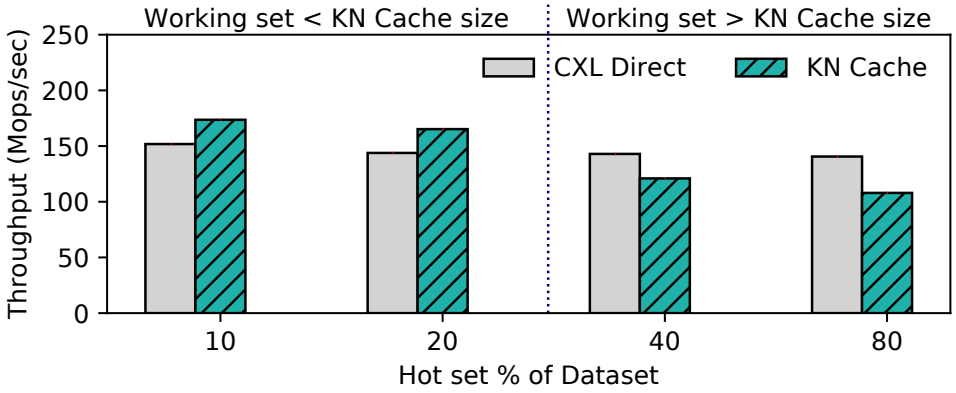
In summary, the existing index designs for RDMA disaggregated memory can be sub-optimal or are not reliably applicable to CXL disaggregated memory. There is a need both for a new CPU-cache conscious design and a way to address partial failure for CXL index structures.

Caching. KNs in CXL disaggregated memory settings have processor-attached local memory like in RDMA-based disaggregation settings. We may consider caching data into the local memory of KNs to reduce the accesses to the slower CXL disaggregated memory, as in RDMA-based disaggregation.

We observe, however, caching data into KN-side local memory does not always bring better performance than direct CDM accesses. Figure 5.1 shows the performance comparisons between using KN-side caching over a hashtable index at CDM (**KN Cache**) and using the index directly while bypassing KN-side caching



(a) Low load (8 KN threads)



(b) High load (128 KN threads)

Figure 5.1: **Performance comparison between direct and caching accesses.** The plot compares the performance of using KN-side cache and directly using the index (hashtable) in CXL disaggregated memory while bypassing the cache. The hot set percentage is the proportion of the dataset that is accessed by every operation.

(**CXL Direct**) to retrieve data. We emulate CXL disaggregated memory using remote NUMA-node memory and use local NUMA as KN while pinning all the threads at the local NUMA socket. The size of KN Cache is configured to 4GB over the 16GB key-value dataset at CDM. We run read-only workloads and measure the throughput of KN Cache and CXL direct while varying the request skews under two different request loads. The full details of the experimental setups are specified in Section 5.4.

		Low load				High load			
Hot set % of Dataset		10	20	40	80	10	20	40	80
KN Cache hit rate (%)		99.5	99.5	57	10	99.5	99.5	57	10
KN Cache miss penalty (% consumed time)		0.4	0.4	18	27	0.3	0.3	19	30
CPU cache misses per op	CXL Direct	9	9	9	9	8	8	8	8
	KN Cache	13	12	16	14	11	10	15	13

Table 5.1: **Performance profiling of direct and caching accesses.** The table compares the KN cache hit ratio, KN cache miss penalty rate of the total runtime, and LLC misses per operation of CXL Direct and KN cache at different request skews.

As shown in Figure 5.1, using KN Cache does not always bring better performance than CXL Direct. In the low load scenario (Figure 5.1 a), CXL Direct performs 1.3× better than KN Cache when the hot set percentage is 80%. Moreover, under the high load, CXL Direct outperforms KN Cache in the hot set percentage 40% and 80%, 1.2× and 1.3× better respectively. Finally, the performance gap between CXL Direct and KN Cache in the high hot set percentages (10% and 20%) becomes reduced as increasing the load. Note that these performance trends are different from those in RDMA-based setups; we confirm that KN Cache can constantly outperform the counterpart bypassing the cache only even with the 1% cache hit ratio.

These results come from non-negligible KN cache miss penalty with respect to the media access costs and less cache-efficient design nature of caching mechanisms than a standalone index design. Our performance analysis (Table 5.1) shows KN Cache miss penalty can account for up to 30% of the total runtime. The cache miss penalty includes the wasted overheads caused by cache index searches and cache updates due to the misses and it was measured at around $270 - 450$ns per KN cache miss in our experimental setup. This is a significant overhead considering the expected CDM access latency ($170 - 300$ns), explaining the lower KN Cache performance than CXL Direct at the low cache hit rates. Furthermore, KN Cache incurs more CPU cache misses per operation than the direct CXL accesses through the single index, as

shown in Table 5.1. The more CPU cache misses of KN Cache are derived from the CPU cache-inefficient design nature of caching algorithms. The caching algorithms include two features, one is an index to map cached entries and another is for cache eviction. The additional cache eviction mechanism causes more memory accesses than a standalone index design, thus incurring more CPU-cache misses. The better CPU-cache efficiency makes CXL Direct perform closer to KN Cache at high cache hit rates than expected by the raw device latency gap. Furthermore, it increases performance gap from KN Cache at low cache hit rates by better scaling its performance under the high load.

To summarize, we show that caching data into KN-side local memory does not always give us better performance than that of directly accessing CXL disaggregated memory while bypassing the cache. This is because the KN cache miss penalty can add considerable overheads to total runtimes and the additional cache eviction in caching mechanisms can cause more CPU cache misses degrading overall performance and scalability than a standalone index design. These results motivate us to rethink how and when to utilize KN-side local memory properly.

Partitioning. Two kinds of coherence mechanisms can exist in sharing CDM across multiple KNs: hardware-managed coherence and software-managed coherence. Even if CXL guarantees the coherent load/storage accesses to the same CDM from the hardware level (hardware-managed coherence), the coherence of the data copies cached locally at each KN-side memory still must be managed by software (software-managed coherence). Although their support makes programming more convenient for application developers over CXL, their coherence overheads from sharing the same CDM can easily become a bottleneck in scaling the number of KNs [190].

Some existing KVSs for RDMA disaggregated memory sidestep the coherence issues by using partitioning techniques [116, 137]. Partitioning schemes can be the simplest solution to address both the hardware-managed [127] and software-managed [116, 137] coherence traffic across KNs. However, as observed in our previous

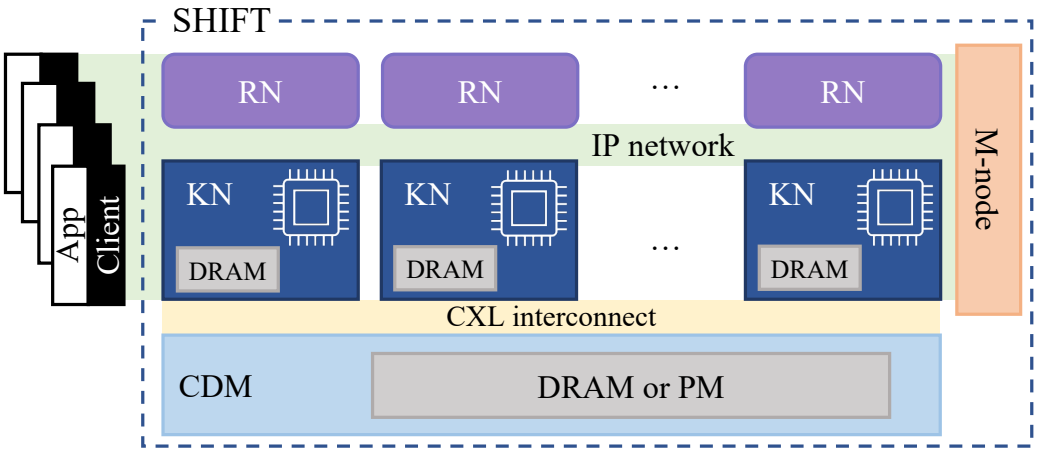


Figure 5.2: **Overview of the KVS cluster for Shift.**

work (§4), total partitioning across all the data, metadata, and ownership (shared nothing) can prevent elastic reconfiguration which is a critical requirement for dynamic resource provisioning [137].

5.2 Shift

We now present Shift, cache-conscious KVS design approaches for CDM. Shift aims at building an elastic, scalable, high-performance, and partial-failure resilient key-value store for CDM. Shift targets applications with dynamic working sets/sizes and non-uniform workloads with varying skew [156, 171, 204]. In this section, we first provide the overview of the Shift architecture. Then, we introduce new cache-conscious indexing, caching, and partitioning techniques in Shift.

5.2.1 Overall architecture

Figure 5.2 shows the high-level architecture of Shift. Shift's architecture comprises clients, routing nodes (RNs), KVS nodes (KNs), CDM, and a monitoring/management node (M-node). We reuse the existing design and implementation of clients, RNs, and M-node in our prior DPM KVS [116] for Shift. We describe these components and how a request flows between them.

Applications interact with SHIFT through clients. The routing tier (RNs) provides cluster membership to the clients and isolates them from the internal variation of the KVS cluster. A client first contacts an RN to obtain cluster membership. Then, the client sends get or put requests to the appropriate KN, which will then perform the requests on its behalf. Each KN is equipped with general-purpose processors and a small amount of memory relative to the CDM capacity. KNs can access CDM over the CXL interconnect using load and store instructions. CDM has the large CPU-less memory pool shared across KNs. CDM is the source of ground truth in SHIFT, storing data (key-value pairs) and metadata (indexing data structures). KNs cache data fetched from CDM in their local DRAM and respond to client requests. The M-node monitors KNs statuses and workload characteristics to detect failures or load variance at KNs, and triggers a suitable reconfiguration or recovery.

Assumptions. We assume that KN failures are fail-stop and independent of DPM failures; when an KN fails, the KN abruptly terminates its execution and its local DRAM contents are lost. Furthermore, we assume CDM and M-node are always available and alive. Lastly, we assume M-node detects the KN failures correctly without false-positive or false-negative detection.

5.2.2 Reusing PM indexes for CDM

The CPU-cache oblivious designs of the index structures for RDMA disaggregated memory make them unsuitable for CDM. The index structures for CDM must be designed in a CPU-cache conscious manner for high performance, scalability, and partial failure tolerance. To achieve these goals, we propose reusing existing PM indexes for CDM.

Overall intuition. We observe a number of existing PM index structures have been already designed to be CPU-cache efficient, concurrent, and crash consistent. Some of the mechanisms to tolerate partial KN failures such as online inconsistency detection/recovery and write ordering (*e.g.,* using cache-line flushes and fences) are

already equipped with many PM indexes themselves [82, 112]. With these indexes, writer threads (KNs in the context of CXL) have the ability to detect permanent inconsistencies and fix them in an online manner without blocking the progress of other reader threads (KNs). If we apply the PM indexes to CDM, they would be cache efficient and partial-failure-tolerable.

Challenge. The existing PM indexes, however, cannot be used as-is for CDM due to their distinct failure assumptions in design. The PM index structures are designed while assuming full-system-crash failure and rely on a specific post-crash procedure in an offline manner that releases all the permanent locks before resuming threads. The offline lock release after a crash can be appropriate for the full-system-crash failure model, where all resources fail together in the event of a crash and the timing of their resumption can be controlled.

This offline approach, however, is not suitable for partial-failure models such as in CXL settings, where some KNs can be partially crashed while the other KNs and shared CXL disaggregated memory are still operational. Simply applying the offline approach to the CXL settings can result in concurrency safety violation or largely need to compromise elasticity. For example, even if a KN crashes, the locks held by other alive KNs should not be released for correct mutual exclusion. Releasing all the locks at CXL disaggregated memory without considering which KNs held them can cause safety violation. Although we could consider blocking all KNs while the exhaustive lock release is underway, this would significantly reduce elasticity.

There are two challenges to release the permanent lock after a KN crash safely and elastically. First, we need to identify lock ownership to correctly distinguish between valid locks held by alive KNs and permanent ones that must be released due to a KN crash. Second, we need to efficiently narrow down locks to be checked for the permanent lock release in order to quickly identify which locks are permanent and minimize blocking time of other KNs due to the deadlock. We solve these challenges by using lock intention log.

Lock intention log. To correctly identify lock ownership across crashes, it is required that all the locks are stored in a shared CXL disaggregated memory, have the ownership information along with the locking status indicator, and can be changed (lock/unlock with the ownership) using atomic operations (*e.g.,* Compare-and-Swap). Furthermore, KNs maintain a per-KN log, *lock intention log*, at CXL disaggregated memory to quickly identify a set of locks to be checked and released after a KN crash. Prior to acquiring a lock, a KN first declares its intent of attempting the lock acquisition by recording the lock address into the lock intention log. After that, KNs acquire a lock while recording their unique id (ownership) to that using atomic operations. This mechanism ensures that following a KN crash, the per-KN lock intention logs collectively indicate all locks that might be locked, and the recorded ownership in the locks allow to identify which KN actually holds each lock.

Recovery. The lock intention log coupled with the locks embedding the KN ownership enables fast permanent lock release. When a KN crashes, M-node detects the crash then sends a recovery request to one of alive KNs along with the unique id of the KN that crashes. The recovery KN that receives the recovery request checks the locks recorded in the lock intention log assigned to the crashing KN and compares the ownership recorded inside each lock with the unique KN id. If they are matched, the recovery KN releases the permanent locks by the crashing KN, otherwise skipping. After unlocking all the permanent locks, other alive KNs can detect and fix any inconsistency caused by the failed KN using the online inconsistency detection and recovery mechanisms provided by the PM indexes.

5.2.3 Non-hierarchical processing

Caching data into KN-side local memory (KN cache) does not always result in better performance than that of direct accesses to CXL disaggregated memory (CXL direct). The performance trend is not even deterministic but can vary depending on various factors such as KN cache miss penalty, CPU cache efficiency of each layer's

Algorithm 1 Non-hierarchical processing pseudocode

Require: N > 0
Ensure: 1 ≤ CacheRatio ≤ 99

```
RequestCount = 0
EpochInterval = N
CacheRatio = 50
CacheLatency = 0
DirectLatency = 0
CacheAccessCount = 0
DirectAccessCount = 0
while recvReq() == true do
    if (RequestCount % 100) < CacheRatio then
        ret = Cache.Get(key)
        if ret == Hit then
            CacheLatency += HitLatency
            CacheAccessCount += 1
        else if ret == Miss then
            val = Direct.Get(key)
            Cache.Update(key, val)
            CacheLatency += MissPenalty
        end if
    else if (RequestCount % 100) ≥ CacheRatio then
        Direct.Get(key)
        DirectLatency += GetLatency
        DirectAccessCount += 1
    end if
    if (RequestCount % EpochInterval) == 0 then
        AvgCacheLatency = CacheLatency / CacheAccessCount
        AvgDirectLatency = DirectLatency / DirectAccessCount
        if AvgCacheLatency ≤ AvgDirectLatency then
            CacheRatio += (AvgDirectLatency/AvgCacheLatency)
        else if AvgCacheLatency > AvgDirectLatency then
            CacheRatio -= (AvgCacheLatency/AvgDirectLatency)
        end if
        CacheLatency = 0, DirectLatency = 0
        CacheAccessCount = 0, DirectAccessCount = 0
    end if
end while
```

implementations (*e.g.,* the types of indexes at CXL disaggregated memory), and the level of the loads, as we observed in Section 5.1.2. Even worse, a local trend at a KN may not be global across other multiple KNs. Hardware specs and workload patterns that each host has are likely to be different in the CXL environment that allows heterogenous device setups. Such heterogeneity can result in different performance trends between KN cache and CXL direct in each host. To address these challenge, we present Non-Hierarchical Processing (NHP) to achieve the best-case performance in the presence of those variables.

NHP strikes the right balance between using KN cache and CXL direct to process requests. We employ in-situ A/B testing by monitoring the request processing latency of each layer to dynamically balance the processing ratio between KN cache and CXL direct. For example, we increase the cache processing ratio when the measured latency from KN cache is lower than that of CXL direct. We opt for the latency-based metric because it is not only simple to measure but also a robust single metric that can effectively incorporate all the performance factors into our policy. NHP policies are applied to each KN individually to account for the unique trends of each KN; the processing ratio is adjusted only locally using latency information measured within a KN. Note that NHP can be universally applied for any caching algorithms at KN cache and index type at CDM, irrespective of specific implementations.

Algorithm 1 shows the pseudocode of the NHP policy. NHP initializes the processing ratio of each layer equally at first (Algorithm 1, line 3). The get requests are distributed to either KN cache or CXL direct by following the ratio (Algorithm 1, line $9 - 21$). After processing requests, NHP adds up each measured latency and access counter to keep track of the overheads from each layer during the epoch interval (Algorithm 1, line $11 - 12$, $19 - 20$). When a cache miss happens, NHP adds the cache miss penalty that is the wasted overheads to search and update cache. At the end of each epoch (default epoch size: 100), the processing ratio become adjusted by comparing the average cache and direct latency collected during the epoch.

5.2.4 Reusing ownership partitioning for CDM

When multiple KNs share the same CXL memory addresses, coherence overheads across CPU caches (hardware-managed coherence) or KN-side local memory caches (software-managed coherence) at KNs can easily become a scalability bottleneck in increasing the number of KNs. Simply applying traditional partitioning approaches can end up largely reducing elasticity. To solve these issues, we propose reusing ownership partitioning for scalable, elastic CXL memory sharing.

Ownership partitioning [116] can be reused for CDM sharing to reduce the coherence overheads and to ensure elastic reconfiguration at KNs. By partitioning ownership across KNs, we can avoid the coherence traffic for the accesses both to the data in CDM and their copies in KN local memories. However, as ownership partitioning shares metadata across KNs for the elastic reconfiguration, the accesses and modifications to the shared metadata can incur the coherence traffic. Even if the existing KVSs for RDMA disaggregated memory sidestep this issue by simply not caching metadata structures [116, 198, 228], the shared metadata accessed via loads and stores over CXL can be automatically cached into CPU caches at KNs by the hardware-managed coherence protocol.

The level of the cache-coherence overheads from sharing the metadata can be different depending on workloads; in write-heavy workloads, the coherence traffic across KNs can be substantial due to the frequent metadata changes, but it can be minimal in read-dominant workloads. Although most KVS workloads are read-dominant [12], a recent study analyzing modern KVS clusters reports that write-heavy is also common [212]. Therefore, the shared metadata (*e.g.,* indexes) must be carefully designed to minimize the cache-coherence overheads.

To alleviate this issue, we extensively apply an existing NUMA-aware lock, optimistic locking, to the shared metadata as an optimization for ownership partitioning. In the studies about NUMA-aware locks [22], optimistic locking has shown superior performance than pessimistic locking (*e.g.,* reader-writer locks) due to its

minimal coherence overheads. To be specific, in pessimistic locking, both readers and writers change shared lock variables to specify the lock status. The frequent change to these shared variables causes a large cross-node (cross-KN in CXL settings) coherence traffic that can become a scalability bottleneck. On the other hand, in optimistic locking, writers change the shared lock variable, but readers do not; readers only check the shared lock status to identify if another writer was concurrent. Therefore, readers do not need to take cross-node coherence traffic in common cases. Although it is originally design for NUMA architectures, it is also useful to reduce cross-KN coherence traffic for the shared CXL disaggregated memory.

5.3 Implementation

We implement SHIFT in 10K lines of C++ code. We use the standard C++ library and several open-source libraries including ZeroMQ [169], Google Protocol Buffers [24], the NVMM library [140].

CXL disaggregated memory emulation. As there are no commercially available CXL prototypes, we emulate multi-host shared CXL disaggregated memory using docker containers [60] and remote NUMA node. Each docker container instance is emulated as a KN and implemented to use KN local resources (CPU, memory) only from local NUMA node. We use Kubernetes [71] to monitor the host container instances and to simulate dynamic CXL-host addition/removal scenarios.

Porting PM indexes for CXL disaggregated memory. The original PM indexes rely on the PMDK library [103] for PM allocations. However, the PMDK library has many restrictions in directly applying it for our (emulated) CDM due to the lack of safe memory allocation across multiple processes/containers (multiple KNs in CXL) and partial-failure tolerance mechanisms such as online garbage collection and online recovery. Thus, we replace the original PMDK library with the NVMM library [140].

NVMM is another PM allocator built on the PMDK library that supports

safe multi-process/container/node allocations, online garbage collection, and online recovery. NVMM is specifically designed to work in cache-incoherent multi-node PM environment like The Machine [97], but it also provides an option to enable cache-coherent multi-NUMA node environments. We modify four different PM indexes (P-CLHT, P-Masstree, P-ART, P-HOT) in the RECIPE PM index library [175] to use the NVMM allocator. SHIFT provides these PM indexes as the available main KV indexing structure in CDM, so that application developers can optionally choose them depending on their purpose.

Lock intention log. We implement per-KN lock intention log using a 1KB simple 8-byte record array. Each KN declares the lock acquisition intention by recording the lock address to the 8-byte record in the log array. All the log writes recording the lock intention are performed by non-temporal stores followed by memory fences for crash consistency. After each index operation finishes, the log records become reclaimed before starting a new operation. We call flushes (*e.g.,* `clwb`) to the lock addresses stored in the log records immediately before reclaiming the log records to ensure that the final unlock statuses are consistently reflected in CDM after the reclamation.

When the log array becomes full, another log array can be added by changing the next pointer in the prior log array. However, it is very rare that an additional log array becomes required since the number of required lock acquisitions per operation is bounded by an index traversal depth (*e.g.,* tree depth) which is mostly much smaller than the maximum log entries per array (with 1KB log array, the maximum is 128). The root pointers to each per-KN lock intention log are stored to the known address in CXL disaggregated memory so that the recovery KN can find the right intention log associated with a crashing KN.

KN cache We employ MICA [127] cache as an example caching system for KN-side local memory, which is the state-of-the-art in-memory cache implementation in terms of performance and scalability [213]. MICA cache employs a lossy hash table index, which has a CPU-cache friendly design. It evicts least-recently-used items when its

log-structured memory allocator becomes full or a colliding item in the hash table when a hash collision occurs for high throughput. We chose MICA cache due to its superior performance and scalability. Note that SHIFT does not specifically rely on MICA. It can be freely replaced by other caching implementations, depending on the purpose that requires more functions than what MICA provides like Time-to-Live [213].

5.4 Evaluation

We evaluate the performance of SHIFT and study the breakdown of the benefits from concurrent PM indexes, lock intention log, non-hierarchical processing. We design our experiments to answer the following questions:

- How does PM indexes compare to the state-of-the-art indexes designed for RDMA disaggregated memory in terms of performance and scalability?
- How much does the lock intention log impact the overall throughput of the PM indexes?
- How does NHP fare against other static policies (KN cache, CXL direct) in terms of performance and scalability?

Experimental setup. We use a two-socket 128-core machine with 512MiB LLC (AMD EPYC 7763 64-Core Processor) and 256GB DRAM (128GB per socket) to perform the experiments. We use one of NUMA nodes (node 0) as a KN and emulate another NUMA node (node 1) as CXL disaggregated memory. The estimated latencies in accessing local memory and remote memory (from node0 to node1) are 98ns and 293ns respectively, which is very close to CXL disaggregated memory latency ($170ns - 300ns$). The bandwidth in access the local and remote NUMA memory is measured to 147GB/sec and 80GB/sec respectively.

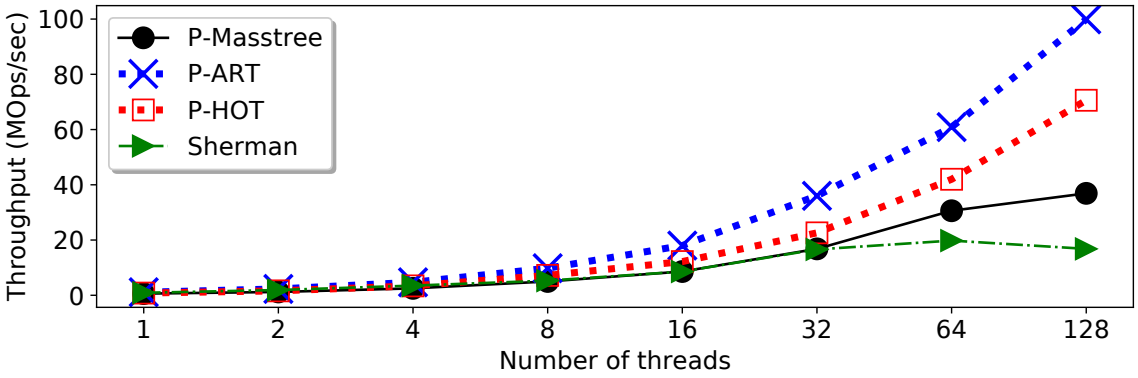


Figure 5.3: **Performance/scalability comparison of PM and RDMA indexes.**

We use 64 million requests with 8-byte keys and 256-byte values for workloads that run on a 16 GB key-value dataset. We measure the performance of the workloads while changing the number of threads at the local NUMA socket up to 128 in a single KN. We further collect low-level performance counters such as the number of LLC and DTLB misses per operation using the `perf` tool [164].

5.4.1 Performance & scalability comparison to RDMA indexes

We use Sherman [198], a state-of-the-art B+tree designed for RDMA disaggregated memory, as a baseline to compare the performance of our PM indexes. Sherman redesigns the original B+tree to optimize it for one-sided RDMA operations by tailoring the B+tree layout and algorithm to use unsorted keys, a dual versioning mechanism, and hierarchical locks. Because the original implementation of Sherman uses one-sided RDMA operations, we modified it to use load/store instructions to run on our emulated CXL while retaining the original data structure layout and algorithm. We compare Sherman to three different PM tree indexes (P-Masstree, P-ART, P-HOT) [112] using a read-only workload with uniform distribution. P-Masstree is a cache-efficient, hybrid tree structure of trie and B+tree. P-ART is a radix tree variant optimized for space consumption and scalable performance. P-HOT is a lookup and space-optimized variant of a trie.

	P-Masstree	P-ART	P-HOT	Sherman
Average LLC misses per op	49	15	29	72
Average DTLB misses per op	10	3	4	3

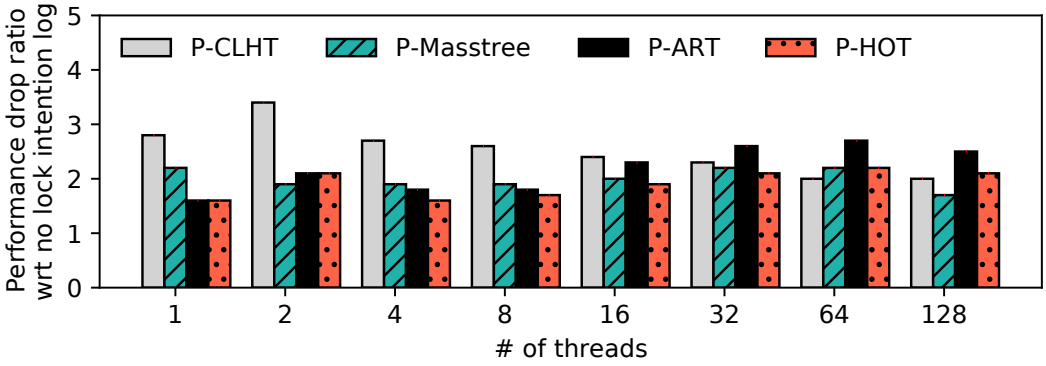
Table 5.2: **Performance profiling of PM and RDMA indexes.** The table shows the average number of last-level cache and data TLB misses per operation of each index. The cache-oblivious, I/O-oriented designs of Sherman cause much higher hardware cache misses than PM indexes.

Figure 5.3 shows the performance and scalability results of PM and RDMA indexes. All the PM indexes scales to 128 threads, but Sherman does not scale after 64 threads. With 128 threads, P-ART, P-HOT, and P-Masstree perform 6×, 4×, and 2× better than Sherman respectively. As shown in Table 5.2, Sherman has much higher average LLC misses per operation, resulting in lower performance and scalability than the PM indexes.

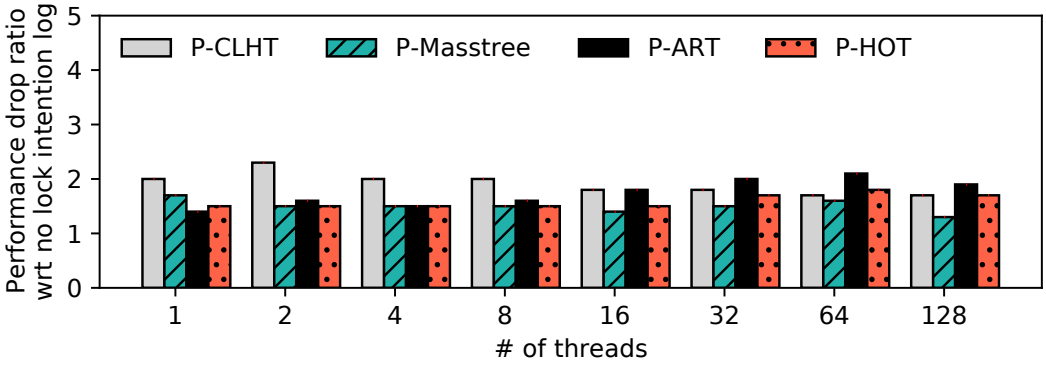
On the other hand, we observe Sherman performs 1.5× better than P-Masstree with the smaller number of threads ($1-4$ threads) due to lower average DTLB misses per operation (Table 5.2). Sherman uses the large node size ($1KB-4KB$) to minimize the number of one-sided RDMA operations by reducing the overall tree depth, and it contributes to lower DTLB misses per operation than P-Masstree. However, this result implies that the cache-oblivious, I/O-oriented index designs can perform properly with low concurrent environments, but make hard to achieve scalability on cache-coherent memory devices like CXL disaggregated memory.

5.4.2 Performance tradeoff of lock intention log

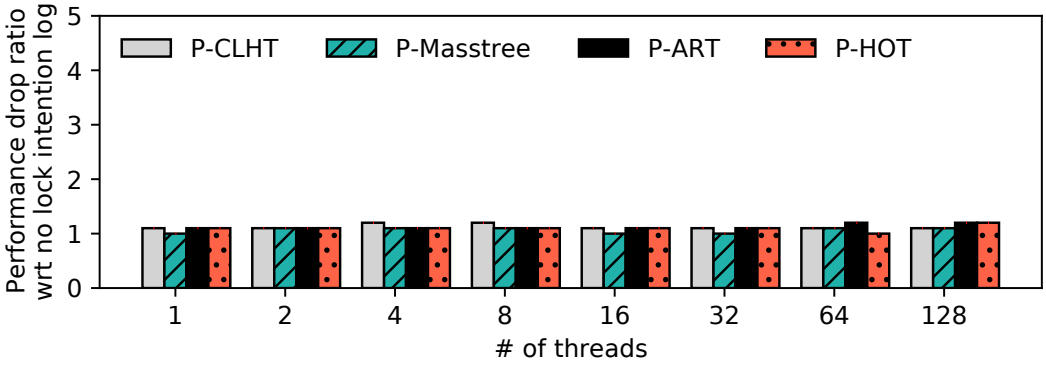
Lock intention log requires KNs to record their lock acquisition intent into per-KN log arrays whenever accessing an index object. This additional overhead can impact the performance of PM indexes, and it can vary depending on the index types. Furthermore, without GPF, KNs should manually flush the recorded log entries and final lock status to CDM for crash consistency; this adds more overheads. To study



(a) Update 100%

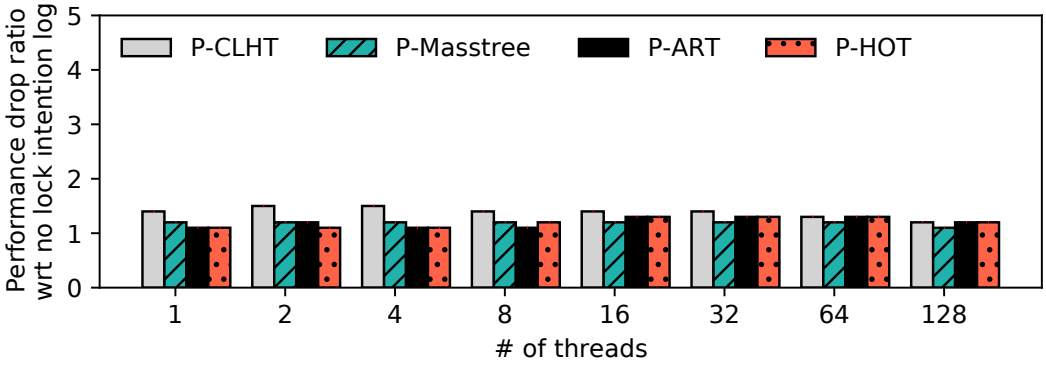


(b) Update 50%

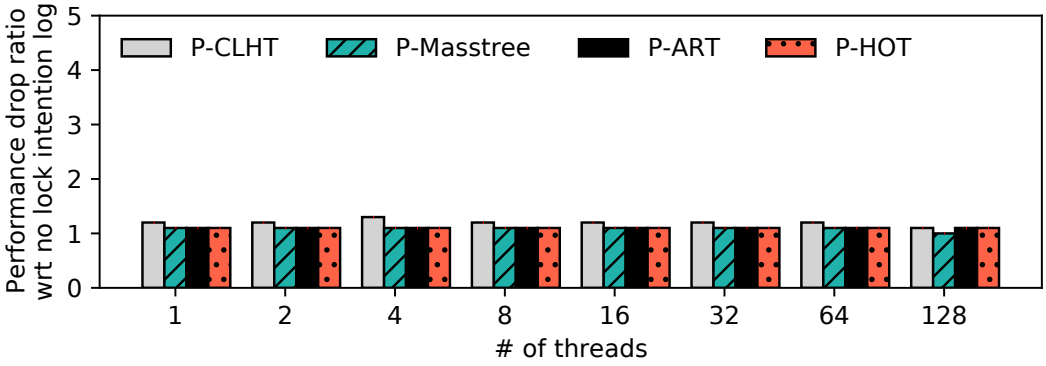


(c) Update 5%

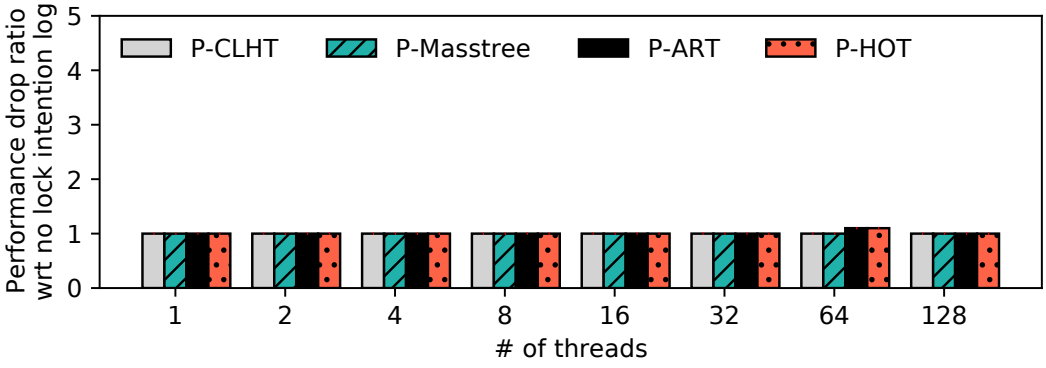
Figure 5.4: **Performance impact by lock intention log without GPF.** These graphs show the performance drops ratio relative to the cases without lock intention log when GPF is disabled. The performance drops due to lock intention log were up to 3.4×, 2.3×, 1.2× in update 100%, 50%, and 5% respectively.



(a) Update 100%



(b) Update 50%



(c) Update 5%

Figure 5.5: **Performance impact by lock intention log with GPF.** These graphs show the performance drops ratio relative to the cases without lock intention log when GPF is enabled. The drop ratios are up to 1.5×, 1.3× in update 100%, 50% respectively, but negligible in update 5%.

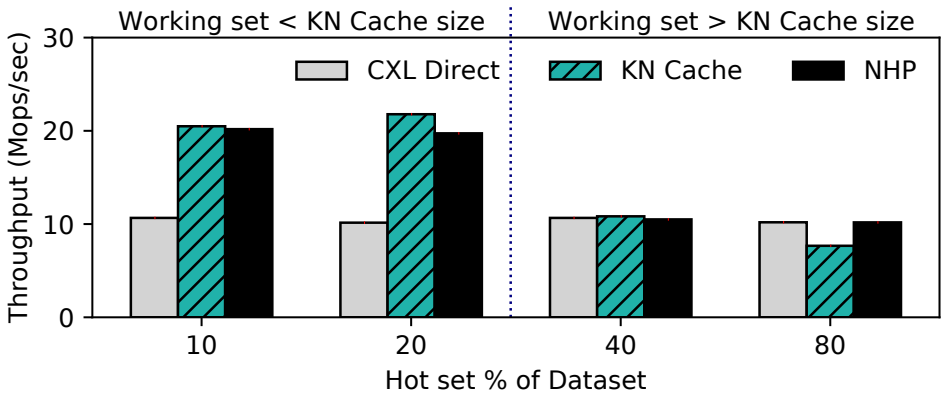
these impacts, we use the four different types of PM indexes, P-CLHT (Hashtable), P-Masstree (Trie and B+tree hybrid), P-ART (Radix tree), P-HOT (Trie). We also measure their performance by dividing the groups into those with GPF and without GPF, using uniformly-distributed workloads with different update ratios. We replace all the flushes and non-temporal stores with corresponding temporal instructions to simulate GPF.

Figure 5.4 shows the performance drop ratio relative to the one without lock intention log, when GPF is disabled. In this configuration, the performance drops due to lock intention log are observed by up to 3.4×, 2.3×, 1.2× in the workloads with update 100%, 50%, and 5% respectively. Indexes with higher original performance show a larger performance drop. And, the impact is reduced as the read ratio increases; it is because the read operations in the PM indexes we employ are lock-free or use optimistic locking, thus do not involve lock intention log. On the other hand, the performance drops become much reduced when GPF is enabled (shown in Figure 5.5); the drop ratios are up to 1.5×, 1.3× in the workloads with update 100%, 50% respectively, but negligible in update 5%.

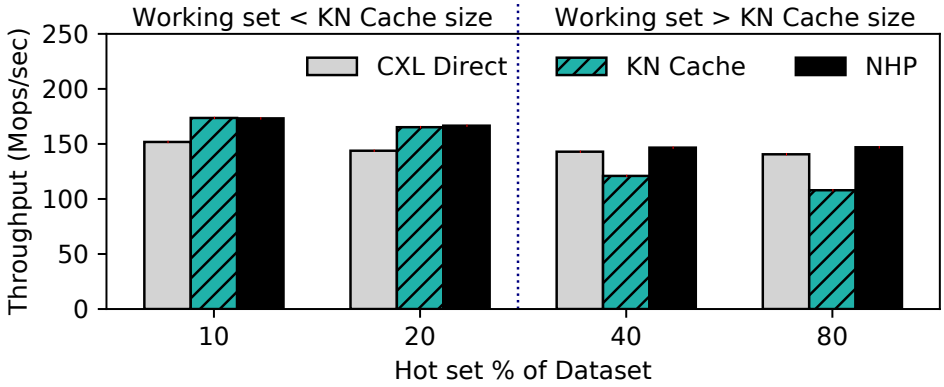
5.4.3 Performance of non-hierarchical processing

We compare NHP with two static counterparts, hierarchical caching (KN cache) and direct CXL access (CXL direct) strategies, using four different PM index structures at CDM. The hierarchical caching (KN cache) is a traditional caching strategy that always first checks the cache in KN-side local memory and then does the index in CDM if cache misses happen. The direct CXL access (CXL direct) checks the index in CDM directly while bypassing KN cache. For this evaluation, the size of KN cache is configured to 4GB over the 16GB key-value dataset at CDM. We use read-only workloads with different request skews (hot set percentages[1]) and evaluate the throughput of KN cache and CXL direct under two distinct request loads to

[1]The hot set percentage is the proportion of the dataset that is accessed by every operation.



(a) Low load (8 KN threads)



(b) High load (128 KN threads)

		Low load				High load			
Hot set % of Dataset		10	20	40	80	10	20	40	80
KN Cache hit rate (%)		99.5	99.5	57	10	99.5	99.5	57	10
KN Cache miss penalty (% consumed time)		0.4	0.4	18	27	0.3	0.3	19	30
CPU cache misses per op	CXL Direct	9	9	9	9	8	8	8	8
	KN Cache	13	12	16	14	11	10	15	13
	NHP	13	13	12	10	11	11	10	9

(c) Average LLC misses per op

Figure 5.6: **Comparison between NHP and static policies on P-CLHT.** The plots demonstrate that NHP can match the performance of the best policies for all hot set percentages under different request loads.

demonstrate the NHP's adaptivity to diverse scenarios.

Figure 5.6 shows the performance comparison between NHP and other static policies when P-CLHT is used as a main index in CDM. In both low load with 8 KN threads (Figure 5.6 a) and high load with 128 KN threads (Figure 5.6 b), NHP matches the performance of the best policies in each hot set percentage almost perfectly, showing error rates less than 9%. Interestingly, we also observe the average LLC misses per operation does not exactly follow the performance trends of NHP (shown in Figure 5.6 c). However, this mismatch rather demonstrates that NHP is able to balance the processing loads across two layers while reflecting other factors into its policy decision, based on the simple latency metric. On the other hand, we confirm NHP also shows the similar adaptability regardless of the types of indexes in CDM, shown in Figure 5.7, Figure 5.8, Figure 5.9.

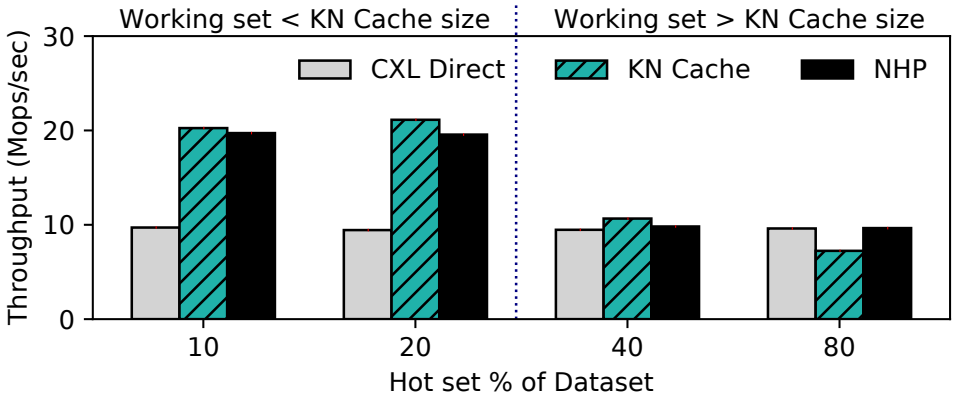
5.5 Limitations and Discussion

Shift has a number of limitations. First, while we address challenges of scaling KNs and tolerating their partial failures, we have not explored how to make CXL disaggregated memory failure-resilient and scalable. Second, we assume the system precisely detects KN failures without any false-positive or false-negative detection. The incorrect failure detection can have severe consequences in both system performance and correctness. However, assessing the wider ramifications of the inaccurate failure detection is not within the purview of this study. Third, the techniques proposed in Shift are derived from key-value store designs for CXL disaggregated memory. They may be applicable to other system domains, but we have not explored their applicability in this work. Fourth, CXL disaggregated memory supports various scales from a single local machine setup to a rack-scale deployment. Performance specifications like the latency and bandwidth to disaggregated memory can vary depending on these scales, but we have not evaluated our system with such a variety of specifications. Fifth, we focus on how to correctly reuse PM indexes on

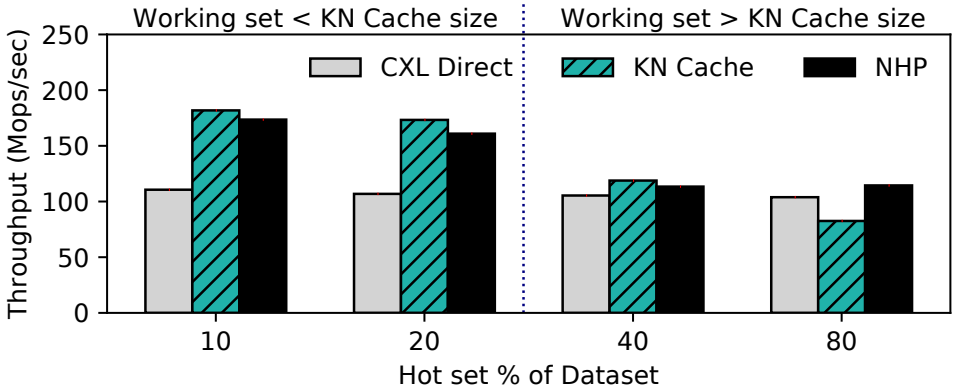
CXL disaggregated memory. However, we have not explored how to further optimize the PM indexes for CXL disaggregated memory specifically, given its unique latency and bandwidth characteristics. Finally, in recent years, various caching policies have emerged, such as caching shortcut pointers instead of data copies, utilizing the byte-addressable characteristics of PM and disaggregated memory [116, 190]. However, we have not explored these caching policies in our study for our non-hierarchical processing. We consider these as attractive areas of future work.

5.6 Summary

This chapter presents Shift, a novel cache-concious key-value store for CXL disaggregated memory. Shift aims at building an elastic, scalable, high-performance, and partial-failure-recoverable KVS for CXL disaggregated memory. Motivated by the limitations of existing solutions for RDMA disaggregated memory, Shift introduces the new indexing, caching, and partitioning approaches to achieve these goals. Shift propose reusing existing PM indexes and ownership partitioning to achieve CPU-cache efficiency, concurrency, and elasticity for CXL disaggregated memory. Shift further improves the PM indexes to be partial-failure-tolerable by using the lock intention log and alleviates cache-coherence traffic by comprehensively applying a NUMA-aware locks, optimistic locking, for the shared metadata. Shift also introduces non-hierarchical processing that synthetically utilize both KN cache and CXL direct accesses to achieve better and more stable performance than the static counterparts. Through the extensive evaluation, we show our PM indexes perform up-to 6× better than Sherman, the state-of-the-art RDMA-based B+tree and show the performance tradeoff of using the lock intention log. Lastly, we show NHP can match the best performance among its static counterparts and even outperforms them in high request loads.



(a) Low load (8 KN threads)

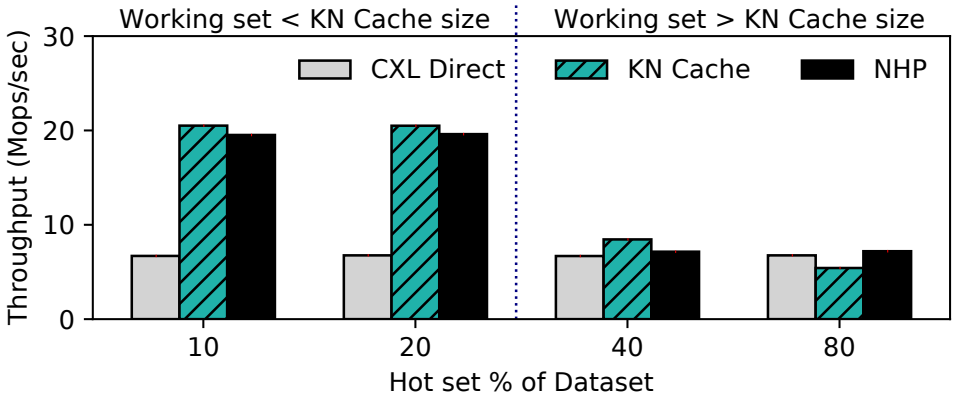


(b) High load (128 KN threads)

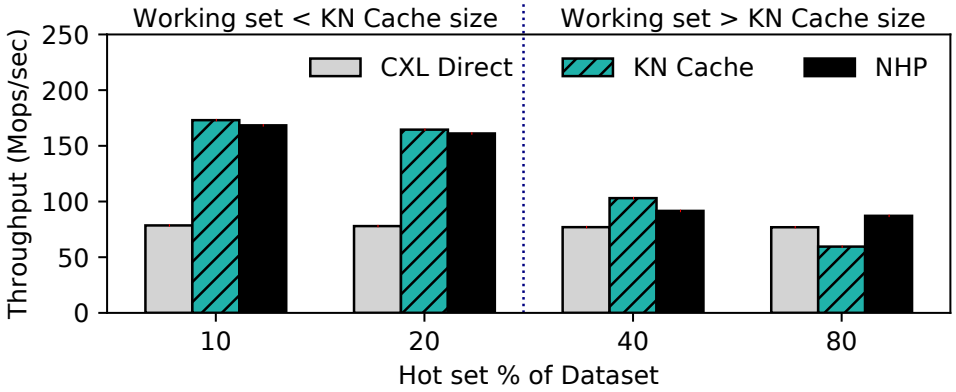
		Low load				High load			
Hot set % of Dataset		10	20	40	80	10	20	40	80
KN Cache hit rate (%)		99.5	99.5	57	10	99.5	99.5	57	10
KN Cache miss penalty (% consumed time)		0.5	0.5	19	27	0.3	0.3	20	24
CPU cache misses per op	CXL Direct	15	15	15	15	14	14	14	14
	KN Cache	13	12	19	22	11	11	18	21
	NHP	13	13	17	15	11	11	16	14

(c) Average LLC misses per op

Figure 5.7: **Comparison between NHP and static policies on P-ART.**



(a) Low load (8 KN threads)

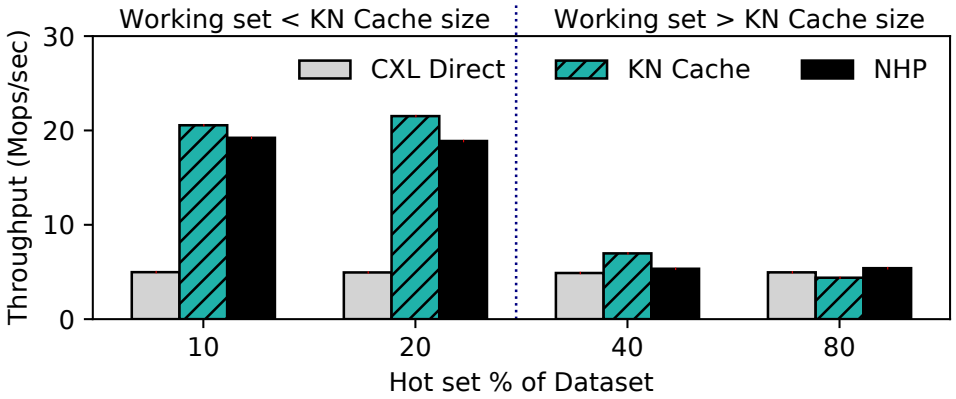


(b) High load (128 KN threads)

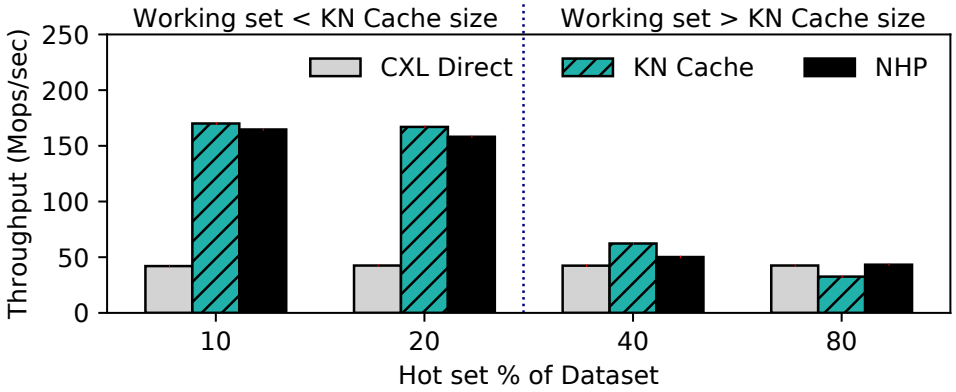
		Low load				High load			
Hot set % of Dataset		10	20	40	80	10	20	40	80
KN Cache hit rate (%)		99.5	99.5	57	10	99.5	99.5	57	10
KN Cache miss penalty (% consumed time)		0.6	0.6	16	22	0.3	0.3	16	19
CPU cache misses per op	CXL Direct	30	30	30	30	26	26	26	26
	KN Cache	14	13	27	38	12	11	24	33
	NHP	14	14	29	28	12	13	26	25

(c) Average LLC misses per op

Figure 5.8: **Comparison between NHP and static policies on P-HOT.**



(a) Low load (8 KN threads)



(b) High load (128 KN threads)

		Low load				High load			
Hot set % of Dataset		10	20	40	80	10	20	40	80
KN Cache hit rate (%)		99.5	99.5	57	10	99.5	99.5	57	10
KN Cache miss penalty (% consumed time)		0.4	0.4	14	17	0.3	0.3	11	11
CPU cache misses per op	CXL Direct	50	50	50	50	46	46	46	46
	KN Cache	13	12	35	56	11	11	32	52
	NHP	14	14	42	44	12	13	38	42

(c) Average LLC misses per op

Figure 5.9: **Comparison between NHP and static policies on P-Masstree.**

Chapter 6: Related Work

In this chapter, we place our contributions of this dissertation in the context of relevant prior work. First, we discuss the prior effort made in guaranteeing crash consistency for PM. Then, we discuss the prior studies about partitioning and caching techniques for data-intensive systems. Finally, we discuss the design techniques proposed in the past for cache-coherent memory devices.

6.1 Crash consistency for PM

Isolation and Crash Recovery. Memory Persistency [163] makes the connection between crash recovery and the semantics of memory consistency by introducing the concept of Recovery Observer. Durable Linearizability [88] and Recoverable Linearizability [17] theoretically define the relationship between crash recovery and non-blocking synchronization. However, these works only propose model semantics, without connecting the findings to practical index structures.

TSP [150] proposes the broad insight that non-blocking indexes can be converted into crash-consistent counterparts by coupling Recovery Observer and Flush-on-Failure. However, Flush-on-Failure technique requires additional hardware support like the backup power supply and kernel modifications. RECIPE exploits and extends these broad observations to build concurrent, crash-consistent PM indexes without any hardware support and kernel changes. While TSP assumes non-blocking writes, RECIPE relaxes the assumption, allowing the commonly-used write exclusion.

Concurrent, crash-consistent PM Indexes. In the past five years, 15 PM indexes have been proposed, out of which only three have open source, concurrent implementations: FAST & FAIR [82], CCEH [149], Level Hashing [37, 227], and Dash [133]. RECIPE is complementary to these efforts in building a concurrent PM index. While these studies design the new indexes for PM from scratch, RECIPE takes a more

principled approach by reusing decades of research in building concurrent in-memory indexes with no modifications to the underlying design of the DRAM index.

Transactional PM Systems. Previous work like Atlas [34], JUSTDO [87], iDO [130], and NVThreads [81], persist data at boundaries of critical sections called Failure Atomic SEctions (FASE). They automatically inject logging for every persistent update [34, 81] or program states [87, 130] within FASE by using compile-time analysis. However, their approaches amplify the overhead of cache line flushes, as they require an additional persistent log. These systems also pay a startup cost to replay the log during recovery, which could be significant for large indexes. However, RECIPE-converted indexes do not employ additional logging mechanisms and pay no startup recovery cost when the index restarts after a crash.

Crash-Consistency Testing for PM Applications. PM application testing frameworks such as Yat [110], Intel PM-Inspector [85], and pmreorder [166] aim at enabling correctness testing and debugging for applications built for PM. However, these tools use either random or exhaustive techniques to construct crash states, which does not scale as the number of writes to the PM increase [85, 110, 166]. Our crash testing strategy used for RECIPE, on the other hand, exploits the fact that operations in PM indexes are comprised of a small set of atomic steps, thereby simulating crashes only after these atomic steps. This technique makes our approach efficient and powerful enough to reveal bugs within a few crash states. PMTest [131] requires that developers manually annotate their source code with assert-like statements to find errors [131]. However, our approach requires lower effort from developers, since changes are localized to the write path.

6.2 Partitioning and caching for data-intensive systems

DPM architectures. OP (Ownership Partitioning) follows the idea that just because you *can* share, it does not mean you *should* share. This observation has been

made before in other contexts. Storage Area Networks provide storage disaggregation in a data center [15], where volumes could be shared among hosts, but often they are not [31]. Key-value stores provide storage disaggregation in the cloud, where data can be shared among nodes, but applications may choose not to [197]. Fine-grained logical partitioning has been proposed to support live reconfigurations in in-memory key-value stores [2, 106], in-memory databases [57], and graph processing [208]. Even multiprocessor shared-memory systems sometimes forgo sharing of data structures among threads, choosing instead to partition data [16, 25, 127]. Our work demonstrates that partitioning logical ownership while sharing physical data and metadata in DPM provides high performance and lightweight reconfigurability.

Distributed transactions for partitioned KVSs. Many distributed key-value stores sacrifice transaction support to create a linearly scalable distributed data storage tailored for partitionable applications [7, 30, 40, 51]. NewSQL key-value stores seek to provide ACID transactions alongside non-transactional key-value stores while maintaining their horizontal scalability [122, 188, 221, 224]. A common design principle of these systems is to decouple transaction management components (*e.g.,* transaction logging, commit management, concurrency control) from data storage components, enabling independent scaling of each component [142].

Ownership Partitioning (OP) is complementary to these studies; OP focuses on enhancing the performance, scalability, and elasticity of data storage components, while NewSQL KVS research focuses on scalable transaction support independently of the data storage components. We believe that OP can be used in conjunction with these studies to achieve the transaction support for Dinomo. However, the shared metadata in OP can introduce more contention overheads between transactions than shared-nothing counterparts. Therefore, further work may be needed to optimize transaction processing performance and scalability by minimizing the contention.

Adaptive caching policy. Adaptive caching policies have been explored in other contexts, illustrating how a single cache can be used for multiple purposes or how a

replacement policy can consider multiple behaviors. For example, the Sprite operating system shared its memory between the file system buffer cache and the virtual memory system [153]. The Adaptive Replacement Cache (ARC) uses a replacement policy that balances between recency and frequency of accesses [145]. In contrast to these systems, which use fixed-size cache entries with uniform miss penalties, DAC (Disaggregated Adaptive Caching) manages a cache where different types of entries (e.g., values vs. shortcuts) have different sizes and varying miss penalties. The novelty of our scheme arises from a new setting (DPM) where adaptivity is essential.

6.3 Design techniques for cache-coherent memory devices

Caching for PM. Similar observations have also been made in systems for PM, another low-latency, byte-addressable media in the cache-coherent hierarchy, that hierarchical caching does not consistently outperform its bypassing counterpart [170]. The drawbacks of having a DRAM caching layer for the fast PM include additional cache management and data copy (between DRAM and PM) overheads, unnecessarily overwhelming CPU caches and outweighing the benefits of direct PM accesses [107]. For example, many PM-aware file systems statically bypass page cache to avoid these overheads [43, 55, 209]. In contrast to these systems, our Non-Hierarchical Processing (NHP) for CDM dynamically adapts the processing ratio between KN caching and direct CDM accesses to achieve performance comparable to the best static policies across diverse scenarios.

NUMA-aware designs. The hardware guaranteed cache coherence in CXL settings brings the benefits of simplified software stacks and programming. However, it comes at a cost; without careful consideration, the cache-coherence traffic can easily become a scalability bottleneck. Conventional systems for NUMA architectures have the similar challenges to reduce the cache-coherence overheads for high performance and scalability. Various solutions to reduce the coherence traffic between different NUMA nodes have been proposed. The core design principle in NUMA-aware designs is to

preserve locality where processes and their memory are kept together on the same NUMA node to reduce contentions (including the cache-coherence traffic, memory controller, and interconnect) and remote access overheads. To ensure this locality, various locality-aware designs have been proposed such as partitioning [127, 138, 147], replication [27, 144], migration [13, 20, 121], and delegation [26, 138, 147, 220].

CXL disaggregated memory, however, is CPU-less without having local computing units different from NUMA architecture. Thus, the locality assumption (pages accessed by a CPU core need to be in its local NUMA node, or vise versa) as well as the designs originated by it do not work for CXL architectures.

Systems for CDM. With the advent of CXL technologies, research on building systems for CDM has been gathering traction recently. These studies aim at building a general-purpose memory management system that can transparently enable CDM for various applications. Pond [125] utilizes CDM as a zero-core virtual NUMA (zNUMA) node to handle stranded memory spaces allocated by cloud virtual machines. It employs a ML-based prediction model to determine how much CDM pool memory to allocate for a VM to while minimizing performance degradation. TPP [143] proposes a new page placement mechanism to utilize CDM as a slower memory tier. It leverages techniques in existing NUMA Balancing and Linux's LRU-based age management mechanisms for page temperature detection and offloads cold pages to CDM. CXL-SHM [218] proposes a partial-failure-resilient memory allocator for CDM using a non-blocking era-based algorithm based on reference counting.

In the contrast to them, Shift focuses on design approaches to achieve high performance, scalability, elasticity, and partial-failure tolerence for a specific system application, a key-value store. Shift employs the NVMM allocator [140] to achieve partial-failure-resilient memory allocations for CDM like CXL-SHM.

Chapter 7: Conclusion

Cloud-based KVSs must evolve to meet rising demands for performance, scalability, elasticity, utilization, and crash resilience. Transitioning to emerging memory and disaggregation technologies like PM, RDMA, and CXL offers promising solutions. However, these technologies require careful KVS design to optimize their benefits. This dissertation explores novel indexing, caching, and partitioning techniques to achieve a high-performance, scalable, elastic, and crash-recoverable KVS for the emerging memory and disaggregation technologies.

7.1 Summary

In this dissertation, we first present RECIPE, a principled approach for converting concurrent DRAM indexes to crash-consistent indexes for PM. RECIPE is based on the following insight that isolation techniques in concurrent DRAM indexes can be translated to crash consistency with minimal modifications. We present a set of conditions that allow developers to identify this class of DRAM indexes and the corresponding actions to convert the target DRAM indexes to be persistent and crash-consistent. Next, we present DINOMO, the first key-value store for RDMA-based DPM that simultaneously achieves high common-case performance, scalability, and elasticity. DINOMO uses a novel combination of techniques such as ownership partitioning, disaggregated adaptive caching, selective replication, and lock-free and log-free indexing to achieve these goals. Finally, we propose SHIFT, a cache-conscious key-value store for CXL disaggregated memory that achieves high performance, scalability, elasticity, and partial-failure tolerance. SHIFT reuses the existing PM indexes and ownership partitioning to enable CPU-cache efficient, KN-scalable, and elastic KVS for CXL disaggregated memory. SHIFT further retrofits lock intention log to make the PM indexes tolerate partial KN failures and propose non-hierarchical pro-

cessing to take right balance between KN cache and CXL direct accesses.

7.2 Lessons learned

In this section, we present a list of general lessons we learned while working on this dissertation.

Effectively reusing existing techniques. Emerging technologies exhibit unique characteristics, and modern cloud platforms must fulfill a diverse range of application requirements. Designing a system from scratch to accommodate these demands is a challenging and time-consuming endeavor. Consequently, effectively reusing existing techniques is of paramount importance and often yields remarkable results. RECIPE exemplifies this approach by converting DRAM indexes into crash-consistent PM indexes with minimal modifications. The indexes generated by RECIPE outperform many other PM indexes designed from scratch. This achievement is a testament to the combined efforts of both the DRAM and PM research communities. A similar approach was employed in SHIFT to develop a partial-crash-resilient, scalable, and elastic KVS for CDM by reusing the existing PM indexing and DPM partitioning techniques.

Application-specific approaches. We have also observed that general-purpose approaches often fall short in addressing the specific needs of emerging technologies and platforms. Therefore, we advocate for tailored approaches. For instance, RECIPE utilizes index-algorithm-specific methods to ensure crash consistency in DRAM indexes, circumventing the need for generic, but expensive logging mechanisms. This strategy preserves the original performance benefits of DRAM indexes. Moreover, DINOMO customizes traditional partitioning techniques to optimize them for DPM.

Cyclical nature of technology. Finally, the rapid rise and fall of technologies is an undeniable reality. However, we must not be discouraged by such setbacks; instead, we should persevere in our research endeavors as long as our vision for the

technology remains valid. Despite the discontinuation of Intel Optane PM, we firmly believe that another PM will eventually emerge, capitalizing on its byte-addressable, high-performance, and durable attributes. Furthermore, lessons learned from one research domain can be applied to others. Therefore, we must remain vigilant, as new opportunities may arise where our expertise can make a significant contribution.

7.3 Closing words

The advent of new memory and disaggregation technologies, coupled with the diverse demands of modern cloud platforms, marks a pivotal moment in systems research. However, it is challenging to suitably integrate these technological advancements into existing platforms to meet their specific requirements. Without careful system designs that consider the unique characteristics of these new technologies, the inherent benefits they offer can easily be overlooked. This dissertation proposes solutions that meticulously account for the distinctive attributes of these new hardware technologies without compromising the diverse requirements of an efficient cloud-based KVS. It demonstrates the feasibility of developing a KVS for emerging memory and disaggregation technologies that simultaneously delivers high performance, scalability, elasticity, and crash recoverability.

Works Cited

[1] S.V. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66–76, 1996.

[2] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, Roberto Peon, Larry Kai, Alexander Shraer, Arif Merchant, and Kfir Lev-Ari. Slicer: Auto-sharding for datacenter applications. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, page 739–753, 2016. ISBN 9781931971331.

[3] Marcos K. Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. Designing far memory data structures: Think outside the box. In *Proceedings of the 17th Workshop on Hot Topics in Operating Systems*, page 120–126, 2019. ISBN 9781450367271.

[4] Ahmad Al-Shishtawy and Vladimir Vlassov. Elastman: Elasticity manager for elastic key-value stores in the cloud. In *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*, 2013. ISBN 9781450321723.

[5] Mohammad Alshboul, Prakash Ramrakhyani, William Wang, James Tuck, and Yan Solihin. Bbb: Simplifying persistent programming using battery-backed buffers. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 111–124, 2021.

[6] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[7] J. Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: The Definitive Guide Time to Relax*. O'Reilly Media, Inc., 1st edition, 2010. ISBN 0596155891.

[8] Thomas E. Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N. Schuh, and Emmett Witchel. Assise: Performance and availability via client-local nvm in a distributed file system. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pages 1011–1027, 2020. ISBN 978-1-939133-19-9.

[9] Sebastian Angel, Mihir Nanavati, and Siddhartha Sen. Disaggregation and the application. In *Proceedings of the 12th USENIX Workshop on Hot Topics in Cloud Computing*, jul 2020.

[10] Dmytro Apalkov, Alexey Khvalkovskiy, Steven Watts, Vladimir Nikitin, Xueti Tang, Daniel Lottis, Kiseok Moon, Xiao Luo, Eugene Chen, Adrian Ong, Alexander Driskill-Smith, and Mohamad Krounbi. Spin-transfer torque magnetic random access memory (stt-mram). *J. Emerg. Technol. Comput. Syst.*, 9(2), may 2013. ISSN 1550-4832.

[11] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. Bztree: A high-performance latch-free range index for non-volatile memory. *Proc. VLDB Endow.*, 11(5):553–565, jan 2018. ISSN 2150-8097.

[12] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, page 53–64, 2012. ISBN 9781450310970.

[13] Linux AutoNUMA. `https://lwn.net/Articles/849095/`, 2022. Accessed: 2022-12-28.

[14] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. Triad: Creating synergies between memory, disk and log in log structured key-value stores. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 363–375, 2017. ISBN 978-1-931971-38-6.

[15] Richard Barker and Paul Massiglia. *Storage Area Network Essentials: A Complete Guide to Understanding and Implementing SANs*. Wiley Publishing, 1st edition, 2001. ISBN 0471034452.

[16] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: A new os architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, page 29–44, 2009. ISBN 9781605587523.

[17] Ryan Berryhill, Wojciech Golab, and Mahesh Tripunitara. Robust shared objects for non-volatile main memory. In *19th International Conference on Principles of Distributed Systems (OPODIS 2015)*, volume 46, pages 1–17, 2016. ISBN 978-3-939897-98-9.

[18] Laurent Bindschaedler, Ashvin Goel, and Willy Zwaenepoel. Hailstorm: Disaggregated compute and storage for distributed lsm-based databases. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, page 301–316, 2020. ISBN 9781450371025.

[19] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. Hot: A height optimized trie index for main-memory database systems. In *Proceedings of the 2018 International Conference on Management of Data*, pages 521–534, 2018. ISBN 978-1-4503-4703-7.

[20] Sergey Blagodurov, Sergey Zhuravlev, Alexandra Fedorova, and Ali Kamali. A case for numa-aware contention management on multicore systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, page 557–558, 2010. ISBN 9781450301787.

[21] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. Using lightweight formal methods to validate a key-value storage node in amazon s3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, page 836–850, 2021. ISBN 9781450387095.

[22] Jan Böttcher, Viktor Leis, Jana Giceva, Thomas Neumann, and Alfons Kemper. Scalable and robust latches for database systems. In *Proceedings of the 16th International Workshop on Data Management on New Hardware*, 2020. ISBN 9781450380249.

[23] William Bridge, Ashok Joshi, M. Keihl, Tirthankar Lahiri, Juan Loaiza, and N. MacNaughton. The oracle universal server buffer. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, page 590–594, 1997. ISBN 1558604707.

[24] Protocol Buffers. `https://developers.google.com/protocol-buffers`, 2022. Accessed: 2022-02-16.

[25] Irina Calciu, Dave Dice, Tim Harris, Maurice Herlihy, Alex Kogan, Virendra Marathe, and Mark Moir. Message passing or shared memory: Evaluating the delegation abstraction for multicores. In *Proceedings of the 17th International Conference on Principles of Distributed Systems - Volume 8304*, page 83–97, 2013. ISBN 9783319038490.

[26] Irina Calciu, Justin E. Gottschlich, and Maurice Herlihy. Using elimination and delegation to implement a scalable numa-friendly stack. In *Proceedings of the 5th USENIX Conference on Hot Topics in Parallelism*, page 7, 2013.

[27] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. Black-box concurrent data structures for numa architectures. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, page 207–221, 2017. ISBN 9781450344654.

[28] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, Bo Wang, Yuhui Wang, Haiqing Sun, Ze Yang, Zhushi Cheng, Sen Chen, Jian Wu, Wei Hu, Jianwei Zhao, Yusong Gao, Songlu Cai, Yunyang Zhang, and Jiawang Tong. Polardb serverless: A cloud native database for disaggregated data centers. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*, page 2477–2489, 2021. ISBN 9781450383431.

[29] Amanda Carbonari and Ivan Beschasnikh. Tolerating faults in disaggregated datacenters. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, page 164–170, 2017. ISBN 9781450355698.

[30] Apache Cassandra. `https://cassandra.apache.org/`, 2022. Accessed: 2022-12-26.

[31] Adrian M. Caulfield and Steven Swanson. Quicksan: A storage area network for fast, distributed, solid state disks. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, page 464–474, 2013. ISBN 9781450320795.

[32] Keren Censor-Hillel, Erez Petrank, and Shahar Timnat. Help! In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 241–250, 2015. ISBN 978-1-4503-3617-8.

[33] Sang K. Cha, Sangyong Hwang, Kihong Kim, and Keunjoo Kwon. Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 181–190, 2001. ISBN 1-55860-804-4.

[34] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 433–452, 2014. ISBN 978-1-4503-2585-1.

[35] 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*, page 275–290, 2018. ISBN 9781450347037.

[36] Shimin Chen and Qin Jin. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment*, 8(7):786–797, February 2015. ISSN 2150-8097.

[37] 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*, pages 799–812, 2020. ISBN 978-1-939133-14-4.

[38] Yue Cheng, Ali Anwar, and Xuejing Duan. Analyzing alibaba's co-located datacenter workloads. In *Proceedings of the 2018 IEEE International Conference on Big Data*, pages 292–297, 2018.

[39] Ping Chi, Wang-Chien Lee, and Yuan Xie. Making b+-tree efficient in pcm-based main memory. In *Proceedings of the 2014 International Symposium on Low Power Electronics and Design*, pages 69–74, 2014. ISBN 978-1-4503-2975-0.

[40] Kristina Chodorow and Michael Dirolf. *MongoDB: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2010. ISBN 1449381561.

[41] Nachshon Cohen, Michal Friedman, and James R. Larus. Efficient logging in non-volatile memory by exploiting coherency protocols. *Proc. ACM Program. Lang.*, 1(OOPSLA), 2017.

[42] Nachshon Cohen, David T. Aksun, Hillel Avni, and James R. Larus. Fine-grain checkpointing with in-cache-line logging. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, page 441–454, 2019. ISBN 9781450362405.

[43] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, page 133–146, 2009. ISBN 9781605587523.

[44] 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*, page 143–154, 2010. ISBN 9781450300360.

[45] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0036-0. doi: 10.1145/1807128.1807152. URL `http://doi.acm.org/10.1145/1807128.1807152`.

[46] RDMA core userspace libraries and daemons. `https://github.com/linux-rdma/rdma-core`, 2022. Accessed: 2022-02-16.

[47] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 752–768, 2018. ISBN 9781450356985.

[48] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. In *Proceedings of the 20th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, page 631–644, 2015. ISBN 9781450328357.

[49] Tudor Alexandru David, Rachid Guerraoui, Tong Che, and Vasileios Trigonakis. Designing ascy-compliant concurrent search data structures. Technical report, EPFL, 2014.

[50] Miyuru Dayarathna, Charuwat Houngkaew, and Toyotaro Suzumura. Introducing scalegraph: An x10 library for billion scale graph analytics. In *Proceedings of the 2012 ACM SIGPLAN X10 Workshop*, 2012. ISBN 9781450314916.

[51] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, page 205–220, 2007. ISBN 9781595935915.

[52] Peter Desnoyers, Ian Adams, Tyler Estro, Anshul Gandhi, Geoff Kuenning, Mike Mesnier, Carl Waldspurger, Avani Wildani, and Erez Zadok. Persistent memory research in the post-optane era. In *Proceedings of the 1st Workshop on Disruptive Memory Systems*, page 23–30, 2023. ISBN 9798400703003.

[53] 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*, page 1243–1254, 2013. ISBN 9781450320375.

[54] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, page 401–414, 2014. ISBN 9781931971096.

[55] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, 2014. ISBN 9781450327046.

[56] 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: 2022-09-16.

[57] Aaron J. Elmore, Vaibhav Arora, Rebecca Taft, Andrew Pavlo, Divyakant Agrawal, and Amr El Abbadi. Squall: Fine-grained live reconfiguration for partitioned main memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, page 299–313, 2015. ISBN 9781450327589.

[58] Jose M. Faleiro and Daniel J. Abadi. Latch-free synchronization in database systems: Silver bullet or fool's gold? In *Proceedings of the 8th Biennial Conference on Innovative Data Systems Research*, page 9, 2017.

[59] Panagiota Fatourou, Nikolaos D. Kallimanis, and Thomas Ropars. An efficient wait-free resizable hash table. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, page 111–120, 2018. ISBN 9781450357999.

[60] Docker: Empowering App Development for Developers. `https://www.docker.com`, 2022. Accessed: 2022-02-16.

[61] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2):5–es, 2007. ISSN 0734-2071.

[62] Xinwei Fu, Wook-Hee Kim, Ajay Paddayuru Shreepathi, Mohannad Ismail, Sunny Wadkar, Dongyoon Lee, and Changwoo Min. Witcher: Systematic crash consistency testing for non-volatile memory key-value stores. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, page 100–115, 2021. ISBN 9781450387095.

[63] Xinwei Fu, Dongyoon Lee, and Changwoo Min. Durinn: Adversarial memory and thread interleaving for detecting durable linearizability bugs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 195–211, 2022. ISBN 978-1-939133-28-1.

[64] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, page 249–264, 2016. ISBN 9781931971331.

[65] Vasilis Gavrielatos, Antonios Katsarakis, Arpit Joshi, Nicolai Oswald, Boris Grot, and Vijay Nagarajan. Scale-out ccnuma: Exploiting skew with strongly consistent caching. In *Proceedings of the Thirteenth EuroSys Conference*, 2018. ISBN 9781450355841.

[66] Gen-Z-Consortium. https://genzconsortium.org/, 2022. Accessed: 2022-02-16.

[67] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pages 29–43, 2003.

[68] Garth A. Gibson and Rodney Van Meter. Network attached storage architecture. *Commun. ACM*, 43(11):37–45, nov 2000. ISSN 0001-0782.

[69] Amit Golander, Sagi Manole, and Yigal Korman. Persistent memory over fabric (pmof). In *Proceedings of the 10th ACM International Systems and Storage Conference*, 2017. ISBN 9781450350358.

[70] V. Gottemukkala, E. Omiecinski, and U. Ramachandran. A scalable sharing architecture for a parallel database system. In *Proceedings of 1994 6th IEEE Symposium on Parallel and Distributed Processing*, pages 110–117, 1994.

[71] Kubernetes: Production grade container orchestration. http://kubernetes.io, 2022. Accessed: 2022-02-16.

[72] Goetz Graefe. Modern b-tree techniques. *Foundations and Trends® in Databases*, 3(4):203–402, 2011. ISSN 1931-7883.

[73] Paul Grun, Stephen Bates, and Rob Davis. Persistent memory over fabrics (pmof). In *Persistent Memory Summit 2018*, 2018. URL https://www.snia.org/educational-library/persistent-memory-over-fabrics-pmof-2018. Accessed: 2022-02-16.

[74] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 649–667, 2017. ISBN 978-1-931971-37-9.

[75] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: A hardware-software co-designed disaggregated memory system. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, page 417–433, 2022. ISBN 9781450392051.

[76] Zvika Guz, Harry (Huan) Li, Anahita Shayesteh, and Vijay Balakrishnan. Nvme-over-fabrics performance characterization and the path to low-overhead flash disaggregation. In *Proceedings of the 10th ACM International Systems and Storage Conference*, 2017. ISBN 9781450350358.

[77] R. Hagmann. Reimplementing the cedar file system using logging and group commit. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 155–162, 1987. ISBN 0-89791-242-X.

[78] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991. ISSN 0164-0925.

[79] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3): 463–492, 1990. ISSN 0164-0925.

[80] Dave Hitz, James Lau, and Michael A. Malcolm. File system design for an NFS file server appliance. In *USENIX Winter 1994 Technical Conference, San Francisco, California, USA, January 17-21, 1994, Conference Proceedings*, pages 235–246, 1994.

[81] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. Nvthreads: Practical persistence for multi-threaded applications. In *Proceedings of the Twelfth European Conference on Computer Systems*, page 468–482, 2017. ISBN 9781450349383.

[82] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable transient inconsistency in byte-addressable persistent b+-tree. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, page 187–200, 2018. ISBN 9781931971423.

[83] Stratos Idreos and Mark Callaghan. Key-value storage engines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, page 2667–2672, 2020. ISBN 9781450367356.

[84] Intel. Intel 64 and IA-32 Architectures Software Developers Manual Combined Volumes. `https://software.intel.com/en-us/articles/intel-sdm`, 2019.

[85] Intel. *Intel Inspector*, 2019. URL `https://software.intel.com/en-us/get-started-with-inspector`.

[86] Intel-Optane-DC-Persistent-Memory. `https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html`, 2022. Accessed: 2022-02-16.

[87] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via justdo logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 427–442, 2016. ISBN 978-1-4503-4091-5.

[88] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *Proceedings of 30th International Symposium on Distributed Computing, DISC 2016, Paris, France, September 27-29, 2016.*, pages 313–327, 2016.

[89] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the

intel optane dc persistent memory module, 2019. URL https://arxiv.org/abs/1903.05714. Accessed: 2022-09-19.

[90] Kyungho Jeon, Hyuck Han, Shin-gyu Kim, Hyeonsang Eom, Heon Y. Yeom, and Yongwoo Lee. Large graph processing based on remote memory system. In *2010 IEEE 12th International Conference on High Performance Computing and Communications (HPCC)*, pages 533–537, 2010. doi: 10.1109/HPCC.2010.88.

[91] 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*, page 45–51, 2022. ISBN 9781450393997.

[92] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance rdma systems. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, page 437–450, 2016. ISBN 9781931971300.

[93] Anuj Kalia, David Andersen, and Michael Kaminsky. Challenges and solutions for fast remote persistent memory access. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, page 105–119, 2020. ISBN 9781450381376.

[94] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: A high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499, 2008. ISSN 2150-8097.

[95] Martin Kaufmann, Amin Amiri Manjili, Panagiotis Vagenas, Peter Michael Fischer, Donald Kossmann, Franz Färber, and Norman May. Timeline index: A unified data structure for processing queries on temporal data in sap hana. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, page 1173–1184, 2013. ISBN 9781450320375.

[96] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. Lessons learned from the chameleon testbed. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, pages 219–233, 2020. ISBN 978-1-939133-14-4.

[97] Kimberly Keeton. The machine: An architecture for memory-centric computing. In *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers*, 2015. ISBN 9781450336062.

[98] Kimberly Keeton. Memory-driven computing. In *Keynote at 15th USENIX Conference on File and Storage Technologies*, 2017. URL `https://www.usenix.org/conference/fast17/technical-sessions/presentation/keeton`. Accessed: 2022-09-16.

[99] Kimberly Keeton, Sharad Singhal, and Michael Raymond. The openfam api: A programming model for disaggregated persistent memory. In *OpenSHMEM and Related Technologies. OpenSHMEM in the Era of Extreme Heterogeneity*, pages 70–89, 2019. ISBN 978-3-030-04918-8.

[100] Alfons Kemper and Thomas Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*, pages 195–206, 2011.

[101] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. Hyperloop: Group-based nic-offloading to accelerate replicated transactions in multi-tenant storage systems. In *Proceedings of the 2018 Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, page 297–312, 2018. ISBN 9781450355674.

[102] Wook-Hee Kim, Jihye Seo, Jinwoong Kim, and Beomseok Nam. clfb-tree: Cacheline friendly persistent b-tree for nvram. *ACM Transactions on Storage (TOS)*, 14(1):5, 2018.

[103] Persistent Memory Development Kit. `https://pmem.io/pmdk/`, 2022. Accessed: 2022-02-16.

[104] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash storage disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems*, 2016. ISBN 9781450342407.

[105] Roland Kühn, Daniel Biebert, Christian Hakert, Jian-Jia Chen, and Jens Teubner. Towards data-based cache optimization of b+-trees. In *Proceedings of the 19th International Workshop on Data Management on New Hardware*, page 63–69, 2023. ISBN 9798400701917.

[106] Chinmay Kulkarni, Aniraj Kesavan, Tian Zhang, Robert Ricci, and Ryan Stutsman. Rocksteady: Fast migration for low-latency in-memory storage. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles*, page 390–405, 2017. ISBN 9781450350853.

[107] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, page 460–477, 2017. ISBN 9781450350853.

[108] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16 (2):133–169, may 1998. ISSN 0734-2071.

[109] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, December 2001. URL `https://www.microsoft.com/en-us/research/publication/paxos-made-simple/`. Accessed: 2022-07-10.

[110] Philip Lantz, Subramanya Dulloor, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. Yat: A validation framework for persistent memory software. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 433–438, 2014. ISBN 978-1-931971-10-2.

[111] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. Wort: Write optimal radix tree for persistent memory storage systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 257–270, 2017. ISBN 978-1-931971-36-2.

[112] 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*, page 462–477, 2019. ISBN 9781450368735.

[113] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe : Converting concurrent dram indexes to persistent-memory indexes (extended version), 2019. URL `https://arxiv.org/abs/1909.13670`. Accessed: 2023-10-04.

[114] Sekwon Lee. *BUG fix : FAIR algorithm*, 2019. URL `https://github.com/DICL/FAST_FAIR/pull/4`.

[115] 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 (extended version), 2022. URL `https://arxiv.org/abs/2209.08743`. Accessed: 2022-09-19.

[116] 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.

[117] Youngmoon Lee, Hasan Al Maruf, Mosharaf Chowdhury, Asaf Cidon, and Kang G. Shin. Hydra : Resilient and highly available remote memory. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies*, pages 181–198, February 2022. ISBN 978-1-939133-26-7.

[118] Philip L. Lehman and s. Bing Yao. Efficient locking for concurrent operations on b-trees. *ACM Trans. Database Syst.*, 6(4):650–670, 1981. ISSN 0362-5915.

[119] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 38–49, 2013.

[120] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. The art of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*, page 3, 2016.

[121] Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. Thread and memory placement on numa systems: Asymmetry matters. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, page 277–289, 2015. ISBN 9781931971225.

[122] Justin Levandoski, David Lomet, , and Kevin Keliang Zhao. Deuteronomy: Transaction support for cloud data. In *Conference on Innovative Data Systems Research (CIDR)*, pages 123–133, 2011.

[123] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. The bw-tree: A b-tree for new hardware platforms. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 302–313, April 2013. doi: 10.1109/ICDE.2013.6544834.

[124] LevelDB. `https://github.com/google/leveldb`, 2022. Accessed: 2022-12-26.

[125] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 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*, page 574–587, 2023. ISBN 9781450399166.

[126] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, page 583–598, 2014. ISBN 9781931971164.

[127] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, page 429–444, 2014. ISBN 9781931971096.

[128] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, page 267–278, 2009. ISBN 9781605585260.

[129] Feilong Liu, Lingyan Yin, and Spyros Blanas. Design and evaluation of an rdma-aware data shuffling operator for parallel database systems. In *Proceedings of the Twelfth European Conference on Computer Systems*, page 48–63, 2017. ISBN 9781450349383.

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

[131] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. Pmtest: A fast and flexible testing framework for persistent memory programs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 411–425, 2019. ISBN 978-1-4503-6240-5.

[132] Xinxin Liu, Yu Hua, and Rong Bai. Consistent rdma-friendly hashing on remote persistent memory. In *Proceedings of the 2021 IEEE 39th International Conference on Computer Design*, pages 174–177, 2021.

[133] 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.

[134] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. Imbalance in the cloud: an analysis on alibaba cluster trace. In *Proceedings of IEEE International Conference on Big Data*, pages 2884–2892, 2017.

[135] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 190–200, 2005. ISBN 1595930566.

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

[137] Teng Ma, Mingxing Zhang, Kang Chen, Zhuo Song, Yongwei Wu, and Xuehai Qian. Asymnvm: An efficient framework for implementing persistent data structures on asymmetric nvm architecture. In *Proceedings of the 25th ACM*

International Conference on Architectural Support for Programming Languages and Operating Systems, page 757–773, 2020. ISBN 9781450371025.

[138] Lukas M. Maas, Thomas Kissinger, Dirk Habich, and Wolfgang Lehner. Buzzard: A numa-aware in-memory indexing system. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, page 1285–1286, 2013. ISBN 9781450320375.

[139] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, page 135–146, 2010. ISBN 9781450300322.

[140] Non-Volatile Memory Manager. `https://github.com/HewlettPackard/gull`, 2023. Accessed: 2023-10-11.

[141] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012*, pages 183–196, 2012.

[142] Alessandro Margara, Gianpaolo Cugola, Nicolò Felicioni, and Stefano Cilloni. A model and survey of distributed data-intensive systems. *ACM Comput. Surv.*, 56(1), 2023. ISSN 0360-0300.

[143] 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.

[144] Ajit Mathew and Changwoo Min. Hydralist: A scalable in-memory index using asynchronous updates and partial replication. *Proc. VLDB Endow.*, 13 (9):1332–1345, jun 2020. ISSN 2150-8097.

[145] Nimrod Megiddo and Dharmendra S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 115–130, March 2003.

[146] Memcached. *A distributed memory object caching system*, 2011. URL `http://memcached.org`.

[147] Zviad Metreveli, Nickolai Zeldovich, and M. Frans Kaashoek. Cphash: A cache-partitioned hash table. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, page 319–320, 2012. ISBN 9781450311601.

[148] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using One-Sided RDMA reads to build a fast, CPU-Efficient Key-Value store. In *Proceedings of the 2013 USENIX Annual Technical Conference*, pages 103–114, June 2013. ISBN 978-1-931971-01-0.

[149] 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 (FAST 19)*, pages 31–44, Boston, MA, 2019. ISBN 978-1-931971-48-5.

[150] Faisal Nawab, Dhruva R. Chakrabarti, Terence Kelly, and Charles B. Morrey III. Procrastination beats prevention: Timely sufficient persistence for efficient crash resilience. In *Proceedings of the 18th International Conference on Extending Database Technology*, pages 689–694, 2015.

[151] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. Agamotto: How persistent is your persistent memory

application? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1047–1064, 2020. ISBN 978-1-939133-19-9.

[152] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, page 291–305, 2015. ISBN 9781931971225.

[153] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the sprite network file system. *ACM Trans. Comput. Syst.*, 6(1):134–154, feb 1988. ISSN 0734-2071.

[154] Edmund B. Nightingale, John R. Douceur, and Vince Orgovan. Cycles, cells and platters: An empirical analysisof hardware failures on a million consumer pcs. In *Proceedings of the Sixth European Conference on Computer Systems*, page 343–356, 2011. ISBN 9781450306348.

[155] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *Proceedings of 10th USENIX Symposium on Networked Systems Design and Implemetnation (NSDI)*, 2013.

[156] Joe Novak, Sneha Kumar Kasera, and Ryan Stutsman. Auto-scaling cloud-based memory-intensive applications. In *Proceedings of the 2020 IEEE 13th International Conference on Cloud Computing*, pages 229–237, 2020.

[157] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. An analysis of load imbalance in scale-out data serving. In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, page 367–368, 2016. ISBN 9781450342667.

[158] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, and Marcos Aguilera. Storm: A fast transactional dataplane for remote data structures. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, page 97–108, 2019. ISBN 9781450367493.

[159] Overview of kubectl. `https://kubernetes.io/docs/reference/kubectl/overview/`, 2021. Accessed: 2022-02-16.

[160] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, page 305–320, 2014. ISBN 9781931971102.

[161] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*, pages 371–386, 2016.

[162] Processor Counter Monitor (PCM). `https://github.com/opcm/pcm`, 2022. Accessed: 2022-07-10.

[163] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory Persistency. In *Proceedings of the 41st Annual International Symposium on Computer Architecuture*, pages 265–276. IEEE Press, 2014.

[164] perf: Linux profiling with performance counters. `https://perf.wiki.kernel.org/index.php/Main_Page`, 2023. Accessed: 2023-10-11.

[165] PMDK. *The libvmmalloc library*, 2019. URL `http://pmem.io/pmdk/libvmmalloc/`.

[166] PMDK. *pmreorder*, 2019. URL `http://pmem.io/pmdk/manpages/linux/master/pmreorder/pmreorder.1.html`.

[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. ISBN 978-1-939133-23-6.

[168] Danica Porobic, Ippokratis Pandis, Miguel Branco, Pınar Tözün, and Anastasia Ailamaki. Oltp on hardware islands. *Proc. VLDB Endow.*, 5(11):1447–1458, jul 2012. ISSN 2150-8097.

[169] The ZeroMQ project. `https://zeromq.org/`, 2022. Accessed: 2022-02-16.

[170] Gianlucca O. Puglia, Avelino Francisco Zorzo, César A. F. De Rose, Taciano Perez, and Dejan Milojicic. Non-volatile memory file systems: A survey. *IEEE Access*, 7:25836–25871, 2019.

[171] Chenhao Qu, Rodrigo N. Calheiros, and Rajkumar Buyya. Auto-scaling web applications in clouds: A taxonomy and survey. *ACM Comput. Surv.*, 51(4), jul 2018. ISSN 0360-0300.

[172] 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 Symposium on Operating Systems Principles*, page 497–514, 2017. ISBN 9781450350853.

[173] Jun Rao and Kenneth A. Ross. Cache conscious indexing for decision-support in main memory. In *Proceedings of the 25th International Conference on Very Large Data Bases*, page 78–89, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc. ISBN 1558606157.

[174] Jun Rao and Kenneth A. Ross. Making b+- trees cache conscious in main memory. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, page 475–486, 2000. ISBN 1581132174.

[175] Recipe. `http://github.com/utsaslab/recipe`, 2023. Accessed: 2023-10-02.

[176] RocksDB. `http://rocksdb.org/`, 2022. Accessed: 2022-12-26.

[177] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. Aifm: High-performance, application-integrated far memory. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, 2020. ISBN 978-1-939133-19-9.

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

[179] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. Log-structured memory for DRAM-based storage. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 1–16, 2014. ISBN ISBN 978-1-931971-08-9.

[180] David Schwalb, Markus Dreseler, Matthias Uflacker, and Hasso Plattner. Nvc-hashmap: A persistent and concurrent hashmap for non-volatile memories. In *Proceedings of the 3rd VLDB Workshop on In-Memory Data Mangement and Analytics*, page 4. ACM, 2015.

[181] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*, pages 69–87, October 2018. ISBN 978-1-939133-08-3.

[182] Debendra Das Sharma. Compute express link (cxl): Enabling heterogeneous data-centric computing with heterogeneous memory hierarchy. *IEEE Micro*, 43(2):99–109, 2023.

[183] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. Strom: Smart remote memory. In *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020. ISBN 9781450368827.

[184] Arjun Singhvi, Aditya Akella, Maggie Anderson, Rob Cauble, Harshad Deshmukh, Dan Gibson, Milo M. K. Martin, Amanda Strominger, Thomas F. Wenisch, and Amin Vahdat. Cliquemap: Productionizing an rma-based distributed caching system. In *Proceedings of the 2021 Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, page 93–105, 2021. ISBN 9781450383837.

[185] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B. Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. Memory errors in modern systems: The good, the bad, and the ugly. In *Proceedings of the 20th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, page 297–310, 2015. ISBN 9781450328357.

[186] Tom Talpey and J.R.H. Pinkerton. Rdma durable write commit. `https://datatracker.ietf.org/doc/html/draft-talpey-rdma-commit-00`, 2016. Accessed: 2022-02-16.

[187] R. Tewari, M. Dahlin, H.M. Vin, and J.S. Kay. Design considerations for distributed caching on the internet. In *Proceedings. 19th IEEE International Conference on Distributed Computing Systems (Cat. No.99CB37003)*, pages 273–284, 1999.

[188] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, page 1–12, 2012. ISBN 9781450312479.

[189] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: The next

generation. In *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020. ISBN 9781450368827.

[190] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, pages 33–48, 2020. ISBN 978-1-939133-14-4.

[191] Philippas Tsigas and Yi Zhang. Evaluating the performance of non-blocking synchronization on shared-memory multiprocessors. In *Proceedings of the 2001 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, page 320–321, 2001. ISBN 1581133340.

[192] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, page 18–32, 2013. ISBN 9781450323888.

[193] Abhishek Verma, Madhukar Korupolu, and John Wilkes. Evaluating job packing in warehouse-scale computing. In *Proceedings of the 2014 IEEE International Conference on Cluster Computing*, pages 48–56, 2014.

[194] Paolo Viotti and Marko Vukolić. Consistency in non-transactional distributed storage systems. *ACM Comput. Surv.*, 49(1), jun 2016. ISSN 0360-0300.

[195] Haris Volos. The case for replication-aware memory-error protection in disaggregated memory. *IEEE Computer Architecture Letters*, 20(2):130–133, 2021.

[196] Haris Volos, Kimberly Keeton, Yupu Zhang, Milind Chabbi, Se Kwon Lee, Mark Lillibridge, Yuvraj Patel, and Wei Zhang. Memory-oriented distributed computing at rack scale. In *Proceedings of the ACM Symposium on Cloud Computing*, page 529, 2018. ISBN 9781450360111.

[197] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building an elastic query engine on disaggregated storage. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation*, pages 449–462, February 2020. ISBN 978-1-939133-13-7.

[198] Qing Wang, Youyou Lu, and Jiwu Shu. Sherman: A write-optimized distributed b+tree index on disaggregated memory. In *Proceedings of the 2022 ACM SIGMOD International Conference on Management of Data*, page 1033–1048, 2022. ISBN 9781450392495.

[199] Ruihong Wang, Jianguo Wang, Stratos Idreos, M. Tamer Özsu, and Walid G. Aref. The case for distributed shared-memory databases with rdma-enabled memory disaggregation. *Proc. VLDB Endow.*, 16(1):15–22, 2022. ISSN 2150-8097.

[200] Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. Easy lock-free indexing in non-volatile memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 461–472. IEEE, 2018.

[201] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. Building a bw-tree takes more than just buzz words. In *Proceedings of the 2018 International Conference on Management of Data*, page 473–488, 2018. ISBN 9781450347037.

[202] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing RDMA-enabled distributed transactions: Hybrid is better! In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*, pages 233–251, October 2018. ISBN 978-1-939133-08-3.

[203] H.-S. Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P. Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E. Goodson. Phase change memory. *Proceedings of the IEEE*, 98(12):2201–2227, 2010.

[204] Chenggang Wu, Vikram Sreekanti, and Joseph M. Hellerstein. Autoscaling tiered cloud storage in anna. *Proc. VLDB Endow.*, 12(6):624–638, feb 2019. ISSN 2150-8097.

[205] Chenggang Wu, Jose M. Faleiro, Yihan Lin, and Joseph M. Hellerstein. Anna: A kvs for any scale. *IEEE Transactions on Knowledge and Data Engineering*, 33(2):344–358, 2021.

[206] Xingbo Wu, Fan Ni, Li Zhang, Yandong Wang, Yufei Ren, Michel Hack, Zili Shao, and Song Jiang. Nvmcached: An nvm-based key-value cache. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, page 18. ACM, 2016.

[207] 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.

[208] Xiating Xie, Xingda Wei, Rong Chen, and Haibo Chen. Pragh: Locality-preserving graph traversal with split live migration. In *Proceedings of the 2019 USENIX Annual Technical Conference*, pages 723–738, July 2019.

[209] Jian Xu and Steven Swanson. Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, page 323–338, 2016. ISBN 9781931971287.

[210] 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, February 2020. ISBN 978-1-939133-12-0.

[211] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. Nv-tree: Reducing consistency cost for nvm-based single

level systems. In *13th {USENIX} Conference on File and Storage Technologies ({FAST} 15)*, pages 167–181, 2015.

[212] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 191–208, 2020.

[213] Juncheng Yang, Yao Yue, and Rashmi Vinayak. Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 503–518, 2021.

[214] Daniel Zahka and Ada Gavrilovska. Fam-graph: Graph analytics on disaggregated memory. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 81–92, 2022. doi: 10.1109/IPDPS53621.2022.00017.

[215] Da Zhang, Vilas Sridharan, and Xun Jian. Exploring and optimizing chipkill-correct for persistent memory based on high-density nvrams. In *Proceedings of the 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture*, pages 710–723, 2018.

[216] Huanchen Zhang, David G Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. Reducing the storage overhead of main-memory oltp databases with hybrid indexes. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*, pages 1567–1581, 2016.

[217] Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu. FORD: Fast one-sided RDMA-based distributed transactions for disaggregated persistent memory. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies*, pages 51–68, 2022. ISBN 978-1-939133-26-7.

[218] Mingxing Zhang, Teng Ma, Jinqi Hua, Zheng Liu, Kang Chen, Ning Ding, Fan Du, Jinlei Jiang, Tao Ma, and Yongwei Wu. Partial failure resilient memory management system for (cxl-based) distributed shared memory. In *Proceedings of the 29th Symposium on Operating Systems Principles*, page 658–674, 2023. ISBN 9798400702297.

[219] Yingqiang Zhang, Chaoyi Ruan, Cheng Li, Xinjun Yang, Wei Cao, Feifei Li, Bo Wang, Jing Fang, Yuhui Wang, Jingze Huo, and Chao Bi. Towards cost-effective and elastic cloud database deployment via memory disaggregation. *Proc. VLDB Endow.*, 14(10):1900–1912, jun 2021. ISSN 2150-8097.

[220] 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 (OSDI 22)*, pages 179–193, July 2022. ISBN 978-1-939133-28-1.

[221] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. Foundationdb: A distributed unbundled transactional key value store. In *Proceedings of the 2021 International Conference on Management of Data*, page 2653–2666, 2021. ISBN 9781450383431.

[222] Yang Zhou, Hassan M. G. Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E. Culler, Henry M. Levy, and Amin Vahdat. Carbink: Fault-Tolerant far memory. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation*, pages 55–71, July 2022. ISBN 978-1-939133-28-1.

[223] 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, July 2018. ISBN 978-1-939133-01-4.

[224] 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. ISBN 978-1-939133-01-4.

[225] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. Designing distributed tree-based index structures for fast rdma-capable networks. In *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data*, page 741–758, 2019. ISBN 9781450356435.

[226] Pengfei Zuo and Yu Hua. A write-friendly hashing scheme for non-volatile memory systems. In *Proceedings of the 33st Symposium on Mass Storage Systems and Technologies*, MSST '17, 2017.

[227] 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 (OSDI 18)*, pages 461–476, 2018. ISBN 978-1-931971-47-8.

[228] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. One-sided RDMA-Conscious extendible hashing for disaggregated memory. In *Proceedings of the 2021 USENIX Annual Technical Conference*, pages 15–29, July 2021. ISBN 978-1-939133-23-6.