Efficient Data Structures and Algorithms for Practical Resource Disaggregated Data Centers

SHEN, Jiacheng

A Thesis Submitted in Partial Fulfilment
of the Requirements for the Degree of
Doctor of Philosophy
in
Computer Science and Engineering

The Chinese University of Hong Kong
May 2024

Thesis Assessment Committee

Professor LEE Pak Ching (Chair)

Professor LYU Rung Tsong Michael (Thesis Supervisor)

Professor XU Qiang (Committee Member)

Professor CHEN Haibo (External Examiner)

Abstract of thesis entitled:

Efficient Data Structures and Algorithms for Practical Resource Disaggregated Data Centers
Submitted by SHEN, Jiacheng
for the degree of Doctor of Philosophy
at The Chinese University of Hong Kong in May 2024

Data centers are heading towards disaggregation to gain better resource efficiency. From mainframes to clusters of commodity servers, from versatile homogeneous clusters to specialized heterogeneous clusters, we have witnessed many successful disaggregation practices in the history of data center evolution. All these approaches decouple hardware into smaller management units, improving flexibility in resource management and achieving better resource efficiency.

Unfortunately, in today's data centers, resources are still coupled in individual monolithic servers, leading to severe resource under-utilization. Resource disaggregation is proposed to achieve better resource efficiency by taking disaggregation one step forward. It achieves near-optimal flexibility in resource management by decoupling hardware from monolithic servers into independent network-connected resource pools. However, such an architecture remains impractical due to its performance issues. Programs suffer from severe performance degradation since their data structures and algorithms are not suitable to the disaggregated architecture with loosely coupled hardware.

In this thesis, we address the performance issue by designing efficient data structures and algorithms native to the dis-

aggregated architecture. We specifically focus on disaggregated memory (DM), the central part of resource disaggregation, and design data structures and algorithms for in-memory storage systems over DM, *i.e.*, memory-disaggregated storage systems. Our work covers the data structure and algorithm design for three main components of a memory-disaggregated storage system, *i.e.*, memory management, index, and fault tolerance.

First, memory management of a memory-disaggregated storage system involves memory allocation and executing caching algorithms, which are the joint effort of data structures and algorithms. However, existing memory allocation and caching algorithms are designed for monolithic-server-based storage systems, which cannot adapt to DM with asymmetric compute power between the compute and memory pool. We design a two-level memory allocator and the first client-centric caching framework on DM to achieve efficient memory management. We integrate the two approaches into Ditto, the first memory-disaggregated caching system. Ditto shows up to 9× higher throughput than the state-of-the-art caching systems under YCSB workloads.

Second, for index data structures, we focus on range indexes. Existing range indexes on DM treat the memory pool as disks and use B+ trees as the index data structure. They suffer from suboptimal performance due to the inherent I/O size amplification of B+ trees. We propose SMART, a high-performance radix-tree-based range index on DM that has nearly no I/O size amplifications. SMART further addresses the challenges regarding concurrency control with a hybrid concurrency control scheme and reduces amplifications in I/O numbers with a read delegation and write combining scheme. SMART outperforms existing approaches by up to 6.1× under YCSB workloads.

Third, reliability is critical to memory-disaggregated storage systems. Unfortunately, existing fault tolerance algorithms cannot adapt to the complicated failure situations on DM due to

the isolated compute and memory failures. Moreover, they suffer from suboptimal performance due to their severe I/O amplifications and high concurrency control overhead. To achieve reliability with high performance, we design high-performance replication and logging algorithms native to DM. We adopt these two algorithms in FUSEE, the first fully memory-disaggregated storage system. Our evaluation results show that FUSEE performs up to 10× better than existing approaches.

Last but not least, we present DMC the industrial practice in Huawei Cloud that uses DM to improve the memory efficiency of its distributed caching service. We integrate some of the above memory management and indexing data structures and algorithms into DMC. We close the gap between academia and industry by introducing our design principles, design decisions, and lessons learned from our experience.

In summary, this thesis contributes to both academia and industry by making resource disaggregation more practical in a bottom-up manner. To industry, we promote the deployment of DM in cloud data centers by investigating the design of a production-level memory-disaggregated caching service. To academia, we provide guidelines on efficient data structures and algorithm design for resource-disaggregated data centers and show the benefits of disaggregating an in-memory storage system. Comprehensive experiments confirm the effectiveness and efficiency of our proposed methods.

論文題目　　：面向資源分離式數據中心的高效數據結構和算法設計
作者　　：沈家誠
學校　　：香港中文大學
學系　　：計算機科學與工程學系
修讀學位　　：哲學博士
摘要　　：
數據中心正步入解耦合時代以獲得更好的資源效率。從大型機到商品服務器集群，從通用同構集群到專用異構集群，我們在數據中心演進的歷史上見證了許多成功的解耦合實踐。所有這些方法都將硬體解耦成更小的管理單位，提高資源管理的靈活性，並實現更好的資源效率。

不幸的是，在當今的數據中心，資源仍然耦合在單體式服務器中，帶來嚴重的資源浪費。順應數據中心解耦合的趨勢，近年來提出了資源解耦合架構來進一步提升數據中心資源利用率。它將硬體從單一的服務器解耦為獨立的網絡連接的資源池，從而實現資源管理的最佳靈活性。然而，由於在其上運行程序的嚴重性能問題，這種架構仍然沒有廣汎應用。這種性能問題的核心是因爲現有的數據結構和算法不再適合於硬體鬆散耦合的解耦合架構。

在這篇論文中，我們通過設計更適合於解耦架構的數據結構和算法來解決性能問題。我主要關注內存分離架構，並為內存分離架構上的存儲系統設計數據結構和算法。我們的工作涵蓋了內存分離存儲系統的三個主要組件的數據結構與算法設計，即，內存管理，索引，和容錯。

首先，內存分離的存儲系統的內存管理涉及 1）分配內存空間，以及 2）執行緩存替換算法。然而現有內存分配以及緩存替換算法面向基於單體式服務器的存儲系統設計，無法適應分離式內存架構中計算池與內存池中不對稱的算力。我們設計了

一個兩級內存分配器以及首個以客戶端爲中心的緩存替換框架來實現分離式內存架構上高效的內存管理。我們將這兩個技術應用在首個內存分離的緩存系統 Ditto 中。實驗表明 Ditto 比現有方法在 YCSB 負載下有超過 9 倍的性能提升。

其次，對於索引數據結構，我們主要關注範圍索引。現有分離式內存架構上的範圍索引方法將內存池看作磁盤，並沿用了基於 B+ 樹的範圍索引數據結構，其性能受制於 B+ 樹嚴重的 I/O 粒度放大。我們提出 SMART ，一個基於基數樹的內存分離架構下的高性能範圍索引。我們通過使用基數樹避免了 B+ 樹的 I/O 粒度放大，并且通過混合并發控制和讀代理和寫合并技術解決了在分離式內存上構建基數樹帶來的竝髮控製和多次 I/O 的挑戰。實驗表明 SMART 相比現有方法帶來了至多 6.1 倍的性能提升。

此外，可靠性也是內存分離架構上的存儲系統的重要要求。然而，現有的故障處理算法無法適應由分離式內存中計算與內存錯誤解耦合帶來的多種錯誤情況。而且現有容錯算法在關鍵路徑上引入了過多的 I/O 導致系統性能下降。爲了在保證性能的同時具有可靠性，我們提出了內存分離架構下高性能的數據副本和日志算法。我們將這兩個算法應用到了 FUSEE 中，并且通過實驗證明了這兩個算法的有效性。實驗表明，相較於現有方法，FUSEE 能帶來至多 10 倍的性能提升。

最後，我們介紹了華為雲設計內存分離的緩存服務的工業實踐（DMC ）。我們研究了現有華為雲分佈式緩存服務的內存利用率問題，介紹了指導 DMC 設計的核心需求和思想，並討論了我們在此過程中學到的經驗教訓。此外，在 DMC 的設計與實現中應用到了上述內存管理和索引數據結構和算法，證明我們設計的有效性。

綜上，本文討論了在資源解耦合數據中心上設計高效數據結構與算法的指導思想，證明了將存儲系統內存分離之後帶來的的好處，並通過我們的工程實踐一定程度上推進了內存分離架構在現有數據中心的部署進度。我們通過全面的實驗確認了我們提出的方法的有效性和效率。

Acknowledgement

First and foremost, I would like to express my gratitude to my supervisor, Prof. Michael R. Lyu. He supported me in exploring research topics in systems and encouraged me whenever I encountered some difficulties. He is always patient in guiding me to achieve my goal and his help is without any reservations. His encouragement and guidance are essential to my Ph.D. study. I truly learned a lot from him not only in doing impactful research but also in being a nice person.

I appreciate Prof. Patrick P.C. Lee and Prof. Qiang Xu at The Chinese University of Hong Kong for their precious time to serve as my thesis assessment committee members. Their comments and suggestions throughout my Ph.D. study are essential to this thesis. I also appreciate Prof. Haibo Chen at Shanghai Jiao Tong University for kindly serving as the external examiner and also for his great comments on this thesis.

I'm grateful to have the opportunity to work with my excellent collaborators at The Chinese University of Hong Kong, Fudan University, Sun Yat-Sen University, and Huawei Cloud. I thank Prof. Yangfan Zhou at Fudan, Dr. Pengfei Zuo at Huawei, and Prof. Yuxin Su at SYSU. They always provide insightful suggestions and help me with their profound experiences in systems research. I also thank Mr. Xuchuan Luo at Fudan and Dr. Tianyi Yang at CUHK, with whom we did many fascinating and meaningful works. It is always a pleasure to discuss interesting ideas and research papers with them.

I want to thank all my peer students in Prof. Michael R. Lyu's and Prof. Yangfan Zhou's research groups. Especially my good friends who worked at Huawei Cloud and Fudan, Zhuangbin Chen, Jiazhen Gu, Jinyang Liu, Yichen Li, Zhihan Jiang, Junjie Huang, Xuchuan Luo, Yunzhe Zhang, Ruiying Zeng, and Bowen Yang. We had plenty of good times together in Hong Kong, Shenzhen, and Shanghai. Discussing with them is crucial to my mental health.

I want to convey my heartfelt appreciation to Ms. Siqi Han, who is not only my cherished girlfriend but also my closest confidant. Her comprehension of my choices, her patience during my absence, her encouragement during my moments of desolation, and the joyous, invaluable moments we've spent together all served as my source of motivation and inspiration. It is fortunate to have her during my Ph.D. study.

Finally, I am immensely grateful to my family. They encourage me to pursue my dream and provide their unconditional love throughout the entire process. There are no words to express my gratitude to my parents, Ms. Xuhui Chen and Mr. Jianming Shen. I dedicate this thesis to them.

Dedicated to my family.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Data centers that host huge computing resources and execute various applications are influencing billions of people's daily lives. Resource efficiency of data centers is critical for users to get cheap computing resources to execute various applications. Reviewing the history of data center evolution, our data centers are constantly heading towards disaggregation, *i.e.*, breaking hardware into smaller management units, to gain better resource efficiency. In the 1980s, data centers transformed from the mainframe architecture, a huge machine hosting abundant CPU, memory, and disks, to clusters of homogeneous servers [168]. Hardware resources in homogeneous clusters are then disaggregated into specialized heterogeneous clusters to better host the various heterogeneous hardware and satisfy applications with diverse resource requirements, *e.g.*, disaggregated storage clusters [9, 48, 46] and GPU farms [49].

Unfortunately, today's data centers still suffer from severe resource under-utilization due to the resource coupling in their server-centric architecture. For decades, the basic unit that allocates and executes programs in data centers is a monolithic server, *i.e.*, a single machine that consolidates various types of hardware devices required to execute a program. Data cen-

ter resource allocation and management thus become a complex multi-dimensional bin-packing problem since programs designed for monolithic servers typically cannot leverage resources across server boundaries [177, 199, 163]. While distributed systems can exploit resources on multiple servers, they are still restricted by the physical boundaries of servers since they are composed of multiple processes on different servers. According to our analysis of resource utilization traces in Google, Alibaba, and Huawei, the average CPU and memory utilization is only about 50% due to the severe fragmentation during resource alloation [47, 45].

The idea of resource disaggregation is proposed to achieve better resource efficiency by taking disaggregation one step further [177, 171, 180, 230, 131]. It breaks the boundaries of monolithic servers by managing different types of hardware into autonomous network-connected resource pools, *e.g.*, CPU pools, GPU pools, DRAM pools, etc. Such an architecture greatly simplifies resource management since resources can be flexibly combined and allocated to different applications from different resource pools, improving overall resource efficiency. Resource elasticity can also be improved since resources can be independently adjusted inside their resource pools without affecting the execution of existing applications [199]. Moreover, the failure domain becomes more fine-grained, *i.e.*, CPU failures no longer lead to the unavailability of the entire server together with in-memory data, potentially improving system reliability [227].

However, adopting the resource-disaggregated architecture in real-world data centers is challenging due to its severe performance issues. Applications suffer from 1.6× to 10× slow down due to the amplified communication latency between different hardware components [73]. While the communication latency can be reduced by advancements in the hardware layer, the latency will still be larger than inside single servers due to the larger physical distance between hardware components. Conse-

quently, the performance issue must be addressed in the software layer by designing better systems.

We identify that the major performance bottlenecks in the software layer are the unsuitable data structures and algorithms. Existing data structures and algorithms are designed for monolithic servers with the assumption that hardware components are closely coupled on the same motherboard. They suffer from severe I/O amplifications, expensive concurrency control overhead, and cannot fully exploit various heterogeneous hardware characters to achieve better performance.

This thesis addresses this performance issue in a bottom-up manner by designing high-performance data structures and algorithms native to the disaggregated architecture. Under this guideline, our work concentrates on exploring the synergy between disaggregated memory (DM) and in-memory storage systems, *i.e.*, memory-disaggregated storage systems. First, DM is the central part of resource disaggregation and is gaining increasing attention from both academia and industry [177, 171, 199, 200]. It decouples CPU and memory from monolithic servers into network-connected compute and memory pools. Second, in-memory storage systems, *e.g.*, Memcached [139] and Redis [166], are widely adopted in today's cloud data centers to accelerate application performance. It is beneficial for cloud providers to port in-memory storage systems to DM due to their severe memory under-utilization. Moreover, they contain many fundamental data structures and algorithms that can be applied in many other systems on DM, *e.g.*, memory management and index.

Figure 1.1 shows the high-level architecture of a memory-disaggregated storage system and the major contributions of this thesis. The contributions of this thesis are two-fold. First, we design high-performance data structures and algorithms for memory-disaggregated storage systems. We then describe and address the real-world challenges in productionizing a memory-

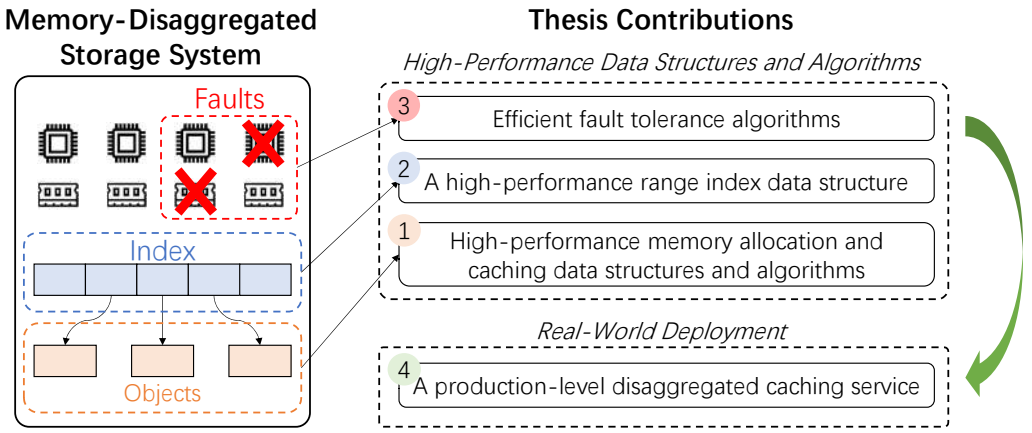


Figure 1.1: The high-level overview of a memory-disaggregated storage system and the contributions of the thesis.

disaggregated caching service.

For the data structures and algorithms design, this thesis designs efficient data structures and algorithms for three major components of a memory-disaggregated storage system, *i.e.*, ***memory management***, ***index***, and ***fault tolerance***. In terms of ***memory management***, we design a two-level memory allocator to efficiently allocate memory spaces to store objects and a client-centric caching framework to execute various caching algorithms with high performance and high hit rates. Both approaches are integrated into Ditto, the first memory-disaggregated caching system. In terms of ***index*** data structures, we design SMART, a high-performance range index on DM. SMART innovatively adopts a radix tree as the range index data structure to reduce the severe I/O size amplifications of traditional B+-tree-based approaches. Finally, in terms of ***fault tolerance*** algorithms, we design an embedded operation log scheme and a high-performance client-centric replication protocol to deal with the failures in the compute and memory pools, respectively. These two techniques are adopted in FUSEE, the first fully memory-disaggregated key-value store that can achieve both high performance and high reliability.

As for dealing with real-world challenges in production, we first statistically analyze the memory under-utilization issues of a production distributed caching service (DCS) cluster in Huawei Cloud. We identify the huge benefit we can get by disaggregating the DCS and introduce DMC, the industrial practice of Huawei Cloud that uses DM to improve the memory efficiency of its DCS. We integrate our designed memory management techniques into DMC and evaluate it with thorough experiments. Our discussions on design principles, design decisions, and lessons learned to close the gap between academia and industry in the field of memory disaggregation.

1.2 Thesis Contributions

This thesis contributes to both academia and industry to achieve a practical resource-disaggregated data center.

First, to academia, we provide guidelines on how to design efficient data structures and algorithms for disaggregated memory and memory-disaggregated storage systems.

- **Efficient memory management data structures and algorithms.**

  Memory management for a memory-disaggregated caching system involves 1) allocating memory spaces to store objects and 2) executing caching algorithms to keep hot objects in memory. Existing memory allocation and caching algorithms are designed for monolithic-server-based storage systems. They rely on the CPUs in the memory pool of DM to execute the memory management computation, which incurs severe performance degradation due to the asymmetric and weak compute power in the memory pool. Moreover, achieving high cache hit rates on DM is difficult due to the changing resources and data access patterns.

We design Ditto, the first memory-disaggregated caching system that simultaneously achieves efficient memory allocation and high cache hit rates. First, a two-level memory allocator is proposed to efficiently allocate memory from the memory pool. The two-level memory allocator separates the memory allocation data structures into compute-light and compute-heavy components and schedules them to the compute and memory pools according to their asymmetric compute capabilities. An adaptive client-centric caching framework is then proposed to execute various caching algorithms with high efficiency and achieve high cache hit rates. The client-centric caching framework approximates various caching algorithms with sampling and adopts a lightweight machine-learning algorithm to select the best caching algorithm according to the current workload. Extensive evaluation under YCSB and real-world Twitter workloads shows the effectiveness and performance of Ditto.

- **A high-performance range index data structure.**

  Range index is widely adopted by storage systems to conduct both point queries and range queries. Existing range indexes on DM are constructed with B+ trees, which sacrifice I/O sizes to reduce I/O numbers. However, the amplified I/O sizes waste the limited network bandwidth in the memory pool and become a severe performance bottleneck on DM when multiple compute nodes simultaneously access the memory pool.

  We propose SMART, a high-performance range index on DM. SMART innovatively adopts a radix tree as a range index data structure on DM, which has nearly no amplifications in I/O sizes due to the fine-grained tree nodes. We further address two challenges regarding concurrency control and the amplified number of I/O operations when con-

structing a high-performance radix tree on DM. SMART introduces a hybrid concurrency scheme to achieve efficient concurrency control and a read delegation and write combining scheme to further reduce the number of concurrent I/O operations. We use extensive evaluation under YCSB workloads to show the effectiveness and performance of SMART.

- **Efficient fault-tolerance algorithms.**

  Memory disaggregation introduces new challenges in terms of fault tolerance due to the physically decoupled but logically coupled failures of CPU and memory. Specifically, data could be lost when there are failures in the memory pool, affecting the execution of user requests in the compute pool. Besides, failures in the compute pool can corrupt data in the memory pool due to the partially executed operations, compromising system correctness.

  We propose FUSEE, the first fully memory-disaggregated storage system that achieves reliability with high performance. To avoid data loss under memory pool failures, we design a client-centric replication algorithm to replicate data in the memory pool with a bounded number of network I/Os and efficient rule-based conflict resolution. To deal with data corruption caused by compute pool failures, we propose an embedded operation log scheme to recover the corrupted data. The embedded logging algorithm stores log entries together with data to reduce the additional I/Os to maintain logs on operation critical paths. Extensive evaluation under YCSB workloads shows the effectiveness and performance of FUSEE.

Second, to industry, our work promotes the deployment of disaggregated memory in cloud data centers and shows a great

improvement in terms of memory efficiency by disaggregating a distributed caching service.

- **Deployment experiences.**

  Many cloud providers offer distributed caching services (DCSes) to speed up various cloud applications. We statistically analyze the severe memory under-utilization issues of a production DCS cluster and identify the potential benefit of porting DCS to DM. We introduce **D**isaggregated **M**emory **C**aching (DMC), a production-level memory-disaggregated caching service in Huawei Cloud. We close the gap between academia and industry by discussing the requirements, design principles and choices, and lessons learned from our experience. Thorough experimental results show that DMC improves memory utilization by up to 2.6 times with less than 9% performance loss introduced by DM.

1.3 Thesis Organization

The remainder of this thesis is organized as follows.

- **Chapter 2: Background and Related Work**

  Chapter 2 introduces resource disaggregation, scopes the problem we try to solve, and discusses the related works in the literature. Specifically, we first introduce the overall architecture of resource and memory disaggregation in Section 2.1. We then discuss general principles of designing high-performance data structures and algorithms native to the disaggregated data center in Section 2.2. We introduce in-memory storage systems over disaggregated memory, *i.e.*, memory-disaggregated storage systems, in Section 2.3 and discuss related works in Section 2.4.

- **Chapter 3: Efficient Memory Management Data Structures and Algorithms**

  In Chapter 3, we introduce our data structure and algorithms design to achieve high-performance memory management for memory-disaggregated storage systems. We introduce the challenges of managing memory and summarize our contributions in Section 3.1. We elaborate on the challenges in detail with experiments in Section 3.2. We introduce Ditto in Section 3.3, the first caching system on DM, and elaborate on its data structures and algorithms for remote memory allocation and caching algorithms execution. We evaluate Ditto with thorough experiments in Section 3.4, discuss the related works in Section 3.5, and summarize the chapter in Section 3.6.

- **Chapter 4: A High-Performance Range Index**

  In Chapter 4 we present SMART, a high-performance radix-tree-based range index on disaggregated memory. We discuss the issues with existing range indexes on DM and summarize our contributions in Section 4.1, introduce existing B+-tree-based range indexes in Section 4.2.2. We analyze issues with existing approaches in Section 4.3 and introduce the design of SMART in Section 4.4. Finally, we evaluate SMART in Section 4.5, introduces the related works in Section 4.6, and conclude the chapter in Section 4.7

- **Chapter 5: Efficient Fault Tolerance Algorithms**

  This Chapter handles the complex failure situations on DM with efficient replication and logging algorithms. Section 5.1 discusses the challenges of achieving fault tolerance with high performance on DM. Existing approaches are introduced in Section 5.2. We present FUSEE in Section 5.4, the first fully memory-disaggregated key-value store that

achieves reliability with high efficiency, and introduces the failure handling process in Section 5.5. We evaluate FUSEE with comprehensive experiments in Section 5.6, discuss the literature in Section 5.7, and summarize the chapter in Section 5.8.

- **Chapter 6: Industrial Practice: Productionizing a Memory-Disaggregated Caching Service**

  Chapter 6 introduces our industrial practice of productionizing a memory-disaggregated caching service. We first summarize the major requirements for a production-level memory-disaggregated caching service in Section 6.1 Then, we introduce the existing distributed caching service in Huawei Cloud and discuss its issues in Section 6.2. We discuss our design principles and introduce our design in detail in Sections 6.3, 6.4, and 6.5. Finally, we evaluate our design in Section 6.6, introduce related works in Section 6.8, and summarize the chapter in Section 6.9.

- **Chapter 7: Conclusion and Future Work**

  We conclude this thesis in Chapter 7 and talk about some interesting future directions. We envision a practical resource-disaggregated data center where various programs can execute on the disaggregated hardware with high performance and propose to achieve a better trade-off between performance and compatibility to achieve this goal.

□ **End of chapter.**

Chapter 2

Background and Related Work

Outline

This chapter introduces the background and related work for this thesis. We first introduce resource disaggregation, memory disaggregation, their motivations and performance issues in Section 2.1. We then highlight three key aspects of designing high-performance data structures and algorithms, *i.e.*, I/O amplifications, concurrency control, and asymmetric compute capabilities, in Section 2.2. Section 2.3 introduces memory-disaggregated storage systems and their three major components, *i.e.*, memory management, data indexing, and fault tolerance. Finally, we introduce the related work regarding resource disaggregation, memory disaggregation, and memory-disaggregated storage systems, in Section 2.4.

2.1 Resource-Disaggregated Data Centers

In this section, we first motivate resource disaggregation by discussing the limitations of server-centric data centers. We then introduce resource disaggregation and memory disaggregation,

as well as their performance issues.

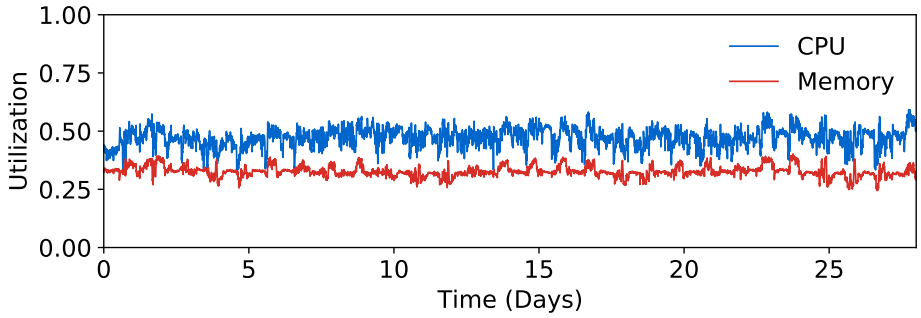
2.1.1 Limitations of Server-Centric Data Centers

The server-centric architecture, *i.e.*, using monolithic servers to deploy and manage resources, has been dominating data centers for decades [225]. However, two trends in modern data center hardware and software are making this architecture problematic.

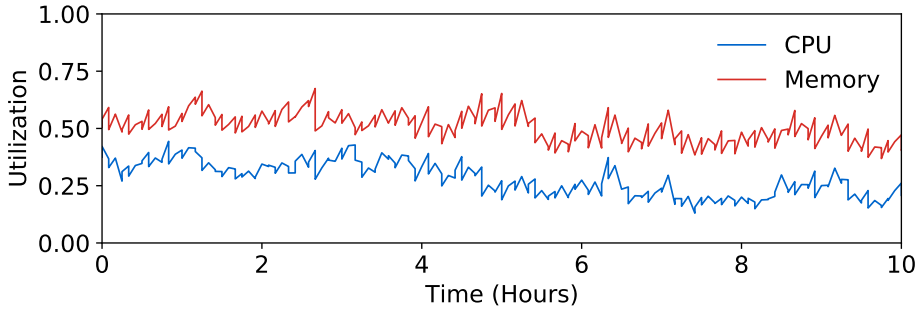
First, from the hardware perspective, due to the slowdown of Moore's law and the diminishing of Dennard scaling, modern data centers are embracing heterogeneous and domain-specific compute and storage devices. Heterogeneous compute devices, *e.g.*, Google TPU [94], FPGA [161], AWS Nitro [10], GPU, and programmable switches [155], provide higher computing efficiency with lower energy consumption than general-purpose processors, improving the overall cost-efficiency of the entire data center. Meanwhile, various heterogeneous storage devices with different speeds and capacities, *e.g.*, SSDs [142], persistent memory [87], disks, and even glasses [13], are deployed to satisfy the storage requirements for different applications.

Second, from the software perspective, today's data centers are hosting various types of applications. These applications exhibit more diverse requirements concerning the types and quantities of hardware resources. For instance, typical data analytics applications, *e.g.*, Hadoop [15] and Spark [16], require strong CPUs, large memory, and large storage to efficiently conduct data computations, while the emerging AI applications [111, 219, 203] mostly rely on strong GPUs to achieve high parallelism in model training and inference.

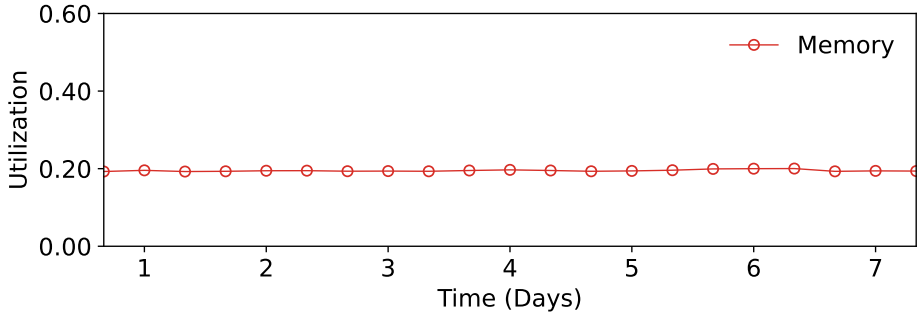
Satisfying these two trends in server-centric data centers suffers from the following three critical issues, *i.e.*, poor resource efficiency, poor hardware elasticity, and coarse-grained failures.



(a) Google.



(b) Alibaba.



(c) Huawei Cloud.

Figure 2.1: The CPU and memory utilization of Google, Alibaba, and Huawei Cloud data centers.

1) Poor resource efficiency. Server-centric data centers consolidate various heterogeneous hardware in individual monolithic servers. They manage hardware resources by creating multiple virtual machines (VM) on the deployed physical servers. Considering 1) monolithic servers as physical boundaries for re-

source management and 2) the diverse resource requirements for various cloud applications, data center resource allocation and management becomes a complex multi-dimensional bin packing problem [225]. We analyze the average CPU and memory utilization of three well-known cloud providers, *i.e.*, Google, Alibaba, and Huawei Cloud. For Google and Alibaba, we analyze 29-day and 10-hour CPU and memory utilization traces, respectively. For Huawei Cloud, we analyze a 7-day memory utilization trace. As shown in Figure 2.1, the average CPU and memory utilization accounts for only 50%. Half of the resources are wasted due to severe resource fragmentation and stranding [123]. Specifically, resource fragmentation refers to small pieces of resources that are insufficient to satisfy any application requirements. The stranded resources are those on physical servers that cannot be leveraged to create VMs due to the lack of another resource on the same physical server [123].

2) Poor hardware elasticity. Hardware elasticity refers to the convenience of reconfiguring hardware inside data centers. The emerging heterogeneous hardware devices and the varying resource requirements for modern data center applications make hardware elasticity more critical than ever. Adding new devices to support new applications and adjusting the resource proportions inside existing monolithic servers to better satisfy application requirements is demanding for cloud providers. Unfortunately, in today's server-centric data centers, reconfiguring hardware after they have been installed in monolithic servers is difficult and inconvenient [177, 65, 161]. Data center owners have to plan out server reconfigurations in advance so that applications executing inside these servers are not affected.

3) Coarse failure domain. Achieving high reliability is a basic requirement for both data center hardware and software [187, 77, 217]. However, the failure unit in existing server-centric data centers is usually coarse-grained [177, 199], *i.e.*, the

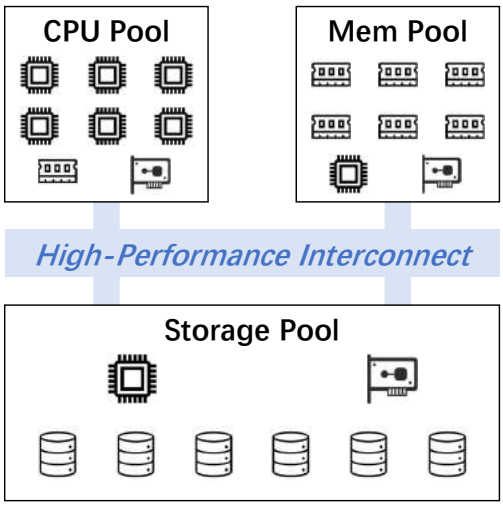


Figure 2.2: The architecture of a resource-disaggregated data center.

entire server becomes unusable or unavailable when a hardware component of the server fails, leading to the crashes of applications. Unfortunately, according to the previous analysis [173], motherboard, CPU, memory, and power supply failures account for more than 50% of hardware failures in server-centric data centers. System reliability in today's server-centric data centers is compromised due to the coupled hardware failures.

2.1.2 Resource Disaggregation

Resource disaggregation is advocated as a promising architecture for next-generation data centers. It can satisfy the hardware and software trends and address all the previous issues with the server-centric architecture [177, 199, 171]. As shown in Figure 2.2, in a resource-disaggregated data center, hardware resources are decoupled from individual monolithic servers and maintained as independent autonomous resource pools. These resource pools are interconnected with high-performance data center networking techniques so that different hardware components can communicate and cooperate with each other.

The resource disaggregated architecture addresses all the previous issues introduced by monolithic servers by providing more flexibility in resource management. First, data center resource allocation is greatly simplified since hardware resources are no longer coupled in a single server. Resource efficiency can be improved since data center owners can freely allocate and combine resources in different resource pools for various applications. Second, the hardware elasticity can also be improved since the decoupling of hardware components makes it possible to scale different types of hardware resources independently without affecting the execution of existing applications. New types of hardware devices can be conveniently deployed by creating a new resource pool and connecting to the data center network. Finally, disaggregating hardware resources creates a more fine-grained failure domain since the failures of different hardware components are isolated by the network.

Such an architecture is enabled by the advances in networking devices in the following three aspects:

High-speed network. Over the past decade, data center networks become more and more scalable. The speed of networking devices has grown over an order of magnitude. Existing high-performance networking techniques, *e.g.*, InfiniBand [86] and remote direct memory access (RDMA) [79], have already reached 400 Gbps and ultra-low latency, *e.g.*, sub-800 nanosecond latency [151]. Such high bandwidth is already comparable to the main memory bus in monolithic servers [177]. With the main memory bus facing a bandwidth wall [170], line rate network bandwidth in the future is even predicted to exceed local DRAM bandwidth [191].

Network-integrated devices. There is a trend in the hardware design to move network interfaces closer to hardware components, *e.g.*, Intel OmniPath [52], RDMA [79], and NVMe over Frabrics [183]. More and more hardware devices will be able to

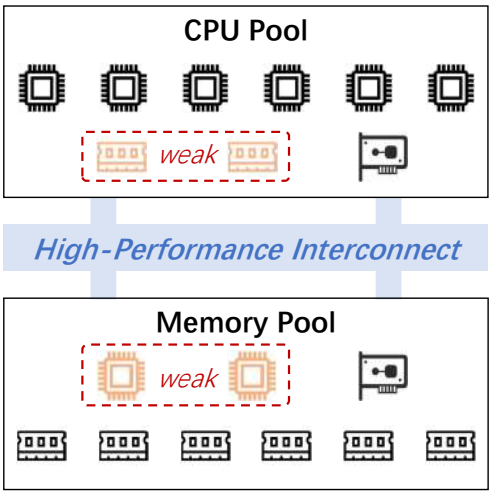


Figure 2.3: The overall architecture of memory disaggregation.

access networks directly without the help of additional general-purpose processors, *e.g.*, CPUs. This makes the idea of network-attached hardware possible in resource disaggregation.

Stronger hardware processing capabilities. Modern hardware devices are having more processing capabilities [86, 189, 52, 150], which enables software logic to be offloaded to hardware, *e.g.*, offloaded network stacks [86], offloaded network functions [178], and even offloaded storage systems [122]. The powerful on-device processing capabilities shift the computing paradigms in modern data centers from CPU-centric to XPU-centric [88], making it possible to create autonomous resource pools in the resource-disaggregated architecture.

2.1.3 Memory Disaggregation

Disaggregated memory (DM) [127, 199, 171, 230, 180, 131, 179] is the most widely-discussed topic in the resource disaggregation research. As shown in Figure 2.3, DM decouples the CPU and DRAM of monolithic servers into independent compute and memory pools. The compute pool contains multiple compute

nodes (CN) with abundant CPUs. Each CN is also equipped with a small amount of DRAM to serve as the runtime cache. The memory pool consists of memory nodes (MN), each hosting a large amount of DRAM. A small number of weak CPU cores are installed in each MN to serve lightweight management tasks, *i.e.*, network connection and memory management. The compute pool and the memory pool are connected with high-performance CPU-bypass networks with high bandwidth and microsecond-scale latency, *e.g.*, InfiniBand [86] and CXL [185].

The memory pool provides the compute pool with both data management and data access interfaces. The data management interfaces, *i.e.*, `ALLOC` and `FREE`, allow CNs to allocate and free memory spaces, which are implemented with the weak CPU cores in the memory pool. The implementation of data access interfaces, *i.e.*, `READ`, `WRITE`, `ATOMIC_CAS` (compare and swap), and `ATOMIC_FAA` (fetch and add), relies on the interconnect techniques. The CPU-bypass feature of the interconnect enables the memory pool to implement the data access interface with high throughput and low latency, without being bottlenecked by the weak CPUs on MNs. Without loss of generality, in this thesis, we assume that the compute pool and memory pool are connected with RDMA networking. However, the techniques in this thesis are also compatible with other types of interconnects as long as they provide the aforementioned data access and management interfaces.

The disaggregated memory architecture inherits all the benefits of the resource-disaggregated architecture, *i.e.*, higher memory efficiency, better CPU and memory elasticity, and fine-grained failure domain. Achieving memory disaggregation is also an urgent task due to the expensive memory prices and the low memory utilization in today's data centers.

2.1.4 The Performance Issue

Unfortunately, we are not seeing the large-scale deployment of the resource-disaggregated architecture in real-world data centers due to the severe performance issues. The performance overhead is rooted in both hardware and software. First, in the hardware layer, although advanced networking technologies greatly improve communication bandwidth, the communication latency is still an order of magnitude higher than that inside a single server. On DM, this amplified latency greatly increases the overhead of accessing remote memory, causing poor performance. Even worse, the communication latency for the resource-disaggregated architecture will stay lower than that on monolithic servers due to the amplified physical distance between hardware components. Consequently, the performance issue has to be addressed in the software layer.

From the software perspective, there are two ways to execute applications over a resource-disaggregated data center, *i.e.*, top-down approaches [177, 127, 171, 199] and bottom-up approaches [230, 228, 201]. Top-down approaches achieve disaggregation in an intermediate layer, *e.g.*, OS [177] and language runtimes [171, 199]. Programs can be executed over these intermediate abstractions without any modifications. However, such approaches suffer from up to 10 times performance degradation since the high-level abstractions in the intermediate layer greatly amplifies the communication overhead between the hardware components.

This thesis follows a bottom-up approach that designs applications and systems native to the resource-disaggregated architecture, *e.g.*, storage systems [223, 230], vector databases [89], databases [35]. For the bottom-up approach, the key performance bottleneck lies in the data structure and algorithms. It is reported that improper data structures and algorithm design severely degrade system performance, inducing more than 15

times slow down [230, 228, 201, 131]. We discuss the guidelines for designing data structures and algorithms in the following section (Section 2.2).

2.2 Data Structures and Algorithms for Resource Disaggregation

Designing efficient data structures and algorithms for the resource-disaggregated architecture requires us to understand the characters the new architecture. In this section, we first introduce the guidelines for efficient data structure and algorithm design by analyzing and comparing the resource-disaggregated architecture with some well-known systems. We then discuss the limitations of existing data structures and algorithms.

2.2.1 Guidelines for Data Structures and Algorithms for Resource Disaggregation

We highlight three critical aspects when designing efficient data structures and algorithms for resource-disaggregated data centers. Specifically, high-performance data structures and algorithms for resource-disaggregated data centers have to simultaneously achieve 1) high-performance concurrency control, 2) efficient I/O, and 3) fully exploited asymmetric compute capabilities of heterogeneous hardware. We summarize these principles by comparing the resource-disaggregated architecture with several well-known architectures.

To begin with, the key idea of resource disaggregation is to break the boundaries of resource management. In this perspective, the entire data center is managed as a huge monolithic server. As shown in Figure 2.4, similar to the canonical Von Neumann architecture, the heterogeneous computing devices are just like the compute unit and the various storage media are just

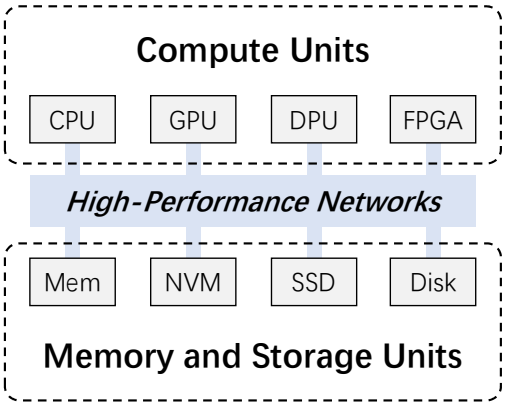


Figure 2.4: Resource disaggregation as a Von Neumann machine.

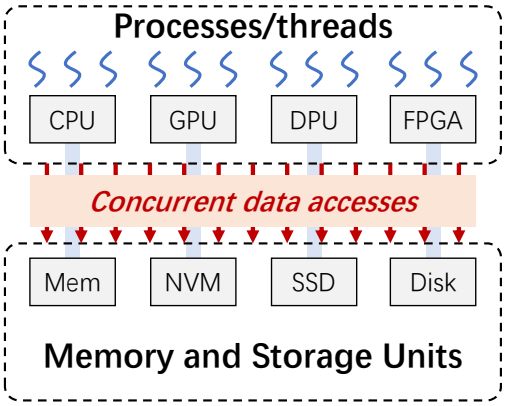


Figure 2.5: Resource disaggregation as a large-scale parallel machine.

like the memory and storage unit. Starting from this, we gain a deeper understanding by separately analyzing the compute unit pool, memory and storage unit pool, and the physical construction of the disaggregated data center.

The compute unit pool. From the perspective of the compute unit pool, a resource-disaggregated data center holds a huge number of heterogeneous computing devices. As shown in Figure 2.5, all these computing devices uniformly access the memory and storage unit pool with high-performance networks. Such an architecture is similar to **multi-processor parallel machines** [82], where data accesses are highly concurrent and possibly conflicting. *Consequently, data structures and algorithms for resource disaggregation have to efficiently resolve conflicts to achieve high-performance concurrency control.*

The memory and storage unit pool. From the perspective of the memory and storage unit pool, there are various different storage media, as shown in Figure 2.6. Different storage media have different characters in terms of speed and capacity, making the entire architecture resemble **the memory hierarchy in tiered memory systems** [103, 214]. Similar to the tiered memory systems, data structures and algorithms for re-

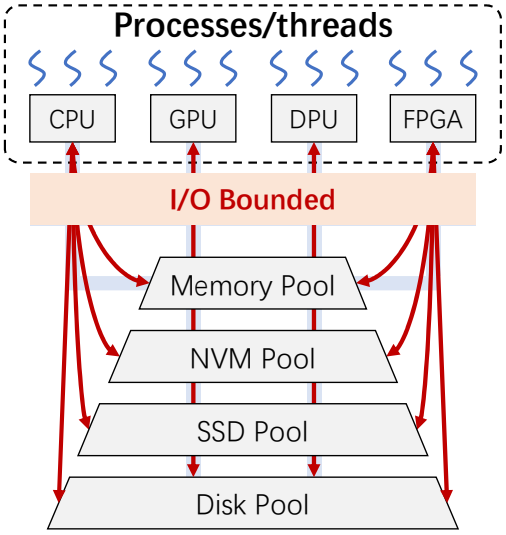


Figure 2.6: Resource disaggregation as a tiered memory system.

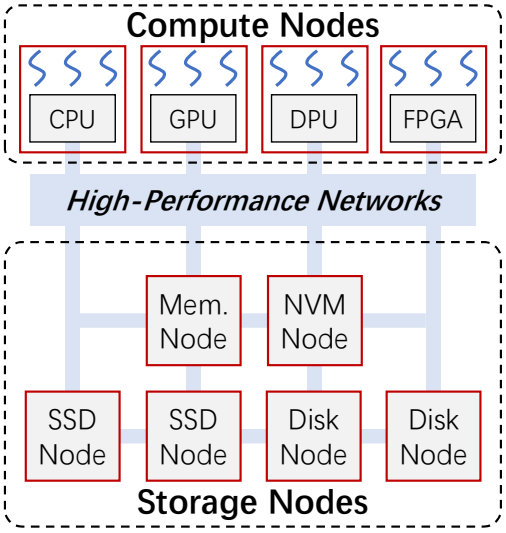


Figure 2.7: Resource disaggregation as a heterogeneous distributed system.

source disaggregation are also I/O bounded since all data access has to go through the network with higher latency and lower bandwidth. *Consequently, data structures and algorithms for resource disaggregation also have to achieve I/O efficiency by reducing the numbers and sizes of I/O operations.*

The physical construction. Finally, from the perspective of the physical construction of a resource-disaggregated data center, the basic unit is still servers, as shown in Figure 2.7. Fundamentally, we are dealing with **distributed systems** [15, 16, 139] and thus facing the same challenges, *i.e.*, high coordination overhead between nodes. However, the resource-disaggregated architecture cannot be directly viewed as a distributed system since servers in a resource-disaggregated data center are specialized and heterogeneous. Different servers host different devices and resources to achieve better resource flexibility. *Consequently, data structures and algorithms for resource disaggregation also have to exploit and optimize the computation over various heterogeneous devices to gain better performance.*

Table 2.1: Characters of data structures and algorithms for existing systems.

Systems	Concurrency	I/O	Asymmetry
Parallel Machines	🙂	☹	☹
Tiered Memory Systems	😐	🙂	☹
Distributed Systems	😐	😐	☹

2.2.2 Limitations with Existing Data Structures and Algorithms

Unfortunately, existing data structures and algorithms suffer from suboptimal performance over the resource-disaggregated architecture since cannot simultaneously achieve all these requirements, as illustrated in Table 2.1.

First, in large-scale parallel machines, many concurrent data structures and algorithms are designed to achieve high concurrency, *e.g.*, concurrent hash tables [84, 110, 61] and concurrent linked lists [140]. They design lock-based and lock-free algorithms to efficiently resolve conflict in data accesses and achieve high-performance concurrency control. However, they assume a uniform memory access model where all data are already loaded in the memory. When executing over the resource disaggregated architecture they suffer from severe I/O amplifications due to their multiple numbers of remote memory accesses. Meanwhile, they are designed for single machines without considering the asymmetry between servers.

Second, there are many I/O efficient data structures and algorithms for tiered memory systems, *e.g.*, B+ trees [146, 201]. They leverage the locality in the data structure and algorithm computation to reduce the numbers and sizes of I/O operations. However, these data structures are not optimized to achieve high concurrency due to their coarse-grained locks [146, 201]. Meanwhile, they also do not consider leveraging the asymmetric het-

erogeneous devices in various servers.

Finally, there are also many high-performance data structures and algorithms designed for distributed systems, *e.g.*, garbage collection [3], replication [194, 154], etc. They reduce the number of I/O operations by reducing the number of cross-node communications and achieve high concurrency by sharding data to multiple servers and dealing with these data individually [16, 166]. However, existing distributed computing data structures and algorithms assume a symmetric architecture where all nodes have comparable compute and storage capabilities. Executing these data structures and algorithms over resource-disaggregated data centers also cannot fully exploit the capabilities of heterogeneous devices to achieve high performance.

2.3 Memory-Disaggregated Storage Systems

Memory-disaggregated storage systems, *i.e.*, in-memory storage systems over disaggregated memory, is gaining attention in both academia and industry due to their better resource efficiency and elasticity. This section first introduces in-memory storage systems and then introduces the challenges of constructing such a system over disaggregated memory.

2.3.1 In-Memory Storage Systems

In-memory storage systems, *e.g.*, Memcached and Redis, are widely adopted in modern cloud data centers to serve as an intermediate caching layer. They provide upper-level applications with easy-to-use `GET` and `SET` interfaces to access data with high throughput and low latency.

Figure 2.8 shows the architecture of a typical in-memory storage system. There are multiple client threads in upper-level applications and multiple server nodes. Data are typically parti-

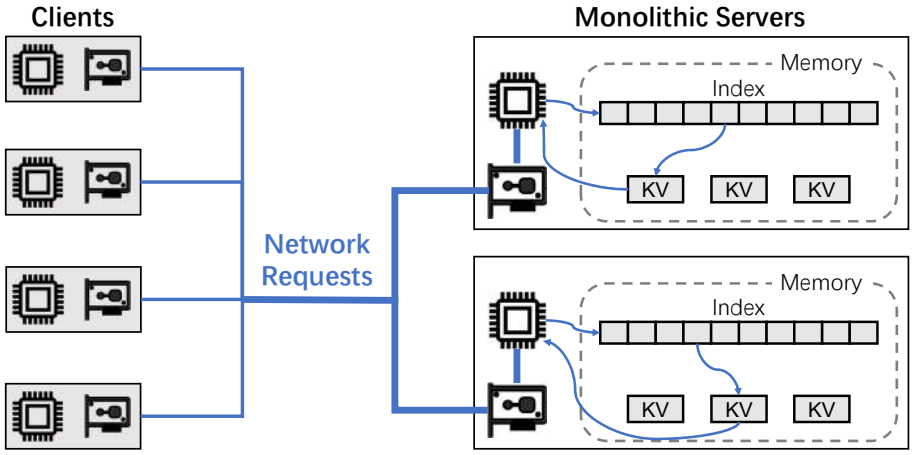


Figure 2.8: The general architecture of an in-memory storage system on monolithic servers.

tioned among all server nodes. Each server manages local memory with a memory allocator to store objects with variable sizes and a cache engine to only keep hot objects in the limited memory space. All objects are organized with an index to efficiently locate data when serving requests. Besides, objects are replicated among various servers with a data replication protocol to achieve high availability on node failures.

When executing `GET` and `SET` requests, clients send RPC requests to the node that contains the primary replica of the required data. When serving `GET` requests, the server searches its local index and returns the required data. When serving `SET` requests, the server first allocates new memory space to hold the newly written objects or evict an object when the memory is full. The index is then modified to reflect this object modification. The written data is synchronized to all nodes containing its backup replica according to the replication protocol. On node failures, requests are routed to nodes that contain the backup replica of the requested data.

Unfortunately, existing in-memory storage systems inherit the issues with the server-centric architecture.

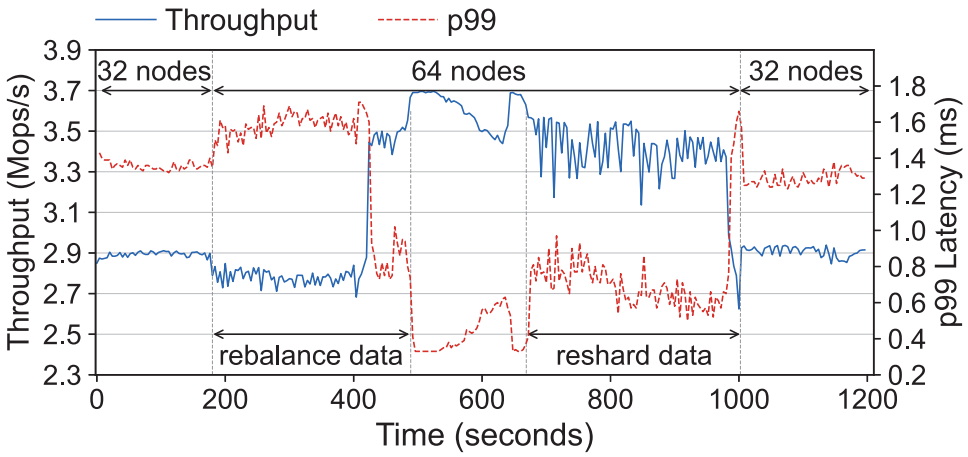


Figure 2.9: The performance of Redis when adjusting resources.

1) Resource inefficiency. Resources of existing in-memory storage systems on monolithic servers, *e.g.*, Redis [60], Memcached [139], ElastiCache [60], MemoryStore [75], are allocated with fix-sized virtual machines (VMs) with both CPU and memory, *e.g.*, 1 CPU with 2 GB DRAM, to facilitate resource management over monolithic servers. During dynamic resource scaling, resources are wasted when coupled CPU and memory are allocated, but only CPU or memory needs to be dynamically increased. Moreover, applications' resources requirements must be rounded up to fit in these fix-sized VMs, causing low resource utilization in the entire datacenter [19].

2) Poor resource elasticity. Existing in-memory storage systems shard data to multiple monolithic servers to leverage more CPU and memory resources and achieve higher throughput [126, 60, 75, 166]. Cached data have to be resharded and migrated when new VMs are added to the caching cluster. The migration cost [63] is unavoidable when either CPU or memory needs to be adjusted due to the coupled allocation of CPU and memory on monolithic servers. The performance gain when increasing resources and the resource reclamation after shrinking re-

sources is delayed for minutes due to the time-consuming data migration [112]. Moreover, the throughput drops and latency increases due to the consumption of additional CPU cycles and network bandwidths spent on moving data [109, 164].

Figure 2.9 shows the migration cost on Redis [166], the backend of many cloud caching services [60, 75], during resource adjustments under the read-only YCSB-C workload [50] with 10 million 256B key-value pairs. We first use 32 Redis nodes, each with 1 CPU core and 1 GB DRAM, then add 32 more nodes after 3 minutes of execution, and shrink back to 32 nodes after 3 minutes of stable execution with 64 nodes. We launch all 64 Redis nodes initially to calculate the pure cost of data migration and use 512 client threads to get the maximum throughput. When scaling to 64 nodes, Redis takes 5.3 minutes to migrate data. The throughput drops up to 7%, and the 99th percentile latency increases up to 21% in the process. When shrinking back to 32 nodes, the resource reclamation is delayed for 5.6 minutes due to data migration. Such migration cost is unavoidable even if using advanced migration techniques [109, 63] since CPU and memory are allocated in a coupled manner in VMs and objects are sharded to individual VMs.

3) Coupled failures. Finally, the reliability of existing in-memory caching systems is restricted by the coupled failure in server-centric data centers. Specifically, in a monolithic server, when a CPU fails data in the memory on the same server becomes unavailable. The situation is similar when it comes to memory failures. The reliability would become better if CPU and memory failures can be handled separately.

Due to all these issues, it is gaining increasing attention in both academia and industry to port in-memory storage systems over disaggregated memory, *i.e.*, constructing memory-disaggregated storage systems [119, 179, 180].

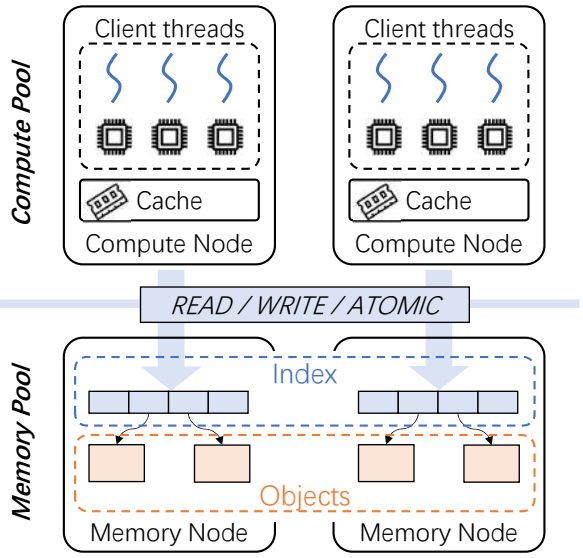


Figure 2.10: The architecture of a memory-disaggregated storage system.

2.3.2 Challenges over Disaggregated Memory

Figure 2.10 shows the architecture of a memory-disaggregated storage system. There is a compute pool and a memory pool. CNs in the compute pool contain abundant CPUs executing multiple threads to execute user requests. A small local memory is installed to serve as the runtime cache for these client threads. There are multiple MNs in the memory pool, each equipped with large memory space and a weak CPU core to execute management tasks. Both objects and the index are scattered and replicated among all MNs.

Three challenges have to be addressed to make memory-disaggregated storage systems practical.

1) Remote memory management. Memory management involves efficiently allocating memory spaces and executing caching algorithms. Both involve the maintenance of expensive memory management data structures and algorithms, *e.g.*, free lists for memory allocation, and heaps or lists for caching algorithms [153, 193, 92]. Unfortunately, existing memory man-

agement data structures and algorithms are designed with the assumption that both the memory and the data structures are maintained locally with low concurrency. They suffer from severe concurrency control overhead and amplifications in I/O numbers when being executed over disaggregated memory.

2) Efficient data indexing. The performance of the index data structure is critical to the efficiency of the request serving efficiency of the entire memory-disaggregated storage system. Unfortunately, existing approaches on DM overlook the memory character of the disaggregated memory pool. They treat remote memory as local high-performance disks due to the similar latency and bandwidth and increase I/O sizes to reduce the number of I/O operations [201, 229]. Consequently, they suffer from suboptimal performance since they induce severe waste in network bandwidths when executing over disaggregated memory with highly concurrent requests and limited network bandwidth.

3) Efficiently handling complex failures. Adopting memory disaggregation is a double-edged sword. On the one hand, it isolates the failures of compute and memory resources, potentially improving reliability. On the other hand, it introduces more complicated failure situations, *e.g.*, CNs and MNs can fail separately [14]. Efficiently handling all these failures is critical to achieve high reliability and availability. Unfortunately, existing data replication and logging algorithms suffer from severe I/O amplifications, resulting in severe overhead when executing on disaggregated memory.

In this thesis, we address all these challenges by designing data structures and algorithms native to DM that can simultaneously achieve high-performance concurrency control, efficient I/O, and optimized for asymmetric hardware.

2.4 Related Works

This section introduces the related work in terms of resource disaggregation, memory disaggregation, and memory-disaggregated storage systems.

2.4.1 Resource Disaggregation

The idea of resource disaggregation has been discussed for decades. Recent works focus on two main fields, *i.e.*, improving existing systems for storage disaggregation and exploring the disaggregation of a broader spectrum of devices.

The idea of storage disaggregation is first explored in the field of resource disaggregation due to the requirement to improve storage efficiency and the increase in network bandwidth [11]. In modern cloud data centers, block storage is commonly virtualized and disaggregated to meet various application requirements [9, 48, 46]. Works in this field typically focus on achieving good performance according to the characteristics of the storage device, *i.e.*, HDDs and SSDs. In terms of disaggregating traditional HDDs, Parallax [205], Blizzard [141], and Petal [117] provides a virtual block interface for operating systems and applications. They all focus on achieving higher performance and In terms of disaggregating traditional HDDs, systems like Parallax [205], Blizzard [141], and Petal [117] focus on the better implementation of distributed virtual block store with high performance and other features like replication and failure recovery. Snowflake constructs an elastic query engine with disaggregated object store in cloud data centers [197]. For SSDs, systems like Gimbal [142], disaggregated flash [106], CORFU [24], and FAWN [12], explores systems design to achieve high performance combining the block-access characters of novel SSDs. All these works are orthogonal to this thesis since this thesis focuses on the design of data structures and algorithms for in-memory

storage systems over disaggregated memory. Meanwhile, disaggregated memory is byte-addressable, which is fundamentally different compared with disks or SSDs.

Another thread of work focuses on the management of various heterogeneous disaggregated hardware. Among them, LegoOS [177] is the first work towards the management of hardware resources for disaggregated data centers. It proposes the idea of splitkernel, a new OS architecture to manage various hardware and connect them to the RDMA network. SuperNIC [178] is another approach that disaggregates network resources and computation with advanced FPGA-based SmartNICs. There are also works that manages disaggregated hardware resources in serverless platforms to make serverless platforms more resource efficient [51, 157]. These works majorly focuses on making hardware resource disaggregation possible, which is complementary to this thesis where we address the performance issues.

2.4.2 Memory Disaggregation

Memory disaggregation is the core of existing resource disaggregation research. Existing work on disaggregated memory can be classified into top-down and bottom-up approaches.

Top-down approaches achieve memory disaggregation with existing system intermediate abstraction layers, *e.g.*, operating systems [128, 76, 149, 8, 163, 124], language runtimes [171, 199, 80, 200, 227, 162], and co-designed systems software [127, 189, 185, 118, 202, 81, 178]. Operating-systems-based approaches view disaggregated memory pool as a fast swap device [149, 8]. They enable a process in a monolithic server to leverage memory resources in other servers by swapping in and out 4 KB memory pages. However, they suffer from poor performance since they amplify fine-grained remote memory accesses into 4 KB memory

pages. Runtime-based approaches [171, 199, 80] address this issue by enabling object-level swap in and out. However, the high-level abstraction hides the real memory organization detail from user programs, resulting in poor performance.

Bottom-up approaches port individual applications to the disaggregated memory architecture from scratch. Applications like key-value stores [192, 119, 180], transactional storage systems [55, 223, 56], compilers [102], and vector databases [89] are ported to the disaggregated memory to enjoy the larger memory spaces and the better resource efficiency.

All these approaches are orthogonal to this thesis. This thesis focuses on the data structures and algorithm design for systems over disaggregated memory. The technique is applicable to both system software and upper-level applications. For instance, the proposed memory management data structures can also be applied to disaggregated operating systems and language runtimes to achieve better performance. Also, the proposed index data structure can be applied to many bottom-up systems over disaggregated memory.

2.4.3 Memory-Disaggregated Storage Systems

Existing works on memory-disaggregated storage systems can be classified into two categories. The first explores different systems architectures to better exploit the benefits of disaggregated memory. Clover [192] and Dinomo [119] are the state-of-the-art memory-disaggregated storage systems. Clover [192] adopted an architecture that decouples the data and control plane so that the data access can be more efficient. Dinomo [119] partitions the ownership of data to compute nodes to achieve better elasticity and scalability. Different from their architecture, approaches in this thesis are designed for a memory-disaggregated storage system that adopts a shared everything architecture, which is

complementary to these works.

The second thread of work explores efficient data structures and algorithms design [230, 201, 229]. Race hashing [230] is the state-of-the-art hash table over disaggregated memory. It achieves high performance by batching RDMA requests and achieves a high load factor by sharding data to multiple buckets with multiple hash functions. Sherman [201] and FG [229] are B+ trees for disaggregated memory. The former reduces lock contention with a hierarchical lock design and the latter achieves better performance with fine-grained tree node partitions. Different from existing works on designing and improving dedicated data structures, this thesis identifies and summarizes the general principles for high-performance data structure design. Besides, we design new data structures and algorithms for memory management and fault tolerance, and we further improve the performance of existing range indexes on DM.

□ **End of chapter.**

Chapter 3

Efficient Memory Management Data Structures and Algorithms

Outline

Memory management is critical to a high-performance memory-disaggregated storage system. Unfortunately, existing data structures and algorithms for memory management are unsuitable for disaggregated memory (DM) due to their severe amplifications in I/O numbers and the high concurrency control overhead. This chapter describes the design and implementation of Ditto, the first memory-disaggregated caching system that can efficiently manage memory in the memory pool. Specifically, Ditto proposes a two-level memory allocation scheme to achieve high-performance memory allocation and a client-centric caching framework to efficiently execute various caching algorithms. Moreover, a distributed adaptive caching scheme is proposed to achieve high cache hit rates by adapting to changing workloads and resources on DM.

3.1 Introduction

Efficient memory management is critical to achieving high performance for memory-disaggregated storage systems [192, 55]. First, existing memory-disaggregated storage systems adopt an out-of-place update manner [230, 119]. Memory needs to be allocated from the memory pool before objects can be inserted and updated, making memory allocation on the critical path that can affect performance [192, 230]. Besides, memory-disaggregated storage systems need to identify and keep hot objects in the expensive disaggregated memory pool with limited capacities [166, 139, 27, 169]. This requires executing caching algorithms with high performance and achieving high cache hit rates.

However, existing data structures and algorithms for memory management are constructed for monolithic-server-based in-memory storage systems. They suffer from severe I/O amplifications, high concurrency control overheads, and cannot achieve high cache hit rates on DM with loosely coupled and dynamically changing compute and memory resources.

Specifically, three challenges have to be addressed to achieve practical memory management on DM.

1) Centralized memory allocation. For in-memory storage systems over monolithic servers, memory is allocated and managed directly with the CPU on the same monolithic server. They use various data structures, *e.g.*, linked lists [107], trees [221], etc., to organize and quickly locate free memory blocks. Unfortunately, these data structures and algorithms are unsuitable for disaggregated memory due to their severe I/O amplifications and high concurrency control overhead. First, memory nodes (MNs) in the memory pool cannot handle the compute-heavy fine-grained memory allocation for variable-sized objects due to their weak compute power [192, 81]. Besides, allocating memory directly from the compute nodes (CNs) in the compute

pool suffers from multiple numbers of I/Os and high concurrency control overhead due to the large number of concurrent CNs and the multiple numbers of memory accesses to maintain memory allocation data structures [118].

2) Bypassing remote CPUs hinders the execution of caching algorithms. In-memory storage systems use various caching algorithms under different workloads [166, 27, 139]. Caching algorithms monitor the hotness of cached objects and select eviction victims by maintaining the hotness information in various caching data structures, *e.g.*, queues [93], heaps [169], etc. Since data access changes object hotness, existing caching algorithms rely on the CPUs of caching servers, where all data accesses are executed, to monitor object hotness and maintain caching data structures [139]. However, on DM, clients in the compute pool bypass CPUs in the memory pool when accessing objects. Evaluating object hotness becomes difficult due to the lack of a centralized hotness monitor on data paths. Evicting objects also becomes inefficient since caching data structures have to be maintained with multiple network I/Os from the compute pool, where the data accesses are executed. Moreover, supporting various caching algorithms for different workloads [166, 27] is even more difficult on DM since different caching algorithms evict objects with specified rules and tailored data structures [93, 193].

3) Changing resources affects hit rates of caching algorithms. Cache hit rates closely relate to the data access patterns [195] and the cache size [169]. On DM, both attributes change when dynamically adjusting compute or memory resources. Specifically, data access patterns change with the number of concurrent clients (*i.e.*, compute resources), and the cache size changes with the allocated memory spaces (*i.e.*, memory resources). As a result, the best caching algorithm that maximizes hit rate changes dynamically with resource settings. In-memory storage systems with fixed caching algorithms cannot adapt to these dynamic

features of DM and can lead to inferior hit rates.

We address these challenges with Ditto[1], an elastic and adaptive memory-disaggregated caching system. First, to achieve efficient remote memory management, Ditto employs a two-level memory management scheme that splits the memory management data structures into compute-light and compute-heavy components. The compute-light coarse-grained memory blocks are managed by MNs with weak compute power, and the compute-heavy fine-grained objects are handled by clients on CNs with strong compute capabilities. Second, we propose a client-centric caching framework to efficiently execute various caching algorithms on DM. The framework employs *distributed hotness monitoring* and *sample-based eviction* to achieve high performance. The distributed hotness monitoring uses one-sided RDMA verbs to record the access information from distributed clients in the compute pool, uses eviction priority to formally describe object hotness, and assesses objects' eviction priorities by applying priority calculation rules on the recorded access information. The sample-based eviction scheme selects eviction victims by sampling multiple objects and selecting the one with the lowest priority on the client side without maintaining remote data structures [166]. Since the key difference among different caching algorithms are their definitions of eviction priorities, various caching algorithms can be integrated by defining tailored priority calculation rules with little coding effort. Finally, we propose a distributed adaptive caching scheme to achieve high cache hit rates under dynamically changing resources. Ditto simultaneously executes multiple caching algorithms and uses regret minimization [69, 71, 220], an online machine learning algorithm, to perceive their performance and select the best one according to the current resource setting.

We implement Ditto and evaluate its performance with both

[1]DMC is a Pokémon that can arbitrarily change its appearance.

synthesized and real-world workloads [108, 184, 215]. Ditto is more elastic than Redis regarding resource efficiency and the speed of resource adjustments. On YCSB and real-world workloads, Ditto outperforms CliqueMap [182], the state-of-the-art key-value cache, by up to 9× and 3.6×, respectively. Moreover, Ditto can flexibly extend 12 widely-used caching algorithms with 12.5 lines of code (LOC) on average. The implementation of Ditto is open-source available at: `https://github.com/dmemsys/Ditto.git`.

The contributions of this work include the following:

- We design a two-level memory management scheme that leverages both MNs and CNs to efficiently allocate and manage the disaggregated memory pool.
- We propose a client-centric caching framework where various caching algorithms can be integrated flexibly and executed efficiently on DM.
- We propose distributed adaptive caching to provide high hit rates by selecting the best caching algorithm according to the dynamic resource change and various data access patterns on DM.
- We implement Ditto and evaluate it with various workloads. Ditto outperforms the state-of-the-art approaches by up to 9× under YCSB synthetic workloads and up to 3.6× under real-world workloads.

3.2 Challenges

This section elaborates on the challenges in terms of memory management and executing caching algorithms.

3.2.1 Remote Memory Allocation

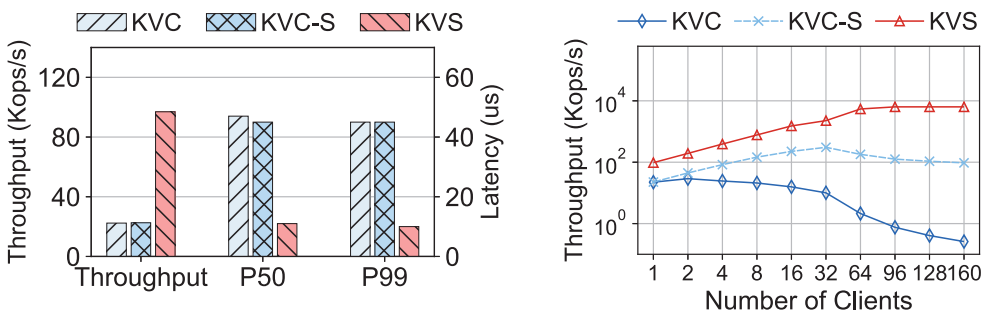
The key challenge of allocating memory from the disaggregated memory pool is where to execute the memory allocation computation, *i.e.*, maintaining memory allocation data structures. There are two possible memory management approaches on DM, *i.e.*, compute-centric ones and memory-centric ones [118].

The compute-centric approaches store the memory management metadata on MNs and allow clients to allocate memory by directly modifying the metadata with remote memory accesses. Unfortunately, such an approach suffers from severe I/O amplifications and concurrency control overhead. Specifically, maintaining memory allocation data structure typically requires multiple remote memory accesses, *e.g.*, popping free blocks from linked lists. Moreover, clients' access have to be synchronized since the memory management metadata are shared by all clients.

The memory-centric approaches manage all memory allocation data structures with the compute power on MNs. Such approaches are also infeasible since the weak memory-side compute power can be easily overwhelmed by the frequent fine-grained object allocation requests from clients. Although there are several approaches that conduct memory management on DM, they all target page-level memory allocation and rely on special hardware, *e.g.*, programmable switches [118] and SmartNICs [81], which are orthogonal to our problem.

3.2.2 Executing Caching Algorithms on DM

Existing caching algorithms are designed for *server-centric* in-memory storage systems where all data are accessed and evicted by the server-side CPUs in a centralized manner. Such a setting, however, no longer holds on DM since 1) memory-disaggregated storage systems are *client-centric*, where clients directly access



(a) Single-client performance.

(b) Multi-client throughput.

Figure 3.1: The cost of maintaining caching data structures on DM.

and evict the cached data in a CPU-bypass manner, and 2) the compute power in the memory pool of DM is too weak to execute caching algorithms on the data path. Two problems need to be addressed to execute caching algorithms on DM.

The first problem is how to evaluate the hotness of cached objects in the *client-centric* setting. Existing caching algorithms assess objects' hotness by monitoring and counting all data accesses [193, 18, 34]. The monitoring can be trivially achieved on server-centric in-memory storage systems since the CPUs of monolithic caching servers access all data. However, on DM, accesses to cached objects cannot be monitored either in the memory pool or on clients because 1) RDMA bypasses the CPUs in the memory pool, and 2) individual clients in the compute pool are not aware of global data accesses.

The second problem is how to efficiently select eviction victims on the client side. Caching algorithms maintain various caching data structures, *e.g.*, lists [193], heaps [18, 34], and stacks [92], to reflect the hotness of cached objects and select eviction victims based on these data structures. The data structures are maintained by the CPUs of in-memory storage servers on each data access since access changes object hotness. However, the maintenance of caching data structures has to be executed by clients in the compute pool since clients directly ac-

cess objects with one-sided RDMA verbs. Maintaining these data structures thus becomes inefficient due to the multiple I/Os on the critical path. Besides, locks are required to ensure the correctness of caching data structures under concurrent accesses [139]. System throughput will be severely bottlenecked by the microsecond-scale lock latency and the network contention caused by iteratively retying on lock failures [201].

To illustrate the problem of maintaining caching data structures, we compare the performance of a linked-list-based LRU key-value cache (KVC), a key-value cache with sharded LRU lists (KVC-S), and a key-value store (KVS) on DM [180] under the read-only YCSB-C benchmark [50]. All approaches use a lock-free hash table to index cached objects. KVC maintains a lock-protected linked list to execute LRU. KVC-S shards the LRU list into 32 sub-lists to avoid lock contention and sleeps 5 us on lock failures to reduce the wasted RDMA requests on lock failures. Figure 3.1a shows the throughput and latency of the three approaches with a single client, ruling out lock contention. The throughput of KVC and KVC-S is only 23% of that of KVS, and the tail latency is more than 4.5× higher due to the additional RDMA operations on the critical path of data accesses. Figure 3.1b shows their throughput with growing numbers of client threads. The throughputs of KVC and KVC-S drop with more than 32 client threads because the RNIC of the MN is overwhelmed by the useless `RDMA_CAS`es on lock-fail retries. The throughput of KVC-S drops more mildly due to the 5 us backoff on lock failures.

3.2.3 Dynamic Resource Changes Affect Hit Rate

Hit rates of caching algorithms closely relate to the data access patterns and the cache size [169]. However, both aspects are affected when dynamically adjusting compute and memory

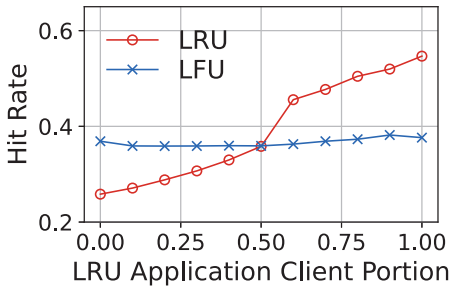


Figure 3.2: Hit rates under different numbers of clients under different applications.

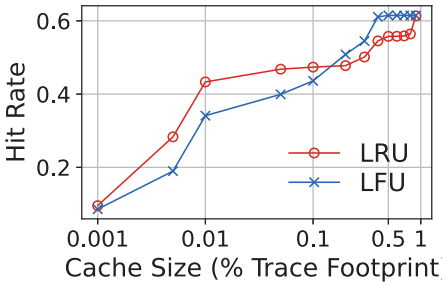
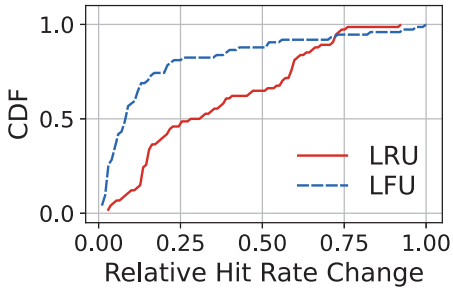
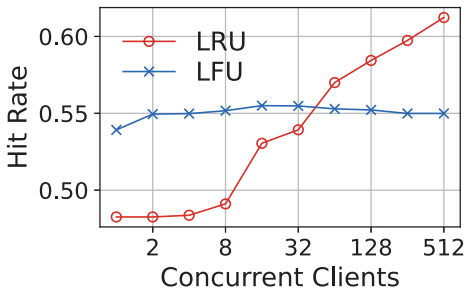


Figure 3.3: Hit rates of LRU and LFU on the same workload with different cache sizes.



(a) The CDF of relative hit rate changes on 74 workloads.



(b) Hit rates under different number of concurrent clients.

Figure 3.4: The effect of concurrent clients on hit rates.

resources, making the best caching algorithm that maximizes the hit rate changes accordingly. Since DM enables resources to be adjusted fleetly and frequently, the effect of changing resource settings is amplified. In-memory storage systems with fixed caching algorithms cannot adapt to these dynamic features of DM and can lead to inferior hit rates.

1) Changing compute resources affects hit rates. For memory-disaggregated storage systems, applications execute multiple client threads on CPU cores in the compute pool to access cached data in the memory pool. The access pattern on cached objects is the mixture of access patterns of all applications. The change in compute resources, *i.e.*, the number of client threads of an application, alters the overall mixture of

access patterns and affects the hit rate of individual caching algorithms in two ways.

First, the percentage of the data accesses of an application in the mixture changes with the number of client threads. The overall access pattern on the cached objects thus changes since applications have dissimilar access patterns [44]. Figure 3.2 shows the simulation result on a single machine with two applications under varying numbers of client threads. One application executes an LRU-friendly workload and the other executes an LFU-friendly one from the FIU block trace [108]. The hit rates of LRU and LFU are affected by the change of the compute resources in applications, where LFU exhibits a better hit rate when the LFU-friendly application has more compute resources and vice versa.

Second, the number of concurrent clients in an application changes the original access pattern of a workload due to concurrent executions. We simulate on 74 real-world workloads from Twitter [215] and FIU [108] with numbers of clients ranging from 1 to 512. Figure 3.4a shows the cumulative distribution function (CDF) of the relative hit rate change in these workloads. The relative hit rate change is calculated as $\frac{h_{max}-h_{min}}{h_{max}}$, where h_{max} and h_{min} are the highest and lowest hit rates of a workload under different numbers of clients. As we increase the number of client threads, 80% of workloads have 60% hit rate change in LRU and 21% in LFU. Meanwhile, the best caching algorithms on 36% of workload change with the varying number of concurrent clients. Figure 3.4b shows an example FIU trace where the hit rate of LFU performs better with a small number of concurrent clients but becomes inferior to LRU when the number of clients increases.

2) Changing memory resources affects hit rates. Changing memory resources leads to changing cache sizes of caching systems on DM. For individual workloads, the best

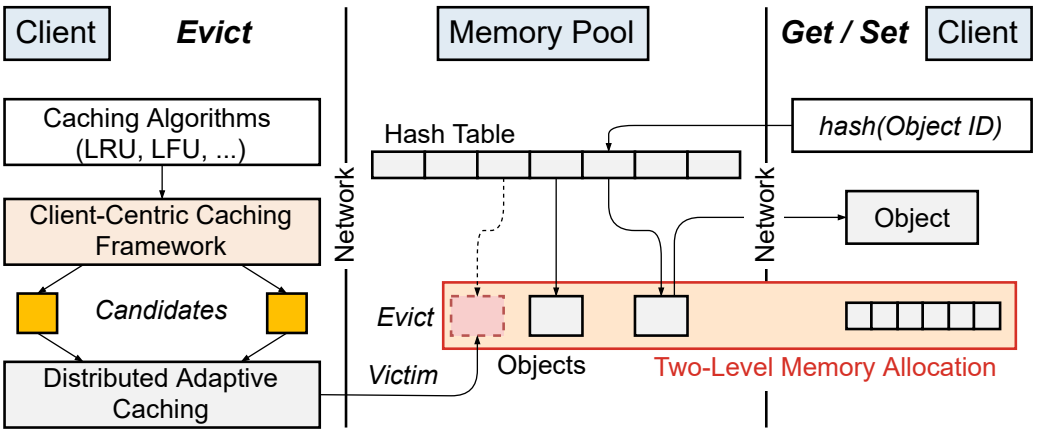


Figure 3.5: The overview of Ditto.

caching algorithm that maximizes the hit rate changes with cache sizes [169], *e.g.*, one workload can be LRU-friendly with a small cache size but becomes LFU-friendly under bigger cache sizes. Our simulation finds that the best algorithm changes in 22 of the 74 real-world workloads when the cache size changes. Figure 3.3 shows an example FIU trace where LRU performs better with small caches and LFU performs better with larger cache sizes.

Consequently, it is necessary for caching systems on DM to dynamically select the best caching algorithm according to the changing resource settings. However, achieving adaptivity is difficult on DM due to its decentralized and distributed nature, as we will introduce in § 3.3.4.

3.3 The Ditto Design

3.3.1 Overview

Figure 3.5 shows the overall architecture of Ditto. Ditto adopts a hash table to organize objects stored in the memory pool. The hash table stores pointers to the addresses of the cached objects. Following existing architectures of storage systems on DM [182,

180], applications execute on CNs and each application owns a local Ditto client as a subprocess. Each Ditto client has multiple threads on dedicated cores and applications communicate with Ditto clients with local shared memory to execute *Get* and *Set* operations. Under this architecture, applications can freely scale compute resources by adding or removing the number of threads and CPU cores assigned to Ditto. The adjustment on compute resources is independent against cached data because there is no need to increase or decrease the cache size in the memory pool when adding or reducing CPU cores.

Ditto clients execute *Get* and *Set* operations with one-sided RDMA verbs similar to RACE hashing [230], the state-of-the-art hashing index on DM. For *Get*s, a client searches the address of the cached object in the hash table and fetches the object from the address with two `RDMA_READ`s. For *Set*s, a client searches the slot of the cached object in the hash table with an `RDMA_READ`, writes the new object to a free location with an `RDMA_WRITE`, and atomically modifies the pointer in the slot with an `RDMA_CAS`.

Ditto adopts a two-level memory allocator (§ 3.3.2), a client-centric caching framework (§ 3.3.3) and a distributed adaptive caching scheme (§ 3.3.4) to achieve efficient memory management on DM. The two-level memory allocator allocates memory blocks with high performance to efficiently execute the out-of-place insert and update operations. The client-centric caching framework efficiently executes multiple caching algorithms on DM by selecting multiple eviction candidates of various caching algorithms. The distributed adaptive caching scheme uses machine learning to learn the characteristics of the current data access pattern and evicts the candidate selected by the caching algorithm that performs the best.

Table 3.1: The recorded access information.

Name	Description	Global?	Stateful?
size	Object size	✓	✗
insert_ts	Insert timestamp	✓	✗
last_ts	Last access timestamp	✓	✗
freq	The number of accesses	✓	✓
latency	Access latency	✗	✗
cost	Cost to fetch the object from the storage server	✗	✗

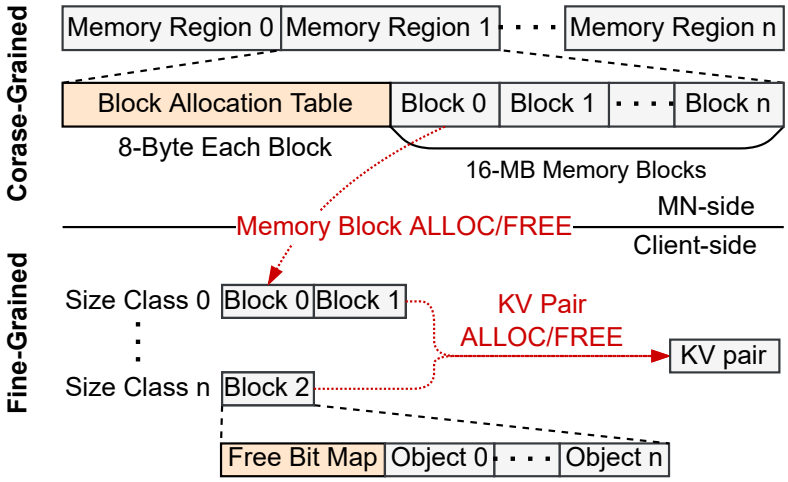


Figure 3.6: The two-level memory management scheme.

3.3.2 Two-Level Memory Allocation

As discussed in Section 3.2, the key challenge of DM management is that conducting the management tasks solely on clients or on MNs cannot satisfy the performance requirement of frequent memory allocation for KV requests. Ditto addresses this issue via a two-level memory management scheme. We split the server-centric memory management computations into compute-light coarse-grained allocation and compute-heavy fine-grained allocation and execute on MNs and clients accordingly.

To manage the huge memory space in the disaggregated mem-

ory pool, Ditto partitions the 48-bit memory space on multiple MNs. Similar to FaRM [55], Ditto shards the memory space on MNs into 2 GB memory regions and maps each region to an MN with consistent hashing [100].

Figure 3.6 shows the two-level memory allocation of Ditto. The first level is the coarse-grained MN-side memory block allocation with low computation requirements. Each MN partitions its local memory regions into coarse-grained memory blocks, *e.g.*, 16 MB, and maintains a block allocation table ahead of each region. The block allocation table records a client ID (CID) that allocates each block in a region. Clients allocate memory blocks by sending `ALLOC` requests to MNs. On receiving an `ALLOC` request, an MN allocates a memory block from one of its memory regions, records the client ID in the block allocation table, and replies with the address of the memory block. The second level is the fine-grained client-side object allocation. Specifically, each client manages the blocks allocated from MNs exclusively with a slab allocator [28]. The client-side slab allocators split memory blocks into objects of distinct size classes. An object is always allocated from the smallest size class that fits it.

The allocated objects can be freed by any client. To efficiently reclaim freed memory objects on client sides, Ditto stores a free bit map ahead of each memory block, as shown in Figure 3.6, where each bit indicates the allocation state of one object in the memory block. The free bit map is initialized to be all zeros when a block is allocated. To free an object, a client sets the corresponding bit to '1' in the free bit map with an `RDMA_FAA` operation. By reading the free bit map, clients can easily know the freed objects in their memory blocks and reclaim them locally. Ditto frees and reclaims memory objects periodically using background threads in a batched manner to avoid the additional RDMA operations on the critical paths of KV accesses. The correctness of concurrently accessing stored objects and re-

claiming memory spaces is guaranteed by checking the key and the CRC of each object whenever they are accessed [230].

3.3.3 Client-Centric Caching Framework

The client-centric caching framework addresses the challenges of evaluating object hotness and selecting eviction candidates when executing caching algorithms on DM.

First, to assess the hotness of cached objects, Ditto records objects' access information and decides objects' hotness by defining and applying priority functions to the recorded access information. Specifically, Ditto associates each object with a small metadata recording its global access information, *e.g.*, access timestamps, frequency, etc. The metadata is updated collaboratively by clients with one-sided RDMA verbs after each *Get* and *Set*. On the client side, Ditto offers two interfaces to integrate caching algorithms, *i.e.*, priority functions (`double priority()`) and metadata update rules (`void update()`). Both functions take the recorded metadata as input. The `priority` function maps the metadata of an object to a real value indicating its hotness. Since the key difference between caching algorithms is their definition of object hotness, various caching algorithms can be integrated by defining different priority functions with the `priority` interface. To support as many algorithms to be simply integrated with the `priority` interface as possible, we summarize the access information commonly used by existing caching algorithms [159] in Table 3.1 and record them in Ditto by default.

For advanced caching algorithms that require more access information, we allow algorithms to extend and define their own rules to update the metadata with the `update` interface. Listing 3.1 shows an example implementation of LRU-K [153]. LRU-K evicts objects with the smallest timestamp at its last K^{th} ac-

cess. We split the *last_ts* field into K timestamps with lower precision and construct a ring buffer with the $freq$ counter. If the object is accessed more than K times, then its priority is its last K^{th} access timestamp, which is indexed by $(freq - K + 1) \bmod K$. Otherwise, we return the *insert_ts* of the object to achieve FIFO eviction in the access buffer [93]. We resort to storing the modified timestamp of LRU-K with cached objects if we need to simultaneously execute LRU-K with caching algorithms that rely on *last_ts*, *e.g.*, LRU.

Second, to efficiently select eviction candidates of various caching algorithms on DM, Ditto adopts sampling with client-side priority evaluation. The overhead of maintaining expensive caching data structures is then avoided. Specifically, on each eviction, Ditto randomly samples K objects in the cache and applies the defined priority functions to the access information of the sampled objects. The eviction victim is approximated as the object with the lowest priority among K sampled objects.

However, such a framework suffers from severe amplifications in the number of I/O operations due to the multiple number of RDMA operations spent on recording access information and sampling objects. Ditto proposes a *sample-friendly hash table* and a *frequency-counter cache* to reduce the overhead of sampling objects and recording access information on DM.

Sample-friendly hash table

The sample-friendly hash table reduces the overhead of recording access information and sample objects on DM. Specifically, sampling objects on DM suffers from high access latency because multiple `RDMA_READs` are required to fetch the metadata of objects scattered in the memory pool. Moreover, updating access information affects the overall throughput because these additional RDMA operations consume the bounded message rate of RNICs in the memory pool.

Listing 3.1: The pseudocode of LRU-K.

```
def update(Metadata m):
    m.freq += 1
    idx = m.freq % K
    m.last_ts[idx] = current_timestamp()

def priority(Metadata m):
    if m.freq < K:
        return m.insert_ts
    idx = (m.freq - K + 1) % K
    return m.last_ts[idx]
```

The sample-friendly hash table co-designs the sampling process with the hash index to address these two problems. First, instead of storing all metadata together with objects, Ditto stores the most widely used metadata (*i.e.*, the default ones) together with the slots in the hash index but retains the metadata extensions required by advanced caching algorithms in objects. With the co-designed hash table, sampling can be conducted with only one `RDMA_READ` by directly fetching continuous slots with a random offset in the hash table. Second, Ditto reduces the number of I/Os on updating object metadata by organizing access information according to their update frequency. The well-organized access information enables multiple access information to be updated with a single `RDMA_WRITE`.

Hash table structure. Figure 3.7 shows the structure of the sample-friendly hash table. The hash table has multiple buckets with multiple slots. Each slot consists of two parts, *i.e.*, an atomic field and a metadata field. The atomic field is similar to the slot of Race Hashing [230], which is 8-byte in length and modified atomically with `RDMA_CAS`es when objects are inserted, updated, or deleted. The atomic field contains a 6-byte *pointer* referring to the address of the object, a 1-byte *fp* (fingerprint) accelerating object searching, and a 1-byte *size* recording the

size of the stored object. Similar to RACE hashing [230], we use a 1-byte size field and measure the sizes of objects in the granularity of 64B memory blocks. For large objects, Ditto stores the remaining data in a second memory block that links to the first one. The metadata field records the access information required by most caching algorithms, as summarized in Table 3.1. An additional *hash* field is recorded for the distributed adaptive caching scheme, which will be discussed in § 3.3.4.

Access information organization. Ditto organizes the stored access information in two ways to reduce the number of RDMA operations on metadata updates. First, Ditto reduces the number of access information that has to be included in the metadata by distinguishing local and global information. Global information has to be maintained collaboratively by all clients and thus must be included in the metadata. Local information can be decided locally by distributed clients and hence does not need to be included. The *latency* and *cost* are local information because we assume that the latency and cost are approximately the same among clients and can be estimated based on the size of objects and the latency and cost of accessing other objects. Second, global information is further classified into stateless and stateful information. Stateless information is updated by overwriting its old value, while stateful information is updated based on its old value. For instance, the *insert_ts* and *last_ts* are stateless because the old timestamps are no longer useful. The *freq* is stateful because it is always updated to increase by 1. Ditto groups the stateless information together in the metadata so that they can be updated with a single `RDMA_WRITE`. The stateful information is updated with `RDMA_FAA`s.

Frequency-counter cache

A client-side frequency-counter (FC) cache is proposed to further reduce the overhead of updating metadata. With the sample-

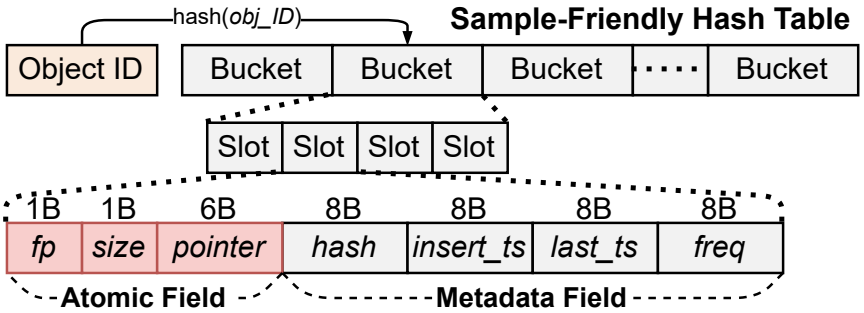


Figure 3.7: The sample-friendly hash table structure.

friendly hash table, updating metadata still requires two RDMA operations, *i.e.*, an `RDMA_WRITE` to update the stateless information and an `RDMA_FAA` to update the stateful *freq*. These RDMA operations consume the message rate of the RNIC and thus limit the overall throughput of Ditto. Besides, executing `RDMA_FAA` on DM is expensive due to the contention in the internal locks of RNICs [96]. The FC cache aims to reduce the number of `RDMA_FAA` on metadata updates.

The FC cache stems from the idea of write-combining on modern processors [53]. In modern processors, several write instructions in a short time window are likely to target the same memory region, *e.g.*, a 64-byte cache line. The write combining scheme adopts a buffer to absorb writes to the same region in a short time window and convert them into a single memory write operation to save memory bandwidths.

Similar to write-combining, Ditto employs an FC cache as the write-combining buffer. The FC cache contains entries recording the object ID, the address of the slot in the hash table and the delta value of the counter. We track the insert time of each cache entry to ensure that the frequency counters in the memory pool do not lag too much. Each time an object is accessed, its update to the frequency counter is buffered in the FC cache. The update to the remote frequency counter is deferred until a cache entry

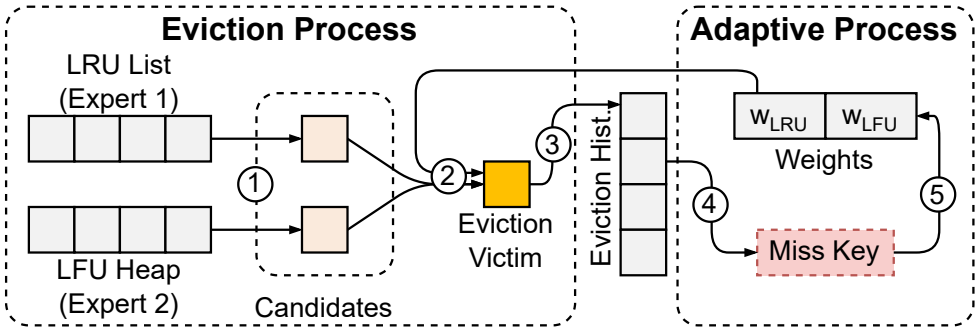


Figure 3.8: Adaptive caching on monolithic servers.

is evicted.

There are two situations when an entry will be evicted from the FC cache. First, if the space of the FC cache is full, an entry with the earliest insert timestamp will be evicted. Second, if the buffered delta value of an object is greater than a threshold t, the entry will be evicted. On entry eviction, the buffered counter value is added to the slot metadata with a single `RDMA_FAA` according to the recorded slot address, reducing the number of `RDMA_FAA` to up to $1/t$.

3.3.4 Distributed Adaptive Caching

Adaptive caching on monolithic servers is proposed to adapt to changing data access patterns in real-world workloads. Ditto proposes a distributed adaptive caching scheme to adapt to both changing workloads and dynamic resource settings on DM. The key problem is how to achieve adaptive caching in a distributed and client-centric manner on DM.

Recent approaches on monolithic servers formulate adaptive cache as a multi-armed bandit (MAB) problem [169, 195, 17, 138]. As shown in Figure 3.8, caching servers simultaneously execute multiple caching algorithms, named experts in MAB [17]. Each expert is associated with a weight, reflecting its performance in the current workload. The execution of the adap-

tive caching consists of an eviction and an adaptive process. During the eviction process, each expert proposes an eviction candidate according to their own caching data structures (①). Eviction victims are then decided opportunistically according to the weights of the experts (②), *i.e.*, candidates of experts with higher weights are more likely to be evicted. The metadata of the evicted object, *i.e.*, the object ID and the experts choosing it as a candidate, are inserted into a fix-sized FIFO queue named eviction history (③). During the adaptive process, existing approaches use *regret minimization* [69, 71, 220] to adjust expert weights. Specifically, when a missed object ID is found in the eviction history (④), the weights of experts deciding to evict the object are decreased (⑤). Intuitively, finding a missed object ID in the eviction history is a *regret* because a more judicious eviction decision could have rectified the cache miss [195].

Two challenges have to be addressed to achieve adaptive caching on DM. First, maintaining the global FIFO eviction history is expensive due to the high I/O amplifications when accessing remote data structures on DM, as mentioned in § 3.2. Second, managing expert weights on distributed clients is costly since clients need to be synchronized to get the updated weights.

The distributed adaptive caching scheme addresses these DM-specific challenges. First, Ditto evaluates multiple priority functions with the client-centric caching framework to simultaneously execute multiple caching algorithms on DM. Second, to avoid maintaining an additional FIFO queue on DM, Ditto embeds eviction history entries into the hash table with a lightweight eviction history (§ 3.3.4). Finally, to efficiently update and utilize expert weights on the client side, Ditto proposes a lazy weight update scheme to avoid the expensive synchronization among clients (§ 3.3.4).

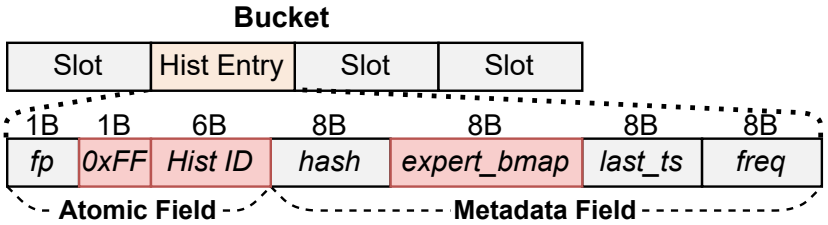


Figure 3.9: The structure of a lightweight history entry.

Lightweight eviction history

The eviction history on monolithic servers needs to maintain an additional FIFO queue and an additional hash index to organize and index history entries [169, 195]. The lightweight eviction history adopts two design choices to reduce I/O amplifications of maintaining these additional data structures on DM. First, it uses an *embedded history design* that reuses the slots of the sample-friendly hash table to store and index history entries. No additional space needs to be allocated and no additional hash index needs to be constructed for history entries. Second, the lightweight eviction history proposes a *logical FIFO queue with a lazy eviction scheme* to efficiently achieve FIFO replacement on history entries. No additional FIFO queue needs to be maintained to evict history entries.

Embedded history entries. Figure 3.9 shows the structure of an embedded history entry of the lightweight history. History entries are stored in the slots of the sample-friendly hash table with three differences. First, the *size* stores a special value (*0xFF*) to tag the slot as a history entry. We use *0xFF* instead of 0 since we use 0 to indicate empty slots. Second, the pointer field stores a 6-byte history ID instead of the address of the object. Finally, the history entry uses the *insert_ts* of the slot to store a bitmap indicating which experts have decided to evict the object (*expert_bmap*). Besides, each entry stores the hash value of the evicted object ID in the *hash* field to check if

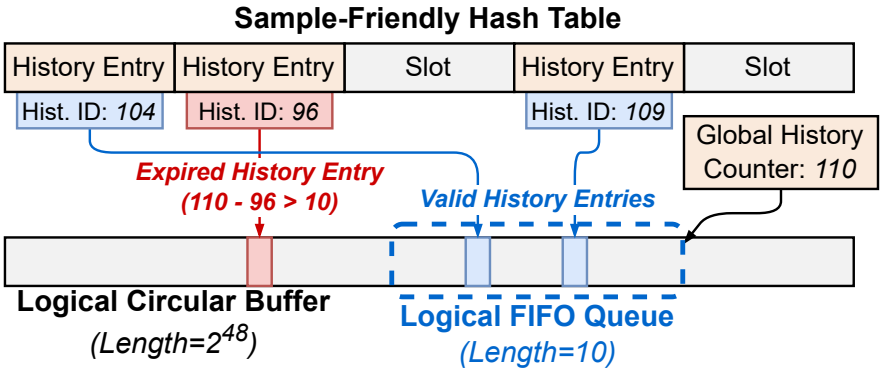


Figure 3.10: The logical FIFO queue structure.

a missed object is contained in the eviction history. The hash value is written to the metadata when the object is inserted into the cache and will not be modified until its history entry is evicted from the FIFO eviction history.

The logical FIFO queue. The logical FIFO queue simulates FIFO eviction without actually maintaining a FIFO queue on DM. It is constructed with a global history counter and the history IDs in history entries. The global history counter is a 6-byte circular counter that generates history IDs for new history entries. It is stored in an address in the memory pool known to all clients. The history IDs of history entries are acquired by atomically reading the global history counter and increasing it by one (*i.e.*, atomic fetch-and-add). As shown in Figure 3.10, the global history counter and history IDs of history entries can be viewed as locations in a logical circular buffer with 2^{48} entries. Combined with the size of the FIFO eviction history, the logical FIFO queue is then constructed, where the global history counter is the tail of the FIFO queue and the history IDs represent the location of history entries in the queue.

Figure 3.11 shows the operations of the lightweight history:

History insertion. A client inserts a history entry when it decides to evict a victim object from the cache. The client first

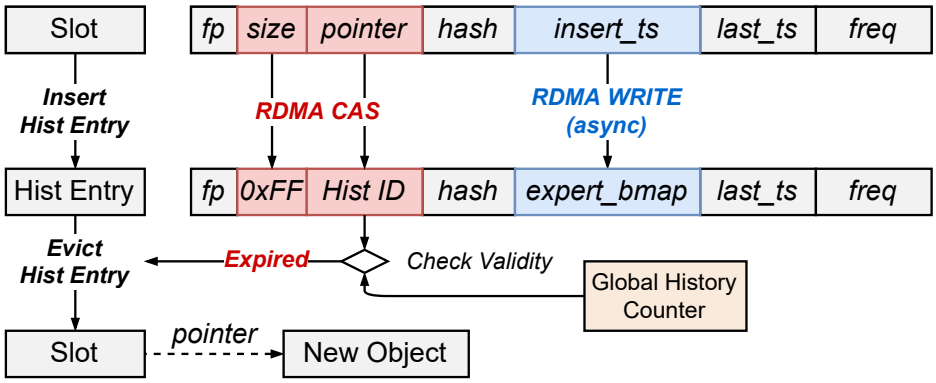


Figure 3.11: Inserting and evicting a history entry.

acquires a history ID by performing an `RDMA_FAA` on the global history counter, which atomically returns the current value of the counter and increases it by one. Then the client issues an `RDMA_CAS` to atomically modify the *size* and the *pointer* in the slot of the victim object to be *0xFF* and the acquired history ID, respectively. The expert bitmap is then asynchronously written to the *insert_ts* field of the slot metadata with an `RDMA_WRITE`.

Lazy history eviction. Ditto adopts a lazy eviction scheme to achieve FIFO eviction on history entries, *i.e.*, expired history entries are kept in the history for a while before their evictions. To prevent clients from accessing expired history entries, Ditto proposes a client-side expiration checking mechanism. Suppose the global history counter is v_1, the history ID is v_2, and the size of the FIFO history is l. If $v_1 > v_2$, the history entry is invalid when $v_1 - v_2 > l$. Otherwise, the history entry is invalid if $v_1 + 2^{48} - v_2 > l$, considering the wrap-up of the 48-bit global history counter. The actual evictions happen when inserting new objects into the cache. As shown in Figure 3.11, when inserting new objects, the expired slots are considered empty slots and are overwritten to be ordinary slots, which transparently evicts the history entry.

Regret collection. A regret is defined as a client finding an object to be missed in the cache but contained in the eviction

history. The embedded history entry makes collecting regrets the same process as searching objects in the cache. When a client searches for an object, it calculates the hash value of the object ID, locates a bucket based on the hash value, and iteratively matches the slots in the bucket to see if the pointed object has the same object ID as the target. During the process, clients also match the hash value of the encountered history entries in the bucket. Regrets can then be collected if the object has not been found but a history entry has a matching hash value.

Lazy expert weight update

Ditto formulates the problem of cache replacement as MAB and uses regret minimization to dynamically adjust the weights of experts. When a regret is found, *i.e.*, a missed object hits in the eviction history, the weights of the experts that evicted the object should be penalized. Suppose expert E_i made a bad eviction decision and the decision is the t-th entry in the eviction history. The weight of the expert is then updated to be $w_{E_i} = w_{E_i} \cdot e^{\lambda * d^t}$, where λ is the learning rate and d^t is the penalty. The penalty $e^{\lambda * d^t}$ is related to the position of the entry in the FIFO history because an older regret should be penalized less, where d is a fixed discount rate[2]. The challenge of updating weights on DM is that regrets are no longer collected and expert weights are no longer used in a centralized manner by monolithic caching servers. Updating and using expert weights from distributed clients incurs nonnegligible overhead due to the high synchronization overhead on DM [192].

The idea of the lazy weight update scheme is to let clients batch the regrets locally and offload the weight update lazily to the weak CPUs of MNs. In this way, the frequency of updating weights is reduced and the overhead of synchronization is

[2]Similar to [195], the discount rate is $0.005^{1/N}$, where N is the cache size.

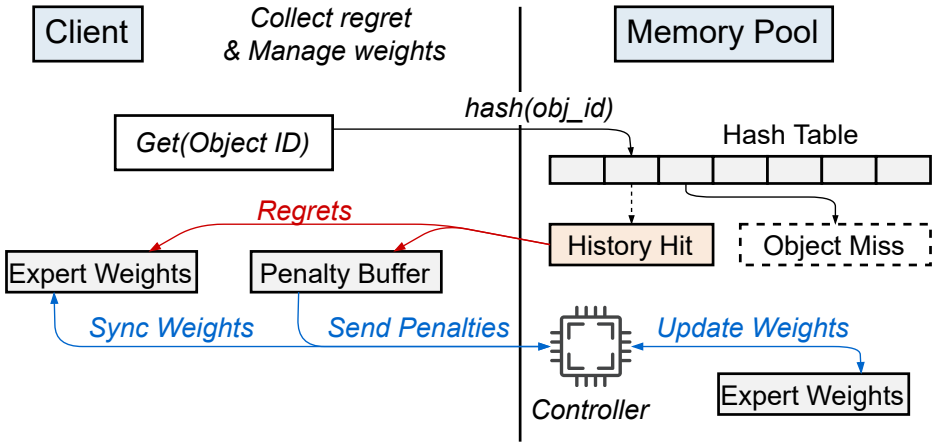


Figure 3.12: The lazy weight update scheme.

avoided. Meanwhile, the weak controller of MNs will not become a bottleneck due to the infrequent update.

Figure 3.12 shows the process of the lazy expert weight update scheme. Each client maintains expert weights locally to make eviction decisions. When a client finds a missed object hit in the eviction history, it applies the penalty to the local expert weights according to the history bitmap in the history entry. The penalties are recorded in a penalty buffer. When the number of buffered penalties exceeds a threshold, the client sends all the penalties to the controller of the memory node holding the expert weights with an RDMA-based RPC request. On receiving clients' penalties, the controller of the MN first applies the penalties to the global expert weights and then replies the updated global weights to clients.

To reduce the bandwidth consumption of transferring the penalties over the network, Ditto compresses the penalties using the attribute of exponential functions. Specifically, the sum of the penalties is stored in the penalty buffer and transferred to the MN instead of a list of individual penalties.

With the lazy weight update scheme, clients' eviction decisions are made on local weights, which are not always synchro-

nized with global weights. However, such asynchrony does not affect the adaptivity of Ditto, as shown in our experiments.

3.3.5 Discussions

Metadata extensions. As mentioned in § 3.3.3, Ditto stores extended metadata together with cached objects for advanced caching algorithms. In this situation, the extended metadata is stored as a metadata header ahead of each object. The `update` and `priority` functions take all metadata, *i.e.*, the default ones in the hash table and the extended ones in the metadata header, as input and call user-defined metadata update and priority calculation rules to deal with the extended metadata. After executing *Get* and *Set* operations, an additional `RDMA_WRITE` is required to update the metadata stored with objects. Finally, on cache eviction, additional `RDMA_READ`s are required to fetch the metadata header to calculate eviction priorities.

Security and fairness issues. Since Ditto clients and applications cooperate closely on the same CNs, it is possible that some malicious users can manipulate Ditto clients to make them disproportionately advantaged against other users' applications. We can enforce security techniques, *e.g.*, control flow integrity (CFI) [1], on standalone Ditto clients to prevent Ditto clients from being manipulated. We can also integrate the expected delaying technique [160] in Ditto clients to ensure that applications fairly share the cache.

3.4 Evaluation

The evaluation of Ditto answers the following questions:

- **Q1:** How elastic is Ditto compared with caching systems on monolithic servers?
- **Q2:** How efficient is Ditto in managing memory on DM?

Table 3.2: Real-world workloads used in the evaluation.

Workload	Workload Type	# Requests
IBM	Object Store	10 - 40 million
CloudPhysics	Block IO	50 million
Twitter-Transient	Transient key-value cache	10 million
Twitter-Storage	Storage key-value cache	10 million
Twitter-Compute	Compute key-value cache	10 million
webmail	Block IO	7.8 million

- **Q3:** How adaptive is Ditto to real-world workloads and the changing resources on DM?
- **Q4:** How flexible is Ditto in integrating various caching algorithms on DM?
- **Q5:** How does each design point contribute to Ditto?

3.4.1 Experimental Setup

Testbed. Without explicitly mentioning, we evaluate Ditto with 9 physical machines (8 CNs and 1 MN) on the Clemson cluster of CloudLab [57]. Each machine has two 36-core Intel Xeon CPUs, 256 GB DRAM, and a 100Gbps Mellanox ConnectX-6 NIC. All machines are connected to a 100Gbps Ethernet switch. In all our experiments, we use a single physical machine and use one CPU core to simulate the memory pool of DM with weak compute power [192, 180]. Ditto is compatible with memory pools with multiple MNs as long as the memory pool offers the required interfaces presented in §2.2. Besides, we use up to 32 cores on CNs, with each executing a client thread because they are on the same NUMA node with the RNIC.

Workloads. We evaluate Ditto with both YCSB synthetic workloads [50] and real-world key-value traces [184, 215, 108]. For YCSB synthetic workloads, we use 4 core workloads: A (50%

GET, 50% UPDATE), B (95% GET, 5% UPDATE), C (100% GET), and D (95% GET, 5% INSERT). For all four workloads, we pre-generate 10 million keys with 256-byte key-value pairs, load these generated keys by sharding them to all clients, and execute the corresponding workloads. The requests are generated with Zipfan distribution with $\theta = 0.99$. For real-world key-value traces, we use workloads from IBM [64], CloudPhysics [198], Twitter [215], and FIU [108], as shown in Table 3.2. The *IBM* trace is collected from IBM Cloud Object Storage [64]. We ignore traces with less than 10 million requests since they have too few unique objects and use all 23 traces in our experiments. The *CloudPhysics* dataset includes block I/O traces on VMs with different CPU/DRAM configurations [198]. We use the first 10 traces with more than 50 million requests to evaluate Ditto. For the Twitter traces, we randomly select three traces, *i.e.*, *Twitter-Compute*, *Twitter-Storage*, and *Twitter-Transient*, from a compute cluster, a storage cluster, and a transient caching cluster, respectively. The *webmail* trace is a 14-day storage I/O trace collected from web-based email servers. We use *webmail* as a representative FIU trace similar to existing approaches [169]. In our experiments, we randomly select traces to accelerate our evaluation to show the performance of Ditto in different use cases, *i.e.*, block IO, KV cache on different clusters, and object-store. We truncate traces to allow concurrent trace loading from 32 independent clients on a single CN.

Implementations. We implement Ditto with 20k LOCs. We use LRU and LFU, the two most widely used caching algorithms, as two experts in the distributed adaptive caching scheme. These two caching algorithms are chosen as adaptive experts since existing adaptive caching schemes have found that using a recency-based and a frequency-based caching algorithm can adapt to most workloads [169, 195]. For memory management, we use a two-level memory management scheme [180] so that clients can

dynamically allocate memory spaces in the MN. We pre-register all memory on the MN to its RNIC to eliminate the overhead of memory registration on the critical path of memory allocation.

Parameters. The parameters of Ditto include the number of samples, the size of lightweight eviction history, the threshold and size of the FC Cache, and the learning rate and the number of batched weight updates of distributed adaptive caching. Specifically, the number of samples affects the precision of approximating caching algorithms with sampling. We sample 5 objects on cache eviction according to the default value of Redis [166]. The size of the lightweight eviction history exhibits a tradeoff between the speed of adaptation and the metadata overhead. Setting the history size larger makes adaptation faster since more penalties can be collected during execution. In return, a larger history size requires more space to store history entries. We set the history size as the cache size (calculated in the number of objects) according to LeCaR [195]. The threshold of FC Cache can affect the precision of LFU. We set the FC cache threshold to 10 and set the FC cache size to 10MB according to our grid search. The superior hit rates in our experiments show that using 10 as the FC threshold does not affect hit rates much. Finally, we configure the learning rate of Ditto to be 0.1 and update global weights every 100 local weight updates according to our grid search.

Baselines. We compare Ditto with Redis [166], CliqueMap [182], and Shard-LRU. First, we use Redis, one of the most widely adopted in-memory caching systems that support dynamic resource scaling [166, 60], to show the elasticity of Ditto. Second, we use CliqueMap, the state-of-the-art RDMA-based KV cache from Google, to show the efficiency and adaptivity of Ditto. CliqueMap initiates `RDMA_READ`s on the client side to directly *Get* cached objects, and relies on server-side CPUs to execute *Set* operations. Since *Get*s involves only one-sided `RDMA_READ`s,

no access information can be recorded. Clients of CliqueMap record access information locally and send the information to servers periodically to enable servers to execute caching algorithms. We implemented an LRU (CM-LRU) and LFU (CM-LFU) version of CliqueMap according to its paper due to no open-source implementations. We disable the replication and fault-tolerance of CliqueMap to focus on comparing the execution of caching algorithms. Finally, we use Shard-LRU, a straightforward implementation of a caching system on DM, to show the effectiveness of the client-centric caching framework of Ditto. Clients of Shard-LRU maintain lock-protected LRU lists in the memory pool with one-sided RDMA verbs. We shard objects into 32 LRU lists according to their hash values and force clients to sleep 5 us on lock failures to mitigate lock and network contention. By default, we use one CPU core on MNs to simulate the poor compute power in the memory pool. Each CPU core on CNs exclusively runs a client thread.

3.4.2 Q1: Elasticity

To show the elasticity of Ditto, we run the same experiment as in § 2.3 and force Ditto to use the same amount of CPU or memory resources as Redis on the YCSB-C workload.

Compared with Redis, the elasticity of Ditto is improved in both resource utilization and speed of resource adjustments. First, due to the decoupled CPU and memory on DM, Ditto can adjust CPU cores and memory spaces separately in a fine-grained manner. Resources can be allocated precisely according to the dynamic demands of applications. Second, Ditto does not require data migration when adjusting resources, making the performance gain and resource reclamation more agile than Redis. The throughput of Ditto improves immediately from 5 Mops to 8.5 Mops with 32 more CPU cores added and resumes

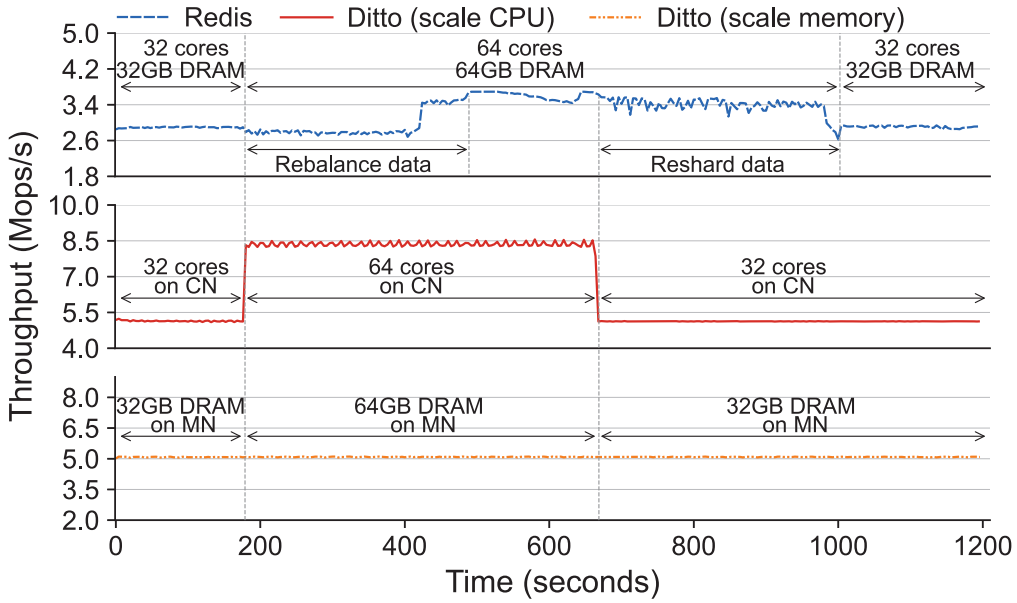
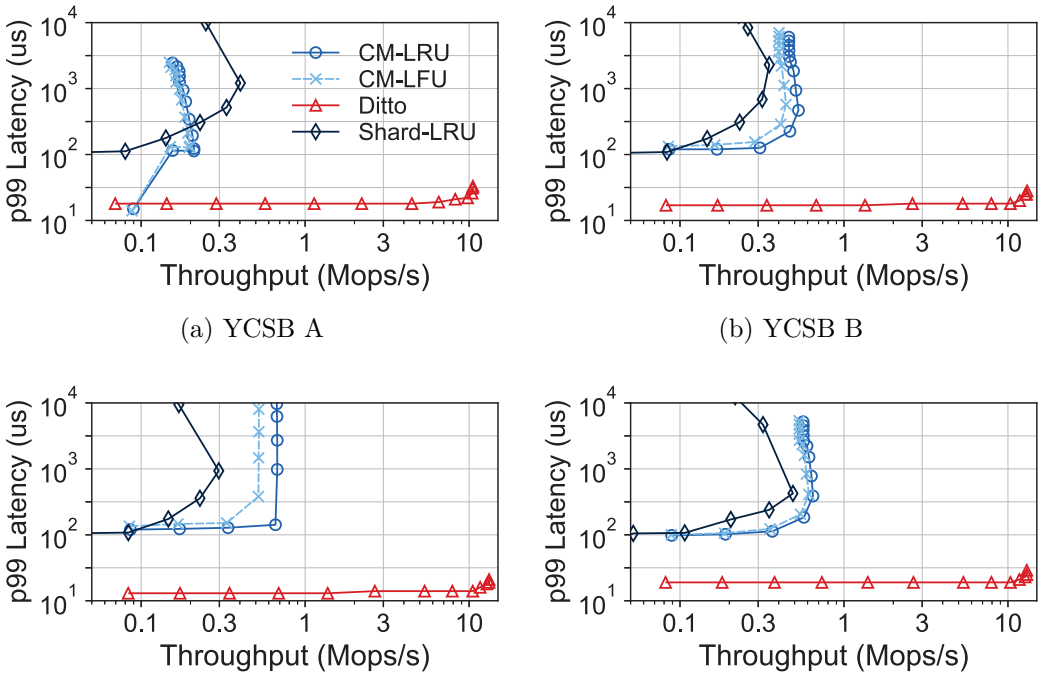


Figure 3.13: The throughput of Ditto when dynamically adjusting compute and memory resources.

immediately back to 5 Mops as we shrink the number of CPU cores back to 32. The throughput doesn't scale linearly as we add CPU cores due to the extra overhead of coroutine scheduling on CNs. The median latency stabilizes at 12 us and the 99th percentile latency fluctuates slightly around 14 to 21 us. As for adjusting memory spaces, the throughput stabilizes on 5 Mops and the tail latency stays on 14 us. Besides, the throughput of Ditto is more than 2 times higher than that of Redis during the experiment. This is because Ditto allows CPU cores to equally access all data, avoiding a single core becoming the performance bottleneck. However, Redis shards data to VMs, which makes the CPU core of some VMs bottleneck the throughput of the entire caching cluster on the skewed YCSB workloads.

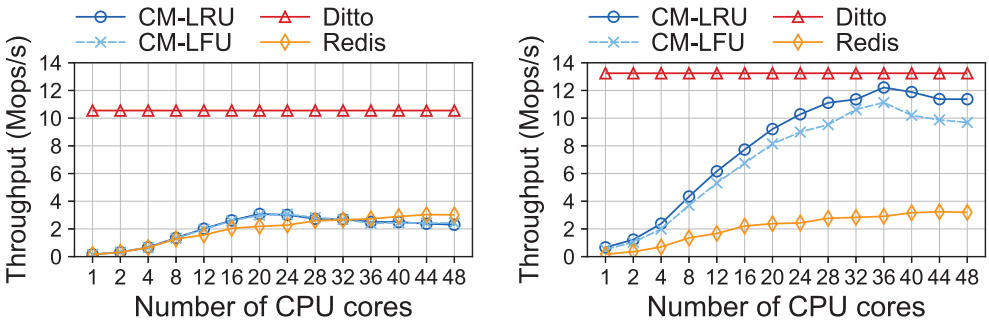
Besides, Ditto does not require more client-side computation than Redis. In the experiment, clients of Ditto consume 32 CPU cores on the CN. In contrast, clients of Redis consume on average



(a) YCSB A (b) YCSB B

(c) YCSB C (d) YCSB D

Figure 3.14: The throughput and tail latency of caching systems on DM.



(a) YCSB A (write-intensive) (b) YCSB C (read-only)

Figure 3.15: The throughput of CliqueMap, Redis, and Ditto with more CPU cores on MN.

36.3 CPU cores out of 128 assigned cores on two CNs. This is because the Redis client library spends CPU cycles to encapsulate and decapsulate data according to the Redis communication protocol and network protocols. Moreover, Ditto saves compute power regarding the overall CPU utilization since Redis servers consume an additional 32 cores on the MN.

3.4.3 Q2: Efficiency

To show that Ditto can efficiently manage memory on DM, we evaluate the throughput and tail latency of Shard-LRU, CliqueMap, and Ditto in the case of no cache misses on YCSB benchmarks. We vary the number of clients from 1 to 256, with each CN holding up to 32 clients.

As shown in Figure 3.14, Shard-LRU is bottlenecked by its remote lock contention even if the sharded LRU list and the 5 us back-off scheme mitigate the lock and network contention. The throughput of CliqueMap is limited by the weak compute power of MNs. For write-intensive workloads (YCSB A), the CPU of the MN is overwhelmed by frequent *Set*s. For read-intensive workloads (YCSB B, C, and D), the CPU of the MN is busy with merging the object access information received from clients. The overall performance is affected by the periodic synchronization of access information and the amplified network bandwidth when sending the access information from clients to the MN.

For all workloads, Ditto is bottlenecked by the message rate of the RNIC on the MN. It achieves 10.5, 13.1, 13.2, and 13.0 Mops respectively on YCSB A, B, C, and D workloads, which is up to 9× higher than Shard-LRU and CliqueMap. Compared with Shard-LRU, Ditto records the access information and selects eviction victims in a lock-free manner, eliminating the expensive lock overhead on DM. Compared with CliqueMap, Ditto accesses data and maintains access information with one-

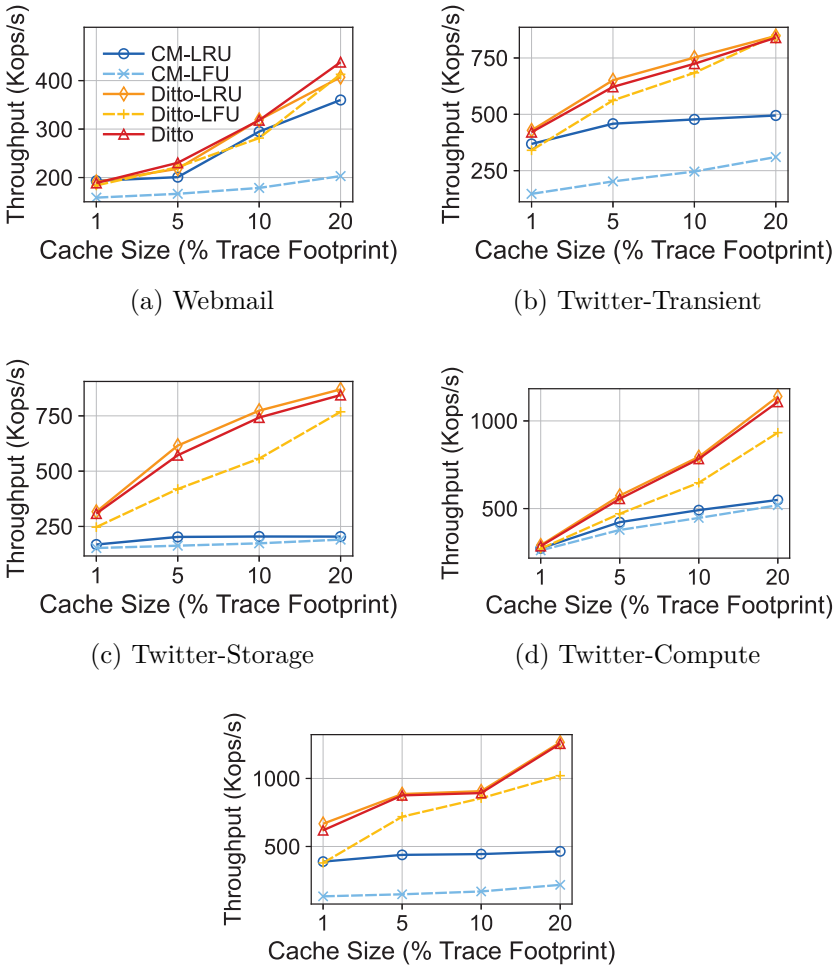
sided RDMA verbs, preventing the weak compute power on the MN from becoming the throughput bottleneck on both write-intensive and read-intensive workloads. However, Ditto performs worse than CliqueMap under the write-intensive YCSB-A workload with a single client, *i.e.*, the first point in Figure 3.14a. This is because the *Set*s of CliqueMap use only a single RTT, while Ditto needs three RTTs to search the remote hash table, read the object, and modify the pointer in the hash table.

Figure 3.15 shows the performance of CliqueMap, Redis, and Ditto under YCSB-A and YCSB-C workloads with increasing numbers of MN-side CPU cores under 256 clients. We shard the LRU list (and the LFU heap) of CliqueMap into 128 shards to avoid server-side lock contention. The throughput of Ditto stays the same since Ditto does not rely on compute power on MNs to execute data accesses. With the same compute resource in the compute pool, CliqueMap consumes more than 20 additional cores to get comparable performance with Ditto on YCSB-C. Ditto achieves 3.3× higher throughput than CliqueMap on the write-intensive YCSB-A workload since CliqueMap relies only on the server-side compute power to execute *Set* operations and maintain caching data structures. The throughputs of Redis on both workloads are bottlenecked by the CPU core of the hottest data shard due to the skewed YCSB workloads. Redis performs slightly better than CliqueMap on YCSB-A workload with more CPU cores since its sample-based eviction eliminates the overhead of maintaining caching data structures locally.

3.4.4 Q3: Adaptivity

Adapt to real-world workloads

To show the adaptivity of Ditto on real-world workloads with



(a) Webmail

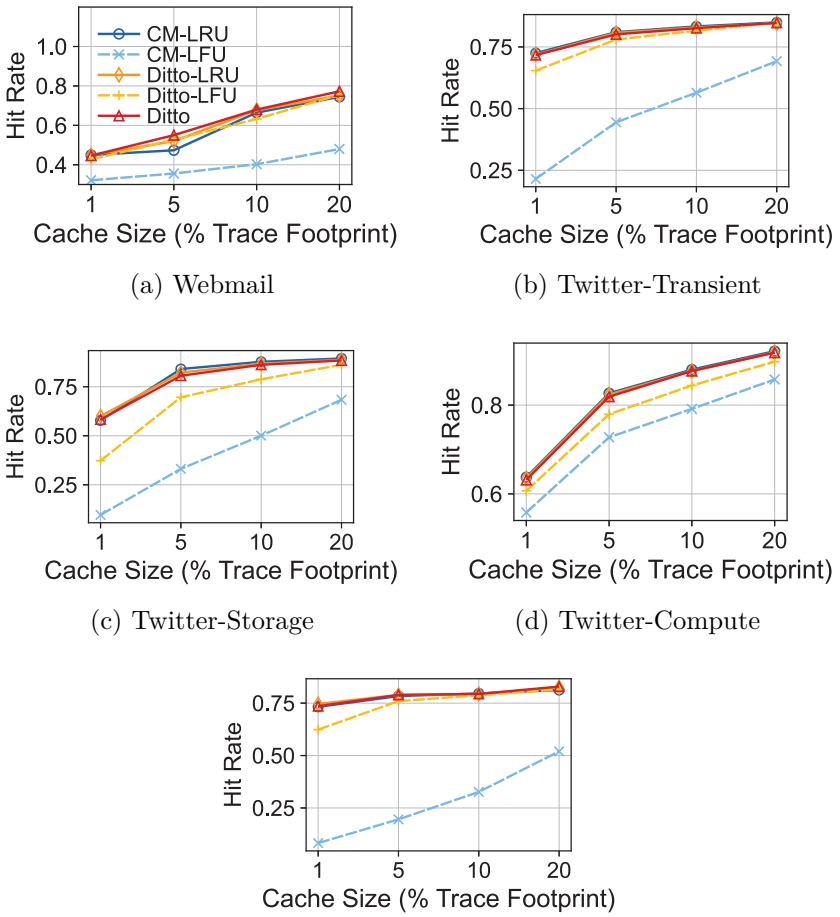
(b) Twitter-Transient

(c) Twitter-Storage

(d) Twitter-Compute

(e) IBM

Figure 3.16: Penalized throughputs under different real-world workloads.



(a) Webmail

(b) Twitter-Transient

(c) Twitter-Storage

(d) Twitter-Compute

(e) IBM

Figure 3.17: Hit rates under different real-world workloads.

different affinities of caching algorithms, we evaluate the throughput and the hit rate of real-world workloads with different cache sizes. For all traces, we use 256-byte object sizes and set cache sizes relative to the size of each workload's footprint, *i.e.*, all unique data items accessed, similar to [169]. For each workload, we use 64 clients to first execute 10 seconds to warm up the cache and then let all clients iteratively run the workload for 20 seconds to calculate the hit rate and the throughput. We use a penalized throughput to simulate real-world situations where caching systems cooperate with a distributed storage system. For each *Get* miss, we force clients to sleep for 500 us before inserting the missed object into the cache with *Set*. The penalty simulates the overhead of fetching data from distributed storage services and 500 us is selected according to the latency of the state-of-the-art distributed storage systems [210, 132, 156].

We compare Ditto with four baseline approaches. We use CM-LRU and CM-LFU to show the performance of precise LRU and LFU implementation with CliqueMap on DM. We introduce Ditto-LRU and Ditto-LFU to show the performance of Ditto with only a single caching algorithm.

Since Ditto is an adaptive caching framework that can execute various caching algorithms and dynamically adapt to the best one based on workloads and resource settings, the performance of Ditto largely depends on the candidate caching algorithms configured by users. We configure Ditto to execute LRU and LFU as examples to show its adaptivity. Under workloads that are friendly to either LRU or LFU, the performance of Ditto should be bounded by Ditto-LRU and Ditto-LFU and approach to the better one since it adaptively selects the better one among the two algorithms.

Figures 3.16 and 3.17 show the penalized throughput and the hit rates under five real-world key-value traces. In all five workloads, the hit rate and penalized throughput of Ditto can

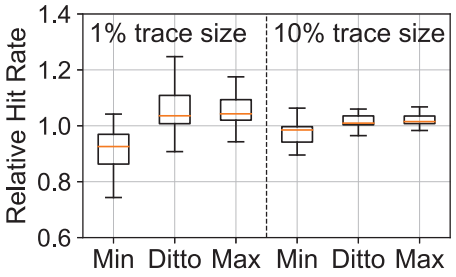


Figure 3.18: The relative hit rate of Ditto, Ditto-LRU, and Ditto-LFU on 33 workloads.

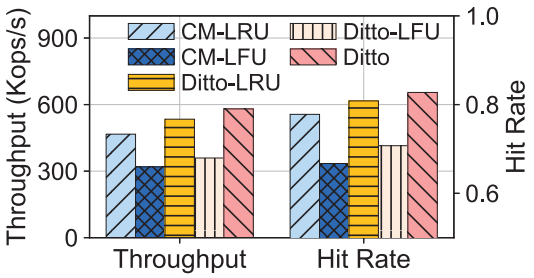


Figure 3.19: The penalized throughput and hit rate under a changing workload.

effectively approach the better one of Ditto-LRU and Ditto-LFU. Meanwhile, Ditto outperforms CliqueMap in all workloads due to higher hit rates and the higher throughput upper-bound. Particularly, the throughput of CliqueMap is bounded by the compute power on the MN under the Twitter workloads, where the hit rates are high. One exception is the throughput of CM-LRU in Figure 3.16a, which has comparable throughput with Ditto. This is because all approaches are bounded by the hit rate on the *webmail* workload and CM-LRU has a slightly lower hit rate compared with Ditto. For most of the workloads, the throughput of Ditto is lower than that of Ditto-LRU when their hit rates are the same due to the additional overhead of adaptive caching, *i.e.*, accessing and increasing the global history counter. However, the overhead is less than 5%, which is acceptable compared with the up to 63% performance gain of using an inferior caching algorithm, since users do not know in advance which caching algorithm performs better.

Figure 3.18 shows the box plot of relative hit rates of Ditto, max(Ditto-LRU, Ditto-LFU), and min(Ditto-LRU, Ditto-LFU) normalized over random eviction on 33 *IBM* and *CloudPhysics* workloads. The hit rate of Ditto significantly exceeds min(Ditto-LRU, Ditto-LRU) and approaches the box of max(Ditto-LRU, Ditto-LFU), showing the adaptivity of Ditto.

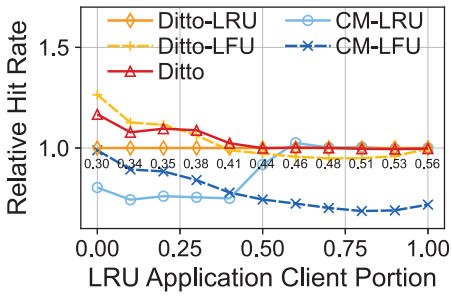


Figure 3.20: The relative hit rates under different proportions of clients assigned to LRU and LFU applications.

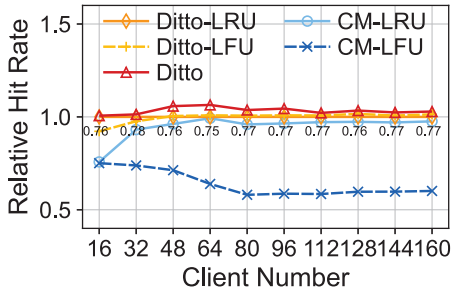


Figure 3.21: The relative hit rates of Ditto and CliqueMap when dynamically adding the number of concurrent clients.

Under changing workloads that iteratively switch between LRU- and LFU-friendly, Ditto should outperform both Ditto-LRU and Ditto-LFU. We show the performance of the four approaches on a synthetic changing workload used in [195]. The workload is synthesized to have four phases that periodically switch back and forth from being favorable to LRU to being favorable to LFU. As shown in Figure 3.19, Ditto outperforms all baselines on both penalized throughput and hit rate because only Ditto can adapt to workload changes.

Adapt to dynamic resource adjustments

To show the adaptivity of Ditto on DM, we evaluate its hit rates with dynamically changing compute and memory resources on the same workload as Figures 3.2, 3.3, and 3.4b, *i.e.*, *webmail*.

Adapt to changing compute resources. Figure 3.20 shows the relative hit rates normalized to Ditto-LRU under different proportions of clients allocated to two applications with LRU and LFU access patterns. The hit rate of Ditto-LFU is higher when the LRU portion is less than 0.4, while Ditto-LRU performs better when the LRU portion grows higher. The hit rate of Ditto is higher than that of Ditto-LRU with a low LRU por-

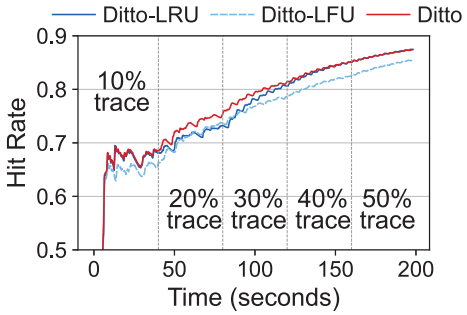


Figure 3.22: The hit rate under dynamic cache sizes.

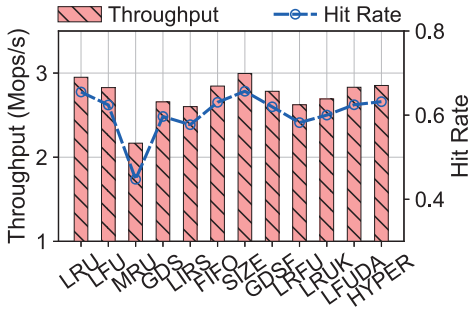


Figure 3.23: The throughput and hit rates of 12 algorithms.

tion and becomes close to Ditto-LRU with a high LRU portion, indicating the adaptivity of Ditto. Besides, Ditto can adapt to the change of access pattern when multiple clients concurrently execute the same workload. Figure 3.21 shows the relative hit rates of Ditto and CliqueMap normalized to Ditto-LRU under dynamically increasing numbers of concurrent clients[3]. The hit rate of Ditto stays above the hit rates of both Ditto-LRU and Ditto-LFU because there are access pattern changes in the *webmail* workload, and only Ditto can adapt to these changes.

Adapt to changing memory sizes. Figure 3.22 shows the hit rate of Ditto when we dynamically increase the memory space. The hit rate of Ditto approaches Ditto-LRU for most cases, outperforming Ditto-LFU. When the cache size is 20% and 30% footprint size, the hit rate of Ditto-LFU exceeds Ditto-LRU. Ditto performs better than both approaches because it can adaptively adjust its algorithm according to the affinity of caching algorithms on different cache sizes.

[3]The absolute hit rates in Figures 3.18, 3.20, and 3.21 can be found in our open-source repository.

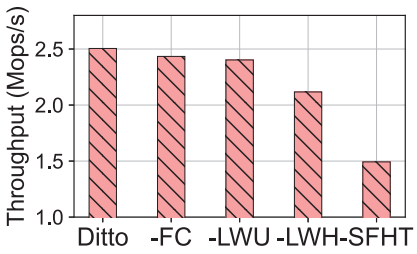


Figure 3.24: Contributions of different techniques on the *webmail* workload.

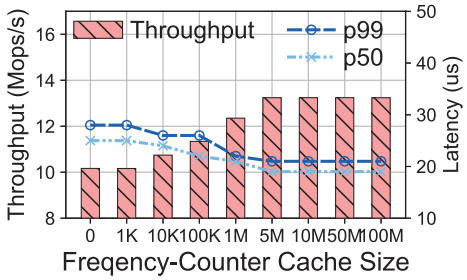


Figure 3.25: The YCSB-C performance of Ditto with different FC Cache sizes.

Table 3.3: LOCs of different caching algorithms on Ditto.

Algs.	LRU	LFU	MRU	GDS	LIRS	FIFO
LOC	9	9	9	14	12	9
Algs.	SIZE	GDSF	LRFU	LRUK	LFUDA	HYPERBOLIC
LOC	9	14	17	23	14	11

3.4.5 Q4: Flexibility

To show that Ditto can flexibly integrate various caching algorithms, we integrate 12 commonly used caching algorithms into Ditto and evaluate their throughput, hit rate, and coding effort. Since evaluating the feasibility of executing different caching algorithms is independent of workloads, we only show the throughput and hit rates on the *webmail* workload in Figure 3.23. Among all the algorithms, SIZE exhibits the best throughput and hit rate, while MRU exhibits the worst. All these algorithms can be easily implemented in Ditto with less than 23 lines of code, as shown in Table 3.3.

3.4.6 Q5: Contribution of Each Technique

We show the contribution of techniques proposed in the chapter by gradually disabling each technique of Ditto. Due to the space

limit, we show the performance of different techniques without miss penalties on the *webmail* workload in Figure 3.24. Ditto performs similarly on other workloads and more results can be found in our open-source repository. The sample-friendly hash table (SFHT) improves the overall throughput by 42% since it reduces the number of RDMA operations on data paths when updating the access information and sampling objects. The lightweight history scheme (LWH) improves the throughput by 13% due to the reduced number of RTTs when maintaining eviction history. Finally, the lazy weight update scheme (LWU) and the frequency-counter cache (FC) contribute to 4% of the overall throughput because the reduced number of RDMA requests saves the message rate of the RNICs on MNs.

Figure 3.25 shows the performance of Ditto under the YCSB-C benchmark with 256 clients and different FC cache sizes. We limit FC cache size in MB since the size of each cache entry varies with the size of its recorded object ID. We only show the result under YCSB-C due to the space limit. Ditto performs similarly on other workloads and more results can be found in our open-source repository. The throughput increases from 10 Mops to 13.2 Mops with increased sizes of the FC cache since more `RDMA_FAA`s can be cached locally to save the message rate of RNICs. The tail latency drops from 28 us to 21 us due to the reduced number of RDMA operations and less contended network. Also, the performance gain of the FC cache becomes insignificant when the size of the FC cache exceeds 5 MB, indicating that the FC cache can improve overall performance with small additional memory consumption on clients.

Figure 3.26 shows the performance of Ditto under YCSB-A write-intensive and YCSB-C read-only workloads with different memory allocation techniques. We execute this experiment with the following experiment setting due to the lack of the same nodes on CloudLab. We use 16 CNs and 2 MNs, each equipped

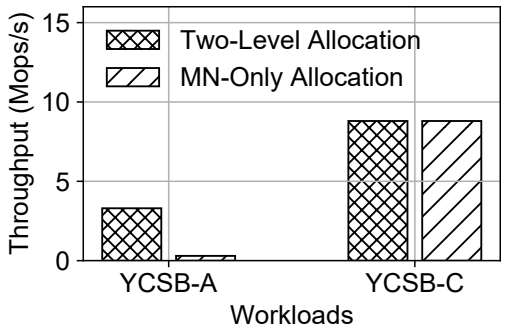


Figure 3.26: The throughput of Ditto with different memory allocation methods.

with an 8-core Intel Xeon E5-2450 processor, 16 GB DRAM, and a 56 Gbps Nvidia ConnectX-3 IB RNIC. The throughput of Ditto is lower than the previous setting since the RNIC has lower bandwidth. To show the effectiveness of the two-level memory allocation scheme, we compare Ditto with an MN-centric memory allocation scheme, as shown in Figure 3.26. The YCSB-A throughput drops 90.9% due to the limited compute power on MNs, while the YCSB-C throughput remains the same since no memory allocation is involved in the read-only setting.

3.5 Related Work

In-Memory Caching Systems. Many approaches aim at improving the performance of Memcached [139] and Redis [166], the two most popular in-memory caching systems. Some [42, 43, 172] optimize the hit rate under objects of varying sizes. Others [165, 148, 216, 66, 126] improve memory efficiency and overall throughput. The work closest to Ditto is CliqueMap [182], an RDMA-based caching system. It uses one-sided `RDMA_READ` for *Get* operations and RPC for *Set* operations, improving the throughput due to the higher bandwidth and CPU-bypass nature of one-sided `RDMA_READ`. However, all these approaches are

designed and optimized for monolithic servers, which inevitably inherit the elasticity issues of monolithic servers. Ditto exhibits better elasticity by leveraging the hardware benefits of DM.

RDMA-Based KV stores. There are two types of RDMA-based KV stores, *i.e.*, server-centric and hybrid ones. The former uses RDMA to construct fast RPC primitives and rely on server CPUs to access data [97, 95, 55, 126]. The latter uses one-sided RDMA verbs to execute *Get* operations and relies on server CPUs to execute *Set* operations [207, 206, 143]. Compared with these approaches, Ditto achieves efficient in-memory caching without relying on server-side CPUs. Besides, the design of Ditto is not limited to RDMA. Other interconnects are also compatible.

Caching Algorithms. Caching algorithms distinguish the hotness of objects using recency [193, 222], frequency [59] and other access information [27], or combining various information together [25, 27, 34, 18] to get higher hit rates. Recently, there are many machine-learning-based adaptive caching algorithms [138, 17, 169, 195]. Among them, CACHEUS [169] is the most related. It uses regret minimization to adaptively select a better caching algorithm. However, all these caching algorithms are designed for server-centric caching systems to optimize specific workloads. Ditto, on the one hand, is designed for caching systems on DM where clients directly access data without involving CPUs in the memory pool. On the other hand, Ditto is an adaptive caching framework where multiple caching algorithms can be integrated and adaptively selected according to workload and resource change.

Disaggregated Memory Management. MIND [118] and Clio [202] are the two state-of-the-art memory management approaches on DM. But they both rely on special hardware to manage memory spaces. The two-level memory management of Ditto resembles the hierarchical memory management of The

Machine [113, 67]. The difference is that Ditto focuses on fine-grained fine-grained object allocation with commodity RNICs, while The Machine relies on special SoCs and directly manages physical memory devices.

3.6 Summary

This chapter introduces efficient memory management data structures and algorithms for memory-disaggregated storage systems. We propose a two-level memory allocator to efficiently allocate memory space, and a client-centric caching framework to efficiently execute various caching algorithms. Both approaches are the joint effort of data structure and algorithm design. We reduce the I/O amplifications, optimize concurrency control, and adapt to the asymmetric compute capabilities to achieve high performance. We integrate our proposed memory management data structures and algorithms in Ditto, the first memory-disaggregated caching system. Experimental results show that Ditto outperforms the state-of-the-art caching system on monolithic servers by up to 9× on YCSB synthetic workloads and 3.6× on real-world key-value traces.

□ **End of chapter.**

Chapter 4

A High-Performance Range Index Data Structure

Outline

Range indexes are fundamental building blocks for memory-disaggregated storage systems. A range index can query both single keys and all keys within a given range. Unfortunately, existing range indexes on disaggregated memory (DM) treat remote memory as high-performance disks and use B+ trees as their underlying data structures. Consequently, they suffer from severe amplifications in I/O sizes since B+ trees sacrifice I/O sizes to reduce the number of I/O operations. The performance for the B+-tree-based range indexes on DM is suboptimal since accessing the coarse-grained B+ tree nodes wastes the bounded bandwidth of the memory pool. This chapter introduces SMART, a high-performance range index data structure tailored for DM that achieves minimal I/O size amplifications. Moreover, SMART also achieves high-performance concurrency control and adapts to DM with asymmetric compute power.

4.1 Introduction

Range indexes are fundamental building blocks for memory-disaggregated storage systems to conduct range queries [201, 229]. Existing approaches treat the disaggregated memory pool as disks and use B+ trees as range indexes [201, 229] to reduce the number of I/O operations to access remote memory. However, B+ trees sacrifice I/O sizes to reduce the I/O numbers, resulting in suboptimal performance due to the severe amplifications in I/O sizes.

Specifically, B+ tree nodes are coarse-grained. When reading or writing an key-value object, one should traverse B+ trees with coarse-grained tree nodes. Multiple irreverent keys, pointers, and objects are fetched from the memory pool to the compute pool through network I/O, inevitably amplifying the network bandwidth consumption. As the network bandwidth is generally the bottleneck of disaggregated memory (DM) [96], the amplified bandwidth consumption leads to low overall throughput and high access latency. Our experimental study shows that the I/O size amplification can dramatically degrade the throughput of Sherman [201], the state-of-the-art B+ tree index on DM. The throughput is 10.8× lower than the theoretical bound of RNICs under the YCSB workloads [50].

This chapter attacks the severe I/O size amplifications in existing range index data structures on DM. We propose that radix trees are more suitable to serve as range indexes than B+ trees on DM due to their better I/O efficiency. Compared with B+ trees, radix trees have smaller I/O size amplifications since they do not store the entire keys in internal nodes. Moreover, radix trees can further reduce I/O size by adaptively adjusting the sizes of their internal nodes, *i.e.*, adaptive radix trees (ARTs) [120]. However, constructing radix trees on DM introduces new challenges in terms of concurrency control and addi-

tional amplifications in I/O numbers.

(1) Expensive and complicated concurrency control. Existing ARTs rely on locks to ensure correctness under concurrent accesses [121]. However, on DM, locks are more expensive than in local memory due to an order of magnitude higher remote memory access latency. In addition, compute-side caches are required on DM to reduce operation latency. Traditional lock-free read-copy-update (RCU) schemes for radix trees introduce frequent changes in the addresses of cached nodes, leading to severe cache thrashing.

(2) Redundant I/Os deteriorate the throughput. Using radix trees generates multiple small-sized remote memory read and write I/Os when traversing and modifying the tree data structure. Many of these I/Os are redundant when multiple clients on the same compute node concurrently traverse the tree. Since RNICs in the disaggregated memory pool have bounded IOPS (I/O per second) [186], these redundant I/Os can waste the limited IOPS and result in suboptimal performance.

To address the above challenges, we propose **SMART**, a di**S**aggregated-me**M**ory-friendly **A**daptive **R**adix **T**ree. First, for better concurrency control, we present a *hybrid ART concurrency control* scheme with a lock-free internal node design and a lock-based leaf node design to achieve high performance without cache thrashing. To mitigate the amplifications in I/O numbers, we propose a *read delegation and write combining (RDWC)* technique to reduce computing-side redundant I/Os.

We implement SMART from scratch and evaluate it using the YCSB benchmark [50]. Compared with Sherman [201], the state-of-the-art B+-tree-based range index on DM, SMART achieves up to 6.1× higher throughput and 1.4× lower latency for typical write-intensive workloads and 2.8× higher throughput with similar latency for read-only workloads. The code of SMART is available at `https://github.com/dmemsys/SMART`.

In summary, this chapter makes the following contributions:

- We propose that radix trees are better range index data structures on DM, based on our theoretical analysis and experimental results.
- We present the first memory-disaggregated radix tree, SMART, with a hybrid ART concurrency control scheme and a read-delegation and write-combining technique.
- We implement SMART and evaluate it using YCSB workloads [50]. The evaluation results demonstrate the efficacy and efficiency of SMART.

4.2 Background

4.2.1 B+ Trees on Disaggregated Memory

Existing range indexes on DM are variants of the B+ tree, including FG [229] and Sherman [201]. FG is the first RDMA-based index supporting DM. It uses a B-link tree structure and completely leverages one-sided verbs to perform index operations, with RDMA-based spin locks for concurrency control. Since FG directly ports the spin-lock-based concurrency control and B-link tree node designs on monolithic servers to DM, its performance suffers from severe network contention on lock retries and I/O amplifications for write operations.

Sherman [201] is the state-of-the-art B+ tree on DM that addresses the network contention and write amplification issues of FG. First, it addresses the network contention on lock-fail retires with a *hierarchical on-chip lock (HOCL)* scheme. The network requests on lock-fail retries are reduced with a local lock table shared among clients on the same CN. Second, it mitigates the write amplification by allowing fine-grained modification to B+

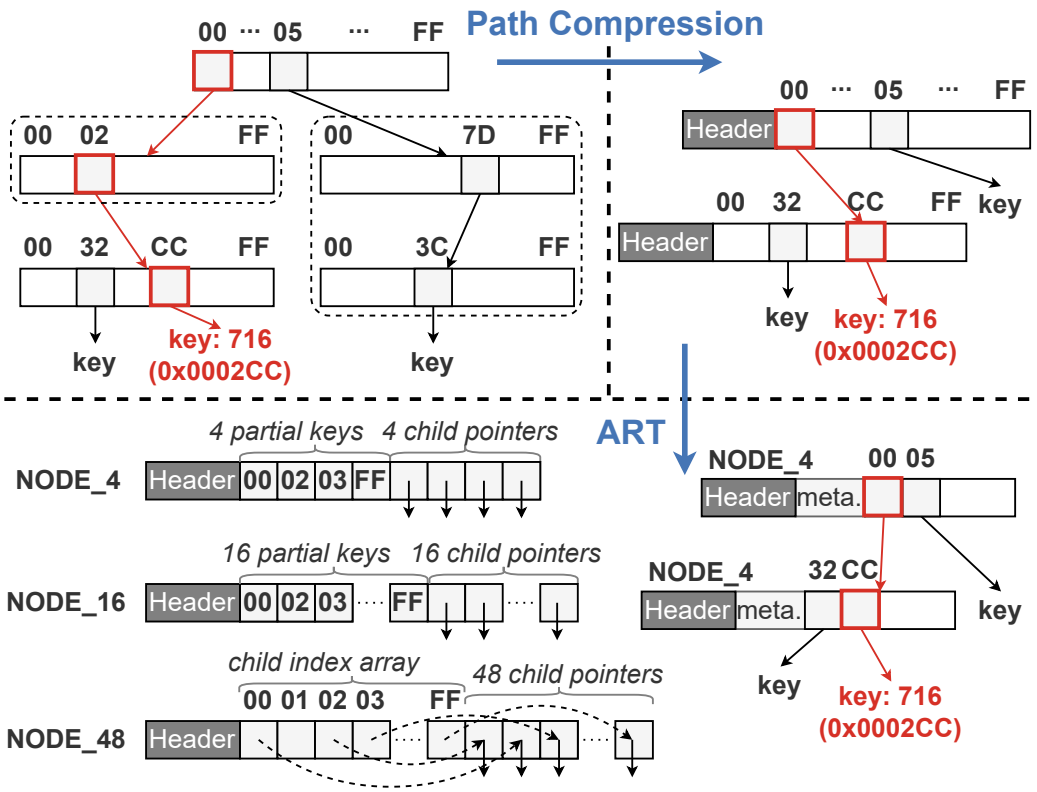


Figure 4.1: The optimization process from the basic radix tree to ART. For clarity, hexadecimal partial keys are shown. `NODE_256` is simply an array of 256 pointers, which is not shown due to the space limitation.

tree nodes with a *two-level version mechanism*. Therefore, Sherman achieves much better performance than FG. Unfortunately, the throughput of Sherman is still limited by its lock-based concurrency control and I/O size amplifications for read operations, which we will analyze in Section 4.3.

4.2.2 Radix Tree

The radix tree is a widely adopted tree index structure. It stores the segmented key in the search path over the tree rather than storing the whole key in the internal node. Specifically, each internal node in the radix tree consists of an array of child pointers. Each pointer is associated with a segment of bits of the whole

key, called *partial key*, as shown in Figure 4.1.

Path compression. Path compression is an optimization method for the radix tree to reduce tree height by removing internal nodes that only contain one single child node, and can be implemented in three ways [120]: ***1) The optimistic method*** directly drops the partial keys in removed nodes and stores a depth value to ensure the subsequent traversal process instead. ***2) The pessimistic method*** stores all the partial keys of removed nodes in the header of the subsequent node. ***3) The hybrid method*** integrates the two methods above by storing partial keys into the fixed-sized header of the subsequent node, together with a depth value to ensure the subsequent traversal if some partial keys overflow from the header.

Adaptive radix tree (ART). ART [120] is the state-of-the-art variant of the 8-bit-span radix tree, designed to optimize the memory utilization of traditional radix trees. Traditionally, an internal node of a radix tree has all 256 pointers representing all possible partial keys. Many pointers are empty due to the sparse key distribution [120], wasting memory space in these internal nodes. ART addresses the issue by proposing four variant internal node sizes with different numbers of pointers, *i.e.*, 4, 16, 48, and 256. It dynamically chooses the best-fit internal node structure to reduce memory consumption. As for concurrency control, ART is synchronized using a lock-based algorithm, *i.e.*, the *read-optimized write exclusion (ROWEX)* protocol [121], which suffers from severe lock contentions when executing on DM.

4.3 Analysis of Tree Indexes Built on DM

In this section, we first theoretically and experimentally compare B+ trees with a vanilla ART (§ 4.3.1). We then present the challenges of designing ART on DM (§ 4.3.2).

All the experiments in this section are conducted with 8 CNs

Table 4.1: Read and write amplification factors of different trees.

	ART	B+ Tree	Sherman
Read	$\frac{M_1+E}{E} = 1.10$	$\frac{M_2+S\cdot E}{E} = 32.7$	$\frac{M_2+S\cdot(M_3+E)}{E} = 33.0$
Write	$\frac{M_1+E}{E} = 1.10$	$\frac{M_2+S\cdot E}{E} = 32.7$	$\frac{M_3+E}{E} = 1.01$

and 1 MN, each equipped with a 100Gbps Mellanox ConnectX-6 RNIC. Each CN launches 32 clients with one shared 600MB cache. We use YCSB workloads [50] with 60 million entries, 32-byte string keys, and 64-byte values, which is typical in real-world workloads [215, 21].

4.3.1 Motivations: B+ Tree vs. ART on DM

The main problem of B+ trees on DM is their severe amplifications in I/O sizes. In internal nodes, the B+ tree stores the whole keys. In leaf nodes, the B+ tree stores multiple keys together. Without optimizations, the B+ tree needs to read and write the entire nodes during each index operation. In the following, we first theoretically compare the I/O size amplifications of ART with the B+ tree and the write-optimized B+ tree (*i.e.*, Sherman [201]). We then experimentally show the performance impacts due to the I/O size amplifications for read operations.

Theoretical Analysis

The I/O size amplification factors of different tree structures are shown in Table 4.1, respectively. We assume the internal nodes are cached and no node split occurs for brevity. M_1 and M_2 denote the metadata size of the leaf node of the radix tree and B+ tree, respectively. M_3 denotes the size of the additional metadata (*i.e.*, entry-level versions) that Sherman applied to each key-value object. S denotes the span size of the B+ tree

node. E denotes the key-value item size.

The amplification factor is defined as the ratio of bandwidth consumption from the server and bandwidth returned to the application. Without optimizations, when a client reads or writes a single object in a range index, the whole leaf node should be read or written. We use the same size of the key-value objects, *i.e.*, 96 bytes, for all trees as an example.

The leaf node of the ART contains one object with its metadata. In our implementation, 10 bytes of metadata is enough for each item in ART. The I/O size amplification factors are $\frac{M_1+E}{E} = \frac{10B+96B}{96B} = 1.10$.

The leaf node of the B+ tree contains S objects together with their metadata. The metadata at least includes two fence keys ($2 \cdot 32B$), a valid bit, a lock bit, a 1-byte level field, and two 7-bit versions [201], *i.e.*, 67 bytes in total. We use the default span size in Sherman, which is 32. The read and write amplification factors are $\frac{M_2+S \cdot E}{E} = \frac{67B+32 \cdot 96B}{96B} = 32.7$.

For Sherman, each key-value object in the leaf node is surrounded by a pair of 4-bit entry-level versions. Thus the read amplification factor is $\frac{M_2+S \cdot (M_3+E)}{E} = \frac{67B+32 \cdot (1B+96B)}{96B} = 33.0$. When writing an object without node splitting, the client only requires to write back the modified item with its associated entry-level versions. Thus the write amplification factor is $\frac{M_3+E}{E} = \frac{1B+96B}{96B} = 1.01$.

Experimental Results

To show the impact of I/O size amplifications, we compare the performances of Sherman and ART under the YCSB-C *read-only* workloads. The impact of is similar for write operations. We observe that the amplification leads to low throughput and high latency of B+ trees on DM.

Observation 1: ***The throughput of the B+ tree is bounded by network bandwidth.*** The memory-side network

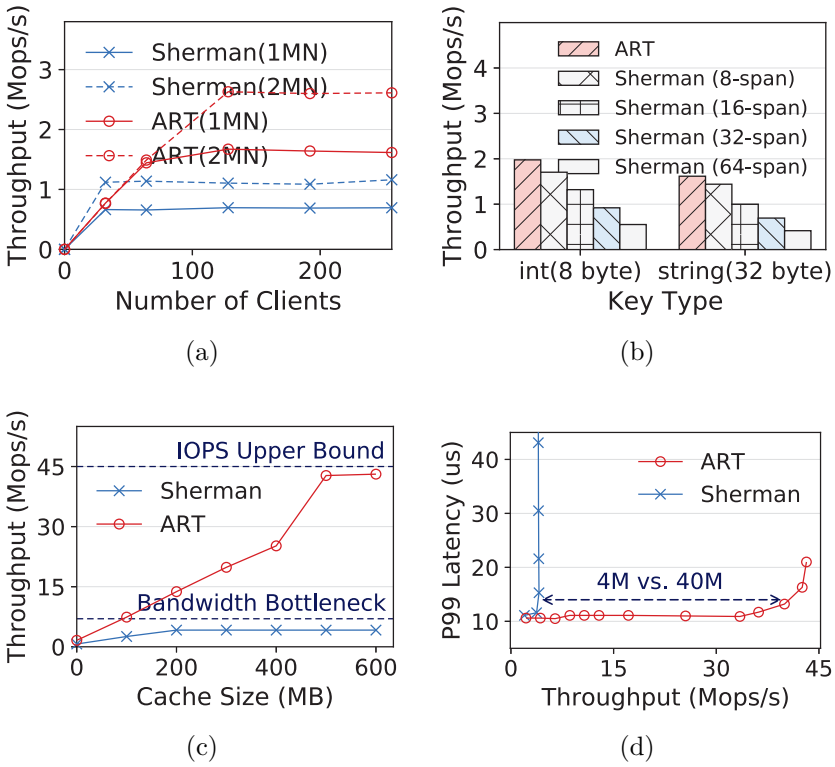


Figure 4.2: The read performances of Sherman and ART under the YCSB C workload (100% read). (a) The throughput bottleneck with no cache. (b) The impact of key size and span size with no cache. (c) The peak throughput with various sizes of caches. (d) The latency deterioration with excess requests.

bandwidth is generally the performance bottleneck for B+ trees on DM [96]. The I/O size amplifications of B+ trees cause more bandwidth consumption for each request, exacerbating the network bottleneck and resulting in low throughput.

As shown in Figure 4.2a, with an increasing number of clients, the limited bandwidth prevents the throughput of Sherman and ART from continually rising. With the same RNIC bandwidth, Sherman has a lower peak throughput than ART due to the severe I/O size amplifications. As shown in Figure 4.2b, the larger the key size or the span size (*i.e.*, the number of keys stored in a leaf node), the larger the amplification, which decreases the

peak throughput of Sherman.

A compute-side cache is usually used for caching the internal nodes of the B+ tree on DM. As shown in Figure 4.2c, with the increasing size of the cache, the throughput of Sherman keeps bounded by the bandwidth bottleneck and finally saturates at 4.17 Mops/s. The bandwidth consumption from the server equals the maximum network bandwidth of 100 Gbps (12.5 $GBps$), and the bandwidth returned to the application is $4.17\ Mops/s \cdot 96B = 0.39\ GBps$. Thus the measured read amplification factor of Sherman is $12.5\ GBps\ /\ 0.39\ GBps = 32.1$, which is close to our theoretical analysis in § 4.3.1.

In contrast, without the read amplification from leaf nodes, the throughput of ART reaches about 45 Mops/s, which is the IOPS upper bound of the RNIC we use. This indicates that ART can make full use of the RNIC capacity and achieve the best resource efficiency as DM desires.

Observation 2: ***The latency of the B+ tree is worsened by early network congestion.*** Network congestion occurs when compute-side requests saturate the bandwidth or IOPS upper bound of RNICs. As the number of clients keeps growing, excess client requests need to queue up across the network, which results in latency deterioration. The I/O size amplifications make B+ trees consume the bandwidth rapidly, expediting the process of network congestion.

As shown in Figure 4.2d, with the increase of throughput, the latency of Sherman and ART is stable in the beginning and then experiences a sudden surge due to the network congestion. Moreover, with the same memory-side RNIC bandwidth, Sherman has a much smaller inflection point (*i.e.*, the throughput threshold that triggers network congestion) than ART. As a result, Sherman shows an extremely high latency with relatively few clients. By contrast, ART has a high tolerance to this latency deterioration thanks to its small amplifications.

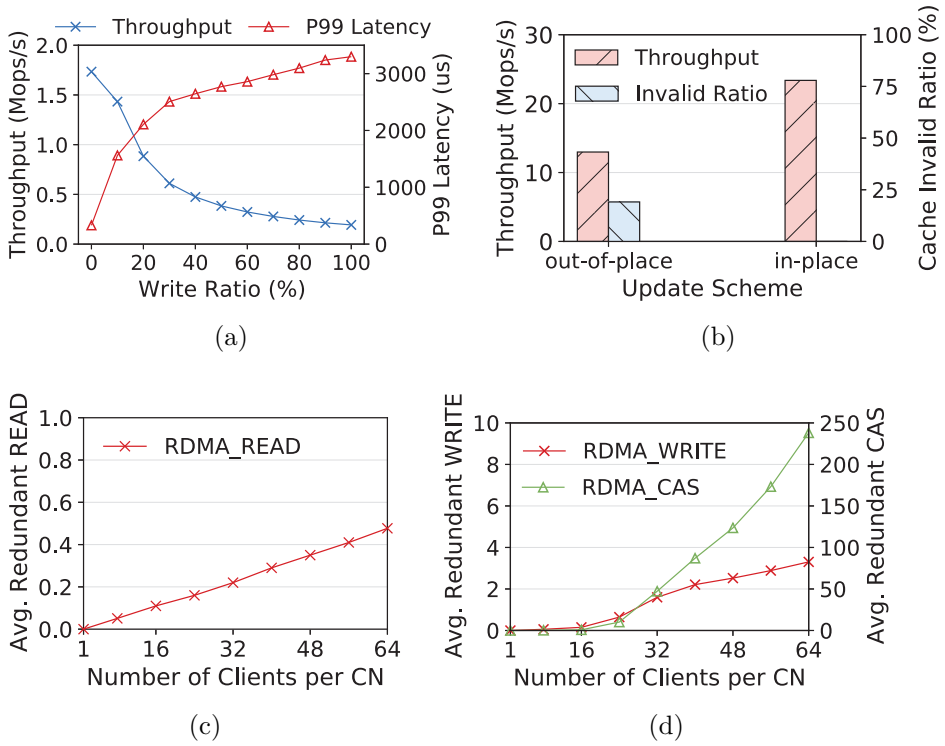


Figure 4.3: (a) The write performance of ART under the YCSB insert workload (100% insert) with no cache. (b) The performance degradation caused by cache thrashing under the YCSB A workload (50% read + 50% update) with sufficient caches. (c-d) The inter-client redundant I/Os on DM in terms of reads and writes.

4.3.2 Challenges: ART on DM

Although ART is more beneficial in terms of I/O size amplifications than B+ trees on DM, we still need to address two critical challenges in terms of concurrency control and I/O number amplifications to make it a practical range index.

Challenge 1: *Lock-based concurrency control of ART causes poor write performance.* Existing ART adopts lock-based algorithms to perform synchronization [121]. However, lock operations are expensive on DM and lead to poor write performance, as shown in Figure 4.3a. Specifically, unlike local memory, each lock operation on DM requires additional network

transmission (*e.g.*, `RDMA_CAS`). Furthermore, existing locks busy waits at the entry point when there are conflicts, causing frequent RDMA operations on retrying to get the lock. This also limits the overall throughput due to the limited IOPS of RNICs.

One feasible solution is to design lock-free algorithms. However, lock-free design is also not a good choice for ART. Specifically, an out-of-place update scheme is required for lock-free algorithms to update items larger than 8 bytes. It atomically compares and swaps the corresponding 8-byte addresses instead of modifying the items in place, as the latter cannot be realized atomically. However, in high-concurrency scenarios, a mass of out-of-place updates lead to frequent changes in the addresses of items. This leads to severe cache coherence issues since the old addresses of the items have been cached in other CNs. Even worse, in skewed workloads, the addresses of hot items are changed repeatedly, resulting in cache trashing.

To verify this, we evaluate the two update schemes in ART with the YCSB A write-intensive workload, as shown in Figure 4.3b[1]. The out-of-place scheme exhibits an average of 19.1% invalid cached addresses of leaf nodes, which results in a 44.5% throughput drop compared with the in-place scheme.

Challenge 2: ***Inter-client redundant I/Os on DM waste the limited IOPS of RNICs.*** ART can achieve better throughput compared with B+ trees due to its small-sized read and write operations. However, we find that there are redundant I/Os that waste the limited IOPS of RNICs in the DM architecture, hindering ART from achieving a high throughput. Specifically, taking read operations as an example, when several clients on the same CN read the same key-value item concurrently, they send identical `RDMA_READ`s across the network. This is a duplication of effort since all these requests do the same

[1]To eliminate the impact of concurrency conflicts, we scatter the update part of workloads among clients without intersection.

transmission work.

To measure the extent of underlying inter-client redundant reads, we launch various numbers of clients on the same CN. Each client continuously issues 1 KB `RDMA_READ`s, with their destination addresses following a Zipfian distribution of skewness 0.99 (*i.e.*, the same as YCSB's). As shown in Figure 4.3c, during each read time window, the average number of redundant `RDMA_READ`s increases with the number of clients and achieves up to 0.48 with 64 clients, implying 48% read performance improvement potential.

As for inter-client redundant writes, we issue `RDMA_WRITE`s continuously with lock-based concurrency control via `RDMA_CAS`es from each client. As shown in Figure 4.3d, during each write time window (including lock acquirement and release), the average number of redundant `RDMA_WRITE`s grows and reaches up to 3.3, indicating around 330% write performance improvement space with 64 clients. Interestingly, the number of redundant writes is more than reads since redundant writes inevitably exacerbate the concurrency conflicts, leading to a longer write time window and thus more redundant writes in return. The near-exponential growth of redundant `RDMA_CAS`es rapidly saturates the IOPS upper bound of RNICs, leading to poor performance.

4.4 SMART Design

We propose SMART, a high-performance ART for DM. Figure 4.4 shows the overview of SMART. To improve the efficiency of concurrency control (**Challenge 1**), we present a *hybrid ART concurrency control* scheme. The scheme contains a lock-free internal node design and a lock-based leaf node design to achieve high write performance without cache thrashing (§ 4.4.1). To save the limited IOPS of RNICs (**Challenge 2**), we propose a *read-delegation and write-combining (RDWC)* technique to elim-

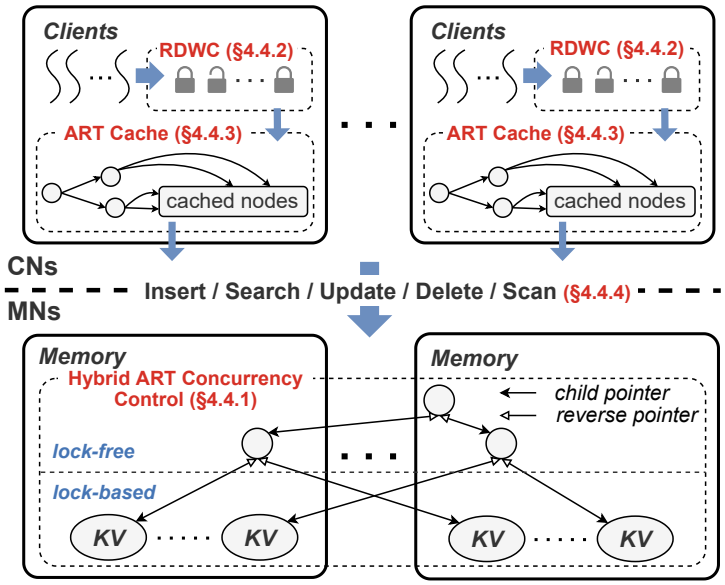


Figure 4.4: The overview of SMART.

inate inter-client redundant I/Os (§ 4.4.2). We further design an *ART cache* to reduce operation latency and achieve better performance (§ 4.4.3). Lastly, we summarize the operations that SMART supports (§ 4.4.4).

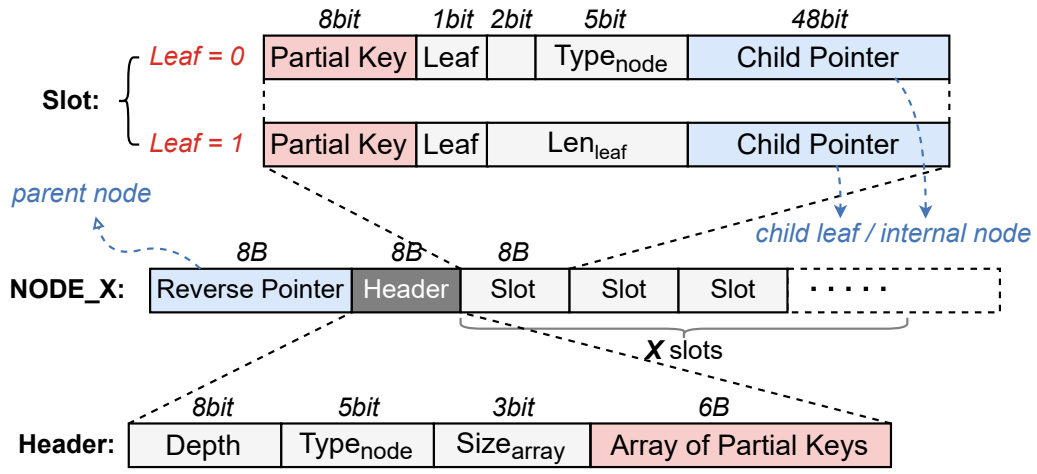
4.4.1 Hybrid ART Concurrency Control

In this section, we first describe the data structures and concurrent operations of the hybrid concurrency control scheme in SMART. We then introduce RDMA-related optimizations.

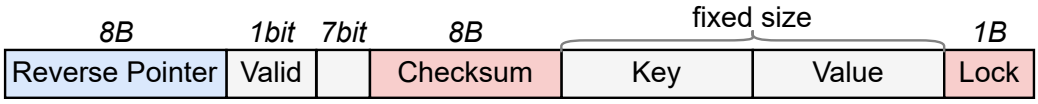
Tree Node Data Structures

Lock-free internal node. As the addresses of internal nodes change more infrequently, internal nodes do not cause cache thrashing like leaf nodes. Hence, it is feasible for lock-free internal nodes to achieve high performance. We modify the internal nodes of ART as follows.

(1) Homogeneous adaptive internal node. As illus-



(a) The homogeneous adaptive internal node with the pessimistic 8-byte header.



(b) The update-in-place leaf node with the rear embedded optimistic lock.

Figure 4.5: The structure of the internal node and the leaf node in SMART. The reverse pointer and the in-header $Type_{node}$ field are used for cache validation.

trated in Figure 4.1, a naive ART stores partial keys and child pointers separately. Such a heterogeneous design makes it hard to design a lock-free algorithm since the separated partial key and child pointer should be modified atomically. Besides, it incurs additional read amplification due to the inflexible fixed-sized internal nodes.

We come up with a homogeneous internal node design that embeds the partial keys into slots. First, this enables a child pointer to be modified together with its corresponding partial key atomically, laying the foundation for lock-free algorithms. Second, the read amplification can be reduced since internal nodes can have an arbitrary number of slots.

As shown in Figure 4.5a, an internal node of SMART consists of an 8-byte reverse pointer, several 8-byte slots, and an 8-byte header. The reverse pointer is used for cache validation, which

will be presented in § 4.4.3. As for each slot, apart from the embedded 8-bit partial key and the 48-bit child pointer, we add a 1-bit $Leaf$ field to indicate whether the pointer is pointing to a leaf node. When $Leaf$ is set, a Len_{leaf} field is provided, which is used to support variable-sized keys (§ 4.4.5). When $Leaf$ is unset, there is a 5-bit $Type_{node}$ field to indicate the type of the following internal node. Note that SMART mainly uses the $Type_{node}$ to reduce the network bandwidth consumption rather than memory consumption. When fetching an internal node, SMART can `RDMA_READ` only the required number of slots according to the $Type_{node}$ field, reducing the read amplification and thus saving the network bandwidth.

(2) Pessimistic 8-byte header of the internal node. We choose the pessimistic method for path compression since both the optimistic and hybrid methods need two tree traversals to insert a nonexistent key. One entire tree traversal is required to search for the nonexistent key since not all compressed partial keys are stored in the header. The other traversal executes the actual insertion. In contrast, the pessimistic method can insert the nonexistent key through one traversal.

Besides, following previous designs [121, 134], we fixed the header size to 8 bytes, which can be changed atomically. If some partial keys overflow from the header, we store them in an empty following node. Although this may increase the tree height, we can mitigate this with the help of cache (§ 4.4.3).

As shown in Figure 4.5a, a header consists of an 8-bit $Depth$ field, a 5-bit $Type_{node}$ field, a 3-bit $Size_{array}$ field, and a 6-byte array of partial keys. The $Depth$ field indicates the start position for matching the target key. The $Type_{node}$ field is used for cache validation, which will be illustrated in § 4.4.3. The $Size_{array}$ field records the length of the partial key array, where at most six partial keys can be stored.

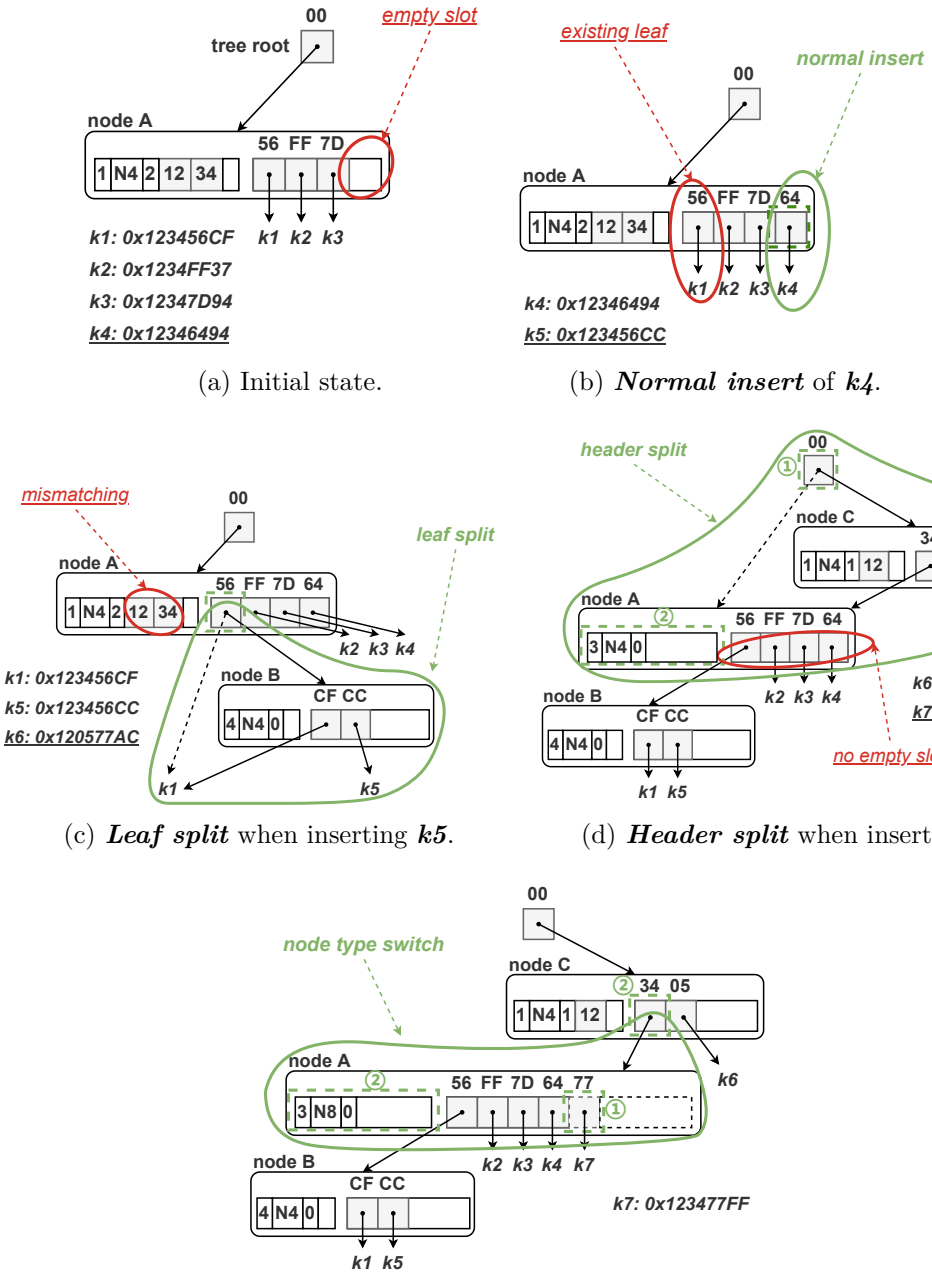
Lock-based leaf node. In-place update schemes are pre-

ferred as it does not cause cache thrashing. To adopt the in-place update, lock-based concurrency control for the leaf node is required. This is acceptable since locks are fine-grained, as each leaf node in the radix tree only contains one key-value item. We design the leaf node structure as follows for concurrency control.

(1) Checksum-based update-in-place leaf node. The in-place update scheme overwrites the leaf node at the same address, causing conflicts among readers and writers. To avoid conflicts, we adopt an optimistic lock in each leaf node with a checksum-based consistency check mechanism [143, 201], where the fixed-sized key-value object in the leaf node is protected by a checksum. For write-write conflicts, an exclusive lock is used to synchronize the writers. As for read-write conflicts, when a writer modifies the leaf node, the checksum is re-calculated based on the new content of the leaf node and written with the new content. The readers verify the checksum after reading the leaf node and retry their operations when they find the checksums do not match.

(2) Rear embedded lock. To further reduce the overhead of locks, we combine the lock release with the writing back of the updated leaf node by embedding the lock into each leaf node. Therefore, the two remote memory I/Os can be reduced to one single `RDMA_WRITE`. Particularly, to avoid premature lock release, we ensure that the lock release is always triggered after the completion of writing back. We achieve this by placing the lock at the rear of a leaf node, which leverages the in-order delivery property of RNICs [55].

As shown in Figure 4.5b, a leaf node of SMART consists of an 8-byte reverse pointer, a $Valid$ bit, an 8-byte checksum, a 1-byte rear lock, and a fixed-sized key-value item. The reverse pointer is used for cache validation, which will be illustrated in § 4.4.3. The $Valid$ bit is used to indicate the deleted state.



(a) Initial state.

(b) ***Normal insert*** of ***k4***.

(c) ***Leaf split*** when inserting ***k5***.

(d) ***Header split*** when inserting ***k6***.

(e) ***Node type switch*** when inserting ***k7***.

Figure 4.6: A step-by-step example of inserting several new keys into SMART with 8-bit partial keys. For clarity, hexadecimal partial keys are shown and reverse pointers are omitted. Each thick dotted box indicates an atomic `CAS`.

Concurrent Operations

Based on the above structural modifications, we demonstrate essential write-related sub-operations with a step-by-step example, as shown in Figure 4.6. Except for the in-place leaf update, all the sub-operations are lock-free. The complete operation process will be described in § 4.4.4.

Normal insert. During an insert, the target partial key may not be in the internal node yet. As shown in Figure 4.6b, after the `WRITE` of the new leaf node (k_4), the client `CAS`es the first empty slot in the node, together with the new partial key. If the `CAS` fails, the client checks whether the return value (*i.e.*, a new value of the slot written by a concurrent client) contains the target partial key. If yes, the client continues to traverse the tree following the return pointer. Otherwise, the client tries the insert again with the next empty slot.

Leaf split. If an existing leaf node is found during an insert, a leaf split is needed as shown in Figure 4.6c. Specifically, the client first calculates the rest of the longest common key prefix of the two leaf nodes (k_5 and k_1). Then it allocates sufficient sequentially connected internal nodes to store the common key prefix in their headers. The last internal node will contain two child pointers pointing to the old and new leaf nodes. All internal nodes and the new leaf node can be written in parallel, after which the client `CAS`es the parent slot to point to the first new internal node. If the `CAS` fails, the client continues to traverse following the return pointer.

Header split. If a mismatching for in-header partial keys is found, a header split is required as shown in Figure 4.6d. Specifically, the client allocates a new `NODE_4` pointing to the split internal node and new leaf node (k_6), with its header storing the matched part of partial keys. The new internal and leaf node can be written in parallel. Then the client `CAS`es the parent slot to make it point to the new internal node (①). If `CAS` succeeds,

the redundant in-header old partial keys are removed via an additional `CAS` (②). Otherwise, the client continues to traverse following the return pointer.

Note that the correctness of concurrent searches can be guaranteed by the in-header *Depth* value, which indicates the start position for matching the current key. A concurrent search `READ`s the parent node and then the child node. Therefore, there are two situations of read-write conflicts. First, the `READ` of the parent node occurs after the `CAS` of the parent slot (①), while the `READ` of the child node occurs before the `CAS` of the split header (②). In this situation, redundant in-header partial keys are read, which does not affect the correctness. Second, the former `READ` occurs before the former `CAS` (①), while the latter `READ` occurs after the latter `CAS` (②). In this case, the reader re-reads the parent slot if finding partial keys missing according to the *Depth* value.

Node type switch. To avoid copy-on-write (COW) overhead and additional cache coherence introduced by out-of-place updates (**Challenge 1**), we conduct an in-place node type switch. This is feasible thanks to the homogeneous adaptive internal node design (§ 4.4.1). To be specific, we pre-allocate the contiguous space of `NODE_256` on MNs for each internal node. This consumes a little additional memory but enables lock-free operations during the node type switch. When neither a matching partial key nor an empty slot is found in the current internal node, the client can try to `CAS` the following empty slots one by one, whose addresses are behind the node (①) as shown in Figure 4.6e. After a successful `CAS`, the current best-fit node type can be determined by the index of the newly inserted slot. The client then tries to update the two old $Type_{node}$ values (on the header and the parent slot) with the new one via two concurrent `CAS`es (②), making the newly inserted leaf visible by subsequent search. If both `CAS`es succeed or fail with return values contain-

ing $Type_{node}$ values larger than or equal to the expected one, the node type switch is finished. Otherwise, the client retries the `CAS`es.

In-place leaf update. To update a leaf node, the client first acquires the rear embedded lock in the leaf node. It then `WRITE`s back the updated leaf node with the re-calculated checksum and the unset lock, after which the in-place leaf update is finished with the lock properly released.

RDMA-related Optimizations

To further optimize performance on DM, SMART adopts the following RDMA-related optimizations [96].

Inline write. For small-sized `WRITE` (*e.g.*, writing internal nodes smaller than `NODE_16` or leaf nodes), the `INLINE` flag is set, enabling the RNIC to encapsulate payload into the work queue entry (WQE) and thus reducing PCIe overhead.

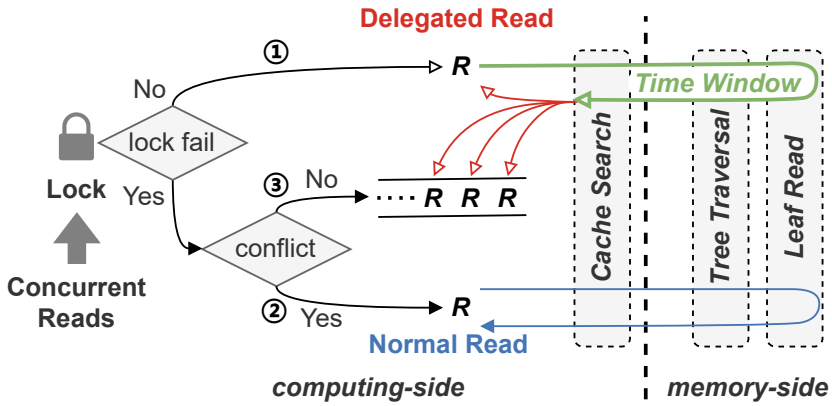
Unsignaled verbs. As for writing commands allowing asynchronous execution (*e.g.*, `CAS` of the header during ***header split***), SMART unsets the `SIGNALED` flag to reduce the overhead of polling RDMA completion queues.

Doorbell batching. If a client issues multiple WQEs to the same queue pair (*e.g.*, to the same MN), a doorbell batching is conducted to reduce PCIe overhead.

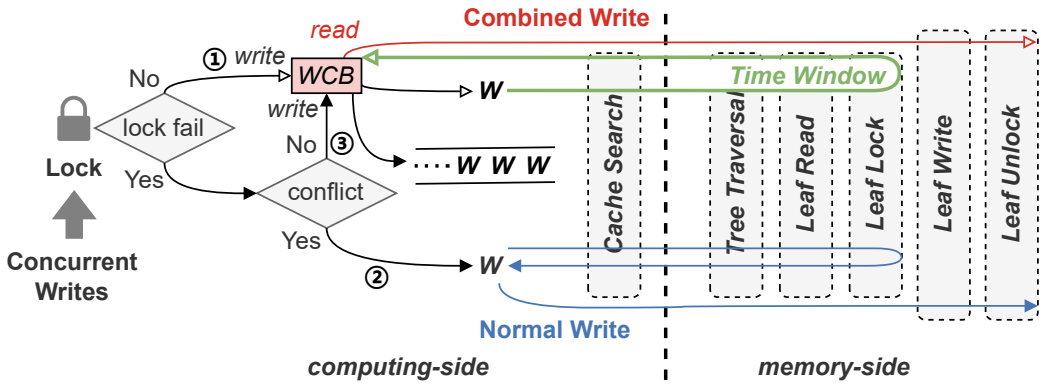
4.4.2 Read Delegation and Write Combining

We propose a read-delegation and write-combining (RDWC) technique on DM to eliminate inter-client redundant I/Os in terms of reads and writes, respectively.

Hash-based local locks. The inter-client redundant I/Os on each CN occur among the concurrent read and write operations on the same key or address. Therefore, computing-side local locks are needed to collect the concurrent operations.



(a) Read delegation.



(b) Write combining.

Figure 4.7: The processes of the read delegation and the write combining on SMART respectively.

We maintain the local locks in each CN as a table, similar to the local lock table of HOCL in Sherman [201]. However, unlike Sherman, which maintains each local lock for a coarse-grained global lock, SMART maintains each local lock for a key (*i.e.*, fine-grained leaf node). It is challenging to store all such locks in each limited computing-side memory. To address this, we use hash-based local locks, where a lock corresponds to a set of keys with the same hash value.

We dynamically maintain a *unique key* in each local lock to solve the hash-conflict problem of our hash-based scheme. Specifically, the first client who acquires a local lock successfully

will record its target key as the unique key of this local lock. The subsequent clients who fail to acquire this local lock will conduct a hash-conflict check by comparing their target key with the unique key. If the target key is exactly the same as the unique key, the client can be involved in the read delegation or write combining. Otherwise, a hash conflict is found, and the client should execute a normal remote read or write on its own for correctness. The unique key is freed when the first client releases the local lock.

Read delegation. To reduce inter-client redundant I/Os for reads, a *delegation client* can be elected on each CN to execute the same read, and then share its `READ` result with other waiting clients. The first client who acquires the local lock successfully is the delegation client and the subsequent clients who fail to acquire the lock are the waiting clients. The relationship between the delegation client and the waiting clients is similar to that between the first cache miss and the subsequent delayed cache hits in the cache system [22].

We implement this as shown in Figure 4.7a. After acquiring the corresponding local lock successfully, the delegation client records its target key as the unique key and then conducts the remote tree search (*i.e.*, including cache search, tree traversal, and leaf node read), which is the time window of read delegation (①). During the time window, the subsequent clients failing to acquire the local lock first execute the hash-conflict check by comparing their target key with the unique key. If a hash conflict is found, the client executes a normal tree search by itself (②). Otherwise, it pushes itself into a read-waiting queue and waits for the search result from the first client (③). Finally, the delegation client shares its search result with the waiting clients and releases the local lock.

Write combining. Write combining (WC) is a normal technology in modern processors [53]. When a processor intends to

issue multiple writes to the same memory region in a small time window, it combines the writes into a single burst write so as to save the system bus bandwidth. This idea, also known as write coalescing, is applied to many storage systems [91, 116]. Inspired by this, we find it feasible to conduct a WC on each CN. When clients intend to make several concurrent key-value writes to the same memory-side key or address, they can combine the writes into a single consensus write so as to save the network bandwidth and the limited IOPS of RNICs.

We implement WC on DM as shown in Figure 4.7b. A client that succeeds in acquiring the corresponding local lock first records its target key as the unique key and writes its new value into the write combining buffer (WCB), and then conducts the remote tree insert or update (①). Differently, the time window of write combining is the former partial period of tree insert or update (*i.e.*, cache search, tree traversal, and lock acquirement on leaf node). After that, the client reads the combined consensus result from WCB and then makes a `RDMA_WRITE` to write back the result and release the remote lock. Finally, the client releases the local lock. During the write-combining time window, the subsequent clients first perform the same hash-conflict check. If a hash conflict is found, the client performs a normal tree insert or update on its own (②). Otherwise, it first writes its expected value into the WCB (with local lock-based concurrency control), making the value visible to the first client. Then the client pushes itself into a write-waiting queue to wait for the completion of the remote write (③).

Put together. Naively putting read-delegation and write-combining together may introduce incorrect read results when a client reads a key-value object after writing it. Specifically, the latter read may be delegated by a client whose read happens before the write operation. In this case, the old value (*i.e.*, the value of the item before the client's write) is returned to

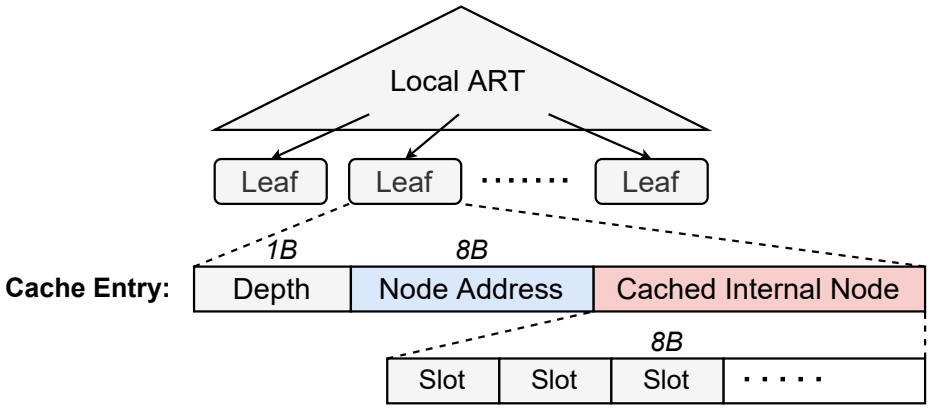


Figure 4.8: The structure of the ART cache.

the read operation that happens after the write, breaking the causality of the read and write. We use the same time window for read-delegation and write-combining to address this issue. In this way, the write and read operations with causal relations are included in two non-overlapped time windows, and thus, the above issue can be avoided. To achieve this, we let readers and writers operating on the same key fairly acquire the same local lock, where the winner decides the time window. Each local lock is associated with two waiting queues, *i.e.*, a read queue and a write queue, so as to conduct read delegation and write combining exclusively and concurrently. In our implementation, 4M 32-bit local locks are sufficient on each CN, consuming only nearly 3% of cache size.[2]

4.4.3 ART Cache

ART-indexed cache. To reduce remote access during tree traversal, a memory-efficient ART-indexed cache is designed on each CN to store partial internal nodes of SMART. To be specific, utilizing the feature that each radix tree node (excluding

[2]Note that with N clients in each CN, there are at most N dynamically-allocated WCBs and unique keys at the same time, whose memory consumption (*i.e.*, size of N key-value items) is negligible.

header) can be uniquely identified by a key prefix, we adopt a local ART on each CN to index the cached internal nodes. As shown in Figure 4.8, each leaf node (*i.e.*, cache entry) of the local ART contains the snapshot of a traversal context (*i.e.*, the content of an internal node being read from MNs, the *Depth* value, and the address of the node).

There are two situations the cache may be invalidated. First, a node in the disaggregated memory pool may no longer be the child of the cached parent node, which happens on ***leaf split*** and ***header split***. We use the reverse pointer in each internal and leaf node to check if the current node is still the child of the cached parent node. Second, the node type of each internal node can change when continuously inserting the node. We also need to get the correct node type since otherwise we may fail to find newly inserted objects. We use the $Type_{node}$ field in each node to detect node type changes.

4.4.4 Operations

All operations first search in the cache for the deepest slot that is matched by the prefix of the target key. If none of the cached slots hits, start the traversal from the tree root slot.

Search. The client first reads the node according to the slot, after which a ***reverse check*** is conducted to check if the cache entry expires. If yes, invalidate the cache entry and retry this search. As for a leaf node being read, the target item is found if its key is the same as the target key. Otherwise, it does not exist. As for an internal node, if all the in-header partial keys are matched, and the next target partial key can be found in a slot, read the next node along the child pointer in the slot and repeat the process. Otherwise, the target item does not exist.

Insert/Update. The client first reads the node and conducts a ***reverse check*** like the search. After that, as for a

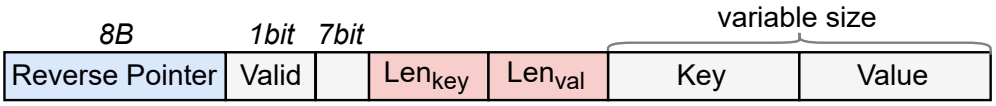


Figure 4.9: The structure of the variable-sized leaf node.

leaf node, if its key is the same as the target key, execute an ***in-place leaf update***. Otherwise, a ***leaf split*** is needed. As for an internal node, if a mismatching for the in-header partial keys is found, conduct a ***header split***. Otherwise, turn to search among the slots. If the current target partial key can be found in a slot, read the next node along the corresponding child pointer in the slot and start the process again. Otherwise, conduct a ***normal insert*** with the next empty pointer slot. If no empty slot can be found, a ***node type switch*** is needed.

Delete. Delete operations have a similar process as insert operations. A normal delete clears the slot pointing to the target leaf node via `RDMA_CAS` and unsets the $Valid$ bit of the deleted leaf node. Opposite operations of ***leaf split*** and ***header split*** are conducted for path compression.

Scan. At each level of traversal, the client conducts parallel `RDMA_READ`s to fetch all nodes inside the target key range. For each `RDMA_READ`, the client processes the node being read in the same way as the search operation, with an additional comparison between partial keys and target key range to exclude unwanted concurrent search paths. Like many other existing tree indexes [229, 201] on DM, SMART does not guarantee the scan is atomic with concurrent insert or update operations.

4.4.5 Discussion

Support for variable-sized keys and values. SMART currently supports fixed-sized keys and values. For variable-sized keys and values, the optimizations of *update-in-place leaf node* and *rear embedded lock* in SMART are no longer applicable. In-

stead, SMART can use the RCU scheme to out-of-place update the leaf node to support variable-sized keys and values. The search, insert, and delete operations on variable-sized key-value items are the same as those on fixed-sized ones.

As for the leaf node structure, SMART can follow the design in RACE [230]. As shown in Figure 4.9, the leaf node includes a Len_{key} field and a Len_{val} field, which indicate the sizes of Key and $Value$. SMART can use the 7-bit Len_{leaf} field in the parent slot and a pre-configured *length_unit* value to indicate the length of the leaf node. The maximum length of a leaf node is $2^7 \cdot length\_unit$. When a key-value item exceeds the maximum length, SMART can store the remaining content in a second key-value block linked to the leaf node.

Generality of techniques in SMART. Some techniques in SMART can also be applied to other kinds of indexes. Particularly: 1) The *RDWC* technique can benefit any tree indexes since it is transparent to the lower-level index structures. When applied to other index structures, it brings about the same performance improvement as applied to ART. 2) The *ART cache* can benefit any radix-tree-based indexes. It is designed to reduce operation latency and handle the cache validation problems caused by ART's features. 3) The *rear embedded lock* can be adopted in any lock-based structures on DM to save one RTT.

The first lock-free ART design. A pure lock-free ART can be formed with the lock-free node design in Figure 4.5a and a lock-free leaf node design with a traditional RCU scheme. To the best of our knowledge, this is the first lock-free ART design. In our implementation, SMART can degenerate into the pure lock-free ART by disabling the optimizations of *update-in-place leaf node* and *rear embedded lock*.

4.5 Evaluation

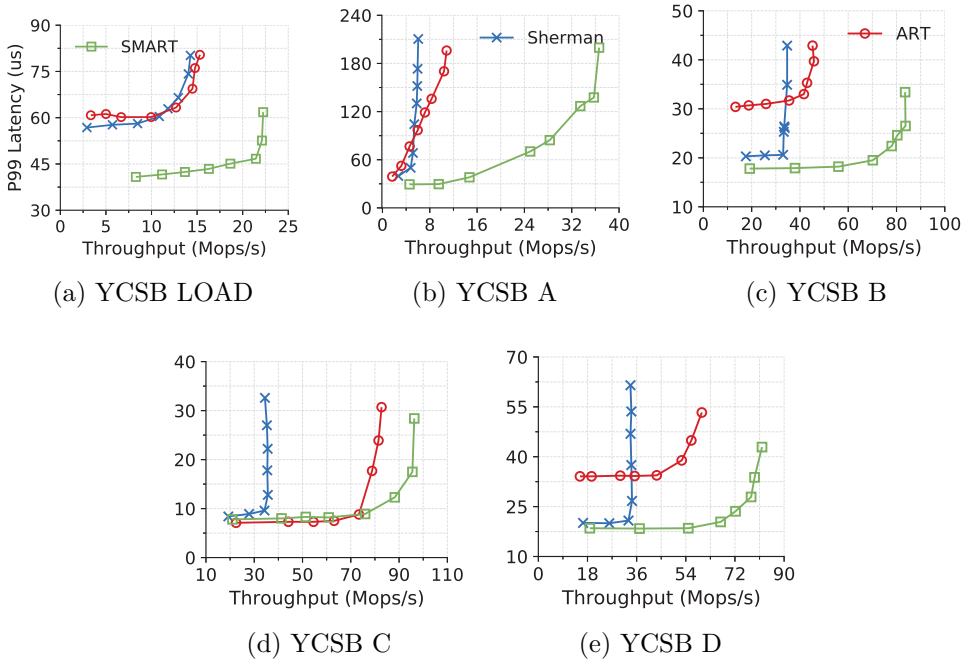
4.5.1 Experimental Setup

Testbed. We run all experiments on 16 nodes (16 CNs and 2 MNs)[3] on the Clemson cluster of CloudLab [57]. Each node has two 36-core Intel Xeon CPUs, 256GB of DRAM, and one 100Gbps Mellanox ConnectX-6 RNIC. Each RNIC is connected to a 100Gbps Ethernet switch. Each MN owns 64GB DRAM and 1 CPU core for network connection and memory allocation. Each CN owns 4GB DRAM and 64 CPU cores, where each core serves as a client. The MNs register memory with huge pages to reduce page translation cache misses of RNICs [55].

Workloads. Without explicit mention, we use the index microbench [204] to generate YCSB [50] workloads like previous work [105, 26, 137]. We evaluate SMART with 6 YCSB core workloads: A (50% read, 50% update), B (95% read, 5% update), C (100% read), D (latest-read, 95% read, 5% insert), E (95% scan accessing up to 100 items, 5% insert) and an additional LOAD (100% insert) workloads, using the default Zipfian distribution for all workloads except for YCSB LOAD and D. For most workloads, we test 2 key types, *i.e.*, integer (8-byte) and string (32-byte). For string workloads, we use 125 million publicly available email addresses [70] and conduct a common pre-processing (*i.e.*, swap username and domain fields of email addresses) like previous work [204, 136, 120]. We use 8-byte values consistent with prior work [201, 206, 136, 144, 98, 26]. For each workload, we populate 60 million keys before conducting 60 million operations, except for the LOAD test.

Comparisons. We compare SMART with two state-of-the-art tree indexes, *i.e.*, Sherman [201] and ART [120]. We use the default configuration of Sherman (*e.g.*, a span size of 32 for long key) with all optimizations enabled (*e.g.*, on-chip mem-

[3]Like Sherman [201], we make two nodes act as both CN and MN.



(a) YCSB LOAD (b) YCSB A (c) YCSB B

(d) YCSB C (e) YCSB D

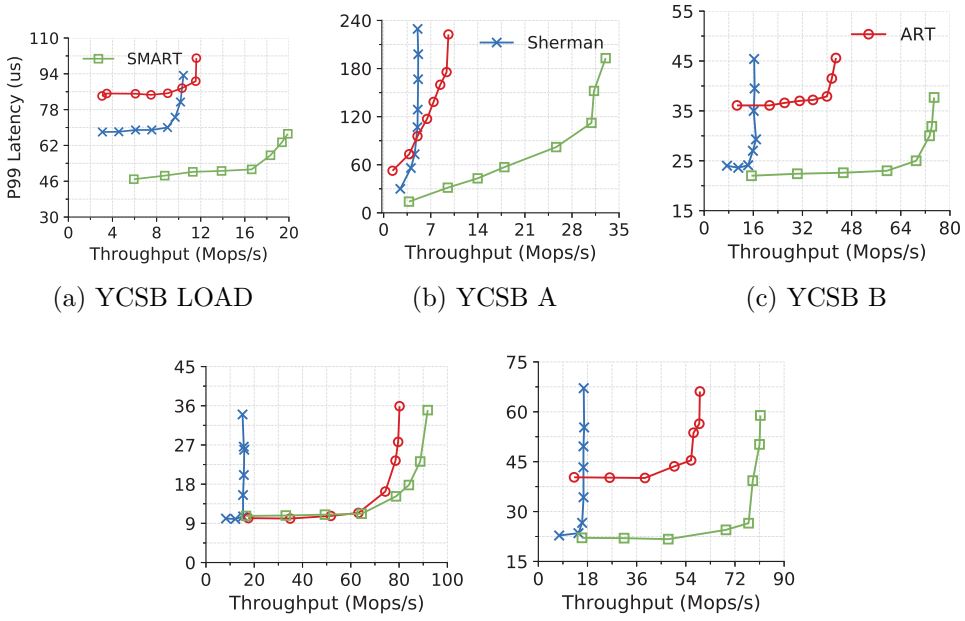
Figure 4.10: The performance comparison of tree indexes on DM under YCSB workloads of integer keys.

ory). Since ART is not designed for DM, we port it to DM by re-implementing it from scratch (as mentioned in § 4.3), including its synchronization design (*i.e.*, ROWEX [121]). For better baseline performance, we apply the HOCL of Sherman to ART and any other baselines of SMART. Coroutines are used in each client to hide RDMA polling overhead.

4.5.2 Performance Comparison

Figures 4.10 and 4.11 present the throughput-latency curves of the three indexes with integer and string keys respectively, using various numbers of clients (16 at least and 896 at most, evenly distributed across 16 CNs). Without loss of generality, we discuss the performance of integer keys in the following.

Search-only workload (*YCSB C*). For the YCSB C workload, SMART outperforms Sherman by 2.8× due to no leaf read



(a) YCSB LOAD (b) YCSB A (c) YCSB B

(d) YCSB C (e) YCSB D

Figure 4.11: The performance comparison of tree indexes on DM under YCSB workloads of string keys.

amplification, as mentioned in § 4.3. Moreover, it outperforms ART by 1.2× due to the read delegation mechanism for reducing redundant I/Os. It is worth noting that SMART achieves up to 96M requests per second, which breaks through the total IOPS upper bound of memory-side RNICs (about 90 Mops in total with the two MNs). This is because the read delegation can perform concurrent duplicated reads with only one delegated read. Besides, the similar P99 latency of SMART and ART shows that the read delegation causes near-zero overhead.

Insert workload (*YCSB LOAD, D*). For the YCSB LOAD workload, SMART outperforms Sherman and ART by 1.6×, 1.5× in throughput and achieves 1.4×, 1.5× lower P99 latency respectively. This can be attributed to the design of the lock-free internal nodes. Specifically, both Sherman and ART have low throughput and high latency due to the node-

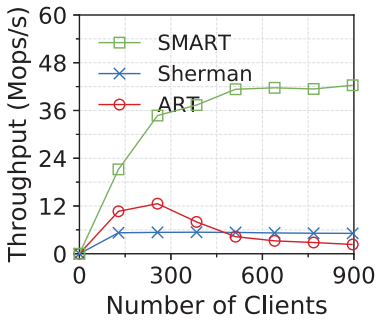


Figure 4.12: The scalability of tree indexes under the YCSB A workload of integer keys.

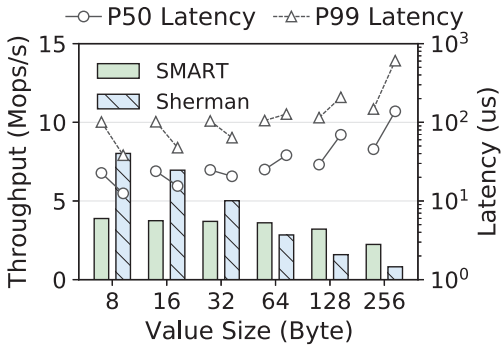


Figure 4.13: The performance of scan under the YCSB E workload of integer keys with different value sizes.

grained locks, which introduce additional RTTs with frequent lock-fail retries, thus wasting the limited IOPS of RNICs in write-intensive scenarios (*i.e.*, 50% insert). Interestingly, with string workloads, the latency of ART becomes much worse since the smaller set of string partial keys (*e.g.*, alphanumeric characters) aggravates concurrency conflicts.

For the YCSB D workload, SMART achieves 2.4× and 1.4× higher throughput and 1.1× and 1.8× lower P99 latency, compared with Sherman and ART respectively. With fewer write conflicts (*i.e.*, only 5% insert), read and write amplifications become the main reason for the poor performance of Sherman. ART still has a high tail latency since concurrent writes cause cache misses, leading to remote tree traversals and thus continuous lock operations on the remote tree.

Update workload (*YCSB A, B*). Compared with Sherman and ART, SMART gains 6.1× and 3.4× improvement in throughput and 1.4× and 1.3× reduction in latency for YCSB A, and achieves 2.4× and 1.8× higher throughput and 1.1× and 1.7× lower P99 latency for YCSB B, respectively.

Unlike the insert workload, YCSB A and B follow a Zipfian

distribution of skewness 0.99, indicating a high amount of update concurrency conflicts. Consequently, Sherman performs poorly with YCSB A due to its coarse-grained, lock-based concurrency control. ART performs better than Sherman since update operations do not modify the partial key fields and thus do not need to acquire locks. However, the out-of-place update scheme used by ART causes cache thrashing, resulting in huge cache-miss overhead and thus much higher latency than SMART. Note that the cache thrashing also impacts search performance, leaving a poor performance of ART on YCSB B (with only 5% update). As shown in Figure 4.12, ART experiences performance collapse with increasing clients due to severe cache thrashing. In contrast, SMART shows excellent scalability due to the cache-friendly in-place leaf node design and fine-grained concurrency control.

Scan workload (*YCSB E*). We evaluate the performance of scan operations with 128 clients using varying value sizes as shown in Figure 4.13. For a small value size (*e.g.*, 8 bytes), SMART shows poorer performance than Sherman since the small-sized leaf nodes saturate the memory-side IOPS upper bound, which is an inherent shortcoming of radix trees. However, for a value size larger than 64 bytes, which is common in real-world workload [215, 21], the scan performance of Sherman becomes worse than SMART since the large-sized leaf nodes rapidly saturate the bandwidth bottleneck.

4.5.3 Factor Analysis for SMART Design

Figure 4.14 presents the factor analysis on SMART. We start with the naive ART and apply each proposed technique one by one. We use 16 CNs (each launches 24 clients) and integer keys for experiments in this section.

+ **Lock-free internal node.** The lock-free internal nodes

mainly contribute to the insert workload. With YCSB LOAD, it brings 1.5× improvement in throughput and 1.8×/1.4× reduction in P50/P99 latency. Unlike ROWEX, lock-free internal nodes eliminate expensive lock overhead during insertion and thus improve performance.

+ Update-in-place leaf node. The in-place update scheme mainly contributes to the update workload. It achieves 1.5× improvement in throughput and 1.4×/1.7× reduction in P50/P99 latency with YCSB B. The in-place update scheme alleviates the cache coherence problem, as the addresses of the cached leaf nodes never expire until being deleted.

+ Rear embedded lock. The rear embedded locks further optimize the in-place update scheme. It eliminates the lock-releasing overhead, saving one RTT during each update. With YCSB A, it improves throughput by 3.0× and reduces tail latency by 11.3×.

+ Read delegation. The read delegation mechanism contributes to the search workload. It brings 1.1× throughput improvement and 1.3× tail latency reduction with YCSB C. It eliminates superfluous reads and thus saves network I/O consumption, so as to support more client requests.

+ Write combining. The write-combining mechanism contributes to write-intensive workloads. It improves the throughput by 1.1× and reduces tail latency by 1.3× with YCSB A.

As the RDWC technique can reduce concurrency conflicts similar to HOCL, we compare their efficiency by applying them on SMART respectively. As shown in Figure 4.15, when applying the primitive HOCL design, SMART shows poor performance with an average of 0.76 lock-fail retry count, due to the limited on-chip memory space (128MB per RNIC in our evaluation) with only 2 MNs, which is insufficient for a large number of fine-grained locks. With E-HOCL (*i.e.*, integrating the rear embedded lock technique into HOCL), SMART achieves much

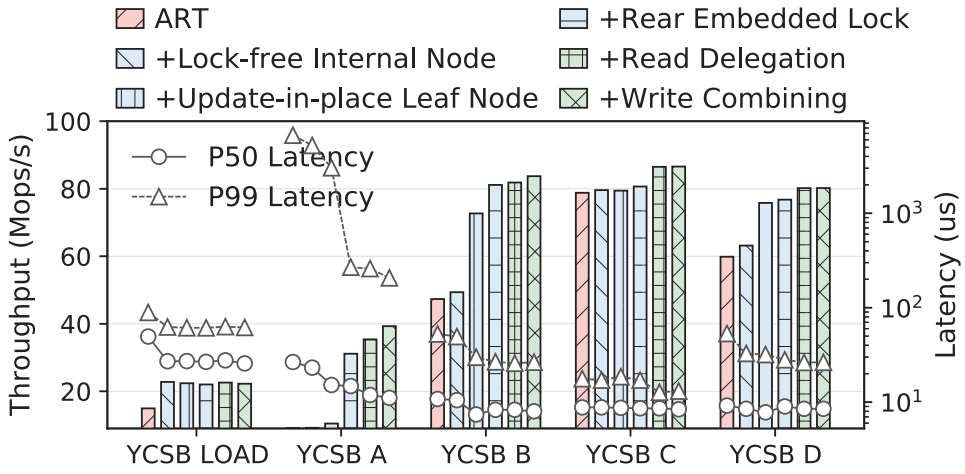


Figure 4.14: The factor analysis of overall performance on SMART.

better performance with an average of 0.29 lock-fail retry count. However, despite the optimization, HOCL still shows lower improvement efficiency than RDWC, which can introduce a 26.2% higher throughput. This is because RDWC saves not only the lock overhead but also the superfluous bandwidth consumption of reads and writes.

As the design of RDWC is transparent to the lower-level index structures, it will lead to the same amount of performance improvements on Sherman, *i.e.*, 1.3× and 1.1× under write-intensive and read-only workloads (Figure 4.14). After applying RDWC to Sherman, SMART can still achieve 4.7× (= 6.1/1.3) higher throughput under write-intensive workloads and 2.5× (= 2.8/1.1) higher throughput under read-only workloads.

Cache-related techniques. Some cache-related techniques contribute to cache efficiency: ***1) Homogeneous adaptive internal node.*** Due to the homogeneous adaptive internal node design, more fine-grained and flexible adaptive nodes are available, saving cache space with smaller sizes of cached nodes. ***2) ART-indexed cache.*** Compared with a traditional hash-based compute-side cache, the ART cache can efficiently save

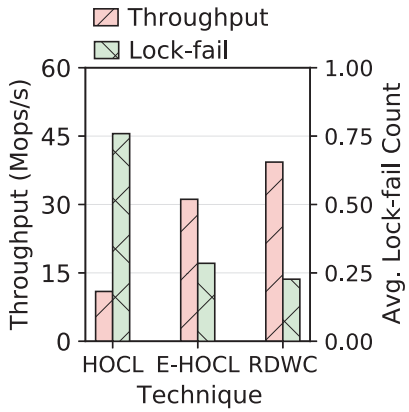


Figure 4.15: Comparison of HOCL, E-HOCL, and RDWC under the YCSB A workload.

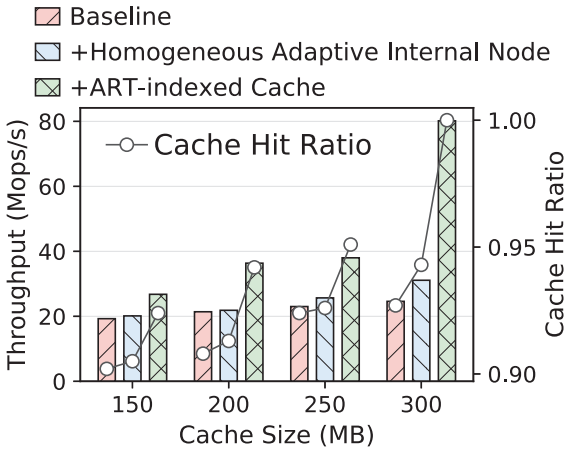
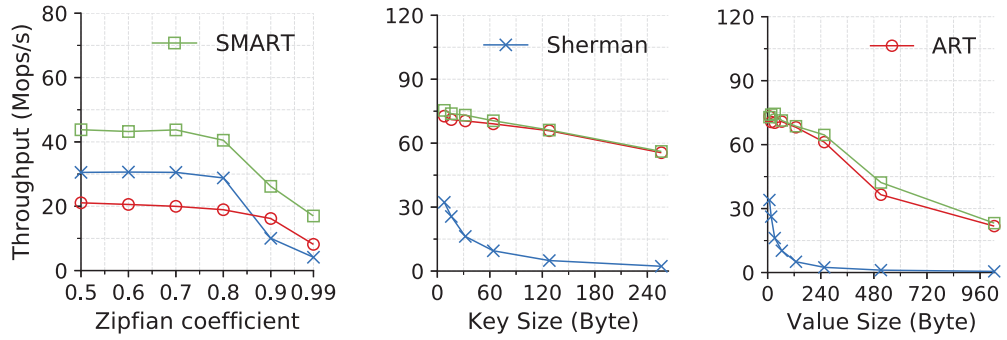


Figure 4.16: Cache efficiency of SMART under the YCSB C workload of string keys with different cache sizes.

memory since key prefixes are no longer stored repeatedly. As shown in Figure 4.16, after applying the above two techniques one by one, SMART achieves an increasing cache hit ratio and overall throughput under each specific limited cache size.

4.5.4 Sensitivity

Skewness. Figure 4.17a shows the performances of different tree indexes on a generated Zipfian workload [126] (50% search + 50% update) with various skewness. SMART performs best under both slightly and highly skewed workloads. Sherman shows a good performance in slightly skewed workloads while having the poorest performance in highly skewed workloads because of its coarse-grained lock-based concurrency control design. ART performs better than Sherman in highly skewed workloads due to the lock-free RCU scheme but performs worst in slightly skewed workloads due to cache thrashing. Note that the RDWC in SMART does not benefit the overall throughput since the network bandwidth is unsaturated. As the Zipfian skewness grows



(a) Skewness. (b) Key size. (c) Value size.

Figure 4.17: The sensitivity analysis.

from 0.5 to 0.99, the performance of ART and SMART decreases by the same multiple (2.6×), and thus their performance gap is reduced. The performance of Sherman decreases by 7.4×, indicating the poor efficiency of coarse-grained lock-based design.

Key-value sizes. Figures 4.17b and 4.17c show the impact of key size and value size on the performances of the three tree indexes under YCSB C with sufficient caches. As the key size grows from 8 to 256 bytes, SMART and ART show a slight performance decline (1.3×), while Sherman experiences a rapid drop in performance (14×). As the value size grows from 8 to 1024 bytes, the performance declines of SMART, ART, and Sherman are 3.1×, 3.4×, and 64×, respectively. This is because, during each search, Sherman needs to fetch the whole leaf node, whose size grows with key and value size, causing the rapidly increasing consumption of network bandwidth. On the contrary, SMART and ART only need to fetch the fine-grained small-sized leaf node. Thus, they are not bounded by the network bandwidth bottleneck, showing a stable performance with varying key sizes and value sizes. The performances of ART and SMART are close since the read delegation in SMART does not benefit the throughput under the unsaturated network. This is consistent with the results shown in Figure 4.10d.

4.6 Related Work

Attracted by the high performance of RDMA, there are increasing studies focusing on building RDMA-based tree data structures [229, 201, 5, 144, 176]. Many of them are built on top of RDMA-based remote procedure calls (RPCs) [144, 176]. Unfortunately, these approaches are infeasible for disaggregated memory due to the asymmetric compute capabilities on compute and memory nodes. Specifically, the CPUs on the memory nodes are too weak to execute index traversal and modifications on the data path.

Two tree data structures built on DM relate most to SMART, *i.e.*, FG [229] and Sherman [201]. FG, designed as a B-link tree, is the first index that completely leverages *one-side RDMA verbs* for write operations. Sherman is the state-of-the-art B+ tree index with several RDMA-friendly software techniques. However, constrained by the structure of the B+ tree, both approaches suffer from low suboptimal throughput and early latency deterioration due to I/O size amplifications. SMART proposes to use radix trees as range indexes on DM to address the severe I/O size amplifications of B+ trees.

Moreover, extending RDMA interfaces is another approach to designing tree indexes on DM. They offload index write operations into memory-side NICs via SmartNICs or other customized hardware [5, 31, 68, 105, 129, 174, 181]. However, they all rely on dedicated hardware. To the best of our knowledge, SMART is the first radix tree index on DM that achieves high performance with commodity RNICs.

4.7 Summary

This chapter presents our design and implementation of a high-performance range index data structure tailored for DM. We in-

novatively propose to use radix trees as range indexes to reduce I/O size amplifications. We further address the challenges in concurrency control and I/O number amplifications with a hybrid concurrency control scheme and a read delegation and write combining scheme. Our evaluation results show that SMART outperforms the existing approaches by up to 6.1× under write-intensive workloads and 2.8× under read-only workloads.

□ **End of chapter.**

Chapter 5

Efficient Fault Tolerance Algorithms

Outline

Achieving reliability on disaggregated memory (DM) is challenging since the isolated failures introduce more complicated failure situations. Existing memory-disaggregated storage systems maintain critical metadata on monolithic servers, simplifying failure handling by directly adopting existing fault-tolerance algorithms. However, they suffer from suboptimal cost-efficiency due to the adoption of monolithic metadata servers. We propose to bring disaggregation also to metadata management and design DM-native fault tolerance algorithms to achieve both reliability and high performance. Unfortunately, existing fault tolerance algorithms suffer from suboptimal performance due to their I/O amplifications and reliance on the weak compute power in the memory pool. This chapter introduces FUSEE, the first fully memory-disaggregated storage system with high-performance replication and logging algorithms to efficiently handle the complex failure situations on DM.

5.1 Introduction

Reliability is a critical aspect for in-memory storage systems [77, 187, 217]. However, the hardware failure isolation provided by disaggregated memory (DM) becomes a double-edged sword. On the one hand, DM isolates the failures of CPU and memory from a single monolithic server, *e.g.*, CPU failures in compute nodes (CNs) no longer lead to the unavailability of data on memory nodes (MNs). Reliability can be potentially improved if the failures can be handled separately. On the other hand, isolated failures between compute and memory introduce more complex failure situations, *e.g.*, MN failures cause data loss and affect the request processing of client applications (clients) on CNs. Existing fault tolerance algorithms become insufficient to deal with the decoupled failures and resources, making it challenging to achieve reliability with high performance.

Existing work [192] adopts a *semi-disaggregated* design that stores objects in the memory pool but retains metadata, *i.e.*, shared system states that can affect correctness, managed on monolithic metadata servers. The metadata server is also replicated with state machine replication [154, 7] to ensure high availability and strong consistency for the critical shared metadata, *i.e.*, index and memory management data structures. Data, *i.e.*, key-value (KV) objects, are also replicated on multiple MNs to avoid data loss under MN failures. While this design can handle the failures correctly, many additional resources have to be exclusively assigned to the metadata servers to prevent it from becoming a performance bottleneck [41, 211, 167], compromising the resource efficiency of DM.

To achieve better cost-efficiency, it is critical to bring disaggregation to metadata management, *i.e.*, storing metadata in the memory pool and directly managing them with clients in the compute pool. However, it is non-trivial to achieve such a fully

memory-disaggregated design due to the following challenges.

1) Replicating the shared index. Memory-disaggregated storage systems typically adopt hash indexes to organize KV objects. The hash index data structure is shared by all clients in the compute pool to execute data access requests. Memory-disaggregated storage systems have to replicate the index data structure on multiple MNs to avoid index loss under MN failures. Existing replication algorithms for shared data, *e.g.*, state machine replication [154, 147, 101, 194] and shared register protocols [133, 30, 23], assumes that the data are exclusively managed by the same CPUs that execute data access requests. They heavily rely on the CPUs to resolve conflicts and achieve strong consistency. However, they are infeasible on DM since the shared index is stored on MNs while requests are executed on CNs. Meanwhile, simply employing consensus protocols [154, 115, 147] or remote locks [192] on CNs suffer from poor performance and scalability due to their severe I/O amplifications and lock contentions [213, 20, 39, 201].

2) Metadata corruption under client failures. For existing semi-disaggregated storage systems, failures in CNs do not affect metadata because the CPUs of monolithic metadata servers exclusively modify metadata. However, clients directly access and modify metadata on memory nodes in the fully memory-disaggregated setting. As a result, client failures can leave partially modified metadata accessible by others, compromising the correctness of the entire KV store.

We propose FUSEE, a fully disaggregated storage system to address the above challenges. First, to replicate the shared index data structure with high performance and strong consistency, FUSEE proposes the SNAPSHOT replication protocol. SNAPSHOT reduces the number of sequential I/O operations with a broadcast-based write protocol and efficiently resolves concurrency conflicts with simple yet efficient rule-based con-

flict resolution. Besides, to deal with the metadata corruption, FUSEE adopts an embedded operation log scheme to recover clients' partially executed operations. The embedded operation log reduces the additional I/O on the critical path for log maintenance by reusing the memory allocation order and embedding log entries in KV objects.

We implement FUSEE from scratch and evaluate its performance using both micro and YCSB benchmarks [50]. Compared with Clover and pDPM-Direct [192], two state-of-the-art semi-disaggregated storage systems, FUSEE achieves up to 4.5 times higher overall throughput and exhibits lower operation latency with less resource consumption. The code of FUSEE is available at `https://github.com/dmemsys/FUSEE`.

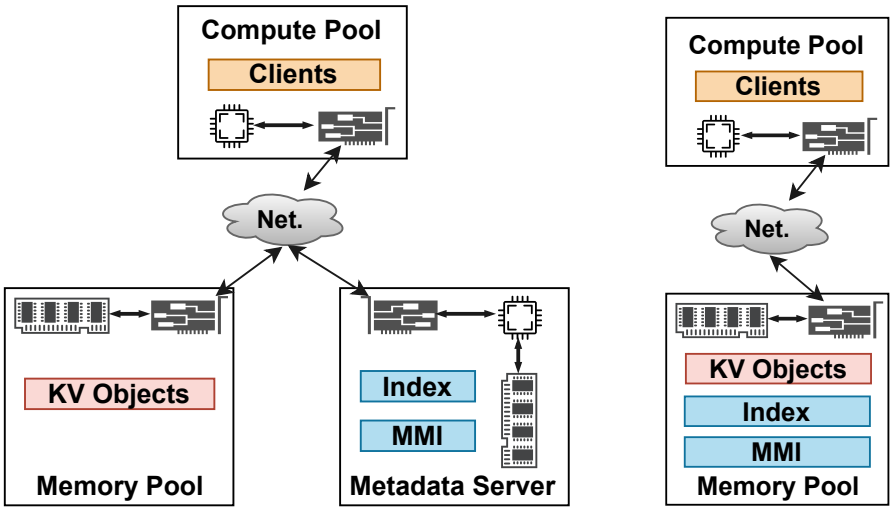
In summary, this chapter makes the following contributions:

- A fully memory-disaggregated storage system that achieves reliability with high performance.
- A client-centric replication protocol that uses conflict resolution rules to enable clients to collaboratively resolve conflicts. The protocol is verified with TLA+ [114] for safety and the absence of deadlocks under crash-stop failures.
- An embedded operation log scheme to recover the corrupted metadata with low log maintenance overhead.
- The implementation and evaluation of FUSEE to demonstrate the efficiency and effectiveness of our design.

5.2 Background and Motivation

5.2.1 Semi-Memory-Disaggregated Storage System

Clover [192] is a state-of-the-art storage system built on DM. It adopts a semi-disaggregated architecture that separates data



(a) Clover (b) FUSEE

Figure 5.1: Two architectures of memory-disaggregated storage systems. (a) The semi-disaggregated architecture (Clover [192]). (b) The fully disaggregated architecture proposed in this work.

and metadata to lower the ownership cost and prevent the compute power of data nodes from becoming the performance bottleneck. As shown in Figure 5.1a, Clover deploys clients on CNs and stores KV objects on MNs. It adopts additional monolithic metadata servers to manage the metadata, including *memory management information (MMI)* and the *hash index*. For `SEARCH` requests, clients look up the addresses of the KV objects from metadata servers and then fetch the data on MNs using `RDMA_READ` operations. For `INSERT` and `UPDATE` requests, clients allocate memory blocks from metadata servers with RPCs, write KV objects to MNs with `RDMA_WRITE` operations, and update the hash index on the metadata servers through RPCs. To prevent clients' frequent memory allocation requests from overwhelming the metadata servers, clients allocate a batch of memory blocks one at a time and cache the hash index locally. As a result, Clover achieves higher throughput under read-intensive workloads with less resource consumption.

However, the semi-disaggregated design cannot fully exploit the resource efficiency of DM due to its monolithic-server-based metadata management. On the one hand, monolithic metadata servers consume additional resources, including CPUs, memory, and RNICs. On the other hand, many compute and memory resources have to be reserved and assigned to the metadata server of Clover to achieve good performance due to the CPU-intensive nature of metadata management [41, 211, 167]. To show the resource utilization issue of Clover, we evaluate its throughput with 2 MNs, 64 clients, and a metadata server with different numbers of CPU cores. We control the number of CPU cores by assigning different percentages of CPU time with cgroup [37]. As shown in Figure 5.2, Clover has a low overall throughput with a small number of CPU cores assigned to its metadata server. At least six additional cores have to be assigned until the metadata server is no longer the performance bottleneck.

To attack the problem, FUSEE enables clients to directly access and modify the hash index and manage memory spaces on MNs with a fully disaggregated design, as shown in Figure 5.1b. Compared with Clover, resource efficiency can be improved because client-side metadata management eliminates the additional metadata servers. The overall throughput can also be improved because the computation bottleneck of metadata management no longer exists.

5.3 Challenges

This section elaborates on the challenges of constructing a fully memory-disaggregated storage system, *i.e.*, index replication and metadata corruption.

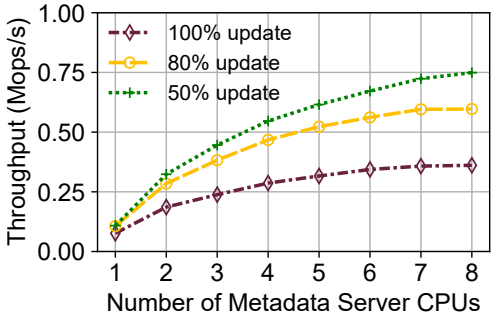


Figure 5.2: The throughput of Clover with an increasing number of metadata server CPUs.

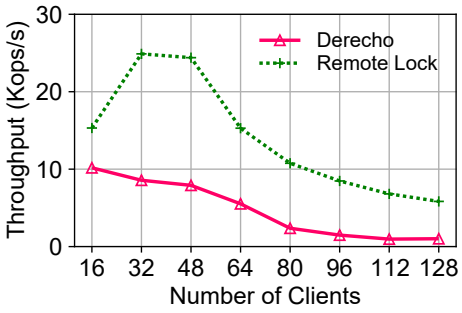


Figure 5.3: The throughput of Derecho [90] and lock-based approaches.

5.3.1 Client-Centric Index Replication

The index must be replicated to prevent data loss on MN failures. Strong consistency, *i.e.*, linearizability [83], is the most commonly adopted correctness standard for data replication because it reduces the complexity of implementing upper-level applications [30, 6, 40]. Linearizability requires that operations on an object appear to be executed in some total order that respects the operations' real-time order [83]. The key challenge of achieving a linearizable replicated hash index under the fully memory-disaggregated setting comes from the client-centric computation nature of DM.

First, existing replication methods are not applicable in the fully memory-disaggregated setting due to their server-centric nature. State machine replication (SMR) [154, 147, 101, 194, 190, 135, 152] and shared register protocols [30, 133] are two major replication approaches that can achieve strong consistency, *i.e.*, linearizability [83]. However, both approaches are designed with the assumption that a data replica is exclusively managed by the CPUs that execute data access and modification requests.

SMR considers the CPUs and the data replica as a state machine and achieves strong consistency by forcing the state ma-

chines to execute deterministic KV operations in the same global order [154, 152]. They require strong server CPUs to reach a consensus on a global operation order and apply state transition to replicas. Besides, shared register protocols view the CPU and the data replica as a shared register with `READ` and `WRITE` interfaces. Linearizability is achieved with a last-writer-wins conflict resolution scheme [133] that forces a majority of shared registers to always hold data with the newest timestamps. This approach also heavily relies on server-side CPUs to compare timestamps and apply data updates. The challenge with the server-centric approaches is that in the fully memory-disaggregated scenario, there is no such management CPU because all clients directly access and modify the hash index with one-sided RDMA verbs.

Second, naively adopting consensus protocols or remote locks among clients results in poor performance due to the amplified number of I/O operations and high concurrency control overhead on synchronizing requests. To show the performance issues of consensus protocols and remote locks, we store and replicate a shared object on two MNs and vary the number of concurrent clients. We use a state-of-the-art consensus protocol Derecho [90] and an RDMA `CAS`-based spin lock to ensure the strong consistency of the replicated object. As shown in Figure 5.3, both Derecho and lock-based approaches exhibit poor overall throughput and cannot scale with the growing number of concurrent clients.

5.3.2 Metadata Corruption

In fully memory-disaggregated storage systems, crashed CNs can leave partially modified metadata and data accessible by other healthy CNs. Since the metadata contains important system states, metadata corruption compromises the correctness of the entire system. Specifically, crashed clients may leave the in-

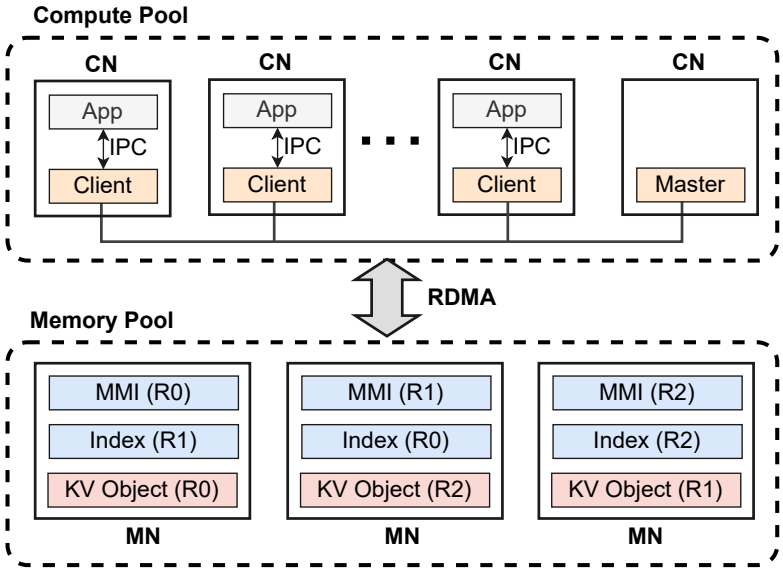


Figure 5.4: The FUSEE overview *(MMI, Index, and KV objects have multiple replicas, i.e., R_0, R_1, and R_2. R_0 is the primary replica.).*

dex in a partially modified state. Other healthy clients may not be able to access data or even access wrong data with the corrupted index. Moreover, crashed clients may allocate memory spaces but not use them, causing severe memory leakage.

This problem is akin to the crash-consistency problem in file systems [29, 158]. Existing file systems typically use write-ahead logs to recover the corrupted metadata on recovery [104, 145]. However, existing logging algorithms introduce additional I/O operations on the critical path of executing operations, resulting in high operation latency.

5.4 The FUSEE Design

5.4.1 Overview

As shown in Figure 5.4, FUSEE consists of clients, MNs, and a master. Clients provide `SEARCH`, `INSERT`, `DELETE`, and `UPDATE` interfaces for applications to access KV objects. MNs store the

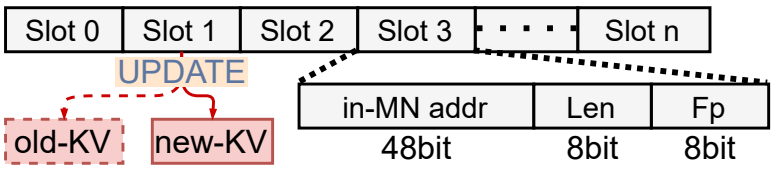


Figure 5.5: The structure of an index replica.

replicated memory management information (MMI), hash index, and KV objects. The master is a cluster management process responsible only for initializing clients and MNs and recovering data under client and MN failures.

FUSEE replicates both the hash index and KV objects to tolerate MN failures. We adopt RACE hashing (Section 5.4.2) as the index data structure and the two-level memory allocation scheme proposed in the previous chapter to allocate and replicate memory space on multiple MNs (Section 5.4.3). The SNAPSHOT replication protocol is proposed to enforce the strong consistency of the replicated hash index (Section 5.4.4). Moreover, FUSEE uses logs to handle the corrupted metadata under client failures and adopts an embedded operation log scheme to reduce the log maintenance overhead (Section 5.4.5). Other optimizations are introduced in Section 5.4.6 to further improve the system performance.

5.4.2 RACE Hashing

RACE hashing is a one-sided RDMA-friendly hash index. As shown in Figure 5.5, it contains multiple 8-byte slots, with each storing a pointer referring to the address of a KV pair, an 8-bit fingerprint (Fp), *i.e.*, a part of the key's hash value, and the length of the KV object (Len) [230]. For `SEARCH` requests, a client reads the slots of the hash index according to the hash value of the target key and then reads the KV object on MNs according to the pointer in the slot. For `UPDATE`, `INSERT`, and

`DELETE` requests, RACE hashing adopts an *out-of-place modification* scheme. It first writes the KV object to the memory pool and then modifies the corresponding slot in the hash index atomically with an `RDMA_CAS`. Nevertheless, RACE hashing only supports a single index replica.

5.4.3 Two-Level Memory Allocation

FUSEE adopts the two-level memory allocator scheme introduced in Section 3.3.2 to allocate and replicate memory blocks that hold KV objects.

The two-level memory allocation scheme shards the memory space on MNs into 2 GB memory regions. FUSEE maps each region to r MNs with consistent hashing, where r is the replication factor. Specifically, consistent hashing maps a region to a position in a hash ring. The replicas are then stored at the r MNs successively following the position and the primary region is placed on the first of the r MN.

Allocating a memory space for a KV object happens before writing the KV pair, as introduced in Section 5.4.2. A client first allocates coarse-grained memory blocks by sending `ALLOC` requests to MNs. On receiving an `ALLOC` request, an MN first allocates a memory block from one of its primary memory regions. As introduced in Section 3.3.2, the two-level memory allocator maintains a block allocation table for each region to record which client allocates from the region. The MN then records the client ID in the block allocation tables of both primary and backup regions. The coarse-grained memory allocation information is thus replicated on r MNs and can survive MN failures. Finally, it replies to the calling client with the address of the replicated memory blocks. Fine-grained allocation for KV objects is also conducted with the client-side slab allocators.

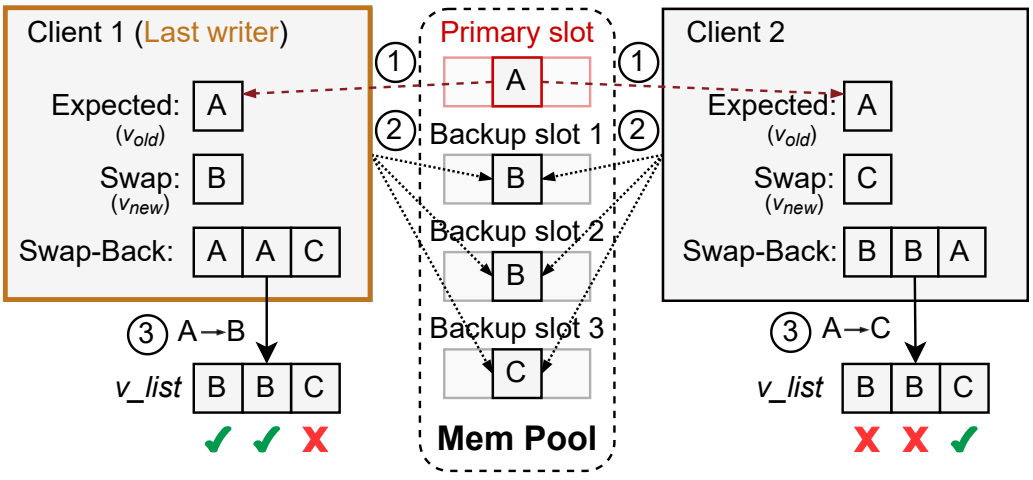


Figure 5.6: The SNAPSHOT replication protocol.

5.4.4 The SNAPSHOT Replication Protocol

In FUSEE, multiple clients concurrently read or write the same slot in the replicated hash index to execute `SEARCH` or `UPDATE` requests, as shown in Figure 5.6. To efficiently maintain the strong consistency of slot replicas in the replicated hash index, FUSEE proposes the SNAPSHOT replication protocol, a client-centric replication protocol that achieves linearizability without the expensive request serialization.

There are two main challenges to efficiently achieving linearizability under the fully memory-disaggregated setting. First, how to protect readers from reading incomplete states during read-write conflicts. Second, how to efficiently resolve write-write conflicts without introducing too many I/Os to serialize all conflicting requests on the critical path.

To address the first challenge, SNAPSHOT splits the replicated hash index into a single primary replica and multiple backup replicas. Write conflicts are resolved in all backup replicas before revising the primary slot. Hence, incomplete states during write conflicts only appear on backup replicas and the primary replica always contains the correct value. Readers can

simply read the contents in the primary replica without perceiving the incomplete states. To address the second challenge, SNAPSHOT adopts a last-writer-wins conflict resolution scheme similar to shared register protocols. SNAPSHOT leverages the *out-of-place modification* characteristic of RACE hashing that conflicting writers always write different values into the same slot because the values are pointers referring to KV objects at different locations. Three conflict-resolution rules are thus defined based on the values written by conflicting writers in backup replicas, which enable clients collaboratively to decide on a single last writer under write conflicts.

Algorithm 1 shows the `READ` and `WRITE` processes of the SNAPSHOT replication protocol. Here we focus on the execution of SNAPSHOT when no failure occurs and leave the discussion of failure handling in Section 5.5. We call the slots in the primary and backup hash indexes primary slots and backup slots.

For `READ` operations, clients directly read the values in the primary slots with `RDMA_READ`. For `WRITE` operations, SNAPSHOT first resolves write conflicts by letting conflicting writers collaboratively decide on a last writer with three conflict resolution rules and then let the decided last writer modify the primary slot. Figure 5.6 shows the process that two clients simultaneously `WRITE` the same slot. The corresponding codes and variables are shown in Algorithms 1 and 2.

Clients first read the value in the primary slot as v_{old} (①). Then each client modifies all backup slots by broadcasting `RDMA_CAS` operations to all backup slots (②) with v_{old} as the expected value and v_{new} as the swap value. On receiving an `RDMA_CAS`, the RNICs on MNs atomically modify the value in the target slot only if v_{old} matches the current value in the slot. Since all writers initiate `RDMA_CAS` operations with the same v_{old} and different v_{new}s and all backup slots initially hold v_{old}, the atomicity of `RDMA_CAS` ensures that each backup slot can only be modified

once by a single writer. As a result, the values in all backup slots will be fixed after each of them has received one `RDMA_CAS` from one writer [1]. Meanwhile, since an `RDMA_CAS` returns the value in the slot before it is modified, all clients can perceive the new values in the backup slots (③) through the return values of the broadcast of `RDMA_CAS` operations. The return values are denoted as *v_list* in Algorithm 1.

Algorithm 1 The SNAPSHOT replication protocol

```
1:  procedure READ(slot)
2:      v = RDMA_READ_primary(slot)
3:      if v = FAIL then deal with failure
4:      return v
5:  procedure WRITE(slot, v_new)
6:      v_old = RDMA_READ_primary(slot)
7:      v_list = RDMA_CAS_backups(slot, v_old, v_new)
8:      // Change all the v_olds in the v_list to v_news.
9:      v_list = change_list_value(v_list, v_old, v_new)
10:     win = EVALUATE_RULES(v_list)        ▷ The last writer returns the
    winning rule while other writers return LOSE.
11:     if win = Rule_1 then
12:         RDMA_CAS_primary(slot, v_old, v_new)
13:     else if win ∈ {Rule_2, Rule_3} then
14:         RDMA_CAS_backups(slot, v_list, v_new)
15:         RDMA_CAS_primary(slot, v_old, v_new)
16:     else if win = LOSE then
17:         repeat
18:             sleep a little bit
19:             v_check = RDMA_READ_primary(slot)
20:             if notified failure then goto Line 24
21:         until v_check ≠ v_old
22:         if v_check = FAIL then goto Line 24
23:     else if win = FAIL then
24:         deal with failure
25:     return
```

With *v_list*, SNAPSHOT defines the following three rules

[1] The process that all conflicting clients broadcast `RDMA_CAS`es to modify backup slots is just like taking a snapshot, which is why the replication protocol is named SNAPSHOT.

to let conflicting clients collaboratively decide on a last writer:

> **Rule 1**: A client that has successfully modified all the backup slots is the last writer.
>
> **Rule 2**: A client that has successfully modified a majority of backup slots is the last writer.
>
> **Rule 3**: If no last writer can be decided with the former two rules, the client that has written the minimal target value (v_{new}) is considered as the last writer.

Algorithm 2 The rule evaluation procedure of SNAPSHOT

```
1:  procedure EVALUATE_RULES(v_list, slot, v_new, v_old)
2:      v_maj = The majority value in v_list
3:      cnt_maj = The number of v_maj in v_list
4:      if FAIL ∈ v_list then
5:          return FAIL
6:      else if cnt_maj = Len(v_list) then
7:          return Rule 1 if v_maj = v_new else LOSE
8:      else if 2 * cnt_maj > Len(v_list) then
9:          return Rule 2 if v_maj = v_new else LOSE
10:     else if v_new ∉ v_list then
11:         return LOSE
12:     v_check = RDMA_READ(slot)
13:     if v_check = FAIL then
14:         return FAIL
15:     else if v_check ≠ v_old then
16:         return FINISH
17:     else if min(v_list) = v_new then
18:         return Rule 3
19:     return LOSE
```

The three rules are evaluated sequentially as shown in Algorithm 2. **Rule 1** provides a fast path when there are no conflicting modifications. **Rule 2** preserves the most successful CAS operations to minimize the overhead of executing atomic operations on RNICs when conflicts are rare [96]. Finally, **Rule**

3 ensures that the protocol can always decide on the last writer. To ensure the uniqueness of the last write, a client issues another `RDMA_READ` to check if the primary slot has been modified (Line 12, Algorithm 2) before evaluating **Rule 3**. If the primary slot has not been modified, then the `RDMA_CAS_backups` (Line 7, Algorithm 1) of the client must happen before the last writer modifies the primary slot. Hence, it is safe to evaluate **Rule 3** because the *v_list* must contain the value of the last writer if it has already been decided. Otherwise, **Rule 3** will not be evaluated because the modification of the primary slot means the decision of a last writer.

Relying on the three rules, a unique last writer can be decided without any further network communications. For example, in Figure 5.6, Client 1 is the last writer according to **Rule 2**. Client 1 then modifies the backup slots that do not yet contain its proposed value using `RDMA_CAS`es and then modifies the primary slot. Other conflicting clients iteratively `READ` the value in the primary slot and return success after finding the change in the primary slot. The primary slot may remain unmodified only under the situation when the last writer crashed, which will be discussed in Section 5.5.

Correctness. The SNAPSHOT replication protocol guarantees linearizability of the replicated hash indexes with last-writer-wins conflict resolution like shared register protocols [30, 133]. We demonstrate the correctness of SNAPSHOT using the notion of the linearizable point of KV operations. A linearizable point is a point when an operation atomically takes effect in its invocation and completion [83]. For `READ` operations, the linearizable point happens when it gets the value in the primary slot. For `WRITE` operations, the linearizable point of the last writer happens when it modifies the primary slot. Linearizable points of other conflicting writers appear instantly before the last writer modifies the primary slot. Conflicts between readers

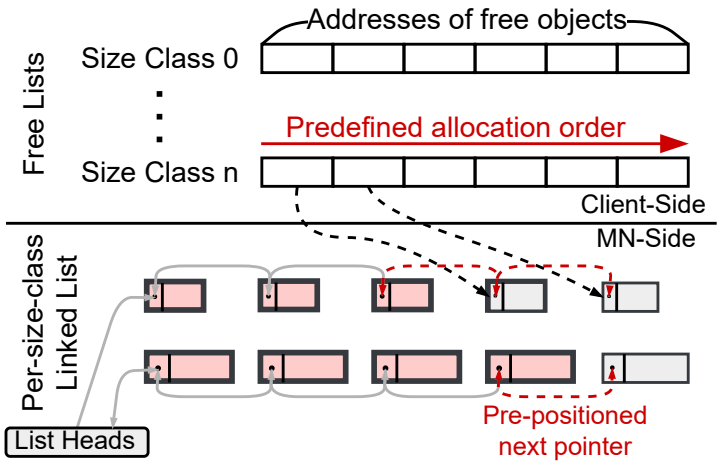


Figure 5.7: The organization of the embedded operation log.

and the last writer are resolved by RNICs because the last writer atomically modifies the primary slot using `RDMA_CAS` operations and readers access the primary slot using `RDMA_READ` operations.

Performance. SNAPSHOT guarantees a bounded worst-case latency when clients `WRITE` the hash index. Under the situation when **Rule 1** is triggered, 3 RTTs are required to finish a `WRITE` operation. Under situations when **Rule 2** or **Rule 3** is triggered, 4 or 5 RTTs are required, respectively.

5.4.5 Embedded Operation Log

Operation logs are generally adopted to repair the partially modified hash index incurred by crashed clients. Conventional operation logs need to record a log entry for each KV request that modifies the hash index. The log entries are written in an append-only manner so that the order of log entries reflects the execution order of KV requests. The recovery process can thus find the crashed request and fix the corrupted metadata by scanning the ordered log entries.

Constructing operation logs incurs high log maintenance overhead on DM since writing log entries adds additional I/Os to the

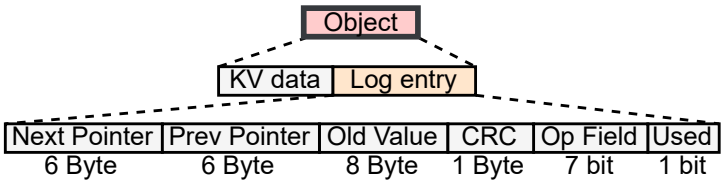


Figure 5.8: The embedded log entry.

critical paths of KV requests. To reduce the log maintenance overhead, FUSEE adopts an *embedded operation log* scheme that embeds log entries into KV objects. The embedded log entry is written together with its corresponding KV object with one `RDMA_WRITE` operation. The additional RTTs required for persisting log entries are thus eliminated.

However, by embedding log entries in KV objects, the execution order of KV requests cannot be maintained because the log entries are no longer continuous. To address this problem, the embedded operation log scheme reuses the memory allocation order to reconstruct the operation execution order. As shown in Figure 5.7, the two-level memory allocator maintains a free list for each size class locally on each client side. FUSEE maintains per-size-class linked lists in the memory pool to organize the log entries of a client in the execution order of KV requests. A per-size-class linked list is a doubly linked list that links all allocated objects of the size class in the order of their allocations. The object allocation order can reflect the execution order of KV requests because all KV requests that modify the hash index, *e.g.*, `INSERT` and `UPDATE`, need to allocate objects for new KV pairs. For `DELETE` operations, FUSEE allocates a temporary object recording the log entry and the target key and reclaims the object on finishing the `DELETE` request. FUSEE stores the list heads on MNs during the initialization of clients, which will be accessed during the recovery process of clients (Section 5.5).

As shown in Figure 5.8, an embedded log entry is a 22-byte data structure stored behind each KV object. It contains a 6-

byte *next pointer*, a 6-byte *prev pointer*, an 8-byte *old value*, a 1-byte *CRC*, a 7-bit *opcode*, and a *used* bit. The *next pointer* points to the next object of the size class that will be allocated and the *prev pointer* points to the object allocated before the current one. The *old value* records the old value of the primary slot for recovery proposes. The *CRC* is adopted to check the integrity of the *old value* under client failures. The *operation field* records the operation type, *i.e.*, `INSERT`, `UPDATE`, or `DELETE`, so that the crashed operation can be properly retried during recovery. The *used bit* indicates if an object is in use or free. Storing the *used bit* at the end of the entire object can be used to check the integrity of an entire object. This is because the order-preserving nature of `RDMA_WRITE` operations ensures that the used bit is written only after all other contents in the object have been successfully written.

FUSEE efficiently organizes per-size-class linked lists by co-designing the linked list maintenance process with the memory allocation process. Since an object is always allocated from the head of a local free list, the allocation order of each size class is pre-determined. Based on the pre-determined order, for each allocation, a client pre-positions the *next pointer* to point to the free object in the head of the local free list and the *prev pointer* to point to the last allocated object of the size class. Both the *next pointer* and the *prev pointer* are thus known before each allocation and the entire log entry can be written to MNs with the KV pair in a single `RDMA_WRITE`.

Combined with the SNAPSHOT replication protocol, the execution process is shown as follows. First, for each writer, a log entry with an empty *old value* and *CRC* is written with the KV object in a single `RDMA_WRITE`. Then, for the last writer of the SNAPSHOT replication protocol, the *old value* is modified to store the old value of the primary slot before the primary slot is modified. For other non-last writers, the used bits in their

corresponding KV log entries are reset to '0' after finding the modification of the primary slot.

5.4.6 Optimizations

Adaptive index cache. Index caching is widely adopted on RDMA-based KV stores to reduce the number of I/O operations spent on remote memory access [209, 207, 206, 192]. For a key, the index cache caches the remote addresses of the replicated index slots and the addresses of the KV objects locally. With the cached addresses, `UPDATE`, `DELETE`, and `SEARCH` requests can read KV objects in parallel with searching the hash index, reducing an RTT on cache hits. To guarantee cache coherence, an invalidation bit is stored together with each object, which is used by clients to check whether the object is valid or invalid. However, by accessing the index cache, invalid KV objects can be fetched into clients, causing read amplification.

To attack the read amplification issue, FUSEE adaptively bypasses the index cache by distinguishing read-intensive and write-intensive keys. For each cached key, FUSEE maintains an access counter and an invalid counter which increases by 1 each time the key is accessed or found to be invalid. A client calculates an *invalid ratio* $I = \frac{\text{invalid counter}}{\text{access counter}}$ for each cached key. The index cache is bypassed when accessing a key with $I > threshold$ because the key is write-intensive and the cached key address points to an invalid KV object with high probability. The *invalid ratio* can adapt to workload changes, *i.e.*, a write-intensive key becomes read-intensive, since the access counter of the key keeps increasing while the invalid counter stops. Besides, the adaptive scheme does not affect the `SEARCH` latency for most cases since only write-intensive keys bypass the cache.

RDMA-related optimizations. KV requests require multiple remote memory accesses. FUSEE adopts doorbell batching and

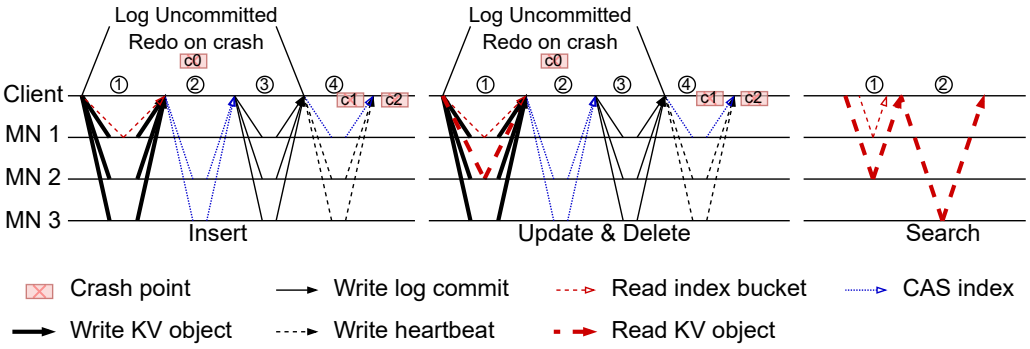


Figure 5.9: The workflows of different KV requests. *INSERT:* ① *write the KV object to all replicas and read the primary index slot.* ② *CAS all backup slots.* ③ *write the old value to the log header.* ④ *CAS the primary slot.* *UPDATE & DELETE:* ① *write the KV object, read the primary slot, and read the KV object according to the index cache.* ② *CAS backup slots.* ③ *write the old value to the log header.* ④ *CAS the primary slot.* *SEARCH:* ① *read the primary slot and the KV object according to the index cache.* ② *read the KV object on cache misses.*

selective signaling [96] to reduce RDMA overhead. Figure 5.9 shows the procedures for executing different KV requests. Each request consists of multiple phases with multiple network operations. For each phase, FUSEE adopts doorbell batching [96] to reduce the overhead of transmitting network operations from user space to RNICs and selective signaling to reduce the overhead of polling RDMA completion queues. Consequently, each phase only incurs 1 network RTT. For `INSERT`, `DELETE`, and `UPDATE` requests, four RTTs are required in general cases. For `SEARCH` requests, at most two RTTs are required and only one RTT is required in the best case due to the index cache.

5.5 Failure Handling

Similar to existing replication protocols [101, 194], FUSEE relies on a fault-tolerant master with a lease-based membership service [78] to handle failures. The master maintains a membership lease for both clients and MNs so that clients always know alive

MNs by periodically extending their leases. The failures of both clients and MNs can be detected by the master when they no longer extend their leases. Master crashes are handled by replicating the master with state machine replication [78, 190, 194]. We model check FUSEE with TLA+ [114] for safety and absence of deadlocks under MN failures.

5.5.1 Failure Model

We consider a partially synchronous system where processes, *i.e.*, clients and MNs, are equipped with loosely synchronized clocks [58, 78, 101]. FUSEE assumes *crash-stop* failures, where processes, *i.e.*, clients and MNs, may fail due to crashing and their operations are non-Byzantine.

Under this failure model, FUSEE guarantees linearizable operations, *i.e.*, each KV operation is atomically committed in a time between its invocation and completion [83]. All the objects of FUSEE are available under an arbitrary number of client crashes and at most $r-1$ MN crashes, where r is the replication factor.

5.5.2 Memory Node Crashes

MN crashes lead to failed accesses to KV objects and hash slots. The complication comes from the unavailable primary and backup slots that affect the normal execution of index `READ` and `WRITE` operations. FUSEE relies on the fault-tolerant master to execute operations on clients' behalves under MN failures. We first introduce how clients `READ`/`WRITE` the replicated slots under MN failures and then introduce the master's operations.

When executing index `WRITE` under MN crashes, FUSEE allows the last writer decided by the SNAPSHOT replication protocol to continue modifying all *alive* slots to the same value. Other writers send RPC requests to the master and wait for the

master to reply with a correct value in the replicated slots. Under situations when no last writer can be decided, the master decides the last writer and modifies all the index slots on behalf of clients. For `READ` operations, executions are not affected under the following two cases. First, if the primary slot is still alive, clients can read the primary slot normally. Second, if the primary slot crashes, clients read all *alive* backup slots. If all *alive* backup slots contain the same value, reading this value is safe because there are no write conflicts. Otherwise, clients use RPCs and rely on the master to return a correct value for the crashed slot. Since `READ` operations are only affected under write conflicts, most `READ` can continue under the read-intensive workloads that dominate in real-world situations [50, 215].

On detecting MN crashes, the master first blocks clients from further modifying the crashed slots with the lease expiration. The master then acts as a representative last writer that modifies all *alive* slots to the same value. Specifically, the master selects a value v in an *alive* backup slot and modifies all *alive* slots to v. Since the SNAPSHOT protocol modifies the backup slots before the primary slot, the values in the backup slots are always newer than the primary slot. Hence, the master choosing a value from a backup slot is correct because it proceeds the conflicting write operations. In cases where all backup slots crash, the master selects the value in the primary slot. Clients that receive old values from the master retry their write operations to guarantee that their new value is written. The master then writes the *old value* in the operation log header to prevent clients from redoing operations when recovering from crashed clients (Section 5.5.3). Finally, the master reconfigures new primary and backup slots and returns the selected value to all clients that wait for a reply. After the reconfiguration of the primary and backup slots, all KV requests can be executed normally without involving the master. During the whole process, only accesses to the crashed

slots are affected and the blocking time can be short thanks to the microsecond-scale membership service [78].

5.5.3 Client Crashes

Crashed clients may result in two issues. First, their allocated memory blocks remain unmanaged, causing memory leakage. Second, other clients may be unable to modify a replicated index slot if the crashed client is the last writer. The master uses embedded operation logs to address these two issues.

The recovery process is executed in the compute pool and consists of two steps, *i.e.*, memory re-management and index repair. Memory re-management restores the coarse-grained memory blocks allocated by the client and the fine-grained object usage information of the client. The recovery process first gets all memory blocks managed by the crashed client by letting MNs search for their local block allocation tables. Then the recovery process traverses the per-size-class linked lists to find all used objects and log entries. With the used objects and the allocated memory blocks, the recovery process can easily restore the free object lists of the crashed client. Hence, all the memory spaces of the crashed client are re-managed.

The index repair procedure then fixes the partially modified hash index. FUSEE deems all requests at the end of per-size-class linked lists as potentially crashed requests. For incomplete log entries, *i.e.*, the *used bit* at the end of the log entry is not set, the client must have crashed during writing the KV object (c0 in Figure 5.9). The object is directly reclaimed without further operation since the writing of the object has not been completed. For a log entry with an incomplete *old value* according to the *CRC* field, FUSEE redoes the request according to the *operation field* and the KV object. Under this situation, either the request belongs to the last writer that crashed before committing the

log (c1 in Figure 5.9), or it belongs to other non-last writers. In the first case, the values in the backup slots may not be consistent and the primary slot has not been modified to a new value. Redoing the request can make the backup and primary slots consistent. In the second case, since the request of crashed non-last writers has not been returned to clients, redoing the request does not violate linearizability. For a request with a complete *old value*, the request must belong to a last writer. However, the request may finish (c3) or crash before the primary slot is modified (c2). The recovery process checks the value in the primary slot (v_p) and the value in the *old value* (v_{old}) to distinguish c2 from c3. If $v_p = v_{old}$, the request crashed before the primary was modified because v_{old} records the value before index modification. Since all backup slots are consistent, the recovery process modifies the primary slot to the new value and finishes the recovery. Otherwise, the request is finished and no further operation is required.

5.5.4 Mixed Crashes

In situations where clients and MNs crash together, FUSEE recovers the failures separately. FUSEE first lets the master recover all MN crashes and then starts the recovery processes for failed clients. KV requests can proceed because the master acts as the last writer for all blocked KV requests. No request is committed twice because the master commits the operation logs on clients' behalves.

5.6 Evaluation

5.6.1 Experiment Setup

Implementation. We implement FUSEE from scratch in C++ with 13k LOC. We implement RACE hashing carefully accord-

ing to the paper due to no available open-source implementations. Coroutines are employed on clients to hide the RDMA polling overhead, as suggested in [97, 230]. The design of FUSEE is agnostic to the lower-level memory media of memory nodes, *i.e.*, any memory node with either persistent memory (PM) or DRAM that provides `READ`, `WRITE`, and 8-byte `CAS` interfaces is compatible. We adopt monolithic servers with RNICs and DRAM to serve as MNs like Clover [209] since we do not have access to smartNICs and PM. Specifically, we start an MN process on a monolithic server to register RDMA memory regions and serve memory allocation RPCs with a UDP socket. MN processes serve memory allocation requests with UDP sockets. Since the socket *receive* is a blocking system call, the process will be in the blocked state with no CPU usage most of the time.

Testbed. We run all experiments on 22 physical machines (5 MNs and 17 CNs) on the APT cluster of CloudLab [57]. Each machine is equipped with an 8-core Intel Xeon E5-2450 processor, 16GB DRAM, and a 56Gbps Mellanox ConnectX-3 IB RNIC. These machines are interconnected with 56Gbps Mellanox SX6036G switches.

Comparison. We compare FUSEE with two state-of-the-art KV stores on DM, *i.e.*, pDPM-Direct and Clover [209]. pDPM-Direct stores and manages the KV index and memory space on the clients. It uses a distributed consensus protocol to ensure metadata consistency and locks to resolve data access conflicts. We extend the open-source version of pDPM-Direct to support string keys for fair comparison in our evaluation. Clover is a semi-disaggregated KV store that adopts monolithic servers to manage memory spaces and a hash index. All `UPDATE` and `INSERT` requests have to go through the metadata server, requiring additional compute power. For both pDPM-Direct and Clover, client-side caches are enabled following their default settings. To show the effectiveness of SNAPSHOT and the adap-

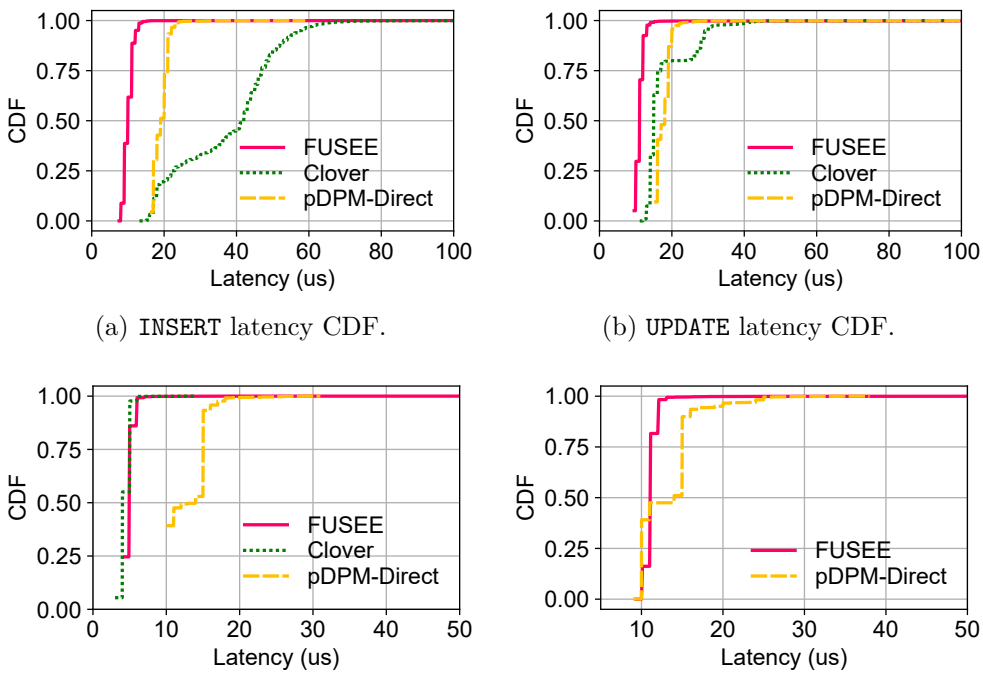
tive index cache, we implement FUSEE-CR and FUSEE-NC, two alternative versions of FUSEE. FUSEE-CR replicates index modifications by sequentially `CAS`ing all replicas to enforce sequential accesses similar to chain replication [194]. FUSEE-NC is the version of FUSEE without a client-side cache. For all these methods, we evaluate their throughput and latency with both micro and YCSB [50] benchmarks.

Since the open-source version of Clover and pDPM-Direct only support one index replica, we compare FUSEE with these two approaches with a single index replica and two data replicas in the microbenchmark (Section 5.6.2) and YCSB performance (Section 5.6.3) evaluations. When evaluating FUSEE with a single index replica, the embedded log is constructed, but the commit of the log is skipped since committing the log is used to ensure the consistency of multiple index replicas. The performance of FUSEE with multiple replicas is evaluated in the fault-tolerance evaluation (Section 5.6.4).

5.6.2 Microbenchmark Performance

We use microbenchmarks to evaluate the operation throughput and latency of the three approaches. For FUSEE and pDPM-Direct, we use 16 CNs and 2 MNs. For Clover, we use 17 CNs and 2 MNs because it needs an additional metadata server, consuming 8 more CPU cores and an additional RNIC. We do not use multiple metadata servers for Clover because the current open-source implementation of Clover only supports a single metadata server. We run 128 client processes on the 16 CNs, where each CN holds 8 clients. The `DELETE` of Clover is not tested because Clover does not support it.

Latency. To evaluate the latency of KV requests, we use a single client to iteratively execute each operation $10,000$ times. Figure 5.10 shows the cumulative distribution functions (CDFs)



(a) `INSERT` latency CDF.

(b) `UPDATE` latency CDF.

(c) `SEARCH` latency CDF.

(d) `DELETE` latency CDF.

Figure 5.10: The CDFs of different KV request latency under the microbenchmark.

of the request latency. FUSEE performs the best on `INSERT` and `UPDATE`, since the SNAPSHOT replication protocol has bounded RTTs. FUSEE has a little higher `SEARCH` latency than Clover since FUSEE reads the hash index and the KV object in a single RTT, which is slower than only reading the KV object in Clover. FUSEE has slightly higher `DELETE` latency than pDPM-Direct because FUSEE writes a log entry and reads the hash index in a single RTT, which is slower than just reading the hash index in pDPM-Direct.

Throughput. Figure 5.11 shows the throughput of the three approaches. The throughput of pDPM-Direct is limited by its remote lock, which causes extensive lock contention as the number of clients grows. For Clover, even though it consumes more hardware resources, *i.e.*, 8 additional CPU cores and an RNIC,

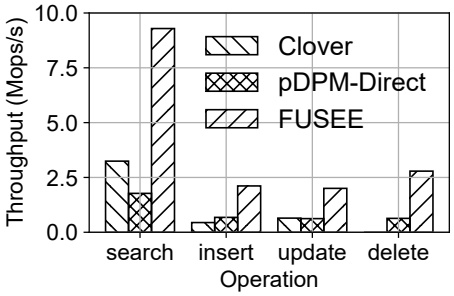


Figure 5.11: The throughputs of microbenchmark.

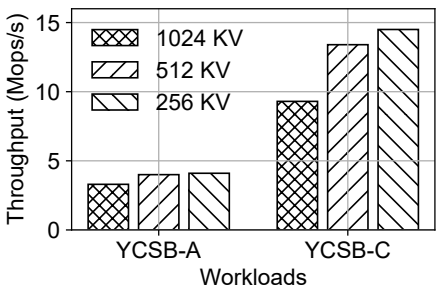


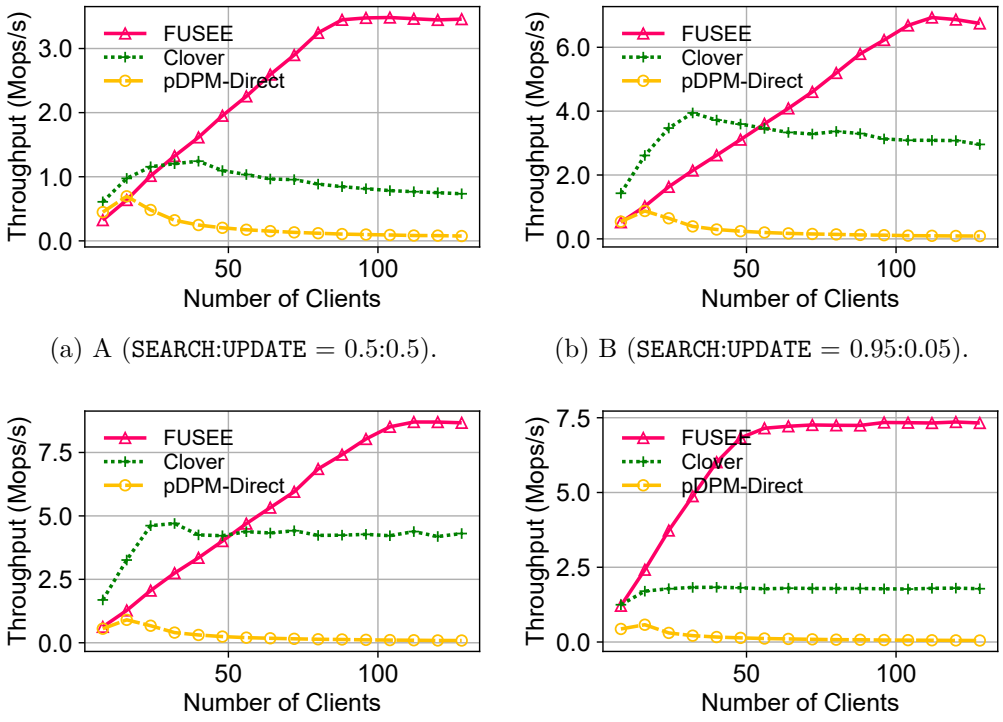
Figure 5.12: The throughput of FUSEE under different KV sizes.

the scalability is still lower than FUSEE. This is because the CPU processing power of the metadata server bottlenecks its throughput. On the contrary, FUSEE improves the overall throughput by eliminating the computation bottleneck of the metadata server and efficiently resolving conflicts with the SNAPSHOT replication protocol.

5.6.3 YCSB Performance

For YCSB benchmarks [50], we generate 100,000 keys with the Zipfian distribution ($\theta = 0.99$). We use 1024-byte KV objects, which is representative of real-world workloads [50, 36, 54]. The hardware setup is the same as microbenchmarks.

YCSB Throughput. Figure 5.13 shows the throughput of three approaches with different numbers of clients. Clover performs the best under a small number of clients since adopting the metadata server simplifies KV operations. Compared with Clover, pDPM-Direct and FUSEE require more RDMA operations to resolve index modification conflicts. As the number of clients grows, the throughput of Clover and pDPM-Direct does not increase because the throughput is bottlenecked by the metadata server and the lock contention, respectively. Compared with Clover, FUSEE scales better with the growing number of clients while consuming fewer resources. Compared with



(a) A (SEARCH:UPDATE = 0.5:0.5).

(b) B (SEARCH:UPDATE = 0.95:0.05).

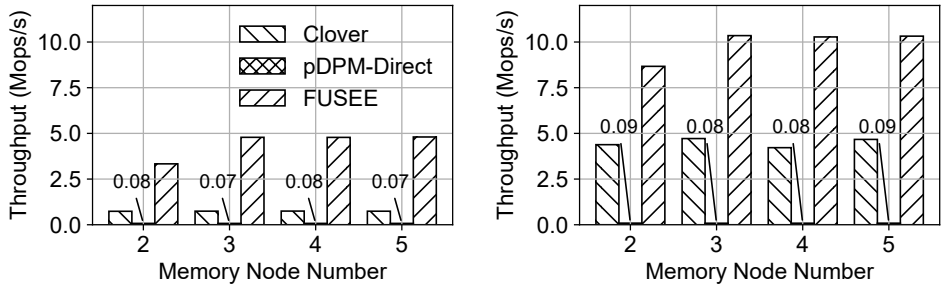
(c) C (100% SEARCH).

(d) D (SEARCH:INSERT = 0.95:0.05).

Figure 5.13: The scalability of FUSEE under different YCSB workloads.

pDPM-Direct, FUSEE improves the throughput by avoiding lock contention. When the number of clients reaches 128, the throughput of FUSEE is 4.9× and 117× higher than Clover and pDPM-Direct, respectively.

Figure 5.14 shows the throughput of the three approaches with a write-intensive workload (YCSB-A) and a read-intensive workload (YCSB-C) when varying numbers of MNs from 2 to 5 using 128 clients. The throughput of pDPM-Direct and Clover does not increase due to being limited by lock contention and the limited compute power of the metadata server, respectively. As for FUSEE, the throughput improves as the number of memory nodes increases from 2 to 3. There is no further throughput improvement because the total throughput is limited by the number of compute nodes.



(a) YCSB-A throughput.

(b) YCSB-C throughput.

Figure 5.14: The throughput with different numbers of MNs.

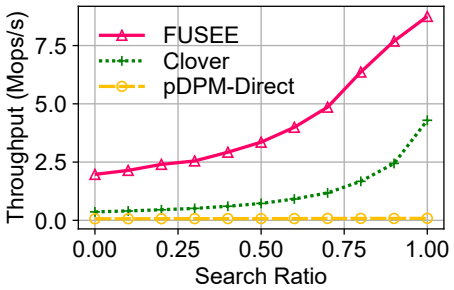


Figure 5.15: Throughput under different `SEARCH-UPDATE` ratios.

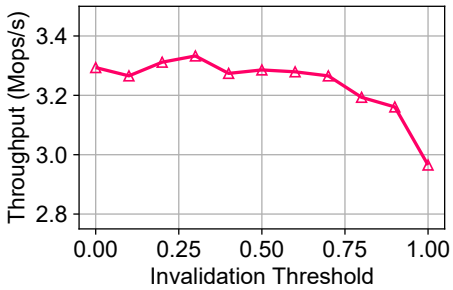
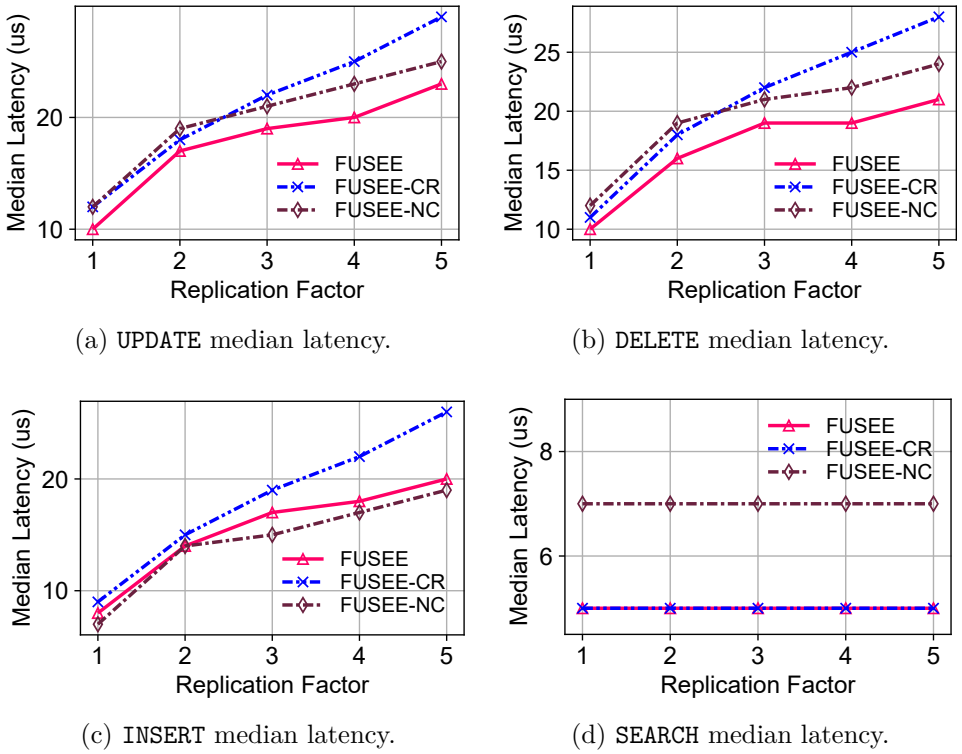


Figure 5.16: Throughput under different adaptive cache thresholds.

Figure 5.12 shows the throughput of FUSEE under smaller KV sizes. Since the throughput of FUSEE is limited by the bandwidth of MN-side RNICs, the YCSB-C throughput of FUSEE increases by 44.1% and 55.9% with 512B and 256B KV objects, respectively. The performance of FUSEE is not affected by the dataset size because the performance depends only on the number of RTTs of KV requests, which is deterministic as presented in Section 5.4.

Read-write performance. Figure 5.15 shows the throughput of the three approaches under different `SEARCH-UPDATE` ratios. As the portion of `UPDATE` grows, the throughput of all three methods decreases because `UPDATE` requests involve more RTTs. However, FUSEE exhibits the best throughput due to



(a) `UPDATE` median latency.

(b) `DELETE` median latency.

(c) `INSERT` median latency.

(d) `SEARCH` median latency.

Figure 5.17: Median operation latency of FUSEE, FUSEE-NC and FUSEE-CR under different replication factors.

eliminating the computation bottleneck of metadata servers.

Adaptive index cache performance. Figure 5.16 shows the YCSB-A throughput of FUSEE with different adaptive index cache thresholds. The throughput of FUSEE decreases with the increasing thresholds because more bandwidth is wasted on fetching invalidated KV objects with a high threshold.

5.6.4 Fault Tolerance & Elasticity

SNAPSHOT Replication Protocol. Figure 5.17 shows the median latency of FUSEE, FUSEE-NC, and FUSEE-CR with different replication factors under microbenchmarks. We set both the numbers of index replicas and data replicas to r where r is the replication factor. The latency of FUSEE-CR on `INSERT`,

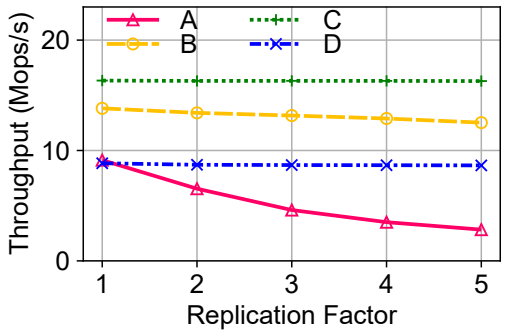


Figure 5.18: YCSB throughput under different replication factors.

`UPDATE`, and `DELETE` grows linearly as the replication factor because it modifies index replicas sequentially, and the number of RTTs equals the replication factor. Differently, the latency of FUSEE grows slightly with the replication factor because SNAPSHOT has a bounded number of RTTs. For `SEARCH` requests, FUSEE and FUSEE-CR have comparable latency since they execute `SEARCH` similarly. Compared with FUSEE-NC, FUSEE has lower latency for `UPDATE`, `DELETE`, and `SEARCH` due to fewer RTTs. The `INSERT` latency is slightly higher than that of FUSEE-NC because FUSEE spends additional time to maintain the local cache.

Figure 5.18 shows the throughput of FUSEE under different replication factors. For YCSB-A and YCSB-B, the throughput drops as the replication factor grows. The YCSB-D throughput slightly drops from 8.8 Mops to 8.6 Mops due to the read-intensive nature of YCSB-D. The YCSB-C throughput remains the same due to no index modifications.

Search under Crashed MNs. FUSEE allows `SEARCH` requests to continue when MNs crash under read-intensive workloads. Figure 5.19 shows the throughput of 9 seconds of execution, where memory node 1 crashes at the 5th second. The overall throughput drops to half of the peak throughput because all data accesses come to one MN. The throughput is then lim-

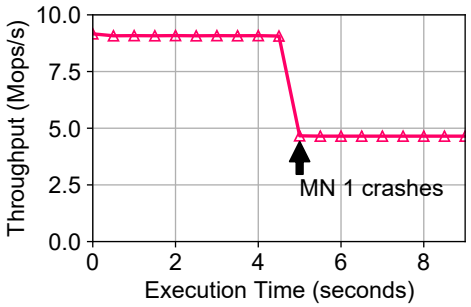


Figure 5.19: YCSB-C throughput under a crashed memory node.

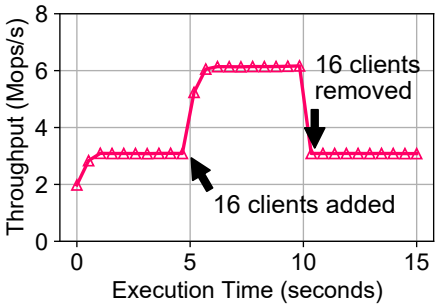


Figure 5.20: The elasticity of FUSEE.

ited by the network bandwidth of a single RNIC.

Recover from Crashed Clients. To evaluate the efficiency of a client recovering from failures, we crash and recover a client after `UPDATE` $1,000$ times. As shown in Table 5.1, FUSEE takes 177 milliseconds to recover from a client failure. The memory registration and connection re-establishment account for 92% of the total recovery time. The log traversal and KV request recovery only account for 4% of the recovery time, which implies the affordable overhead of log traversal.

Elasticity. FUSEE supports dynamically adding and shrinking clients. We show the elasticity of FUSEE by dynamically adding and removing 16 clients when running the YCSB-C workload. As shown in Figure 5.20, the throughput increases when the number of clients increases from 16 to 32 and resumes to the previous level after removing 16 clients.

5.7 Related Work

Replication protocols are widely discussed in distributed storage systems to achieve high availability and reliability. Both traditional [194, 190, 7, 115, 133, 12, 143, 74] and RDMA-based [101, 226, 188] replication protocols are designed to ensure data durability. Unfortunately, both approaches are server-

Table 5.1: Client recovery time breakdown.

Step	**Time (ms)**	**Percentage**
Recover connection & MR	163.1	92.1%
Get Metadata	0.3	0.2%
Traverse Log	3.5	2.0%
Recover KV Requests	3.5	2.0%
Construct Free List	6.6	3.7%
Total	177.0	100%

centric and heavily rely on the CPUs on the storage servers to resolve conflicts and achieve strong consistency. In contrast, SNAPSHOT is a client-centric replication protocol designed for disaggregated memory and achieves high scalability with collaborative conflict resolution.

5.8 Summary

This chapter introduces the design and implementation of fault tolerance algorithms for memory-disaggregated storage systems. We propose a client-centric replication protocol to handle memory node failures and an embedded operation log scheme to deal with compute node failures. Both algorithms achieve high performance by optimizing I/O, concurrency, and asymmetric compute capabilities, the three critical aspects of disaggregated memory. We integrate the proposed algorithms in FUSEE, the first fully memory-disaggregated storage system. Experimental results show that FUSEE outperforms the state-of-the-art approaches by up to 4.5× with less resource consumption.

□ **End of chapter.**

Chapter 6

Industrial Practice: Productionizing a Memory-Disaggregated Caching Service

Outline

Distributed caching services (DCSes) are widely adopted by various cloud applications to achieve high data access performance. Unfortunately, existing DCSes suffer from poor memory efficiency and elasticity due to their deployment on monolithic servers. This chapter introduces the DMC the industrial practice in Huawei Cloud that uses disaggregated memory (DM) to improve the memory efficiency of its DCS. We close the gap between academia and industry regarding disaggregated memory research by discussing the requirements, design principles and choices, and lessons learned in constructing a production-level memory-disaggregated caching service.

6.1 Introduction

Fully-managed distributed caching services (DCSes), *e.g.*, ElastiCache [60], MemoryStore [75], etc., enable applications to store and access data in the cloud with high throughput and low latency. They automatically manage and scale caching service instances, shifting the burden of systems management from application developers to cloud providers.

By offering DCSes, cloud providers seek to maximize resource efficiency and scale resources rapidly according to user requests. However, existing DCSes satisfy neither requirement due to their deployment on monolithic servers. First, they suffer from low memory utilization due to the over-provisioned, stranded [123], and reserved memory. According to our analysis of a large-scale production cluster of Huawei Cloud DCS, the average memory utilization of the entire cluster accounts only for 22%, which significantly increases the cost of the service. Second, scaling compute and memory resources of existing DCS instances takes up to minutes due to the time-consuming data migration on the critical path [109, 63]. Such long resource scaling time cannot adapt to the bursty workloads of cloud applications [175, 212].

Using disaggregated memory (DM) to improve memory efficiency and elasticity of storage systems is widely studied in academia [81, 118, 33, 131, 223, 125]. DM decomposes the CPU and memory from monolithic servers into independent compute and memory pools and connects the two pools with high-performance networking, *e.g.*, InfiniBand [86] and CXL [185]. Resource scaling becomes rapid since data in the memory pool are shared by all CPUs in the compute pool and no longer need to be migrated on the critical path. Memory efficiency can also be improved since 1) the decoupled CPU and memory eliminate the stranded memory, and 2) the fast resource scaling enables efficient on-demand memory allocation and eliminates the need

for users to over-provision memory.

This chapter introduces DMC, *i.e.*, **D**isaggregated **M**emory **C**aching, the industrial practice of Huawei Cloud that leverages memory disaggregation to improve the memory efficiency and elasticity of its DCS. DMC decouples a caching service instance into compute agents in compute pools and data instances in a transactional memory pool named unified memory object (UMO). The design of DMC satisfies the following fundamental requirements.

Compatibility is the primal consideration for DMC. Existing Huawei Cloud DCS is based on Redis [166], which has a large number of users and a mature ecosystem. Breaking the compatibility inevitably results in user and monetary loss.

Reliability is critical to cloud services. However, adopting memory disaggregation is a double-edged sword. On the one hand, DM isolates the failures of compute and memory resources, *e.g.*, CPU failures no longer lead to the unavailability of memory on the same node [199, 230]. Reliability can be improved by handling compute and memory failures separately. On the other hand, sharing a large memory pool introduces a huge failure domain, compromising service reliability. Moreover, DM introduces more complicated failure situations, *e.g.*, compute node failures can corrupt data [180], making it difficult to design fault-tolerance mechanisms. This calls for a careful system design in both cache service instances and the memory pool to achieve high reliability.

Adaptivity to bursty workloads. Cloud applications are featured by their varying [169] and bursty [175] workloads. Satisfying these workloads requires 1) the caching service instance to promptly adjust compute or memory resources and 2) the transactional memory pool to quickly scale memory nodes and balance the workload among all memory nodes. However, existing caching service instances couple the management of data

with the execution of user requests [166, 139, 165], which still incurs time-consuming data migration on the critical path of resource scaling when ported to DM. Besides, the performance of existing transactional memory pools [55, 62, 63] is inevitably hindered for a long period during load balancing and resource scaling due to the expensive data migration. Consequently, DMC has to rearchitect the management of data in existing caching service instances and design efficient data migration techniques for the memory pool.

Performance. Porting DCS to DM incurs performance penalties since the nanosecond-scale local memory accesses are amplified by an order of magnitude [97, 177, 224]. DMC has to mitigate the performance degradation on DM to minimize the negative effects on user applications.

DMC achieves all these requirements. To be ***compatible*** with existing DCSes, we construct DMC by disaggregating a Redis server. To achieve high ***reliability***, DMC adopts a decoupled replication scheme to isolate failures in the software layer and carefully handle the complex failure situations. Besides, in the memory pool, we introduce an on-demand connection management scheme that reduces the failure domain by connecting a compute agent to a limited number of memory nodes in an on-demand manner. To ***adapt to bursty workloads***, we adopt a logical sharding scheme that shards requests to compute agents but preserves their ability to access all data in the memory pool. This enables prompt resource scaling since data migration is no longer on the critical path of resource scaling. In the memory pool design, we propose a novel copy-free memory region migration scheme to efficiently balance loads and scale memory nodes. Finally, to mitigate the ***performance*** penalty caused by memory disaggregation, we design a write-through data cache and a balanced read scheme in our caching service instance. The former hides the remote memory access latency and the latter

prevents a single compute agent from becoming the performance bottleneck.

We deploy the DMC and evaluate it in an experimental cluster. DMC improves memory utilization by up to 2.6×. We also evaluate the throughput and latency of DMC with YCSB [50] and Twitter workloads [215]. The performance loss introduced by DM is less than 10% in normal cases. Moreover, DMC performs better than DCS by up to 9× and 1.25× during resource scaling and in the cluster mode, respectively.

The contribution of this chapter is summarized as follows:

- We analyze the memory utilization of Huawei Cloud DCS, identify its memory utilization issues, and discuss the reasons for the low memory utilization.
- We bridge the gap between academia and the industry by exploring the design choices of memory-disaggregating Redis. We introduce DMC, the disaggregated memory caching service in Huawei Cloud.
- We deploy DMC and evaluate its performance and memory utilization, showing the effectiveness of our design.

6.2 Background and Motivation

In this section, we first introduce the resource utilization and elasticity issues of the monolithic-server-based DCS in Huawei Cloud. We then show how DM can mitigate these issues.

6.2.1 Huawei Cloud's DCS

Figure 6.1 shows the monolithic-server-based DCS in Huawei Cloud. DCS uses Redis [166] to construct a caching service. User applications are executed in virtual machines (VMs) rented from the elastic cloud server (ECS) service of Huawei Cloud. User

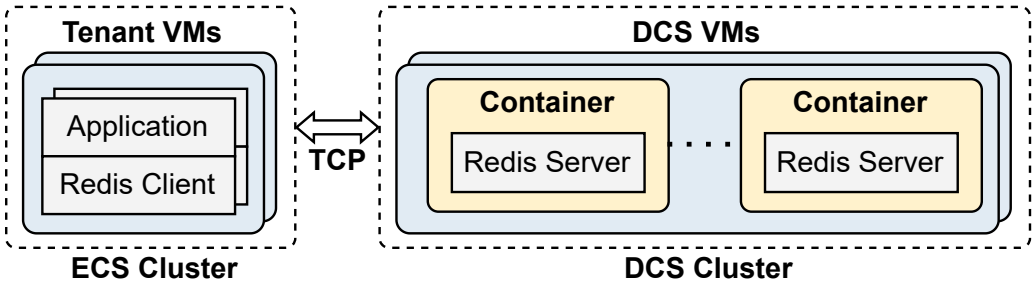


Figure 6.1: The architecture of Huawei Cloud DCS.

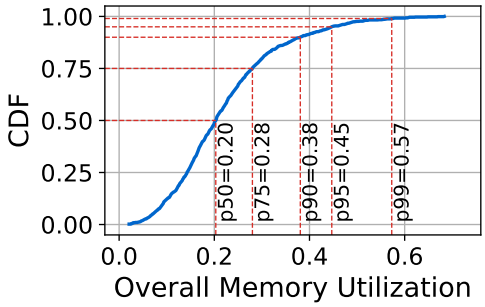


Figure 6.2: The CDF of the memory utilization of all nodes in the production cluster.

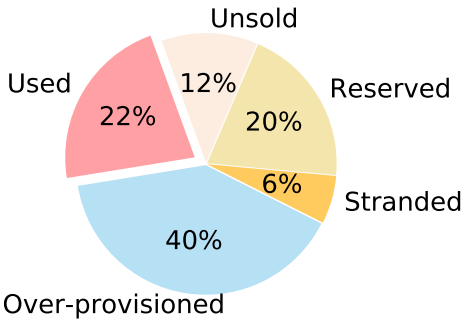
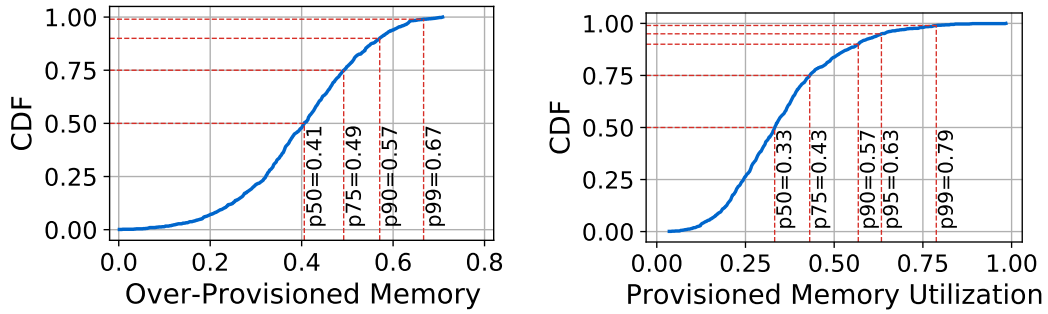


Figure 6.3: The breakdown of memory utilization in the production cluster.

VMs and DCS clusters are connected with TCP networking. Applications use the Redis client library to communicate with Redis servers in the DCS cluster. The DCS cluster is composed of VMs allocated from a lower-level computing infrastructure. Each VM has 64 CPU cores and 256 GB DRAM. DCS deploys Redis servers in containers to achieve lightweight virtualization and performance isolation. There are two problems with such a monolithic-server-based DCS, *i.e.*, low memory utilization and poor elasticity.

Low Memory Utilization

We collect resource utilization traces from a production DCS cluster in Huawei Cloud and analyze its memory utilization. Fig-



(a) Over-provisioned memory.

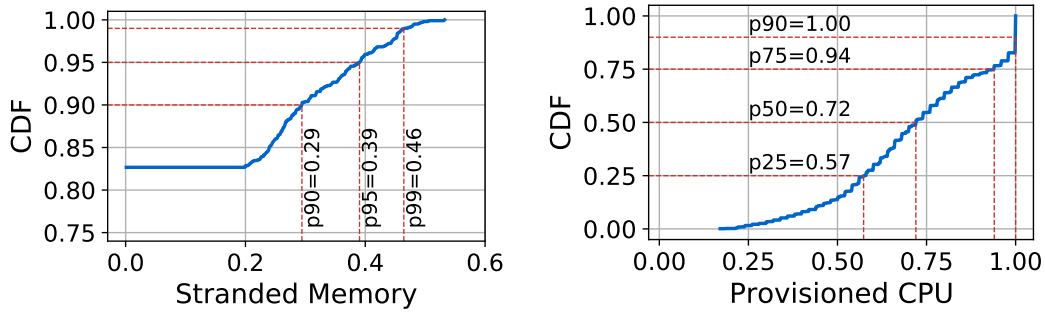
(b) Utilization of provisioned memory.

Figure 6.4: The CDFs of the over-provisioned memory and the utilization of the provisioned memory in the production cluster.

ure 6.2 shows the cumulative distribution function (CDF) of the memory utilization of all nodes in the cluster. 90% of nodes use less than 38% of memory, and the average memory utilization of the cluster is 22%. We classify the unused memory into four categories, *i.e.*, over-provisioned user memory, stranded memory, reserved memory, and unsold memory, and show the memory utilization breakdown in Figure 6.3.

1) Over-provisioned user memory is memory provisioned to DCS instances but not filled to store user data, which accounts for 40% of memory in the cluster. As shown in Figure 6.4, 50% of nodes contain more than 41% of memory over-provisioned by users. The medium utilization rate of provisioned memory is 33% for all users.

We identify two issues of DCS that motivate users to over-provision memory. First, the minute-scale resource scaling urges users to reserve the maximum memory required by their applications so that the number of dynamic resource scaling can be reduced. As a result, memory is wasted under normal use cases most of the time. Second, to prevent the burst of user requests from overwhelming the memory capacities of DCS instances, cloud providers remind users to expand the memory capacities of their DCS instances when their memory utilization exceeds



(a) Stranded memory.

(b) Provisioned CPU.

Figure 6.5: The CDFs of the proportion of stranded memory and provisioned CPU in the production cluster.

80%, which leaves at least 20% of memory in containers unused.

2) Stranded memory refers to the memory in DCS VMs that cannot be used to construct containers due to the lack of available CPUs [123], which account for 6% of the memory in the cluster. As shown in Figure 6.5, 10% of nodes have more than 29% of memory stranded, while only 50% of nodes have provisioned CPU cores of less than 72%. The root cause for the stranded memory is the coupling of CPU and memory on monolithic servers. Such a problem is unavoidable in monolithic-server-based DCS clusters since the proportion of CPU and memory of VMs in the DCS clusters configured by cloud providers cannot always match user requirements.

3) Reserved memory is the memory reserved in the cluster to handle users' burst of instance creation requests, *e.g.*, on Black Fridays [72]. A DCS instance can be created rapidly when its required resources can be satisfied by the reserved resources in the cluster. Otherwise, DCS has to first apply for more VMs from the computing infrastructure to increase the compute and memory capacity, which usually takes minutes to complete. Existing DCS uses a fixed threshold to reserve memory, *i.e.*, new VMs are allocated and added to DCS clusters whenever their

memory utilization exceeds 80%.

4) Unsold memory refers to the remaining unused memory in the cluster that does not belong to the other three categories. Memory in this category is left unused due to the difficulty in achieving optimal memory utilization when provisioning and freeing resources for caching service instances in an online manner [177]. This category accounts for 12% of unused memory in the production cluster.

Poor Elasticity

Elasticity refers to the ability of service instances to quickly and dynamically scale compute and memory resources according to the demand of upper-level applications [19]. Huawei Cloud DCS supports both vertical and horizontal scaling. However, both schemes suffer from slow resource adjustments when users adjust memory capacities.

Vertical scaling adjusts the resources of existing DCS instances. In the process, DCS first launches a new container with the required memory size, copies all data from the current container to the new one, and routes user requests to the new container by modifying the DNS entry in the DCS cluster. The performance of the DCS instance will be affected for minutes due to the time-consuming data migration.

Horizontal scaling scales out resources by adding more Redis servers to a Redis cluster, which is used in the cluster mode of DCS instances. A DCS cluster consists of multiple DCS servers. Objects are sharded to nodes in the cluster according to the hash values of their keys. To execute horizontal scaling, DCS launches a new Redis node in a container and adds the node to the cluster. Then, objects are rebalanced to the newly created node, which involves a large amount of data migration. Although users can normally read and write cached data during horizontal scaling, the performance of the entire cluster will be affected due to the

additional network bandwidth and CPU cycles spent on data migration [109, 164].

6.2.2 Opportunity: Disaggregated Memory

Using DM can improve the elasticity and memory efficiency of existing DCSes. First, resources on DM can be scaled rapidly. For both vertical and horizontal scaling, the key problem that prohibits instant resource scaling is the expensive data migration on the critical path. With DM, data migration can be eliminated from the critical path since data in the memory pool are shared by all CPUs in the compute pool. Only when the CPU or network bandwidth of an MN becomes a bottleneck does data need to be migrated to achieve load balance in the memory pool. Such data migration can be executed asynchronously without affecting request executions [179].

Moreover, DM can reduce the over-provisioned, stranded, and unsold memory in existing DCS clusters due to its rapid resource adjustment and decoupled resource management. Specifically, for over-provisioned memory, cloud providers no longer need to allocate all the required memory to users on instance creation since memory can be allocated promptly and flexibly in an on-demand manner. Meanwhile, users no longer need to provision memory for peak usage since resources can now be scaled rapidly on DM. Besides, DM eliminates stranded memory since most memory is managed in the memory pool and can be used by all CPUs in the compute pool. Moreover, existing disaggregated memory pools manage memory in coarse-grained fix-sized memory regions. The unsold memory caused by resource scheduling can also be reduced due to the simplified memory allocation.

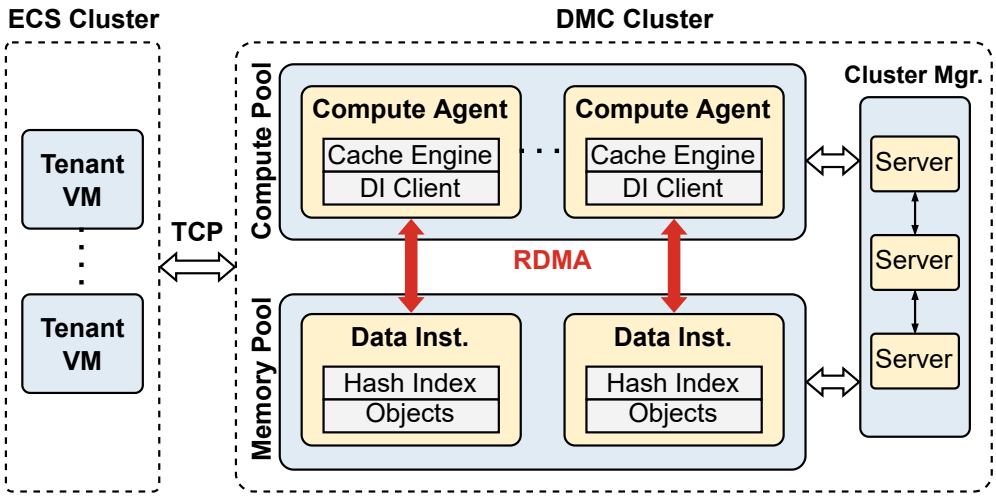


Figure 6.6: The overview of DMC.

6.3 Overview and Design Principles

This section introduces the overall architecture of DMC and discuss three design choices, *i.e.*, replication, data sharding, and caching, in detail.

6.3.1 Overview

Figure 6.6 shows the architecture of DMC. Tenant VMs use TCP-based Redis client library to communicate with caching service instances. The DMC cluster consists of a compute pool and a transactional memory pool, interconnected with RDMA. CNs in the compute pool have low memory-to-CPU ratios, *e.g.*, 1:2. MNs in the memory pool have high memory-to-CPU ratios, *e.g.*, 32:1. The memory pool provides fine-grained object allocation and transactional object read and write interfaces for the compute pool to manage data. Objects are replicated and partitioned to different MNs by the cluster manager to ensure reliability and achieve load balance.

Caching service instances are disaggregated into multiple compute agents and data instances. Compute agents execute user

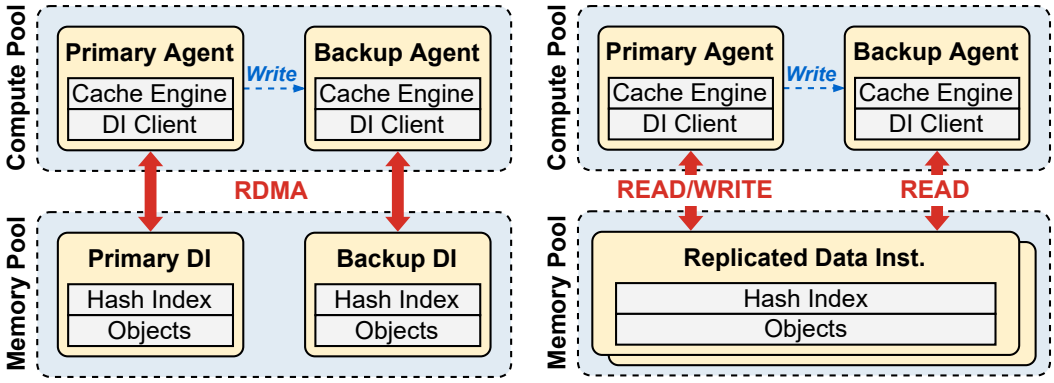
requests and manage the two-level tiered memory, *i.e.*, local DRAM in the compute pool and remote memory in the memory pool. They adopt a cache engine, *i.e.*, a modified version of the Redis server, to manage local DRAM, and a data instance (DI) client to access remote memory. Data instances are passive entities managed by DI clients to store objects and hash indexes. The cluster manager book-keeps all compute agents and data instances owned by each instance.

Over this architecture, we carefully make three design decisions when disaggregating the Redis server in terms of replication, data sharding, and caching. Our design is guided by the following three design principles:

- ***Principle 1***: Decouple compute and memory failures in the software layer to fully exploit the DM's benefit of hardware failure isolation.
- ***Principle 2***: Almost share everything for better elasticity, resource efficiency, and performance.
- ***Principle 3***: Reduce the number of remote memory accesses to achieve higher performance.

6.3.2 Design Choice 1: Replication

Replication is widely adopted in DCSes to ensure upper-level applications do not experience severe performance degradation on node failures. Huawei Cloud DCS adopts the default asynchronous replication scheme of Redis. In a standalone DCS instance, there is one primary node and multiple backup nodes. The primary node is responsible for handling both read and write requests, while backup nodes only serve read requests. When a write request arrives on the primary node, it modifies its data, appends a command recording this modification to a command backlog, and replies to the user. Primary nodes



(a) The coupled replication scheme. (b) The decoupled replication scheme.

Figure 6.7: Design choices regarding replication in DMC.

send their command backlogs asynchronously to all their backup nodes to keep them updated, which prevents user requests from being blocked by the time-consuming data synchronization. On receiving commands from the primary node, backup nodes modify their data and send an acknowledgment back. Acknowledgments are sent in a batched manner to reduce the CPU overhead of processing acknowledgments on the primary node. When a primary node crashes, a backup node is promoted to become the new primary with slightly outdated data. When a backup node crashes, the cluster manager starts a new backup node and sends it a snapshot of data on the primary node to make it fully synchronized.

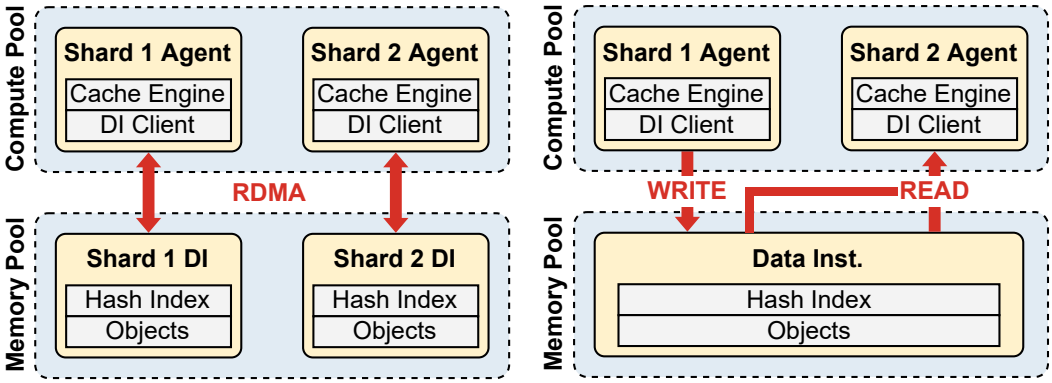
On DM, a straightforward approach that achieves replication and minimizes development overhead would be ***coupled replication***. As shown in Figure 6.7a, the coupled replication disaggregates primary and backup nodes individually. The replication and recovery protocols are the same as that of Redis with two key differences: 1) compute agents execute the replication protocol, and 2) compute agents need to manage both data in local DRAM and in the remote memory pool.

Unfortunately, the coupled replication scheme violates *Prin-*

ciple 1 and suffers from a long failure recovery time. Specifically, on DM, the failures in compute agents and data instances are isolated, *i.e.*, the data instance of a crashed compute agent can still be normally accessed. The coupled replication scheme couples the failures of compute and data in the software layer since the granularity of failure handling is a monolithic DCS node. A data instance shares the fate with its owning compute agent and is no longer useful when its owning compute agent crashes. Consequently, the time-consuming full-synchronization process still needs to be executed when either a compute agent or a data instance crashes.

We propose a ***decoupled replication*** scheme to satisfy *Principle 1*. As shown in Figure 6.7b, compute agents and data instances are replicated separately in the compute and the memory pool. In the memory pool, each data instance is replicated with RDMA-based optimistic concurrency control and two-phase commit protocols, which can achieve strong consistency and high availability. All compute agents share the same replicated data instance. For compute agents, DMC inherits the asynchronous replication protocol of Redis but only allows the primary agent to modify the data instance. When executing write requests, a primary agent updates both its local DRAM cache and the data instance, and then asynchronously forwards requests to all backup agents. Backup agents only update their local caches when they receive a forwarded request. When serving read requests, both primary and backup agents serve the requests with their local caches or fetch data from the data instance on cache misses.

The decoupled replication scheme isolates failures on compute agents and data instances and achieves fast failure recovery. When a compute agent fails, DMC starts a new compute agent in the compute pool and promotes a backup agent on primary agent failures. No data needs to be transmitted since the data



(a) The full data sharding scheme. (b) The logical data sharding scheme.

Figure 6.8: Design choices regarding data sharding in DMC.

instance is not affected. The recovery completes instantly once the new compute agent is launched. When a data instance fails, DMC launches a new data instance and synchronizes it with other alive data instances asynchronously. User applications are not affected since compute agents can still serve requests using alive data instances.

6.3.3 Design Choice 2: Data Sharding

The cluster mode of Redis shards data to multiple nodes to achieve high throughput. Specifically, a standalone Redis server uses a hash table with 16,384 slots to index all objects [166]. A Redis cluster shards data by partitioning hash slots on different nodes. Each node is only responsible for managing objects in its assigned slots. On the client side, the Redis client library caches the mapping of slots to route requests to the correct nodes.

When adding or removing nodes from a Redis cluster, the cluster manager generates a new mapping of slots and migrates data according to the mapping. The cluster stays available during the process. A user request is only blocked for a short time when it accesses an object that is hashed to a migrating slot.

The request will be redirected to the new node and keep retrying until the slot completes migration.

On DM, a straightforward approach to be compatible with Redis clusters is ***full data sharding***. As shown in Figure 6.8a, each compute agent owns a data instance and manages all objects of its assigned shard in the data instance. Inside each data instance, a hash table is adopted to index objects.

However, such an approach fails to satisfy *Principle 2*, leading to poor elasticity and limited performance. First, dynamically scaling compute resources, *i.e.*, compute agents, is still time-consuming. This is because compute and memory resources are still coupled in the software layer when we shard data into isolated data instances and bind them to independent compute agents. Data still needs to be migrated among data instances when remapping hash slots among compute agents. Second, fixing a compute agent to a single shard of data cannot fully exploit the compute resource under skewed workloads. The overall throughput will inevitably be bottlenecked by the compute agents of the hottest shard while other compute agents are not fully utilized.

We propose a ***logical data sharding*** scheme to satisfy *Principle 2*. As shown in Figure 6.8b, DMC only shards the right to write data to compute agents but preserves their abilities to read the entire key space. Specifically, all compute agent shares a common data instance that holds a global hash table and all cached objects. Objects in the data instance are partitioned to multiple MNs for load balance. Each compute agent exclusively owns a segment of the hash table. Write requests are exclusively executed by compute agents that own the target data. Read requests are routed to the owning compute agent of each shard in normal cases. When some compute agents become overloaded, we balance their read requests to all other compute agents to achieve better load balance.

Table 6.1: The design choices of compute-side cache. Each slot indicates which option is better. The underlined is chosen by DMC. AC and DC refer to the address cache and the data cache. WB and WT stand for write-back and write-through strategy. I and E are the abbreviations for inclusive and exclusive caches. C and R refer to coherent and relaxed coherence.

Aspects	**AC VS DC**	**WB VS WT**	**I VS E**	**C VS R**
Performance	DC	WB	E	R
Failure Isolation	Same	WT	I	Same

The logical sharding scheme can achieve better elasticity and resource efficiency. First, compute resources can be scaled rapidly due to the shared data instance. When adding or removing compute agents, we only need to logically reshard the management of hash slots and inform compute agents about the new slot mapping. Second, as the caching workload is read-intensive on Huawei Cloud, the logical partition scheme can effectively achieve load balance and better performance with the balanced read scheme.

6.3.4 Design Choice 3: Compute-Side Cache

Compute-side caches are widely adopted in storage systems on DM [131, 179, 180, 230] to reduce remote memory access overhead. The design of the compute-side cache affects two critical aspects of DMC, *i.e.*, performance and failure isolation. The performance can be evaluated by the number of remote memory accesses reduced by the cache. Failure isolation implies whether the cache introduces additional coupling of failures between data and compute. According to *Principles 1 and 3*, our goal is to decouple the failures between compute and data while maximizing the performance gain of the cache.

DMC carefully inspects the following four basic aspects for a compute-side cache regarding performance and failure isolation,

as shown in Table 6.1:

Address Cache (AC) VS Data Cache (DC). Regarding the content of the cache, there are two design choices. Address caches store the addresses of KV items or part of the data index on CNs [131, 206]. The number of remote memory accesses can be reduced by shortcutting the process of remote index searches. Data cache directly stores KV items on CNs [202, 32], which can reduce more number of remote memory accesses than address caches since no remote memory accesses are required on cache hits. In terms of failure isolation, both approaches perform similarly since the content in the cache does not lead to coupled failures. Consequently, according to *Principle 3*, DMC caches data instead of addresses.

Write-Back (WB) VS Write-Through (WT). Regarding the write strategy of the cache, there are two alternatives. The write-back strategy updates data in the cache and only modifies data in the memory pool during cache evictions. Such an approach achieves better write performance since all write requests are executed in local memory. However, it introduces coupled failure between compute and data since the updated data could be lost together with the crashed compute agent, causing data inconsistency. DMC chooses to use the write-through strategy according to *Principle 1* to achieve failure isolation. The write-through strategy updates the memory pool together with the cache, avoiding data loss on compute agent failures. However, its performance is lower than the write-back strategy due to the additional remote memory accesses on the critical path. We believe this is a price worth paying compared with designing new protocols to deal with coupled failures. Besides, the performance loss is negligible due to the read-intensive workload in Huawei Cloud.

Inclusive (I) VS Exclusive (E). Regarding whether the cached content is in the lower-level storage, *i.e.*, the memory

pool, existing caches can be classified into inclusive and exclusive ones. For exclusive caches, data is either stored in the cache or in the memory pool. The performance of upper-level applications could be better with an exclusive compute-side cache since more data can be held in the entire caching service instance. More data accesses can be served by the faster caching service instance instead of the slower persistent storage services. However, such an approach couples the failure between data and compute since data can be lost on compute agent failures. According to *Principle 1*, DMC chooses to employ an inclusive design to achieve better failure isolation.

Coherent (C) VS Relaxed (R). Finally, regarding the coherence guarantee of the cache, we can have coherent and relaxed caches. Coherent caches guarantee that all data accesses return the most updated data, while the relaxed ones allow the returned data to be temporarily outdated. Both approaches do not affect compute and memory node failures, while relaxed caches can achieve higher performance due to simpler computation overhead. The selection of cache coherence depends on the requirement of upper-level applications. Similar to Redis-based DCSes, DMC provides a relaxed coherence guarantee between replicated compute agents.

Putting all these together, the compute-side cache of DMC is designed to be an *inclusive data cache* with a *write-through strategy* and *relaxed coherence guarantees*. Such a design satisfies *Principles 1 and 3* and balances the failure isolation and performance of caching service instances.

6.4 Caching Service Instance

Figure 6.9 shows a DMC instance with all the previous design choices. A DMC instance contains multiple logical shards in the compute pool and a replicated data instance in the memory

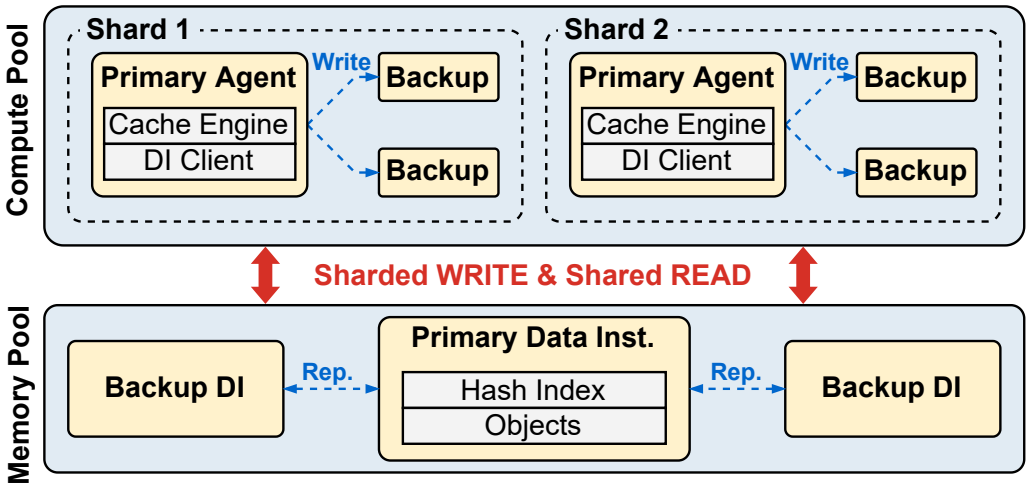


Figure 6.9: The architecture of a DMC instance.

pool. Each logical shard corresponds to a shard of hash table in a Redis cluster which has one primary agent and multiple backup agents. Primary agents serve write requests to their corresponding logical shard. All compute agents, *i.e.*, primary and backup agents, can equally serve all read requests to the entire key space to achieve better load balance and resource efficiency. Each compute agent contains a cache engine and a DI client. A cache engine is responsible for maintaining the coherence and content of the local DRAM cache. A DI client provides a memory-disaggregated caching library that executes the caching algorithm configured by users and provides key-value interfaces, *i.e.*, *Set* and *Get*, for the cache engine to manage data in the memory pool.

6.4.1 Cache Engine

A cache engine manages the DRAM of compute agents as an inclusive data cache with a write-through strategy and relaxed coherence guarantees. To reduce the cache synchronization overhead between compute agents, we restrict the number of com-

pute agents needed to be synchronized by caching data only in compute agents of their own shards. When serving read requests, a compute agent first searches objects in their local caches and relies on DI clients to fetch objects from the memory pool on local cache misses. When serving write requests, primary agents update both the data instance and their local caches if the updated data hits in local caches. Write requests are then asynchronously forwarded to all backup agents. Cache engines of backup agents update objects in their local cache and acknowledge the primary agent in batch when they receive forwarded write requests. To further reduce the replication overhead, DMC adopts a replicated cache scheme which forces the local cache of backup agents to be the same as their primary agents. The number of forwarded write requests can thus be reduced since primary agents only need to write requests that hit their local caches.

The local cache is managed by cache engines with the same caching algorithm as the DMC instance. Cache engines of primary agents leverage the hotness information maintained by DI clients to ensure their local DRAM always holds globally hot objects. When a cache engine fetches an object due to a local cache miss, the DI client returns the object and its hotness. The cache engine then samples multiple objects in its local cache and evicts the coldest one if the fetched object is hotter. Evicted objects are directly dropped due to the write-through strategy. The eviction and the insertion operations of the local cache are also forwarded asynchronously to backup agents to keep their local caches replicated.

6.4.2 Data Instance Client

The DI client provides *Set* and *Get* interfaces for cache engines to manage data in the memory pool. It maintains a global hash

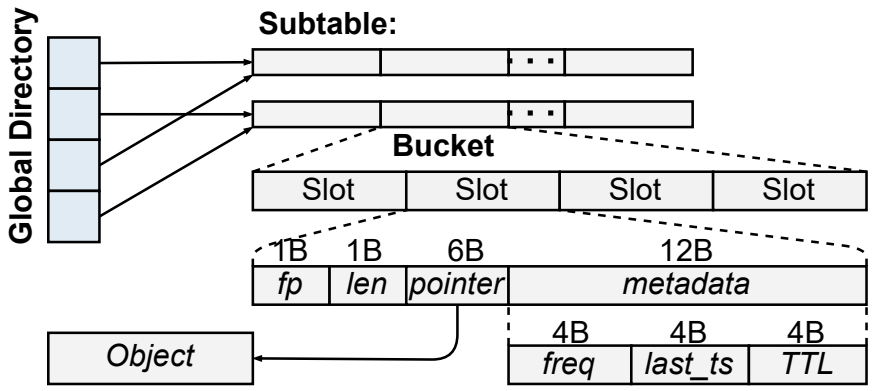


Figure 6.10: The memory-disaggregated hash table structure.

table to index cached objects and record object hotness information to execute caching algorithms. The key challenges are coordinating concurrent accesses from multiple compute agents and efficiently executing caching algorithms. The DI client employs two key techniques, *i.e.*, a memory-disaggregated hash table, and a client-centric caching framework, to address these two challenges.

Memory-Disaggregated Hash Table

The structure of the memory-disaggregated hash table is similar to RACE hashing [230]. As shown in Figure 6.10, the hash table contains a global directory and multiple subtables. The global directory is an array recording the addresses of subtables and is used to achieve on-demand extension. Each subtable contains multiple buckets with each holding multiple slots. Each slot stores a 1-byte fingerprint (*fp*) as a part of the hash value to accelerate *Get* operations, a 1-byte length (*len*) to record the size of the object, a 6-byte pointer (*pointer*) indicating the address of the object in the memory pool, and a 12-byte metadata used for cache eviction.

DI clients leverage the transactional read and write interfaces provided by the memory pool to execute *Get* and *Set* operations.

The process is described as follows:

Get. A DI client first calculates the hash value of the requested key and reads the global directory in the memory pool to identify the subtable and bucket the object belongs to. It then reads the entire bucket from the memory pool, matches the fingerprints of slots in the bucket with the hash value, and fetches the object according to the pointer in the slot that matches the fingerprint. Three remote memory accesses are required in the process.

Set. *Set* operations are executed in an out-of-place manner. A DI client first allocates a new memory block from the memory pool or evicts an object if the data instance is full. Then it writes the object to the allocated memory block and executes a *Get* operation to see if the object is stored in the data instance. If a matching object is found, the DI client atomically modifies the slot of the object to point to the newly allocated memory block with a transactional remote memory write. Otherwise, it finds an empty slot in the bucket and modifies the slot similarly. During the process, five remote memory accesses are involved.

Since the number of remote memory accesses is critical to the performance of DMC, DI clients cache the global directory in the local DRAM of compute agents to avoid reading the global directory before each data access, similar to RACE hashing [230]. The number of remote memory accesses for *Get* and *Set* are thus reduced to two and four in most cases.

Client-Centric Caching Framework

DI clients adopt a client-centric caching framework similar to Ditto [179] to efficiently execute various caching algorithms. The key challenge of executing caching algorithms is that the local DRAM cache and the CPU-bypass RDMA make it difficult to monitor object access and maintain object hotness. Specifically, the local DRAM cache managed by the cache engine hides object

accesses from DI clients on local cache hits. Besides, RDMA bypasses CPUs on MNs when accessing cached objects, prohibiting the memory pool from monitoring object accesses. Moreover, DI clients cannot monitor data accesses individually since they are only aware of their own data accesses while caching algorithms need the global access information from all DI clients to execute.

First, to collect the access information hidden from the local DRAM cache, cache engines record data accesses on cache hits and report these accesses to DI clients in batch. Second, to efficiently monitor object accesses from all DI clients, the client-centric caching framework adopts a distributed access monitoring scheme. Similar to Redis, we associate each cached object with a small metadata in the memory pool to record its access information. The metadata for each object contains an access timestamp, a frequency counter, and a time to live (TTL) value, which is sufficient to execute all caching algorithms supported by Redis, *i.e.*, LRU, LFU, TTL, and Random. Specifically, TTL-based eviction removes all expired objects according to their TTLs. The TTL value is written to the metadata in the memory pool when the object is inserted. LRU and LFU evict objects according to their access timestamps and frequency counters, respectively. Access timestamps and frequency counters are updated atomically with a remote memory write after each object access.

Following the design of Redis, DMC adopts a sample-based eviction scheme. DI clients sample multiple objects and evict the coldest one according to the access information in their metadata. However, Redis stores metadata together with objects, which incurs multiple remote memory accesses when sampling objects in the remote memory pool. Similar to Ditto [179], we store metadata in the hash table to improve the efficiency of sampling objects, as shown in Figure 6.10. In this way, sampling can be efficiently achieved by generating a random integer

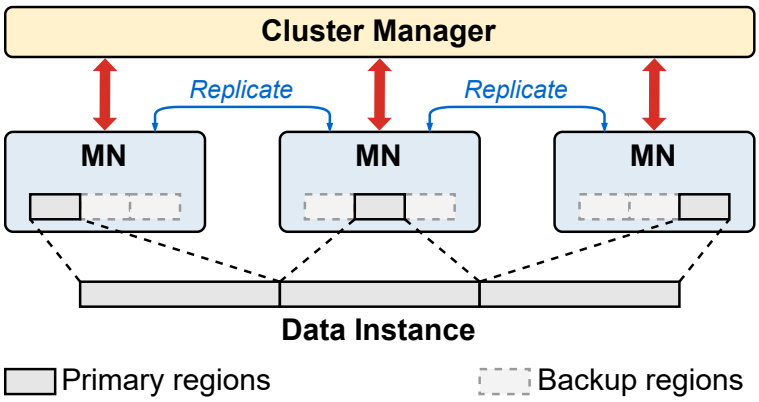


Figure 6.11: The overview of the UMO memory pool.

as a hash value and fetching multiple continuous slots with a single remote memory read.

The client-centric caching framework also provides an `update` and `priority` interface for developers to integrate various caching algorithms [179, 27]. The `update` interface enables developers to customize the recorded access information and define the update rules for access information. The `priority` interface allows developers to define a function to map the recorded metadata to the hotness of cached objects. Since the key differences between caching algorithms are the access information they rely on and the definition of object hotness, various caching algorithms can be integrated by customizing these two interfaces.

6.5 The UMO Memory Pool

Figure 6.11 shows the overall architecture of UMO, the memory pool of DMC. UMO shards and replicates data instances onto multiple MNs and provides interfaces for DI clients to allocate objects and execute transactions on the replicated objects. Similar to existing approaches [223, 56], UMO adopts RDMA-based optimistic concurrency control (OCC) and two-phase commit protocols to execute transactions. A DI client that initiates a

transaction serves as a coordinator. Each transaction has an execution phase and a commit phase. In the execution phase, a coordinator fetches objects in the read and write sets of the transaction with `RDMA_READ`s and updates the write set locally. In the commit phase, the coordinator locks objects in the write set on both primary and backup MNs with RDMA-based RPCs, validates the read set by fetching them again and checking their version numbers, and commits the transaction with RPCs if there is no conflict. Otherwise, the transaction is aborted and retried later. To ensure performance isolation among DMC instances, UMO rate limits DI clients with a token-based QoS mechanism for RDMA [130].

UMO adopts three enabling techniques to satisfy the following three critical requirements. First, to improve memory efficiency, we employ an on-demand allocation scheme to reduce the over-provisioned user memory. Second, to adapt to the bursty cloud workloads, we propose a copy-free memory region migration scheme to rapidly achieve load balance and cluster scaling. Finally, to achieve reliability at scale, we propose an on-demand connection management scheme to reduce the failure domain and the huge connection metadata introduced by the enormous number of connections between compute agents and the MNs.

6.5.1 On-Demand Allocation

UMO adopts a three-level memory management scheme to allocate and free objects. First, at the cluster level, the cluster manager partitions the memory of MNs into 1 GB memory regions. Memory is allocated to data instances one region at a time. Each region is replicated as primary and backup regions on multiple MNs for high availability. Then, at the data instance level, each region is partitioned into coarse-grained memory segments, *e.g.*, 16 MB. DI clients in compute agents get memory one segment

Table 6.2: The statistics of instance sizes in the production cluster.

Size (GB)	**(0, 1)**	**[1, 2)**	**[2, 4)**	**[4, 8)**	**[8, 64]**
Percentage	8.5%	28%	19.8%	17.3%	26.4%

at a time. Finally, inside each DI client, memory segments are further split into memory blocks of 60 size classes [85]. An object is always allocated from the size class that best fits it.

UMO achieves on-demand memory allocation in the granularity of memory regions. When creating a DMC instance, we do not assign all the memory it requires at once but let data instances allocate memory gradually from the memory pool. Although such a scheme leaves unused memory inside 1 GB memory regions, the over-provisioned memory can still be greatly reduced. Table 6.2 shows the statistics of instance sizes in the DCS cluster. Instances with sizes greater than or equal to 2 GB and 8 GB account for 63.5% and 26.4%, respectively. Given the fact that the medium utilization rate of allocated memory is 33% in Figure 6.4b, more than 1 GB of memory can be saved for over 50% of DCS instances. For instances with sizes less than 1 GB, we directly use DRAM in the compute pool to create these small instances.

6.5.2 Copy-Free Memory Region Migration

Efficiently migrating memory regions among MNs is essential to achieve load balance and dynamic scaling. The major challenge is that transactions are inevitably affected during data migrations, *i.e.*, either being blocked or suffering from poor performance [62]. Existing approaches in academia focus on enabling transactions to not be blocked during migrations [109, 62, 63, 99]. However, these approaches are infeasible in production due to their minute-scale migration duration and up to 50% throughput drops [109]. The severe performance degradation and the

long service impact duration make it impossible to satisfy the service level agreement (SLA) in production. To make region migrations practical, *we propose to minimize transaction impact duration instead of allowing transactions to execute with suboptimal performance.*

We propose a copy-free migration protocol with millisecond-scale region migration time and the same blocking time for write transactions. Our key idea is to leverage replicated memory regions to *migrate workloads instead of physically migrating data.* We only focus on migrating primary regions since backup regions can be moved asynchronously without affecting transaction executions [56, 208]. Migrating a primary region consists of two phases, *i.e.*, a preparation phase and an execution phase. The preparation phase decides the target MN to serve the migrated workload. When the cluster manager receives a request indicating a primary region needs to be migrated, it first checks its backup MNs, *i.e.*, MNs that replicate the requested region. If no replica MN can serve the migrated workload, the cluster manager creates a new backup region on another MN asynchronously. The execution phase begins when the target MN is ready for migration. The cluster manager blocks the source MN from serving new write transactions on the primary region and waits for all existing transactions on the region to finish on both source and target MNs. Then the cluster manager updates the mapping of primary and backup regions and informs DI clients about the migration. All transactions can execute normally after the mapping is updated.

UMO efficiently achieves load balance and dynamic scaling with the copy-free region migration scheme:

Load balancing. DI clients sample the remote memory access latency of each region in their transactions and report the samples to the cluster manager periodically. Load balance is triggered whenever the cluster manager finds the average access

latency of a region inside an MN exceeds a threshold. Then the cluster manager iteratively migrates primary regions from the overloaded MN until it is no longer a bottleneck.

Memory pool scaling. Scaling in and out MNs happens when the memory utilization of the memory pool is below or above a threshold. When scaling out MNs, the cluster manager needs to migrate primary regions to the newly added MNs to achieve better load balance. Regions are first replicated to the newly added MNs as backup regions, and then workloads are transferred by migrating some primary regions to them. When scaling in MNs, the cluster manager first replicates all the regions in the MNs to be removed to other MNs as backup regions asynchronously. MNs are then removed after migrating all the primary regions from it.

6.5.3 On-Demand Connection Management

Managing a production-level memory pool with hundreds of MNs introduces a large number of connections between compute and memory pools. Existing transactional memory pools in academia adopt a fully connected architecture, where a single compute agent needs to connect to all MNs [223, 56].

This leads to two issues under large-scale deployments. First, the fully connected architecture generates large connection metadata on compute agents and MNs. Suppose there are K compute agents and N MNs. Each compute agent has to maintain N connections to all MNs. Each MN has to maintain $K + N - 1$ connections to all compute agents and other MNs. This requires more than 4 GB and 80 MB of metadata on each MN and compute agent under a cluster with $10,000$ compute agents and 200 MNs, which is unacceptable considering the 4 GB average instance size. Second, the fully connected architecture creates a huge failure domain. A single MN failure can affect many ir-

relevant compute agents due to the connection error. Such a large failure domain makes it difficult to achieve high service reliability and satisfy SLAs in deployment.

We propose an on-demand connection management scheme to address these issues. Instead of connecting a compute agent to all MNs, we initially connect a compute agent to a small number of MNs. The number of MNs and connections increases in an on-demand manner when 1) the compute agent allocates more memory that cannot be satisfied, or 2) a region is migrated to achieve better load balance. Such an approach effectively reduces the size of connection metadata. Specifically, the maximum memory size of a DMC instance is 64 GB and each MN contains 256 GB DRAM. Each compute agent connects to at least 3 MNs if data instances are replicated three times in the memory pool, and connects to at most 64×3 MNs since data instances are partitioned into 1 GB memory regions scattered in the memory pool. Meanwhile, each MN only needs to maintain $N + 256$ RDMA connections since an MN can only be used by at most 256 compute agents. Under a cluster with $10,000$ DI clients and 200 MNs, the partially connected scheme reduces the memory required for connection metadata to at most 160 MB and 76 MB on MNs and DI clients. Besides, the failure domain is also greatly reduced since a single MN failure now only affects compute agents that store memory regions on it.

6.6 Evaluation

We evaluate DMC in terms of *performance*, *elasticity*, *fault-tolerance*, and *memory efficiency* to show its effectiveness.

Experiment setup. We evaluate DMC with both single-node and cluster instances. The single-node instance is configured with one CPU and 8 GB DRAM. The cluster instance consists of 4 single-node instances with the same setting. Both

single-node and cluster instances are the most widely adopted settings in the DCS cluster. The local cache of DMC is configured to be 600 MB according to our performance and cost analysis. We use an additional user VM with 32 CPU cores and 64 GB DRAM to send requests to caching service instances. The user VM executes 400 client threads to get the maximum throughput of both approaches.

Workload. We use both YCSB [50] and Twitter workloads[215] to evaluate DMC. For YCSB, we use four core workloads, *i.e.*, A (50% GET, 50% UPDATE), B (95% GET, 5% UPDATE), C (100% GET), and D (95% GET, 5% INSERT), with 16 million keys with 256-byte key-value size. Requests are generated with Zipfan distribution with $\theta = 0.99$ following the default setting. For Twitter, we use the first three read-intensive workloads from three types of clusters, *i.e.*, compute (Cluster 1), storage (Cluster 3), and transient (Cluster 16).

6.6.1 Performance

We evaluate the end-to-end performance of DMC compared with DCS in both single-node and cluster instances. Besides, we evaluate how compute-side cache and dynamic memory region migration affect the performance of DMC.

End-to-end performance. Figure 6.12 shows the end-to-end performance of DMC and DCS. For single-node instances, DCS has higher throughput and lower latency since DMC introduces higher remote memory access latency. Under read-intensive workloads, *i.e.*, YCSB B, C, D, and Twitter, the throughput loss is within 10% and the increase of latency is less than 0.3 ms, which is acceptable considering the performance and network fluctuations in production. Under YCSB-A write-intensive workloads, DMC exhibits lower throughput due to the highly conflict transactions in the memory pool. The throughput loss

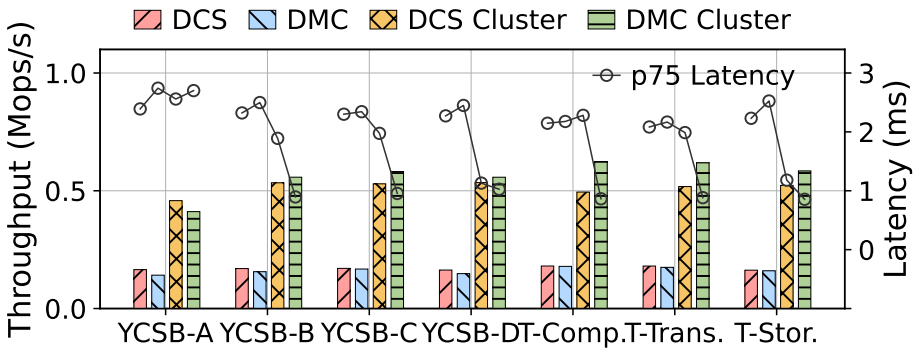


Figure 6.12: The throughput and 75th percentile latency of DCS and DMC under YCSB and Twitter workloads.

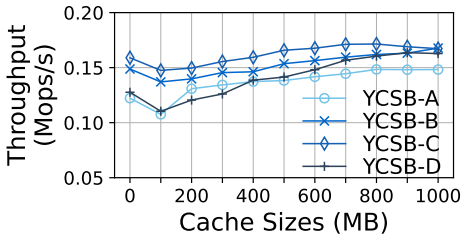


Figure 6.13: Throughput with different compute-side caches.

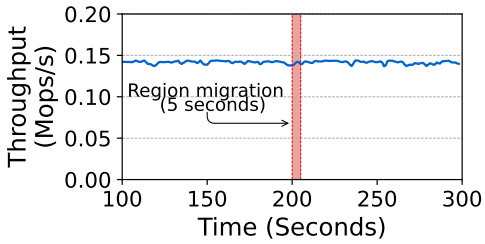


Figure 6.14: Throughput when migrating a memory region.

is still within 15%, which is acceptable in Huawei Cloud since the current workload is read-intensive.

For cluster instances, DMC achieves lower 75th percentile latency and up to 25% higher throughput. The higher throughput is attributed to the logical sharding scheme, which can evenly distribute read requests to all compute agents. In comparison, the throughput of DCS is bottlenecked by a single Redis server due to the highly skewed request patterns. Under YCSB-A, the throughput of DMC is still bounded by the write conflicts, resulting in a 10% throughput loss.

Compute-side cache. Figure 6.13 shows the throughput of a DMC single-node instance with different sizes of compute-side caches. The throughput of DMC increases as the size of the compute-side cache becomes larger. However, DMC instances

with small caches perform worse than those without cache. This is because the frequent cache evictions under small caches introduce too much overhead on the CPUs of compute agents, affecting the request processing speed.

Memory region migration. We sample the throughput of DMC every 500 ms when migrating a memory region under YCSB-B. As shown in Figure 6.14, region migration takes 5 seconds, and the write transactions are blocked for only 15 ms. The throughput of DMC is not affected, indicating the effectiveness of the copy-free region migration protocol.

6.6.2 Elasticity and Fault-Tolerance

Elasticity. Figures 6.15 and 6.16 show the throughput of DMC and DCS during resource scaling under YCSB-C with 1 million keys. We use a smaller dataset due to the poor migration performance of DCS when the dataset is large.

Horizontal scaling. We evaluate horizontal scaling with cluster instances of DCS and DMC. We create a 4-node cluster, scale out to 8 nodes, and scale in to 4 nodes. DCS takes more than 8 minutes to scale out and 5 minutes to scale in. The throughput is up to 9× lower than that of DMC during the process. In contrast, DMC scales out and in instantly without any side effects. The improvement is because DMC only needs to launch new compute agents and inform the cluster manager to adjust the memory assigned to the data instance, eliminating data migration from the critical path.

Vertical scaling. We evaluate the performance of vertical scaling with single-node DMC and DCS instances. We create a single-node instance with 1 CPU and 4 GB DRAM initially, scale it up to 8 GB, and then scale it down to 4 GB. DCS takes more than 15 minutes to scale up or down. This is because DCS creates a new container and iteratively moves all data to the

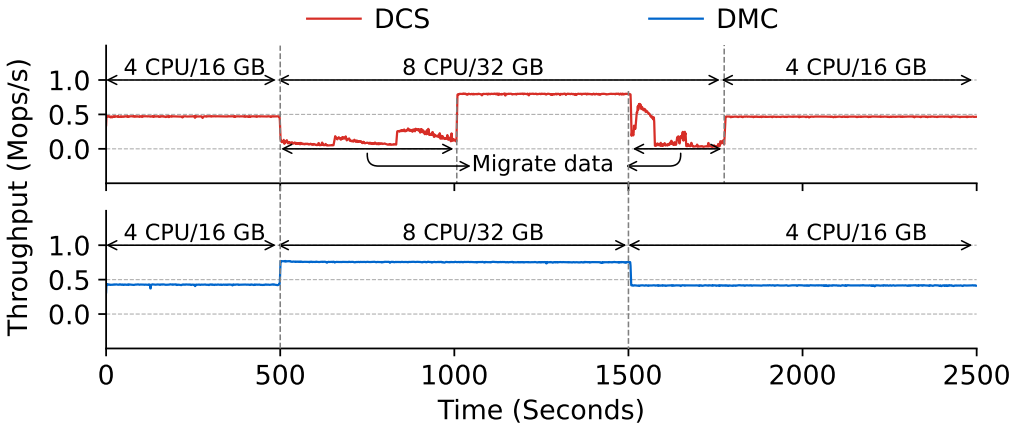


Figure 6.15: The throughput of DMC under horizontal scaling.

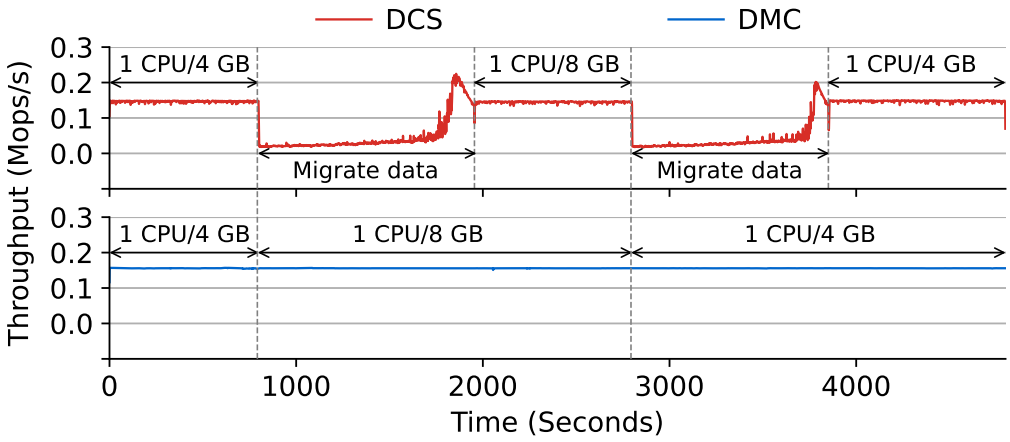
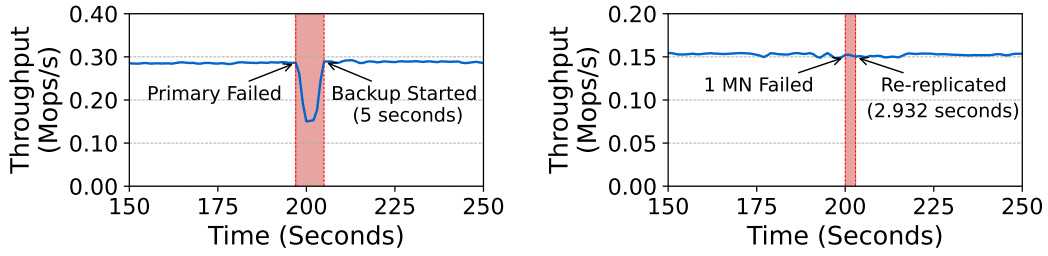


Figure 6.16: The throughput of DMC under vertical scaling.



(a) CN failure.

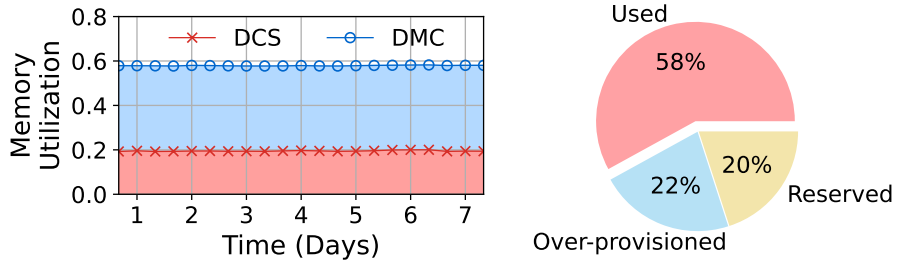
(b) MN failure.

Figure 6.17: The throughput of DMC under MN and CN failures.

.

newly created instance. The throughput drops by up to 90% during the process. On the contrary, the throughput of DMC is not affected since it conducts vertical scaling by asking the cluster manager to allocate and reclaim memory asynchronously. No operation is involved in the data plan.

Fault-tolerance. We evaluate the performance of DMC on MN and CN failures under YCSB-C. For MN failures, we start a single-node DMC instance and manually crash an MN. On MN failures, the memory pool promotes backup regions to primary regions and re-replicates these regions asynchronously. As shown in Figure 6.17b, the performance of DMC is unaffected during the MN failure due to the efficient region migration and re-replication. For CN failures, we launch a replicated DMC instance with one primary agent and one backup agent. We manually crash the primary region during execution and show the throughput of DMC. When dealing with CN failures, the compute pool switches requests to the backup agent and restarts a new backup agent to maintain the replication factor. As shown in Figure 6.17a, the performance of DMC drops by half when the failure happens since both primary and backup agents can serve read requests before failure. The throughput resumes to the previous level after a short period since it takes the cluster manager 5 seconds to notice the failure and re-launch a new



(a) The memory utilization of 7 days. (b) Utilization breakdown.

Figure 6.18: The improvement on memory utilization of DMC.

backup agent.

6.6.3 Memory Efficiency

To show how DMC can improve the overall memory utilization, we collect real-world instance creation and memory allocation traces from a DCS production cluster and feed the trace to the DMC cluster. We sample the memory utilization of all nodes in the DMC cluster three times a day at 1:00, 9:00, and 17:00 and compare with the corresponding points in the trace collected from the DCS cluster.

Figure 6.18a shows the memory utilization of the two clusters in 7 days. DMC reaches up to 58% memory utilization, which is 2.6 times higher than that of the DCS cluster. Figure 6.18b shows the memory utilization breakdown. DMC reduces the over-provisioned memory from 40% to 22% due to the on-demand memory allocation. The remaining memory over-provisioning is caused by the internal fragmentation inside memory regions. Moreover, DMC unifies memory allocation for caching service instances with memory regions, which eliminates the stranded and unsold memory caused by the variable-sized resource scheduling in the DCS cluster.

6.7 Lessons Learned and Future Directions

Disaggregating memory is necessary. Memory utilization can also be improved by achieving on-demand allocation in the monolithic DCS cluster. However, the feasibility of such a design relies on the elasticity of caching service instances. Specifically, to achieve high memory utilization, multiple instances with partially allocated memory have to be compacted on a single node. Instances have to be migrated when the memory in their original node is not sufficient to hold all the instances. Unfortunately, monolithic-server-based caching systems cannot be efficiently migrated since their management of data and execution of user requests are coupled together at the software layer. They suffer from minute-scale data movement on the critical path of instance migrations [109, 99]. Consequently, memory disaggregation is necessary to achieve both elasticity and resource efficiency for DCSes.

The overhead of memory disaggregation is affordable. The performance loss of using DM is a major concern in both academia and industry [218, 196, 4]. DMC has a less than 0.3 ms increase in the request latency and a 10% throughput loss compared with the monolithic DCS. The throughput and latency are sufficient to achieve our SLAs and the performance loss is affordable considering the 2.6 times improvement in the overall memory utilization. Meanwhile, the increase in latency is negligible considering the multiple hops and network fluctuations between user VMs and the DMC cluster.

System design is critical to fully exploit DM. DM has three major benefits thanks to the physically decoupled CPU and memory, *i.e.*, elasticity, resource utilization, and failure isolation [199]. A good software design is a must to exploit these hardware benefits [38]. In the initial design of DMC, we tried multiple combinations of compute-side cache, replication, and

data sharding. DMC with decoupled replication, logical sharding, and data cache outperforms other alternatives in terms of failure isolation, elasticity, and performance.

Memory reservation is still required. Using DM can reduce the over-provisioned memory and eliminate stranded memory in the monolithic DCS cluster. However, memory reservations are still required due to the deployment model of DMC. Specifically, the memory pool of DMC is constructed with VMs allocated from the computing infrastructure of Huawei Cloud. When the capacity of the memory pool is insufficient to create a DMC instance, DMC needs to allocate a new VM and add it to the memory pool. While we have improved the speed of adding VMs to the memory pool, the speed of memory expansion is still limited by the time-consuming VM allocation.

To address this issue, DMC expands the capacity of the memory pool whenever the memory utilization exceeds a fixed threshold. DMC sets heuristic rules to raise the thresholds in advance of events that may cause many bursty instance creation requests, *e.g.*, Black Fridays [72]. Two possible future directions can be investigated to reduce the amount of reserved memory. First, instead of setting a fixed threshold, we can use machine learning to predict the required memory and set the threshold dynamically. Besides, the resource expansion mechanism of the memory pool and even the lower-level infrastructure can also be improved so that resources can be expanded rapidly to satisfy user requests.

Shared everything is desired. In DMC, we adopt a logical partition scheme to achieve better load balance with shared reads. However, there is still room for improvement since write requests are still partitioned and can suffer from imbalanced workloads. Ideally, we could achieve the best load balance and cost efficiency with a shared everything architecture, *i.e.*, compute agents equally serve all requests. However, such an architecture makes it challenging to manage the local DRAM cache of

compute agents. Expensive cache coherence protocols [32, 202] need to be adopted since multiple compute agents can simultaneously access and modify the same object in their local caches. How to efficiently manage compute agents' local DRAM cache under the shared everything architecture is also an important future research topic.

6.8 Related Work

In-memory caching systems. Existing works on in-memory caching systems can be classified into two categories. The first improves cache hit rates of caching algorithms according to the characteristics of cloud workloads [216, 169, 195, 64, 25, 27, 2]. Among them, SegCache [216] proposes a TTL-based eviction algorithm to efficiently evict objects according to TTLs. Hyperbolic [27], and LHD [25] propose new metrics to measure the hotness of cached objects. Cacheus [169] and LeCaR [195] model cache replacement as a multi-armed-bandit problem and use reinforcement learning to adapt to the changing cloud workloads. DMC is orthogonal to these works since we adopt existing caching algorithms and focus on system-level improvements.

The second class improves the performance of caching systems when executing caching algorithms and data access requests. Specifically, MICA [126] employs the request direction techniques on modern NICs to schedule user requests to specific CPU cores of the caching server. MemC3 [66] uses concurrent cuckoo hashing and the CLOCK caching algorithm to improve the throughput of caching servers. CliqueMap [182] uses one-sided RDMA operations to reduce the load on server CPUs when executing read operations and achieve higher overall throughput. However, all these approaches focus on improving the performance of monolithic-server-based caching systems. This work identifies the problem of low memory utilization of production

caching services and improves the cluster-level memory utilization with DMC, a memory-disaggregated caching service.

Memory-disaggregated storage systems. Works that relate to DMC most are Ditto [179] and Dinomo [119]. The client-centric caching framework of DMC is similar to that of Ditto. However, Ditto focuses on executing caching algorithms efficiently and achieving high cache hit rates with an adaptive caching scheme. It is insufficient to be deployed in production since it does not address the challenges of failure handling and scalability issues of the memory pool. DMC addresses these issues with decoupled replication and on-demand connection management. Dinomo is a memory-disaggregated caching system that partitions the management of data to individual CNs to achieve better scalability. The logical data sharding of DMC resembles this scheme but achieves better load balance by allowing all compute agents to equally serve read requests.

6.9 Summary

This chapter presents DMC, Huawei Cloud's industrial practice of using disaggregated memory to improve the memory efficiency of its distributed caching service. We present the severe memory under-utilization issue in Huawei Cloud with real-world statistics and bridge the gap between academia and industry by introducing the industrial requirements and discussing detailed design choices of DMC. Finally, DMC shows 2.6× higher memory utilization, 9× high throughput during resource scaling, and 1.25× higher throughput for the cluster mode instances. The performance loss introduced by DM for single-node instances is less than 10% for most cases.

□ End of chapter.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

Resource disaggregation is a promising next-generation data center architecture. It can achieve near-optimal resource efficiency, hardware elasticity, and scalability. Severe performance degradation is the key problem that prohibits the wide adoption of resource disaggregation in real-world data centers.

This thesis identifies that the root cause for the performance degradation lies in the unsuitable data structures and algorithms and proposes to attack the performance issue in a bottom-up manner, *i.e.*, by designing native data structures and algorithms.

First, by analyzing the characteristics of a disaggregated data center, we summarize three critical aspects to construct high-performance native data structures and algorithms, *i.e.*, I/O, concurrency, and asymmetry.

We then focus on disaggregated memory and design data structures and algorithms for memory-disaggregated storage systems. Our work designs data structures and algorithms for three major components of a memory-disaggregated storage system, *i.e.*, memory management, indexing, and fault tolerance.

To achieve efficient memory management, we propose a two-level memory allocator and a client-centric caching framework. Both approaches achieve high performance by reducing the am-

plified I/O numbers, avoiding and efficiently resolving concurrency conflicts, and scheduling computation according to the compute capabilities of compute and memory pools of DM. We integrate these two approaches in Ditto, the first memory-disaggregated caching system, and verified the effectiveness of our design with thorough experiments.

To index data with high performance, we propose SMART, a high-performance range index data structure on DM. SMART innovatively adopts radix trees as range indexes on DM to attack the severe I/O size amplification of traditional B+-tree-based approaches. We further reduce I/O numbers and improve concurrency control performance in the data structure design. Thorough experiments over YCSB workloads show the prevalence of SMART compared with state-of-the-art approaches.

To achieve reliability with high performance, we design high-performance replication and logging algorithms. Both algorithms reduce the amplified number of I/O operations by shotcutting data-path operations and achieve high concurrency control by simplifying conflict resolution. Moreover, both approaches optimize the asymmetric compute capabilities by relying only on one-sided RDMA operations. We integrate the replication and logging algorithms in FUSEE, the first fully memory-disaggregated storage system, and demonstrate its performance and reliability with thorough experiments.

Finally, I discuss the industrial practice by introducing the design of DMC, the memory-disaggregated caching service in Huawei Cloud. We present the severe memory under-utilization issue in Huawei Cloud with real-world statistics and bridge the gap between academia and industry by introducing the industrial requirements and discussing detailed design choices of DMC. Thorough experiments validate the effectiveness of our designed data structures and algorithms.

In brief, this thesis contributes to both academia and in-

dustry. To academia, we provide guidelines for efficient data structures and algorithm design over DM. To industry, we show the huge benefit of memory-disaggregating a monolithic caching service can have in terms of resource efficiency and elasticity. To some extent, our work promotes the deployment of memory disaggregation in modern cloud data centers.

7.2 Future Work

The problem with resource disaggregation is far from being fully addressed. The goal of resource disaggregation is to allow ***arbitrary*** programs to execute seamlessly over the disaggregated architecture with *high performance*. In this perspective, it is critical to achieve the compatibility of existing programs and simplify the development of future programs. For existing programs, it is crucial to transparently port them to the disaggregated architecture with high performance. For future programs, new programming paradigms need to be proposed to allow engineers to develop efficient disaggregation-native software. However, there is still a huge gap between an arbitrary program and the disaggregated architecture.

Closing this gap calls for a joint effort from both bottom-up and top-down system design. First, many disaggregation-native system components, *e.g.*, data structures, and algorithms, have to be constructed to achieve high performance. This is what we have explored in this thesis. Besides, an internal abstraction layer, *e.g.*, language runtimes and operating systems, has to be deployed to compose bottom-up system components and provide a compatible interface.

Looking forward, I would like to explore combining bottom-up system components with top-down interfaces to achieve better disaggregation for both existing and future programs.

7.2.1 Disaggregating Existing Programs

In existing cloud data centers, there are two major types of programs, *i.e.*, traditional CPU-centric programs and emerging GPU-centric AI applications. The primal challenge of disaggregating these programs in a portable manner is how to mitigate the severe performance degradation. Addressing this challenge requires us to understand the character of program execution and schedule the computation of individual programs to their most suitable lower-level system components. It is promising to approach this at the runtime level, where we can expose upper-level program semantics to the lower-level system components. Concretely, I will explore designing disaggregated language runtimes for CPU-centric programs and machine learning (ML) runtimes for GPU-centric AI applications.

Fine-Grained Disaggregation for CPU-Centric Applications

The large amount of CPU-centric applications in today's data centers makes disaggregating these types of applications a critical task. For these applications, achieving high performance over the disaggregated architecture requires us to decompose a program into fine-grained computation tasks and schedule them to the most suitable hardware accelerators. This makes understanding the semantics of programs critical.

We propose to achieve such fine-grained disaggregation at the runtime level. The benefit of using language runtimes is that they are very close to programs since they directly deal with code. However, it is challenging to correctly divide programs, understand the characters of divided computation tasks, and coordinate these tasks over the disaggregated accelerators. In the future, I will design a disaggregated language runtime that leverages both dynamic profiling and static analysis to decompose the program and construct lower-level system components

to coordinate computation tasks efficiently over the disaggregated architecture.

Disaggregated ML Runtimes for GPU-Centric AI applications

Due to the emergence of large language models (LLM), GPU-centric generative AI workloads, *i.e.*, LLM training and inferencing, are becoming more and more important in modern data centers. Disaggregating these GPU-centric applications is urgent due to the large-scale deployment and the diverse resource requirements of training and inferencing tasks. These applications involve massive parallel floating-point computation and are already scheduled on the most suitable hardware, *i.e.*, GPUs. The problem with the existing AI applications is the memory wall of GPUs, *i.e.*, the compute power of GPUs cannot be fully utilized since they cannot hold all the data in their memory. Inspired by the disaggregation of CPU memory on monolithic servers, I will explore disaggregating GPU memory with high-speed GPU interconnects at the ML runtime level to break the GPU memory wall for generative AI applications.

7.2.2 Disaggregating Future Programs

Eliminating the semantic gap between an arbitrary program and the disaggregated hardware is hard to achieve optimal with the runtime-level program analyses. For future programs, a better approach is to expose some critical lower-level hardware and architectural details to engineers so that they can close the gap when developing programs. This calls for a new programming paradigm for disaggregation-native programs.

Disaggregated Serverless Frameworks for Future Programs

Serverless computing is an emerging programming paradigm that naturally synergizes with the idea of resource disaggrega-

tion. It requires developers to manually decompose a monolithic program into small computation tasks named serverless functions. However, existing serverless frameworks are designed for CPU-centric programs. There is a huge gap between serverless functions and the disaggregated hardware, both in the programming interface and in the execution of serverless functions. To close this gap, in the future, I propose to extend the serverless computing interface to expose more hardware details and design bottom-up system components for more efficient execution of serverless functions.

□ **End of chapter.**

Chapter 8

List of Publications

1. **Jiacheng Shen**, Pengfei Zuo, Xuchuan Luo, Yuxin Su, Jiazhen Gu, Hao Feng, Yangfan Zhou, and Michael R. Lyu. "Ditto: An Elastic and Adaptive Memory-Disaggregated Caching System." In Proceedings of the 29th Symposium on Operating Systems Principles (SOSP 23).

2. **Jiacheng Shen**, Pengfei Zuo, Xuchuan Luo, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R. Lyu. "FUSEE: A Fully Memory-Disaggregated Key-Value Store." In 21st USENIX Conference on File and Storage Technologies (FAST 23).

3. **Jiacheng Shen**, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R. Lyu. "Defuse: A Dependency-Guided Function Scheduler to Mitigate Cold Starts on FaaS Platforms." In 2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS 21).

4. **Jiacheng Shen**, Xu Zhang, Zihao Xiang, Sagiv Goren, Dongxu Li, Ben Che, Zhangyu Chen, Paul Chen, Yonghui Miao, Jia Feng, Pengfei Zuo, and Michael R. Lyu. "Productionizing a Memory-Disaggregated Caching System" In submission. 2024.

5. Xuchuan Luo, Pengfei Zuo, **Jiacheng Shen**, Jiazhen Gu,

Xin Wang, Michael R. Lyu, and Yangfan Zhou. "SMART: A High-Performance Adaptive Radix Tree for Disaggregated Memory." In 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23).

6. Xuchuan Luo, Pengfei Zuo, **Jiacheng Shen**, Jiazhen Gu, Xin Wang, Michael Lyu, and Yangfan Zhou. "A Memory-Disaggregated Radix Tree." ACM Transactions on Storage (2024).

7. Xuchuan Luo, **Jiacheng Shen**, Pengfei Zuo, Xin Wang, Michael R. Lyu, and Yangfan Zhou. "CHIME: A Cache-Efficient and High-Performance Hybrid Index on Disaggregated Memory" In submission. 2023.

8. Tianyi Yang, **Jiacheng Shen**, Yuxin Su, Xiao Ling, Yongqiang Yang, and Michael R. Lyu. "AID: efficient prediction of aggregated intensity of dependency in large-scale cloud systems." In 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE 21).

9. Tianyi Yang, **Jiacheng Shen**, Yuxin Su, Xiaoxue Ren, Yongqiang Yang, and Michael R. Lyu. "Characterizing and mitigating anti-patterns of alerts in industrial cloud systems." In 2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 22).

10. Shuyao Jiang, **Jiacheng Shen**, Shengnan Wu, Yu Cai, Yue Yu, and Yangfan Zhou. "Towards Usable Neural Comment Generation via Code-Comment Linkage Interpretation: Method and Empirical Study." IEEE Transactions on Software Engineering 49, no. 4 (2022): 2239-2254.

11. Tianyi Yang, Cheryl Lee, **Jiacheng Shen**, Yuxin Su, Cong Feng, Yongqiang Yang, and Michael R. Lyu. "MicroRes:

Versatile Resilience Profiling in Microservices via Degradation Dissemination Indexing." In Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 24).

□ **End of chapter.**

Bibliography

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-Flow Integrity Principles, Implementations, and Applications. *ACM Trans. Inf. Syst. Secur.*, 13(1):4:1–4:40, 2009.

[2] M. Abdi, A. Mosayyebzadeh, M. H. Hajkazemi, A. Turk, O. Krieger, and P. Desnoyers. Caching in the Multiverse. In *the 11th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2019, July 8-9, 2019*. USENIX Association, 2019.

[3] S. E. Abdullahi and G. A. Ringwood. Garbage collecting the internet: A survey of distributed garbage collection. *ACM Comput. Surv.*, 30(3):330–373, 1998.

[4] M. K. Aguilera, E. Amaro, N. Amit, E. Hunhoff, A. Yelam, and G. Zellweger. Memory Disaggregation: Why Now and What are the Challenges. *ACM SIGOPS Operating Systems Review*, 57(1):38–46, 2023.

[5] M. K. Aguilera, K. Keeton, S. Novakovic, and S. Singhal. Designing Far Memory Data Structures: Think Outside the Box. In *Workshop on Hot Topics in Operating Systems, HotOS 2019, May 13-15, 2019*, pages 120–126. ACM, 2019.

[6] P. Ajoux, N. Bronson, S. Kumar, W. Lloyd, and K. Veeraraghavan. Challenges to adopting stronger consistency

at scale. In *15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause Ittingen, Switzerland, May 18-20, 2015*. USENIX Association, 2015.

[7] P. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering, San Francisco, California, USA, October 13-15, 1976*, pages 562–570. IEEE Computer Society, 1976.

[8] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. K. Aguilera, A. Panda, S. Ratnasamy, and S. Shenker. Can Far Memory Improve Job Throughput? In *the 15th EuroSys Conference, EuroSys 2020, April 27-30, 2020*, pages 14:1–14:16. ACM, 2020.

[9] Amazon. Amazon elastic block store. `https://aws.amazon.com/ebs`, 2024.

[10] Amazon. Aws nitro system. `https://aws.amazon.com/ec2/nitro`, 2024.

[11] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Disk-locality in datacenter computing considered irrelevant. In *13th Workshop on Hot Topics in Operating Systems, HotOS XIII, Napa, California, USA, May 9-11, 2011*. USENIX Association, 2011.

[12] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: a fast array of wimpy nodes. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 1–14. ACM, 2009.

[13] P. Anderson, E. B. Aranas, Y. Assaf, R. Behrendt, R. Black, M. Caballero, P. Cameron, B. Canakci, T. D.

Carvalho, A. Chatzieleftheriou, R. S. Clarke, J. Clegg, D. Cletheroe, B. Cooper, T. Deegan, A. Donnelly, R. Drevinskas, A. L. Gaunt, C. Gkantsidis, A. G. Diaz, I. Haller, F. Hong, T. Ilieva, S. Joshi, R. Joyce, M. Kunkel, D. Lara, S. Legtchenko, F. L. Liu, B. Magalhães, A. Marzoev, M. McNett, J. Mohan, M. Myrah, T. Nguyen, S. Nowozin, A. Ogus, H. Overweg, A. I. T. Rowstron, M. Sah, M. Sakakura, P. Scholtz, N. Schreiner, O. Sella, A. Smith, I. A. Stefanovici, D. Sweeney, B. Thomsen, G. Verkes, P. Wainman, J. Westcott, L. Weston, C. Whittaker, P. W. Berenguer, H. Williams, T. Winkler, and S. Winzeck. Project silica: Towards sustainable cloud archival storage in glass. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pages 166–181. ACM, 2023.

[14] S. Angel, M. Nanavati, and S. Sen. Disaggregation and the application. In *12th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2020, July 13-14, 2020*. USENIX Association, 2020.

[15] Apache. Apache hadoop. `https://hadoop.apache.org`, 2024.

[16] Apache. Apache spark: Unified engine for large-scale data analytics. `https://spark.apache.org`, 2024.

[17] I. Ari, A. Amer, R. B. Gramacy, E. L. Miller, S. A. Brandt, and D. D. E. Long. ACME: Adaptive Caching Using Multiple Experts. In *Distributed Data & Structures 4, Records of the 4th International Meeting, WDAS 2002, March 20-23, 2002*, Proceedings in Informatics. Carleton Scientific, 2002.

[18] M. F. Arlitt, L. Cherkasova, J. Dilley, R. Friedrich, and T. Jin. Evaluating Content Management Techniques for Web Proxy Caches. *SIGMETRICS Performance Evaluation Review*, 27(4):3–11, 2000.

[19] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, et al. Above the Clouds: A Berkeley View of Cloud Computing. Technical report, Technical Report UCB/EECS, 28, EECS Department, University of California, Berkeley, 2009.

[20] B. Arun, S. Peluso, R. Palmieri, G. Losa, and B. Ravindran. Speeding up consensus by chasing fast decisions. In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017, Denver, CO, USA, June 26-29, 2017*, pages 49–60. IEEE Computer Society, 2017.

[21] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '12, London, United Kingdom, June 11-15, 2012*, pages 53–64. ACM, 2012.

[22] N. Atre, J. Sherry, W. Wang, and D. S. Berger. Caching with delayed hits. In *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020*, pages 495–513. ACM, 2020.

[23] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. In *Proceedings of*

the Ninth Annual ACM Symposium on Principles of Distributed Computing, Quebec City, Quebec, Canada, August 22-24, 1990, pages 363–375. ACM, 1990.

[24] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. D. Davis. CORFU: A shared log design for flash clusters. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 1–14. USENIX Association, 2012.

[25] N. Beckmann, H. Chen, and A. Cidon. LHD: Improving Cache Hit Rate by Maximizing Hit Density. In *the 15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, April 9-11, 2018*, pages 389–403. USENIX Association, 2018.

[26] R. Binna, E. Zangerle, M. Pichl, G. Specht, and V. Leis. HOT: A height optimized trie index for main-memory database systems. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 521–534. ACM, 2018.

[27] A. Blankstein, S. Sen, and M. J. Freedman. Hyperbolic Caching: Flexible Caching for Web Applications. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, July 12-14, 2017*, pages 499–511. USENIX Association, 2017.

[28] J. Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer 1994 Technical Conference, Boston, Massachusetts, USA, June 6-10, 1994, Conference Proceeding*, pages 87–98. USENIX Association, 1994.

[29] J. Bornholt, A. Kaufmann, J. Li, A. Krishnamurthy, E. Torlak, and X. Wang. Specifying and checking file system crash-consistency models. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2016, Atlanta, GA, USA, April 2-6, 2016*, pages 83–98. ACM, 2016.

[30] M. Burke, A. Cheng, and W. Lloyd. Gryff: Unifying consensus and shared registers. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 591–617. USENIX Association, 2020.

[31] M. Burke, S. Dharanipragada, S. Joyner, A. Szekeres, J. Nelson, I. Zhang, and D. R. K. Ports. PRISM: Rethinking the RDMA interface for distributed systems. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 228–242. ACM, 2021.

[32] Q. Cai, W. Guo, H. Zhang, D. Agrawal, G. Chen, B. C. Ooi, K. Tan, Y. M. Teo, and S. Wang. Efficient Distributed Memory Management with RDMA and Caching. *Proceedings of the VLDB Endowment*, 11(11):1604–1617, 2018.

[33] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli. Rethinking Software Runtimes for Disaggregated Memory. In *the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2021, April 19-23, 2021*, pages 79–92. ACM, 2021.

[34] P. Cao and S. Irani. Cost-Aware WWW Proxy Caching Algorithms. In *the 1st USENIX Symposium on Inter-*

net Technologies and Systems, USITS 97, December 8-11, 1997. USENIX, 1997.

[35] W. Cao, Y. Zhang, X. Yang, F. Li, S. Wang, Q. Hu, X. Cheng, Z. Chen, Z. Liu, J. Fang, B. Wang, Y. Wang, H. Sun, Z. Yang, Z. Cheng, S. Chen, J. Wu, W. Hu, J. Zhao, Y. Gao, S. Cai, Y. Zhang, and J. Tong. Polardb serverless: A cloud native database for disaggregated data centers. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 2477–2489. ACM, 2021.

[36] Z. Cao, S. Dong, S. Vemuri, and D. H. C. Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*, pages 209–223. USENIX Association, 2020.

[37] cgroups. cgroups. `https://man7.org/linux/man-pages/man7/cgroups.7.html`, 2022.

[38] A. Chatzieleftheriou, I. A. Stefanovici, D. Narayanan, B. Thomsen, and A. I. T. Rowstron. Could Cloud Storage be Disrupted in the Next Decade? In *the 12th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2020, July 13-14, 2020*. USENIX Association, 2020.

[39] Y. Chen, Y. Lu, and J. Shu. Scalable RDMA RPC on reliable connection with efficient resource sharing. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, pages 19:1–19:14. ACM, 2019.

[40] Y. L. Chen, S. Mu, J. Li, C. Huang, J. Li, A. Ogus, and D. Phillips. Giza: Erasure coding objects across global data centers. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, pages 539–551. USENIX Association, 2017.

[41] Z. Chen, Y. Liu, Y. Wang, and Y. Lu. A gpu-accelerated in-memory metadata management scheme for large-scale parallel file systems. *J. Comput. Sci. Technol.*, 36(1):44–55, 2021.

[42] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti. Dynacache: Dynamic Cloud Caching. In *the 7th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 15, July 6-7, 2015*. USENIX Association, 2015.

[43] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti. Cliffhanger: Scaling Performance Cliffs in Web Memory Caches. In *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, March 16-18, 2016*, pages 379–392. USENIX Association, 2016.

[44] A. Cidon, D. Rushton, S. M. Rumble, and R. Stutsman. Memshare: A Dynamic Multi-Tenant Key-value Cache. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, July 12-14, 2017*, pages 321–334. USENIX Association, 2017.

[45] A. Cloud. Alibaba cluster trace. `https://github.com/alibaba/clusterdata`, 2018.

[46] A. Cloud. Essds. `https://www.alibabacloud.com/help/en/ecs/user-guide/essds`, 2024.

[47] G. Cloud. Google cluster trace. `https://github.com/google/cluster-data`, 2018.

[48] H. Cloud. Object storage service. `https://www.huaweicloud.com/intl/en-us/product/obs.html`, 2024.

[49] Cloud4U. Gpu render fram in the cloud. `https://www.cloud4u.com/cloud-hosting/gpu-render-farm/`, 2024.

[50] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *the 1st ACM Symposium on Cloud Computing, SoCC 2010, June 10-11, 2010*, pages 143–154. ACM, 2010.

[51] M. Copik, M. Chrapek, L. Schmid, A. Calotoiu, and T. Hoefler. Software resource disaggregation for HPC with serverless computing. *CoRR*, abs/2401.10852, 2024.

[52] I. Corporation. Driving Exascale Computing and HPC with Intel. `https://www.intel.com/content/www/us/en/high-performance-computing-fabrics/omni-path-driving-exascale-computing.html`.

[53] I. Corporation. Write Combining Memory Implementation Guidelines. `https://download.intel.com/design/PentiumII/applnots/24442201.pdf`.

[54] S. Dong, A. Kryczka, Y. Jin, and M. Stumm. Evolution of development priorities in key-value stores serving large-scale applications: The rocksdb experience. In *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, pages 33–49. USENIX Association, 2021.

[55] A. Dragojevic, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast Remote Memory. In *the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, April 2-4, 2014*, pages 401–414. USENIX Association, 2014.

[56] A. Dragojevic, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *the 25th Symposium on Operating Systems Principles, SOSP 2015, October 4-7, 2015*, pages 54–70. ACM, 2015.

[57] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra. The Design and Operation of CloudLab. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, July 10-12, 2019*, pages 1–14. USENIX Association, 2019.

[58] C. Dwork, N. A. Lynch, and L. J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

[59] G. Einziger, R. Friedman, and B. Manes. TinyLFU: A Highly Efficient Cache Admission Policy. *ACM Transactions on Storage, TOS 2017*, 13(4):35:1–35:31, 2017.

[60] A. ElastiCache. `https://aws.amazon.com/elasticache/?nc1=h_ls`.

[61] C. S. Ellis. Concurrency in linear hashing. *ACM Trans. Database Syst.*, 12(2):195–217, 1987.

[62] A. J. Elmore, V. Arora, R. Taft, A. Pavlo, D. Agrawal, and A. E. Abbadi. Squall: Fine-grained live reconfiguration for partitioned main memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 299–313. ACM, 2015.

[63] A. J. Elmore, S. Das, D. Agrawal, and A. E. Abbadi. Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms. In *the 2011 ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, June 12-16, 2011*, pages 301–312. ACM, 2011.

[64] O. Eytan, D. Harnik, E. Ofer, R. Friedman, and R. I. Kat. It's Time to Revisit LRU vs. FIFO. In *the 12th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2020, July 13-14, 2020*. USENIX Association, 2020.

[65] Facebook. Wedge 100: More open and versatile then ever. `https://engineering.fb.com/2016/10/18/data-center-engineering/wedge-100-more-open-and-versatile-than-ever`, 2016.

[66] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, April 2-5, 2013*, pages 371–384. USENIX Association, 2013.

[67] P. Faraboschi, K. Keeton, T. Marsland, and D. S. Milojicic. Beyond processor-centric operating systems. In G. Candea, editor, *15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause Ittingen, Switzerland, May 18-20, 2015*. USENIX Association, 2015.

[68] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. M. Caulfield, E. S. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava,

A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. G. Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*, pages 51–66. USENIX Association, 2018.

[69] A. Flaxman, A. T. Kalai, and H. B. McMahan. Online Convex Optimization in the Bandit Setting: Gradient Descent without a Gradient. In *the 16th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005, January 23-25, 2005*, pages 385–394. SIAM, 2005.

[70] Fonxat. 300 million email database. `https://archive.org/details/300MillionEmailDatabase`, 2018.

[71] D. J. Foster, A. Rakhlin, and K. Sridharan. Adaptive Online Learning. *CoRR*, abs/1508.05170, 2015.

[72] B. Friday. `https://www.amazon.com/blackfriday`, 2023.

[73] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network requirements for resource disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 249–264. USENIX Association, 2016.

[74] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh Symposium on Operating System Principles, SOSP 1979, Asilomar Conference Grounds, Pacific Grove, California, USA, 10-12, December 1979*, pages 150–162. ACM, 1979.

[75] Google. Memorystore. `https://cloud.google.com/memorystore`, 2023.

[76] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient Memory Disaggregation with Infiniswap. In *the 14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, March 27-29, 2017*, pages 649–667. USENIX Association, 2017.

[77] J. T. Gu, X. Sun, W. Zhang, Y. Jiang, C. Wang, M. Vaziri, O. Legunsen, and T. Xu. Acto: Automatic end-to-end testing for operation correctness of cloud system management. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pages 96–112. ACM, 2023.

[78] R. Guerraoui, A. Murat, J. Picorel, A. Xygkis, H. Yan, and P. Zuo. uKharon: A membership service for microsecond applications. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 101–120, Carlsbad, CA, 2022. USENIX Association.

[79] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn. RDMA over commodity ethernet at scale. In *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, pages 202–215. ACM, 2016.

[80] Z. Guo, Z. He, and Y. Zhang. Mira: A program-behavior-guided far memory system. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pages 692–708. ACM, 2023.

[81] Z. Guo, Y. Shan, X. Luo, Y. Huang, and Y. Zhang. Clio: A Hardware-Software Co-designed Disaggregated Memory System. In *the 27th ACM International Conference on Architectural Support for Programming Languages and Op-*

erating Systems, ASPLOS 2022, Feb. 28 - Mar. 4, 2022, pages 417–433. ACM, 2022.

[82] M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[83] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[84] M. Hsu and W. Yang. Concurrent operations in extendible hashing. In W. W. Chu, G. Gardarin, S. Ohsuga, and Y. Kambayashi, editors, *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*, pages 241–247. Morgan Kaufmann, 1986.

[85] A. H. Hunter, C. Kennelly, P. Turner, D. Gove, T. Moseley, and P. Ranganathan. Beyond malloc efficiency to fleet efficiency: A hugepage-aware memory allocator. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, Virtual, July 14-16, 2021*. USENIX Association, 2021.

[86] InfiniBand. `https://www.infinibandta.org/`.

[87] Intel. Discover advanced memory with intel optane pmem. `https://www.intel.com/content/www/us/en/products/details/memory-storage/optane-dc-persistent-memory.html`, 2024.

[88] Intel. Match every application to its optimal architecture with xpu. `https://www.intel.com/content/www/us/en/architecture-and-technology/xpu.html`, 2024.

[89] J. Jang, H. Choi, H. Bae, S. Lee, M. Kwon, and M. Jung. CXL-ANNS: software-hardware collaborative memory dis-

aggregation and computation for billion-scale approximate nearest neighbor search. In *2023 USENIX Annual Technical Conference, USENIX ATC 2023, Boston, MA, USA, July 10-12, 2023*, pages 585–600. USENIX Association, 2023.

[90] S. Jha, J. Behrens, T. Gkountouvas, M. Milano, W. Song, E. Tremel, R. van Renesse, S. Zink, and K. P. Birman. Derecho: Fast state machine replication for cloud services. *ACM Trans. Comput. Syst.*, 36(2):4:1–4:49, 2019.

[91] M. Ji, A. C. Veitch, and J. Wilkes. Seneca: Remote mirroring done write. In *Proceedings of the General Track: 2003 USENIX Annual Technical Conference, June 9-14, 2003, San Antonio, Texas, USA*, pages 253–268. USENIX, 2003.

[92] S. Jiang and X. Zhang. LIRS: An Efficient Low Inter-Reference Recency Set Replacement Policy to Improve Buffer Cache Performance. In *the 2002 ACM SIGMETRICS International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS 2002, June 15-19, 2002*, pages 31–42. ACM, 2002.

[93] T. Johnson and D. E. Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *the 20th International Conference on Very Large Data Bases, VLDB 1994, September 12-15, 1994*, pages 439–450. Morgan Kaufmann, 1994.

[94] N. P. Jouppi, D. H. Yoon, M. Ashcraft, M. Gottscho, T. B. Jablin, G. Kurian, J. Laudon, S. Li, P. C. Ma, X. Ma, T. Norrie, N. Patil, S. Prasad, C. Young, Z. Zhou, and D. A. Patterson. Ten lessons from three generations shaped google's tpuv4i : Industrial product. In *48th*

ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Virtual Event / Valencia, Spain, June 14-18, 2021, pages 1–14. IEEE, 2021.

[95] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA Efficiently for Key-Value Services. In *ACM SIGCOMM 2014 Conference, SIGCOMM 2014, August 17-22, 2014*, pages 295–306. ACM, 2014.

[96] A. Kalia, M. Kaminsky, and D. G. Andersen. Design Guidelines for High Performance RDMA Systems. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, June 22-24, 2016*, pages 437–450. USENIX Association, 2016.

[97] A. Kalia, M. Kaminsky, and D. G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *the 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, November 2-4, 2016*, pages 185–201. USENIX Association, 2016.

[98] A. Kalia, M. Kaminsky, and D. G. Andersen. Datacenter RPCs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 1–16. USENIX Association, 2019.

[99] J. Kang, L. Cai, F. Li, X. Zhou, W. Cao, S. Cai, and D. Shao. Remus: Efficient live migration for distributed databases with snapshot isolation. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22. ACM, 2022.

[100] D. R. Karger, E. Lehman, F. T. Leighton, R. Panigrahy, M. S. Levine, and D. Lewin. Consistent hashing and ran-

dom trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing, El Paso, Texas, USA, May 4-6, 1997*, pages 654–663. ACM, 1997.

[101] A. Katsarakis, V. Gavrielatos, M. R. S. Katebzadeh, A. Joshi, A. Dragojevic, B. Grot, and V. Nagarajan. Hermes: A fast, fault-tolerant and linearizable replication protocol. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 201–217. ACM, 2020.

[102] A. Khrabrov, M. Pirvu, V. Sundaresan, and E. de Lara. Jitserver: Disaggregated caching JIT compiler for the JVM in the cloud. In *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 869–884. USENIX Association, 2022.

[103] J. Kim, W. Choe, and J. Ahn. Exploring the design space of page management for multi-tiered memory systems. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, pages 715–728. USENIX Association, 2021.

[104] W. Kim, J. Kim, W. Baek, B. Nam, and Y. Won. NVWAL: exploiting NVRAM in write-ahead logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2016, Atlanta, GA, USA, April 2-6, 2016*, pages 385–398. ACM, 2016.

[105] W. Kim, M. K. Ramanathan, X. Fu, S. Kashyap, and C. Min. PACTree: A high performance persistent range index using PAC guidelines. In *SOSP '21: ACM SIGOPS*

28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021, pages 424–439. ACM, 2021.

[106] A. Klimovic, C. Kozyrakis, E. Thereska, B. John, and S. Kumar. Flash storage disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, pages 29:1–29:15. ACM, 2016.

[107] D. E. Knuth. *The art of computer programming, Volume I: Fundamental Algorithms, 3rd Edition*. Addison-Wesley, 1997.

[108] R. Koller and R. Rangaswami. I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance. In *the 8th USENIX Conference on File and Storage Technologies, FAST 2010, February 23-26, 2010*, pages 211–224. USENIX, 2010.

[109] C. Kulkarni, A. Kesavan, T. Zhang, R. Ricci, and R. Stutsman. Rocksteady: Fast Migration for Low-Latency In-Memory Storage. In *the 26th Symposium on Operating Systems Principles, SOSP 2017, October 28-31, 2017*, pages 390–405. ACM, 2017.

[110] V. Kumar. Concurrent operations on extendible hashing and its performance. *Commun. ACM*, 33(6):681–694, 1990.

[111] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pages 611–626. ACM, 2023.

[112] M. Labib. Amazon ElastiCache Deep Dive. `https://pages.awscloud.com/rs/112-TZM-766/images/Session\%201\%20-\%20ElastiCache-DeepDive_v2_rev.pdf`.

[113] H. Labs. The Machine: A New Kind of Computer. `https://www.hpl.hp.com/research/systems-research/themachine/`, 2014.

[114] L. Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994.

[115] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[116] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable DRAM alternative. In *36th International Symposium on Computer Architecture (ISCA 2009), June 20-24, 2009, Austin, TX, USA*, pages 2–13. ACM, 2009.

[117] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *ASPLOS-VII Proceedings - Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, Massachusetts, USA, October 1-5, 1996*, pages 84–92. ACM Press, 1996.

[118] S. Lee, Y. Yu, Y. Tang, A. Khandelwal, L. Zhong, and A. Bhattacharjee. MIND: In-Network Memory Management for Disaggregated Data Centers. In *the 28th Symposium on Operating Systems Principles, SOSP 2021, October 26-29, 2021*, pages 488–504. ACM, 2021.

[119] S. K. Lee, S. Ponnapalli, S. Singhal, M. K. Aguilera, K. Keeton, and V. Chidambaram. DINOMO: An Elastic, Scalable, High-Performance Key-Value Store for Dis-

aggregated Persistent Memory. *Proceedings of the VLDB Endowment*, 15(13):4023–4037, 2022.

[120] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 38–49. IEEE Computer Society, 2013.

[121] V. Leis, F. Scheibner, A. Kemper, and T. Neumann. The ART of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware, DaMoN 2016, San Francisco, CA, USA, June 27, 2016*, pages 3:1–3:8. ACM, 2016.

[122] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang. Kv-direct: High-performance in-memory key-value store with programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 137–152. ACM, 2017.

[123] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2023, March 25-29, 2023*, pages 574–587. ACM, 2023.

[124] H. Li, K. Liu, T. Liang, Z. Li, T. Lu, H. Yuan, Y. Xia, Y. Bao, M. Chen, and Y. Shan. Hopp: Hardware-software co-designed page prefetching for disaggregated memory. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2023, Mon-*

treal, QC, Canada, February 25 - March 1, 2023, pages 1168–1181. IEEE, 2023.

[125] P. Li, Y. Hua, P. Zuo, Z. Chen, and J. Sheng. ROLEX: A Scalable RDMA-Oriented Learned Key-Value Store for Disaggregated Memory Systems. In *the 21st USENIX Conference on File and Storage Technologies, FAST 2023, February 21-23, 2023*, pages 99–114. USENIX Association, 2023.

[126] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, April 2-4, 2014*, pages 429–444. USENIX Association, 2014.

[127] K. T. Lim, J. Chang, T. N. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *the 36th International Symposium on Computer Architecture, ISCA 2009, June 20-24, 2009*, pages 267–278. ACM, 2009.

[128] K. T. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch. System-level implications of disaggregated memory. In *18th IEEE International Symposium on High Performance Computer Architecture, HPCA 2012, New Orleans, LA, USA, 25-29 February, 2012*, pages 189–200. IEEE Computer Society, 2012.

[129] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta. Offloading distributed applications onto SmartNICs using iPipe. In *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM 2019, Beijing, China, August 19-23, 2019*, pages 318–333. ACM, 2019.

[130] Q. Liu and P. J. Varman. Haechi: A token-based qos mechanism for one-sided i/os in RDMA based storage system. In *41st IEEE International Conference on Distributed Computing Systems, ICDCS 2021, Washington DC, USA, July 7-10, 2021*, pages 171–182. IEEE, 2021.

[131] X. Luo, P. Zuo, J. Shen, J. Gu, X. Wang, M. R. Lyu, and Y. Zhou. SMART: A High-Performance Adaptive Radix Tree for Disaggregated Memory. In *the 17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, July 10-12, 2023*, pages 553–571. USENIX Association, 2023.

[132] W. Lv, Y. Lu, Y. Zhang, P. Duan, and J. Shu. InfiniFS: An Efficient Metadata Service for Large-Scale Distributed Filesystems. In *the 20th USENIX Conference on File and Storage Technologies, FAST 2022, February 22-24, 2022*, pages 313–328. USENIX Association, 2022.

[133] N. A. Lynch and A. A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Digest of Papers: FTCS-27, The Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing, Seattle, Washington, USA, June 24-27, 1997*, pages 272–281. IEEE Computer Society, 1997.

[134] S. Ma, K. Chen, S. Chen, M. Liu, J. Zhu, H. Kang, and Y. Wu. ROART: Range-query optimized persistent ART. In *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, pages 1–16. USENIX Association, 2021.

[135] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building efficient replicated state machine for wans. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008,*

San Diego, California, USA, Proceedings, pages 369–384. USENIX Association, 2008.

[136] Y. Mao, E. Kohler, and R. T. 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. ACM, 2012.

[137] A. Mathew and C. Min. HydraList: A scalable in-memory index using asynchronous updates and partial replication. *Proc. VLDB Endow.*, 13(9):1332–1345, 2020.

[138] N. Megiddo and D. S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *the 2nd USENIX Conference on File and Storage Technologies, FAST 2003, March 31 - April 2, 2003*. USENIX, 2003.

[139] Memcached. `http://memcached.org`, 2022.

[140] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 2002, Winnipeg, Manitoba, Canada, August 11-13, 2002*, pages 73–82. ACM, 2002.

[141] J. W. Mickens, E. B. Nightingale, J. Elson, D. Gehring, B. Fan, A. Kadav, V. Chidambaram, O. Khan, and K. Nareddy. Blizzard: Fast, cloud-scale block storage for cloud-oblivious applications. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, pages 257–273. USENIX Association, 2014.

[142] J. Min, M. Liu, T. Chugh, C. Zhao, A. Wei, I. H. Doh, and A. Krishnamurthy. Gimbal: enabling multi-tenant storage

disaggregation on smartnic jbofs. In *ACM SIGCOMM 2021 Conference, Virtual Event, USA, August 23-27, 2021*, pages 106–122. ACM, 2021.

[143] C. Mitchell, Y. Geng, and J. Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *2013 USENIX Annual Technical Conference, USENIX ATC 2013, June 26-28, 2013*, pages 103–114. USENIX Association, 2013.

[144] C. Mitchell, K. Montgomery, L. Nelson, S. Sen, and J. Li. Balancing CPU and network in the cell distributed B-tree store. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*, pages 451–464. USENIX Association, 2016.

[145] C. Mohan, D. Haderle, B. G. Lindsay, H. Pirahesh, and P. M. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[146] Y. Mond and Y. Raz. Concurrency control in b+-trees databases using preparatory operations. In *VLDB'85, Proceedings of 11th International Conference on Very Large Data Bases, August 21-23, 1985, Stockholm, Sweden*, pages 331–334. Morgan Kaufmann, 1985.

[147] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 358–372. ACM, 2013.

[148] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab,

D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, April 2-5, 2013*, pages 385–398. USENIX Association, 2013.

[149] V. Nitu, B. Teabe, A. Tchana, C. Isci, and D. Hagimont. Welcome to Zombieland: Practical and Energy-Efficient Memory Disaggregation in a Datacenter. In *the 13th EuroSys Conference, EuroSys 2018, April 23-26, 2018*, pages 16:1–16:12. ACM, 2018.

[150] Nvidia. Nvidia bluefield networking platform. `https://www.nvidia.com/en-us/networking/products/data-processing-unit/`.

[151] Nvidia. Connectx nics. `https://www.nvidia.com/en-us/networking/ethernet-adapters/`, 2024.

[152] B. M. Oki and B. Liskov. Viewstamped replication: A general primary copy. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing, Toronto, Ontario, Canada, August 15-17, 1988*, pages 8–17. ACM, 1988.

[153] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *the 1993 ACM SIGMOD International Conference on Management of Data, SIGMOD 1993, May 26-28, 1993*, pages 297–306. ACM Press, 1993.

[154] D. Ongaro and J. K. Ousterhout. In search of an understandable consensus algorithm. In G. Gibson and N. Zeldovich, editors, *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, pages 305–319. USENIX Association, 2014.

[155] P4. P4. `https://opennetworking.org/p4`, 2024.

[156] S. Pan, T. Stavrinos, Y. Zhang, A. Sikaria, P. Zakharov, A. Sharma, S. S. P., M. Shuey, R. Wareing, M. Gangapuram, G. Cao, C. Preseau, P. Singh, K. Patiejunas, J. R. Tipton, E. Katz-Bassett, and W. Lloyd. Facebook's Tectonic Filesystem: Efficiency from Exascale. In *the 19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, pages 217–231. USENIX Association, 2021.

[157] N. Pemberton and J. Schleier-Smith. The serverless data center: Hardware disaggregation meets serverless computing. In *The First Workshop on Resource Disaggregation*, volume 4, 2019.

[158] T. S. Pillai, R. Alagappan, L. Lu, V. Chidambaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Application crash consistency and performance with CCFS. *ACM Trans. Storage*, 13(3):19:1–19:29, 2017.

[159] S. Podlipnig and L. Böszörményi. A Survey of Web Cache Replacement Strategies. *ACM Computing Surveys*, 35(4):374–398, 2003.

[160] Q. Pu, H. Li, M. Zaharia, A. Ghodsi, and I. Stoica. FairRide: Near-Optimal, Fair Cache Sharing. In *the 13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, March 16-18, 2016*, pages 393–406. USENIX Association, 2016.

[161] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Kim, S. Lanka, J. R. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A

reconfigurable fabric for accelerating large-scale datacenter services. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, pages 13–24. IEEE Computer Society, 2014.

[162] Y. Qiao, Z. Ruan, H. Ma, A. Belay, M. Kim, and H. Xu. Harvesting idle memory for application-managed soft state with midas. In *21st USENIX Symposium on Networked Systems Design and Implementation, NSDI 2024, Santa Clara, CA, April 15-17, 2024*. USENIX Association, 2024.

[163] Y. Qiao, C. Wang, Z. Ruan, A. Belay, Q. Lu, Y. Zhang, M. Kim, and G. H. Xu. Hermit: Low-latency, high-throughput, and transparent remote memory via feedback-directed asynchrony. In *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023*, pages 181–198. USENIX Association, 2023.

[164] X. Qin, W. Zhang, W. Wang, J. Wei, X. Zhao, and T. Huang. Optimizing Data Migration for Cloud-Based Key-Value Stores. In *the 21st ACM International Conference on Information and Knowledge Management, CIKM 2012, October 29 - November 02, 2012*, pages 2204–2208. ACM, 2012.

[165] M. Rajashekhar and Y. Yue. Twemcache. `https://blog.twitter.com/engineering/en_us/a/2012/caching-with-twemcache`, 2012.

[166] Redis. `http://redis.io`, 2022.

[167] K. Ren, Q. Zheng, S. Patil, and G. A. Gibson. Indexfs: Scaling file system metadata performance with stateless caching and bulk insertion. In *International Conference for*

High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014, pages 237–248. IEEE Computer Society, 2014.

[168] D. Robertson. The history of data centers: An exponential evolution. `https://blog.enconnex.com/data-center-history-and-evolution`, 2024.

[169] L. V. Rodriguez, F. B. Yusuf, S. Lyons, E. Paz, R. Rangaswami, J. Liu, M. Zhao, and G. Narasimhan. Learning Cache Replacement with CACHEUS. In *the 19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, pages 341–354. USENIX Association, 2021.

[170] B. M. Rogers, A. Krishna, G. B. Bell, K. V. Vu, X. Jiang, and Y. Solihin. Scaling the bandwidth wall: challenges in and avenues for CMP scaling. In *36th International Symposium on Computer Architecture (ISCA 2009), June 20-24, 2009, Austin, TX, USA*, pages 371–382. ACM, 2009.

[171] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay. AIFM: High-Performance, Application-Integrated Far Memory. In *the 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, November 4-6, 2020*, pages 315–332. USENIX Association, 2020.

[172] T. Saemundsson, H. Björnsson, G. V. Chockler, and Y. Vigfusson. Dynamic Performance Profiling of Cloud Caches. In *the 5th ACM Symposium on Cloud Computing, SoCC 2014, November 3-5, 2014*, pages 28:1–28:14. ACM, 2014.

[173] B. Schroeder and G. A. Gibson. Disk failures in the real world: What does an MTTF of 1, 000, 000 hours mean

to you? In *5th USENIX Conference on File and Storage Technologies, FAST 2007, February 13-16, 2007, San Jose, CA, USA*, pages 1–16. USENIX, 2007.

[174] H. N. Schuh, W. Liang, M. Liu, J. Nelson, and A. Krishnamurthy. Xenic: SmartNIC-accelerated distributed transactions. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 740–755. ACM, 2021.

[175] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 205–218. USENIX Association, 2020.

[176] A. Shamis, M. Renzelmann, S. Novakovic, G. Chatzopoulos, A. Dragojevic, D. Narayanan, and M. Castro. Fast general distributed transactions with opacity. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 433–448. ACM, 2019.

[177] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *the 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, October 8-10, 2018*, pages 69–87. USENIX Association, 2018.

[178] Y. Shan, W. Lin, R. Kosta, A. Krishnamurthy, and Y. Zhang. Disaggregating and Consolidating Network Functionalities with SuperNIC. *CoRR*, 2021.

[179] J. Shen, P. Zuo, X. Luo, Y. Su, J. Gu, H. Feng, Y. Zhou, and M. R. Lyu. Ditto: An Elastic and Adaptive Memory-Disaggregated Caching System, 2023.

[180] J. Shen, P. Zuo, X. Luo, T. Yang, Y. Su, Y. Zhou, and M. R. Lyu. FUSEE: A Fully Memory-Disaggregated Key-Value Store. In *the 21st USENIX Conference on File and Storage Technologies, FAST 2023, February 21-23, 2023*, pages 81–98. USENIX Association, 2023.

[181] D. Sidler, Z. Wang, M. Chiosa, A. Kulkarni, and G. Alonso. StRoM: Smart remote memory. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 29:1–29:16. ACM, 2020.

[182] A. Singhvi, A. Akella, M. Anderson, R. Cauble, H. Deshmukh, D. Gibson, M. M. K. Martin, A. Strominger, T. F. Wenisch, and A. Vahdat. CliqueMap: Productionizing an RMA-Based Distributed Caching System. In *ACM SIGCOMM 2021 Conference, SIGCOMM 2021, August 23-27, 2021*, pages 93–105. ACM, 2021.

[183] SNIA. The performance impact of nvme and nvme over fabrics. `https://www.snia.org/sites/default/files/NVMe_Webcast_Slides_Final.1.pdf`.

[184] Z. Song, D. S. Berger, K. Li, and W. Lloyd. Learning Relaxed Belady for Content Distribution Network Caching. In *the 17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, February 25-27, 2020*, pages 529–544. USENIX Association, 2020.

[185] C. Specification. `https://www.computeexpresslink.org/`.

[186] M. Su, M. Zhang, K. Chen, Z. Guo, and Y. Wu. RFP: When RPC is faster than server-bypass with RDMA. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, pages 1–15. ACM, 2017.

[187] X. Sun, W. Luo, J. T. Gu, A. Ganesan, R. Alagappan, M. Gasch, L. Suresh, and T. Xu. Automatic reliability testing for cluster management controllers. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 143–159. USENIX Association, 2022.

[188] Y. Taleb, R. Stutsman, G. Antoniu, and T. Cortes. Tailwind: Fast and atomic rdma-based replication. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 851–863. USENIX Association, 2018.

[189] G.-Z. Technology. `https://genzconsortium.org/`.

[190] J. Terrace and M. J. Freedman. Object storage on CRAQ: high-throughput chain replication for read-mostly workloads. In *2009 USENIX Annual Technical Conference, San Diego, CA, USA, June 14-19, 2009*. USENIX Association, 2009.

[191] S. Thomas, G. M. Voelker, and G. Porter. Cachecloud: Towards speed-of-light datacenter communication. In *10th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2018, Boston, MA, USA, July 9, 2018*. USENIX Association, 2018.

[192] S. Tsai, Y. Shan, and Y. Zhang. Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 33–48. USENIX Association, 2020.

[193] A. Vakali. LRU-Based Algorithms for Web Cache Replacement. In *the 1st International Conference of Electronic Commerce and Web Technologies, EC-Web 2000, September 4-6, 2000*, volume 1875 of *Lecture Notes in Computer Science*, pages 409–418. Springer, 2000.

[194] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In E. A. Brewer and P. Chen, editors, *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 91–104. USENIX Association, 2004.

[195] G. Vietri, L. V. Rodriguez, W. A. Martinez, S. Lyons, J. Liu, R. Rangaswami, M. Zhao, and G. Narasimhan. Driving Cache Replacement with ML-Based LeCaR. In *the 10th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2018, July 9-10, 2018*. USENIX Association, 2018.

[196] L. Vilanova, L. Maudlej, S. Bergman, T. Miemietz, M. Hille, N. Asmussen, M. Roitzsch, H. Härtig, and M. Silberstein. Slashing the Disaggregation Tax in Heterogeneous Data Centers with FractOS. In *the 17th European Conference on Computer Systems, EuroSys 2022, April 5-8, 2022*, pages 352–367. ACM, 2022.

[197] M. Vuppalapati, J. Miron, R. Agarwal, D. Truong, A. Motivala, and T. Cruanes. Building an elastic query engine

on disaggregated storage. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 449–462. USENIX Association, 2020.

[198] C. A. Waldspurger, N. Park, A. T. Garthwaite, and I. Ahmad. Efficient MRC Construction with SHARDS. In *the 13th USENIX Conference on File and Storage Technologies, FAST 2015, February 16-19, 2015*, pages 95–110. USENIX Association, 2015.

[199] C. Wang, H. Ma, S. Liu, Y. Li, Z. Ruan, K. Nguyen, M. D. Bond, R. Netravali, M. Kim, and G. H. Xu. Semeru: A Memory-Disaggregated Managed Runtime. In *the 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, November 4-6, 2020*, pages 261–280. USENIX Association, 2020.

[200] C. Wang, H. Ma, S. Liu, Y. Qiao, J. Eyolfson, C. Navasca, S. Lu, and G. H. Xu. Memliner: Lining up tracing and application for a far-memory-friendly runtime. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 35–53. USENIX Association, 2022.

[201] Q. Wang, Y. Lu, and J. Shu. Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory. In *the 2022 ACM SIGMOD/PODS International Conference on Management of Data, SIGMOD 2022, June 12-17, 2022*, pages 1033–1048. ACM, 2022.

[202] Q. Wang, Y. Lu, E. Xu, J. Li, Y. Chen, and J. Shu. Concordia: Distributed Shared Memory with In-Network Cache Coherence. In *the 19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, pages 277–292. USENIX Association, 2021.

[203] Z. Wang, Z. Jia, S. Zheng, Z. Zhang, X. Fu, T. S. E. Ng, and Y. Wang. GEMINI: fast failure recovery in distributed training with in-memory checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pages 364–381. ACM, 2023.

[204] Z. Wang, A. Pavlo, H. Lim, V. Leis, H. Zhang, M. Kaminsky, and D. G. Andersen. Building a Bw-tree takes more than just buzz words. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 473–488. ACM, 2018.

[205] A. Warfield, R. Ross, K. Fraser, C. Limpach, and S. Hand. Parallax: Managing storage for a million machines. In *Proceedings of HotOS'05: 10th Workshop on Hot Topics in Operating Systems, June 12-15, 2005, Santa Fe, New Mexico, USA*. USENIX Association, 2005.

[206] X. Wei, R. Chen, and H. Chen. Fast RDMA-Based Ordered Key-Value Store using Remote Learned Cache. In *the 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, November 4-6, 2020*, pages 117–135. USENIX Association, 2020.

[207] X. Wei, Z. Dong, R. Chen, and H. Chen. Deconstructing RDMA-Enabled Distributed Transactions: Hybrid is Better! In *the 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, October 8-10, 2018*, pages 233–251. USENIX Association, 2018.

[208] X. Wei, S. Shen, R. Chen, and H. Chen. Replication-driven live reconfiguration for fast distributed transaction processing. In *2017 USENIX Annual Technical Conference*

(USENIX ATC 17), Santa Clara, CA, 2017. USENIX Association.

[209] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 87–104. ACM, 2015.

[210] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *the 7th Symposium on Operating Systems Design and Implementation, OSDI 2006, November 6-8, 2006*, pages 307–320. USENIX Association, 2006.

[211] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, pages 307–320. USENIX Association, 2006.

[212] Q. Weng, W. Xiao, Y. Yu, W. Wang, C. Wang, J. He, Y. Li, L. Zhang, W. Lin, and Y. Ding. MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters. In *the 19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, April 4-6, 2022*, pages 945–960. USENIX Association, 2022.

[213] M. Whittaker, A. Ailijiang, A. Charapko, M. Demirbas, N. Giridharan, J. M. Hellerstein, H. Howard, I. Stoica, and A. Szekeres. Scaling replicated state machines with compartmentalization. *Proc. VLDB Endow.*, 14(11):2203–2215, 2021.

[214] K. Wu, Z. Guo, G. Hu, K. Tu, R. Alagappan, R. Sen, K. Park, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. The storage hierarchy is not a hierarchy: Optimizing caching on modern storage devices with orthus. In *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, pages 307–323. USENIX Association, 2021.

[215] J. Yang, Y. Yue, and K. V. Rashmi. A Large Scale Analysis of Hundreds of In-Memory Cache Clusters at Twitter. In *the 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, November 4-6, 2020*, pages 191–208. USENIX Association, 2020.

[216] J. Yang, Y. Yue, and R. Vinayak. Segcache: A Memory-Efficient and Scalable In-Memory Key-Value Cache for Small Objects. In *the 18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021*, pages 503–518. USENIX Association, 2021.

[217] T. Yang, C. Lee, J. Shen, Y. Su, Y. Yang, and M. R. Lyu. Microres: Versatile resilience profiling in microservices via degradation dissemination indexing. In *Proceedings of ACM SIGSOFT 33rd International Symposium on Software Testing and Analysis*, 2024.

[218] W. Yoon, J. Ok, J. Oh, S. Moon, and Y. Kwon. DiLOS: Do Not Trade Compatibility for Performance in Memory Disaggregation. In *the 18th European Conference on Computer Systems, EuroSys 2023, May 8-12, 2023*, pages 266–282. ACM, 2023.

[219] G. Yu, J. S. Jeong, G. Kim, S. Kim, and B. Chun. Orca: A distributed serving system for transformer-based generative models. In *16th USENIX Symposium on Operating*

Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022, pages 521–538. USENIX Association, 2022.

[220] F. B. Yusuf, V. Stebliankin, G. Vietri, and G. Narasimhan. Cache Replacement as a MAB with Delayed Feedback and Decaying Costs. *arXiv preprint*, 2020.

[221] H. Zhang and Q. Liang. Red-black tree used for arranging virtual memory area of linux. In *2010 International Conference on Management and Service Science*, pages 1–3. IEEE, 2010.

[222] J. Zhang, R. Izmailov, D. Reininger, and M. Ott. Web Caching Framework: Analytical Models and Beyond. In *Proceedings 1999 IEEE Workshop on Internet Applications*, pages 132–141. IEEE, 1999.

[223] M. Zhang, Y. Hua, P. Zuo, and L. Liu. FORD: Fast One-Sided RDMA-Based Distributed Transactions for Disaggregated Persistent Memory. In *the 20th USENIX Conference on File and Storage Technologies, FAST 2022, February 22-24*, pages 51–68. USENIX Association, 2022.

[224] Q. Zhang, X. Chen, S. Sankhe, Z. Zheng, K. Zhong, S. Angel, A. Chen, V. Liu, and B. T. Loo. Optimizing data-intensive systems in disaggregated data centers with TELEPORT. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12-17, 2022*, pages 1345–1359. ACM, 2022.

[225] Y. Zhang. Make it real: An end-to-end implementation of A physically disaggregated data center. *ACM SIGOPS Oper. Syst. Rev.*, 57(1):1–9, 2023.

[226] Y. Zhang, J. Yang, A. S. Memaripour, and S. Swanson. Mojim: A reliable and highly-available non-volatile mem-

ory system. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2015, Istanbul, Turkey, March 14-18, 2015*, pages 3–18. ACM, 2015.

[227] Y. Zhou, H. M. G. Wassel, S. Liu, J. Gao, J. Mickens, M. Yu, C. Kennelly, P. Turner, D. E. Culler, H. M. Levy, and A. Vahdat. Carbink: Fault-tolerant far memory. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 55–71. USENIX Association, 2022.

[228] T. Ziegler, S. T. Vani, C. Binnig, R. Fonseca, and T. Kraska. Designing distributed tree-based index structures for fast rdma-capable networks. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 741–758. ACM, 2019.

[229] T. Ziegler, S. T. Vani, C. Binnig, R. Fonseca, and T. Kraska. Designing Distributed Tree-Based Index Structures for Fast RDMA-Capable Networks. In *2019 ACM SIGMOD/PODS International Conference on Management of Data, SIGMOD 2019, June 30 - July 5, 2019*, pages 741–758. ACM, 2019.

[230] P. Zuo, J. Sun, L. Yang, S. Zhang, and Y. Hua. One-Sided RDMA-Conscious Extendible Hashing for Disaggregated Memory. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, pages 15–29. USENIX Association, 2021.