

# **Scale and Performance in Memory-Centric Storage**

By

Chenhao Ye

A dissertation submitted in partial fulfillment of  
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2026

Date of final oral examination: May 29th, 2026

The dissertation is approved by the following members of the Final Oral Committee:

Andrea C. Arpaci-Dusseau, Professor, Computer Sciences

Remzi H. Arpaci-Dusseau, Professor, Computer Sciences

Ming Liu, Assistant Professor, Computer Sciences

Tsung-Wei Huang, Associate Professor, Electrical & Computer Engineering



*Know You Are Loved.*

# Acknowledgments

My Ph.D. has been a long journey. It began during the COVID-19 pandemic, when the world seemed to slow down, and now concludes amid the rise of agentic AI, when everything seems to move so fast. This journey would have been impossible without the many incredible people who taught, inspired, or supported me. To all of them, I want to express my deepest gratitude.

First and foremost, I would like to thank my advisors, Andrea Arpaci-Dusseau and Remzi Arpaci-Dusseau. From Andrea and Remzi, I learned both how to be a good researcher and how to be a better person. As great systems researchers, they taught me to dig one level deeper into a problem, extract insights from measurements, and formulate interesting research questions. Whenever I felt stuck in research, a conversation with them always helped.

Andrea and Remzi also taught me a lot about communication. Andrea is one of the finest academic writers I have ever known (a “world-class talent,” to quote Remzi). She always gave constructive feedback on my papers, from their logical structure and flow to the wording of individual arguments. Even when receiving a poorly written first draft, she would begin her feedback email with “It is a good start,” followed by the good things she saw and the improvements to make. Remzi is a gifted storyteller. He has a distinctive charm when he shares knowledge, whether explaining a technical idea or offering a lesson about life. From him, I learned to guide audiences through a presentation in the most approachable way, with humor woven in naturally.

Beyond research, I am most grateful for Andrea and Remzi’s unconditional kindness, respect, and patience. When I started my Ph.D., I worked very hard, partly because I felt insecure and was constantly seeking external validation. Working with Andrea and Remzi made me realize that I do not need to impress anyone, not

even my advisors. Their support is unconditional, and I have always felt seen and appreciated. Over time, I learned to find peace and confidence within myself, which is one of the most valuable things I gained during my Ph.D. Their kindness inspires me to carry that same kindness forward.

For their valuable insights and feedback, I thank the other two members of my committee, Ming Liu and Tsung-Wei Huang. Ming always raises sharp questions, many of which prompt me to think about how to improve and extend my work. It was especially kind of Tsung-Wei to devote time to my committee while he was on parental leave.

Several other faculty members supported me in different ways. Mike Swift is the faculty member from whom I took the most courses at Madison, including security, distributed systems, and persistent memory. I enjoyed these courses both for their technical depth and for the many random fun facts along the way. In research, Mike has an impressive ability to look at a problem from unexpected angles and bring fresh perspectives to a discussion. Shivaram Venkataraman opened the door to the world of machine-learning systems for me through two of his courses. He also generously offered his time and help when I was revising the ROS paper. Tej Chajed helped satisfy my curiosity about systems verification and introduced me to a new way of thinking about systems. Xiangyao Yu taught me a great deal about databases and concurrency control. Rahul Chatterjee, who started the Madison Tech Clinic, showed me how we can apply technical knowledge to help the local community.

My internships were among the most memorable parts of my Ph.D. I want to thank Vasileios Zois, Badrish Chandramouli, and Phil Bernstein for a wonderful summer at Microsoft Research. The few mornings I spent writing proofs on the whiteboard right next to the coffee machines remain some of my happiest memories from that summer. I am grateful to Wencong Xiao for the opportunity to intern at ByteDance Seed. When I started the internship, I did not have a strong background in LLMs. Wencong showed tremendous patience and flexibility, giving me the space to learn and grow. The internship was fruitful: it opened my eyes to the many systems problems surrounding LLMs, and we produced work I am proud of. More importantly, Wencong served as a role model for me. Through his own career, he showed me how to bridge research innovation and real-world industrial impact, which deeply influenced how I began to envision my own path after the Ph.D.

I also want to thank Yuhong Zhong, who was my roommate during both internships and has been a supportive friend ever since. During the internships, we split the rent and dining bills, and shared what we learned with each other every day. I feel very fortunate to be friends with Yuhong, who is not only sharp and passionate about research, but also generous enough to discuss my introduction writing with me at 3 a.m. right before a paper deadline.

It was a great pleasure to work with and learn from many talented peers in the computer science department, including Guanzhou Hu, Shawn Zhong, Suyan Qu, Xiangpeng Hao, Sambhav Satija, Vinay Banakar, Kaiwei Tu, Tingjia Cao, Jing Liu, Kan Wu, Anthony Rebello, Yifan Dai, John Shawger, Fariha Tabassum Islam, Caeden Whitaker, Junxuan Liao, Yuvraj Patel, Keren Chen, Konstantinos Kanellis, Sujay Yadalam, Bijan Tabatabai, Ashwin Poduval, Anjali, Hayden Coffey, Tzu-Tao Chang, Johannes Freischuetz, Tareq Mahmood, Mark Mansi, Jinlang Wang, Ziyi Zhang, Hangdong Zhao, Xiating Ouyang, Zifan Liu, Yunjia Zhang, Ling Zhang, Ting Cai, Yifei Yang, Zhihan Guo, Zhenghong Yu, Wenjie Hu, Abigale Kim, Wanzhu Hou, Simon Frisk, Hokeun Cha, Alicia Lyu, James Sorenson, Yuyuan Kang, Wentao Hou, Zerui Guo, Omid Rostamabadi, Joontaek Oh, Yuhao Zhang, Yaxin Hu, Yeping Wang, Sophie Stephenson, Naman Gupta, Jack West, Tim Lu, Julia Nonnenkamp, Shirley Zhang, and Quinn Burke. My thanks also go to everyone who organized, presented at, or attended the MadSystems Seminar. Through everyone's effort, the department became a place where I could always learn something new, exchange ideas, and enjoy the company of wonderful people.

Among these peers, I want to extend my special thanks to Jing Liu, who has been a mentor to me since the uFS project. I have continually learned from Jing, from experiment design to rebuttal writing. Her feedback was invaluable when I revised the ROS paper for the SOSP '26 submission.

I want to thank my family for their unwavering support throughout my education. My parents have always given me the freedom and flexibility to pursue the career path I wanted, while providing generous financial support along the way. My cat, Alioth, deserves thanks too, for being a fluffy source of emotional support and for keeping me company through many long nights before deadlines.

Finally, I am grateful to the many people I met in Madison, at conferences, and during internships, whose names are probably impossible to list exhaustively. I

appreciate all the conversations we shared, whether exchanging ideas about research or simply having fun together. These encounters, big and small, made these years far richer than this dissertation could ever capture.

# Contents

|                                                                           |             |
|---------------------------------------------------------------------------|-------------|
| <b>Acknowledgments</b>                                                    | <b>ii</b>   |
| <b>Contents</b>                                                           | <b>vi</b>   |
| <b>List of Tables</b>                                                     | <b>x</b>    |
| <b>List of Figures</b>                                                    | <b>xi</b>   |
| <b>Abstract</b>                                                           | <b>xiii</b> |
| <b>1 Introduction</b>                                                     | <b>1</b>    |
| 1.1 Resource Efficiency and Fairness Within a Node . . . . .              | 3           |
| 1.2 Scalability and Consistency Under Replication . . . . .               | 5           |
| 1.3 Memory-Centric Storage for LLM Reinforcement Learning . . . . .       | 7           |
| 1.4 Contributions . . . . .                                               | 9           |
| 1.5 Overview . . . . .                                                    | 10          |
| <b>2 HARE: Resource Efficiency and Fairness in Memory-Centric Storage</b> | <b>12</b>   |
| 2.1 Background and Motivation . . . . .                                   | 13          |
| 2.1.1 Diversified Resource Types in the Cloud . . . . .                   | 14          |
| 2.1.2 Multi-Resource Allocation . . . . .                                 | 14          |
| 2.1.3 Characterizing Caches as a Resource . . . . .                       | 15          |
| 2.2 Multi-Resource Allocation With Caches . . . . .                       | 17          |
| 2.2.1 Optimization Goal . . . . .                                         | 18          |

|          |                                                                      |           |
|----------|----------------------------------------------------------------------|-----------|
| 2.2.2    | HARE With a Single Correlated Resource . . . . .                     | 18        |
| 2.2.3    | Generalized Correlation . . . . .                                    | 22        |
| 2.2.4    | HARE With Multiple Correlated Resources . . . . .                    | 23        |
| 2.2.5    | Analysis and Discussion . . . . .                                    | 24        |
| 2.3      | HopperKV . . . . .                                                   | 25        |
| 2.3.1    | HopperKV Architecture and Resources . . . . .                        | 25        |
| 2.3.2    | Spatially-Sampled Ghost Cache . . . . .                              | 27        |
| 2.3.3    | Dynamic and Robust Allocation . . . . .                              | 28        |
| 2.3.4    | HopperKV Evaluation . . . . .                                        | 30        |
| 2.4      | BunnyFS . . . . .                                                    | 38        |
| 2.4.1    | BunnyFS Design and Implementation . . . . .                          | 38        |
| 2.4.2    | BunnyFS Evaluation . . . . .                                         | 40        |
| 2.5      | Conclusion . . . . .                                                 | 42        |
| <b>3</b> | <b>MultiLog: Scalability and Consistency Under Replication</b>       | <b>44</b> |
| 3.1      | Background and Motivation . . . . .                                  | 45        |
| 3.1.1    | Single-Log Scalability Bottleneck . . . . .                          | 46        |
| 3.1.2    | System Goals for Memory-Centric Storage Replication . . . . .        | 47        |
| 3.1.3    | Why Existing Approaches Fall Short . . . . .                         | 48        |
| 3.2      | PrefixLoom: Prefix-Consistent Reads During Parallel Replay . . . . . | 51        |
| 3.2.1    | PrefixLoom Overview . . . . .                                        | 51        |
| 3.2.2    | Replay Protocol . . . . .                                            | 52        |
| 3.2.3    | Prefix-Consistent Read Protocol . . . . .                            | 54        |
| 3.2.4    | PrefixLoom Example . . . . .                                         | 56        |
| 3.2.5    | Scalability Limitation . . . . .                                     | 56        |
| 3.3      | MultiLog: Bottleneck-Free Scaling . . . . .                          | 57        |
| 3.3.1    | MultiLog on the Primary . . . . .                                    | 57        |
| 3.3.2    | MultiLog on the Replica . . . . .                                    | 59        |
| 3.3.3    | Bounded Replay Progress Divergence . . . . .                         | 60        |
| 3.3.4    | Liveness Upon Quiescence . . . . .                                   | 61        |

|          |                                                                   |           |
|----------|-------------------------------------------------------------------|-----------|
| 3.3.5    | Recovery . . . . .                                                | 62        |
| 3.3.6    | Optimization: KRT Table Compression . . . . .                     | 63        |
| 3.4      | Evaluation . . . . .                                              | 63        |
| 3.4.1    | Implementation and Experiment Setup . . . . .                     | 63        |
| 3.4.2    | Append Microbenchmarks . . . . .                                  | 65        |
| 3.4.3    | Replay Microbenchmarks . . . . .                                  | 66        |
| 3.4.4    | Prefix-Consistent Read Microbenchmarks . . . . .                  | 68        |
| 3.4.5    | Replication Macrobenchmarks . . . . .                             | 72        |
| 3.5      | Conclusion . . . . .                                              | 74        |
| <b>4</b> | <b>ROS: Memory-Centric Storage for LLM Reinforcement Learning</b> | <b>75</b> |
| 4.1      | Background and Motivation . . . . .                               | 77        |
| 4.1.1    | Reinforcement Learning Workload Background . . . . .              | 77        |
| 4.1.2    | Weight Transfer System Goals . . . . .                            | 78        |
| 4.1.3    | Limitations of Existing Approaches . . . . .                      | 80        |
| 4.2      | Reference-Oriented Storage . . . . .                              | 82        |
| 4.2.1    | ROS Overview . . . . .                                            | 83        |
| 4.2.2    | Integrity via Mutability Contract . . . . .                       | 84        |
| 4.2.3    | Availability via Retention Protocol . . . . .                     | 85        |
| 4.2.4    | ROS Summary . . . . .                                             | 86        |
| 4.3      | TensorHub Design and Implementation . . . . .                     | 86        |
| 4.3.1    | Naming Scheme . . . . .                                           | 87        |
| 4.3.2    | TensorHub APIs . . . . .                                          | 87        |
| 4.3.3    | Topology-Aware Replication at Scale . . . . .                     | 89        |
| 4.3.4    | Consistency for Model-Parallel Sharding . . . . .                 | 94        |
| 4.3.5    | Robustness to Churn and Failures . . . . .                        | 95        |
| 4.3.6    | Implementation . . . . .                                          | 96        |
| 4.4      | Evaluation . . . . .                                              | 97        |
| 4.4.1    | Microbenchmarks . . . . .                                         | 98        |
| 4.4.2    | RL Training With Standalone Rollouts . . . . .                    | 101       |

|          |                                                                    |            |
|----------|--------------------------------------------------------------------|------------|
| 4.4.3    | Elastic Rollout on Spot Instances . . . . .                        | 103        |
| 4.4.4    | Cross-Datacenter RL Training . . . . .                             | 105        |
| 4.5      | Experience and Lessons . . . . .                                   | 106        |
| 4.6      | Conclusion . . . . .                                               | 107        |
| <b>5</b> | <b>Related Work</b>                                                | <b>108</b> |
| 5.1      | Resource Allocation . . . . .                                      | 108        |
| 5.2      | Scalable Logging and Replication . . . . .                         | 109        |
| 5.3      | Machine Learning Systems and Data Dissemination . . . . .          | 111        |
| <b>6</b> | <b>Conclusions and Future Work</b>                                 | <b>113</b> |
| 6.1      | Summary . . . . .                                                  | 113        |
| 6.1.1    | Resource Efficiency and Fairness Within a Node . . . . .           | 113        |
| 6.1.2    | Scalability and Consistency Under Replication . . . . .            | 114        |
| 6.1.3    | Memory-Centric Storage for LLM Reinforcement Learning . . . . .    | 115        |
| 6.2      | Lessons Learned . . . . .                                          | 116        |
| 6.2.1    | The Elegance of Storage Abstraction Is Under-Appreciated . . . . . | 116        |
| 6.2.2    | Design for Performance, Then Repair Violated Constraints . . . . . | 118        |
| 6.3      | Future Work . . . . .                                              | 119        |
| 6.3.1    | Distributed Cache-Centric Resource Allocation . . . . .            | 119        |
| 6.3.2    | Fair Resource Allocation for Computational Caches . . . . .        | 119        |
| 6.3.3    | Multi-Key Transactions on MultiLog . . . . .                       | 120        |
| 6.3.4    | Unifying Weight Transfer With Checkpointing . . . . .              | 120        |
| 6.3.5    | Co-Designed Data Systems for Emerging Data Types . . . . .         | 121        |
| 6.4      | Closing Words . . . . .                                            | 121        |
|          | <b>Bibliography</b>                                                | <b>123</b> |

## List of Tables

|     |                                                                       |     |
|-----|-----------------------------------------------------------------------|-----|
| 4.1 | Comparison of Different Weight Transfer Approaches Against Five Goals | 79  |
| 4.2 | TensorHub APIs . . . . .                                              | 88  |
| 4.3 | Training Workload Parameters . . . . .                                | 101 |

# List of Figures

|      |                                                                              |    |
|------|------------------------------------------------------------------------------|----|
| 2.1  | Generic Caching System Model . . . . .                                       | 15 |
| 2.2  | Comparison of Existing Approaches and HARE . . . . .                         | 17 |
| 2.3  | Example Systems . . . . .                                                    | 18 |
| 2.4  | An Example of HARE With Visualization . . . . .                              | 21 |
| 2.5  | HopperKV Architecture . . . . .                                              | 26 |
| 2.6  | Construct Miss Ratio Curve With Ghost Cache . . . . .                        | 28 |
| 2.7  | HopperKV Microbenchmarks . . . . .                                           | 32 |
| 2.8  | HopperKV Scaling Macrobenchmark Result . . . . .                             | 34 |
| 2.9  | HopperKV Dynamic Macrobenchmark Result . . . . .                             | 36 |
| 2.10 | HopperKV Twitter Trace Macrobenchmark . . . . .                              | 37 |
| 2.11 | BunnyFS Architecture . . . . .                                               | 39 |
| 2.12 | BunnyFS Scaling Experiment Results as CDF . . . . .                          | 40 |
| 2.13 | BunnyFS Dynamic Experiment Results . . . . .                                 | 41 |
| 3.1  | Garnet Throughput as the Thread Count Scales . . . . .                       | 46 |
| 3.2  | Overview of the PrefixLoom Protocol on a Replica . . . . .                   | 51 |
| 3.3  | PrefixLoom Read Example With a Visualized Logical Timeline . . . . .         | 55 |
| 3.4  | MultiLog Architecture . . . . .                                              | 58 |
| 3.5  | Append Throughput as the Thread Count Scales . . . . .                       | 65 |
| 3.6  | Replay Throughput Over Various Configurations . . . . .                      | 66 |
| 3.7  | Replay and Read With Various Key Distributions . . . . .                     | 69 |
| 3.8  | Sensitivity Analysis of Replay Progress Divergence at MultiLog(16) . . . . . | 71 |
| 3.9  | Effect of <i>KRT</i> Table Size on Read Tail Latency . . . . .               | 71 |

|                                                                  |     |
|------------------------------------------------------------------|-----|
| 3.10 Replication Macrobenchmark Results . . . . .                | 73  |
| 4.1 RL Workload With Diverse Rollout Patterns . . . . .          | 77  |
| 4.2 Reference-Oriented Storage Workflow . . . . .                | 83  |
| 4.3 TensorHub Naming Scheme . . . . .                            | 87  |
| 4.4 TensorHub Examples . . . . .                                 | 90  |
| 4.5 Pipeline Replication . . . . .                               | 92  |
| 4.6 Sharding Consistency Example . . . . .                       | 94  |
| 4.7 Microbenchmark Results . . . . .                             | 99  |
| 4.8 Weight Transfer Flows With Standalone . . . . .              | 101 |
| 4.9 Standalone Rollout Results . . . . .                         | 102 |
| 4.10 Weight Transfer Flows With Standalone and Elastic . . . . . | 103 |
| 4.11 Elastic Rollout Results . . . . .                           | 104 |
| 4.12 Cross-Datacenter Rollout Results . . . . .                  | 105 |

# Abstract

Modern storage systems increasingly center their designs around fast memory. Across diverse workloads and deployment settings, this shift creates new challenges for achieving scale and performance. On a single multi-tenant node, the system must allocate cache space alongside auxiliary resources (e.g., network and I/O), whose demands correlate with the cache miss ratio. Beyond a single node, systems must scale out through replication, with each replica exploiting multi-core parallelism while preserving consistency. Emerging large language model (LLM) workloads operate on massive data in GPU high-bandwidth memory, requiring a rethink of system abstractions. In this dissertation, we ask: how can we improve the scale and performance of memory-centric storage systems under fairness, consistency, and other workload-specific constraints?

In the first part of this dissertation, we present HARE, a cache-centric multi-resource allocation algorithm that exploits the demand correlations between cache and auxiliary resources. HARE first harvests resources by searching for a cache partition that preserves every tenant’s fair-share throughput while reducing the aggregate consumption of auxiliary resources. It then redistributes the harvested resources to improve the throughput of every tenant. We demonstrate HARE’s generality with two systems: HopperKV, a cloud-native key-value cache that modifies Redis to cache data from DynamoDB, and BunnyFS, a microkernel-style local filesystem for NVMe SSDs. Evaluations show that HopperKV achieves up to  $1.9\times$  higher throughput, and BunnyFS up to  $1.4\times$ .

In the second part, we present MultiLog, a highly scalable replication architecture for memory-centric storage. MultiLog scales replication by sharding the log into multiple sublogs across disjoint keyspaces, each of which can be appended and replayed independently. To keep replica reads prefix-consistent across sublogs,

MultiLog introduces PrefixLoom, a lightweight timestamp-based protocol that lazily restores consistency. Implemented in Garnet, MultiLog improves append throughput by up to 19× and replay throughput by up to 36× over a single-log baseline. Under a write-heavy replication workload, its replica lags the primary by only ~10 ms at p99.9, two orders of magnitude lower than in the baselines.

In the final part, we present Reference-Oriented Storage (ROS), a new approach to weight transfer in LLM reinforcement learning. ROS presents the illusion that certain versions of the model weights are stored and can be fetched on demand. Underneath, ROS stores no copies of the weights; it instead tracks the workers that hold each version for inference, and serves reads directly from their GPUs. We build TensorHub, a production-quality system that instantiates the ROS idea with topology-aware transfer, model-parallel consistency, and fault tolerance. Across three distinct rollout workloads, TensorHub reduces total GPU stall time by up to 6.7× for standalone rollouts, accelerates weight updates for elastic rollouts by 4.8×, and cuts cross-datacenter rollout stall time by 19×. TensorHub has been deployed at ByteDance for reinforcement-learning training.

# Chapter 1

## Introduction

Memory-centric storage is a class of storage systems that serve data primarily from fast, volatile memory. The resulting low latency and high throughput place these systems on the critical path of many online services. Systems like Redis [87] and Memcached [33] power a wide range of applications, including session management [86], leaderboards [81], real-time analytics [82], and ads targeting [131], among others. Their importance is reflected both in deployment scale and developer adoption. For example, Netflix operates a cache fleet of roughly 160 clusters and 18,000 servers, handling 350 million operations per second [35]. A recent StackOverflow survey further reports that 43% of AI-agent developers utilize Redis for agent memory and data storage [116]. Industry projections estimate the memory-centric data store market will grow from approximately \$9 billion in 2025 to \$30 billion by 2034 [51].

However, building memory-centric storage systems that satisfy the demands of diverse workloads and deployment settings is challenging. To improve resource utilization, a single-node deployment often needs to serve consolidated workloads from multiple tenants. In such a setting, the memory-centric storage systems must allocate not only the cache space across tenants, but also other auxiliary resources, such as network, I/O, and backend service capacity, whose demands correlate with the cache miss ratio. Achieving high resource efficiency across multiple resources while maintaining fairness among tenants remains an open problem. When workloads exceed the capacity of a single node, the system needs to scale out through replication. However, conventional log-based replication relies on sequential replay to preserve

consistency, fundamentally limiting scalability on multi-core systems. Designing a replication mechanism that scales efficiently without compromising consistency is therefore difficult. Finally, emerging large language model (LLM) training workloads introduce new requirements for memory-centric storage. These workloads operate on massive data residing in GPU high-bandwidth memory (HBM). Designing a memory-centric storage system native to LLM workloads and GPU memory requires a rethink of the entire system stack, from high-level system abstraction to low-level data transfer implementation.

In this dissertation, we ask the following question: how can we improve the *scale* and *performance* of memory-centric storage systems under fairness, consistency, and other workload-specific constraints? We investigate this question across three contexts: multi-tenant resource sharing, primary-replica replication, and emerging LLM reinforcement learning (RL) training workloads.

We begin with multi-tenant sharing within a node. Memory-centric storage systems typically manage multiple resources, such as cache space, network bandwidth, I/O bandwidth, and backend service capacity. Among them, cache space is unique as it directly impacts the demand for other resources; e.g., a larger cache reduces the miss ratio and therefore lowers I/O demand. An allocation framework should capture these correlations to achieve high efficiency while preserving fairness across tenants. Our key idea is to exploit heterogeneous cache sensitivity among tenants: the system allocates more cache to a cache-sensitive tenant, whose demands for other resources then decrease significantly, and harvests the saved resources to benefit all tenants. Based on this idea, we present **HARE**, a novel cache-centric multi-resource allocation algorithm that exploits this heterogeneity through a two-phase harvest/redistribute approach. We built **HopperKV** and **BunnyFS** as case studies to demonstrate HARE’s generality.

Next, we turn to replicated memory-centric storage, where a primary instance is paired with one or more read-only replicas. The primary serves read/write requests and appends every mutation into a log. Each replica asynchronously pulls and replays the log to update its state while serving read requests to clients, ideally guaranteeing that reads reflect a monotonic prefix of the primary state. Unfortunately, existing replication schemes trade off between multi-core scalability and consistency. A single log stream becomes a contention point that fails to scale [20, 87, 105],

while naively sharding the data into multiple log streams breaks the desired prefix consistency [31]. We present **MultiLog**, a highly scalable replication architecture that shards the log while preserving prefix consistency. Unlike prior naive sharding approaches, MultiLog employs a novel coordination protocol called **PrefixLoom**. The protocol maintains lightweight metadata on the replay time and lazily restores the consistency at the read time. By deferring consistency enforcement, PrefixLoom scales up the replay with minimal overhead.

Finally, we examine memory-centric storage in the context of emerging LLM RL training workloads. RL training requires frequently disseminating heavyweight model weights to thousands of rollout workers. Existing storage systems such as parameter servers [59] or Ray object store [67] require massive storage or memory space and data movement, which is prohibitively expensive for model weights at terabyte scale. We seek to design an *RL-native* memory-centric storage system that fully eliminates the redundant data copies and movement. Our key idea is to utilize highly replicated model weights *in place* and provide access to them efficiently via a storage interface, called **Reference-Oriented Storage (ROS)**. ROS presents the illusion that certain versions of the model weights are stored and can be fetched on demand. Underneath, ROS does not physically store any copies of the weights; instead, it tracks the workers that hold these weights for inference and serves reads directly from their GPUs. We build **TensorHub**, a production-quality system based on the ROS idea with user-friendly interfaces, topology-aware transfer, model-parallel consistency, and fault tolerance. TensorHub has been deployed at ByteDance for cutting-edge RL training.

## 1.1 Resource Efficiency and Fairness Within a Node

Modern cloud data systems are commonly organized as a caching layer layered on top of a backend persistent layer. The cache keeps hot data in memory and absorbs a large volume of read requests, preventing the backend storage layer from becoming overloaded. Memory-centric storage systems such as Redis [87] and Memcached [33] are widely used to implement such caching layers.

For better utilization, these storage systems are typically shared by multiple tenants [1, 41, 42, 103, 110]. However, resource management in multi-tenant storage

systems remains a significant challenge, due to the complexity of allocating multiple resource types and meeting diverse application demands. A workload with good locality may demand a large cache space to fit the working set and high network bandwidth to read from the caching layer; a streaming workload, in contrast, needs a minimally sized cache and massive disk I/O bandwidth.

We are thus left with the question: how should storage services allocate multiple resources, including caches, across tenants, to achieve high performance and fairness? Unfortunately, existing multi-resource allocation frameworks—namely, Dominant Resource Fairness (DRF) and its variants [23, 37, 38]—do not apply to caches. DRF assumes resource demands are mutually independent; a change in cache allocation, in contrast, can significantly change a workload’s demands on other resources (e.g., more cache may reduce I/O). Existing cache partition frameworks [1, 15, 25–27, 32, 49, 76, 80] focus on caches alone and do not capture the interactions among other resources, e.g., a disk I/O-bound workload may become network-bound with more caches and a lower miss ratio.

We seek a generic, unified framework that bridges multi-resource and cache allocation. This framework should generalize DRF, capturing each tenant’s demands on I/O, network, and other resources, and their complex interaction with caching as cache sizes change. It should perform allocation accordingly in real time with low overhead.

We present HARE, a cache-centric multi-resource allocation algorithm, which exploits heterogeneous cache sensitivity across tenants for better utility. It uses a novel two-phase harvest/redistribute approach. In the harvest phase, HARE iteratively searches for a “good” partition of caches such that all tenants maintain the same throughput as a baseline “fair” allocation but consume a lower amount of other resources in total; HARE harvests the saved resources. In the redistribute phase, HARE allocates the harvest to improve the throughput of all tenants. Critically, HARE discovers the most valuable resource type to harvest to maximize the gain for redistribution. HARE subsumes DRF: if tenants are equally sensitive to caches, HARE gracefully falls back to DRF.

We built HopperKV to demonstrate the merit of HARE. HopperKV is a cloud-native key-value store that modifies Redis [87] to cache data from the backend DynamoDB [92]. HopperKV uses lightweight ghost caches and resource accounting

to construct tenants’ miss ratio curves and demand vectors. Based on this knowledge, HopperKV optimizes Redis memory allocation among tenants to fully utilize backend read/write capacity and network bandwidth. HopperKV also implements mechanisms to enhance robustness against estimation inaccuracy and bound tail latency. Through microbenchmarks, we show that HopperKV handles a variety of sharing scenarios. In particular, when all tenants have the same dominant resource, DRF does not achieve any gain over the baseline; Memshare [27] only improves a tenant’s performance at the cost of others, hurting fairness; HARE achieves up to  $1.6\times$  higher throughput via an optimized cache partition. We also show that HopperKV maintains fairness as the number of tenants increases. Finally, we show HopperKV adapts to workload changes and handles real-world workloads [126], achieving up to  $1.9\times$  improvement over the baseline.

To further show the generality of HARE, we developed BunnyFS as another case study. BunnyFS is a high-performance microkernel-style filesystem that manages NVMe SSDs. It dynamically allocates resources, including page cache, I/O bandwidth, and worker thread CPU cycles, across tenants. Although the contexts are different, BunnyFS shares many techniques with HopperKV, demonstrating the wide applicability of HARE. Experiments show that BunnyFS is scalable and adaptive, improving throughput by up to  $1.4\times$  compared to the baseline.

## 1.2 Scalability and Consistency Under Replication

Memory-centric storage systems [20, 31, 87] scale out read-dominant workloads through replication. A primary instance appends every mutation to a write-ahead log and ships it to replicas, which replay the log to stay in sync. Having addressed how a single node shares its resources fairly, we now turn to a complementary question: how should memory-centric storage scale across nodes?

The conventional design uses a single log, which introduces two serialization bottlenecks: at the primary, concurrent threads must coordinate on the log tail to allocate space; at the replica, sequential replay on a single thread cannot match the primary’s concurrent write rate. Once replay can no longer keep up with the write rate, replication lag grows without bound, and any record in flight when the primary fails is permanently lost. For example, during GitLab’s 2017 outage [40], PostgreSQL

replication lag grew to several hours, ultimately causing 18 hours of unavailability and the permanent loss of six hours of data.

Ideally, a log-based replicated system for performance-critical memory-centric stores should satisfy four goals. First, the replication path must scale on both append and replay, so that neither becomes a serialization bottleneck. Second, replica reads must reflect a monotonically progressing prefix of the primary’s state, so that clients never observe time-travel anomalies across multiple reads. Third, the replica must keep only a single version of each key, since memory is the key constraint for microsecond-latency stores serving hot data from DRAM. Fourth, reads at the replica must remain as fresh as possible, because extra staleness propagates as added end-to-end latency in downstream services.

Unfortunately, existing approaches have to make trade-offs. Single-log designs in Redis [87], Valkey [111], KeyDB [105], and earlier versions of Garnet [20] guarantee prefix consistency but fail to scale. Naive sharding in Dragonfly [31] replicates each shard independently and thus scales across cores, but a client reading across shards may observe time-travel anomalies. Parallel-replay protocols built for relational databases [46, 118, 127] preserve consistency via multi-version concurrency control (MVCC) or periodic snapshotting, neither of which fits the single-version design of memory-centric storage.

We first present PrefixLoom, a lightweight replica-side protocol that serves prefix-consistent reads during parallel replay. Unlike prior approaches that rely on MVCC or snapshotting, PrefixLoom uses *lazy coordination* between replay and read to weave a prefix-consistent view out of a continuously updated, never-consistent replica image. Parallel replay therefore proceeds at full speed; reads block only when potential inconsistency is detected.

We then present MultiLog, a highly scalable replication architecture that extends PrefixLoom with *physical sharding* of the log on the primary. MultiLog partitions the log into multiple physical sublogs, each owning a disjoint keyspace and maintaining its own log-space allocator and replication stream; each append thread routes its own record to its corresponding physical sublog based on the key hash.

We implement MultiLog on Garnet [20], a high-performance memory-centric store. MultiLog improves append throughput by up to 19× and replay throughput by up to 36× over a single-log baseline. Under a write-heavy replication workload, MultiLog

sustains  $3.8\times$  higher primary throughput than the single-log baseline and  $2\times$  higher replica read throughput than C5 [46]. Its replica stays within  $\sim 10$  ms of the primary at p99.9, two orders of magnitude fresher than either baseline. Together, MultiLog and PrefixLoom scale memory-centric storage across nodes without compromising prefix consistency or single-version memory efficiency.

### 1.3 Memory-Centric Storage for LLM Reinforcement Learning

So far, we have discussed how to improve existing memory-centric storage systems [20, 87] for general-purpose workloads. As LLM-related workloads become increasingly dominant in modern datacenters, it is worth rethinking how to design the next generation of memory-centric storage systems specifically optimized for LLMs. We begin this exploration with reinforcement learning (RL) training workloads, which have driven recent advances in LLM reasoning [28, 43, 97], coding [61, 120], and mathematical capabilities [96, 99, 130].

Weight transfer plays a critical role in RL training. RL operates as a continuous feedback loop between inference and training, with weight transfer closing the loop. In each step, *rollout* workers run inference to generate responses for reward scoring, and *trainer* workers use the resulting data to update the model weights. After each update, the new weights must be transferred to rollout workers for the next training step. In addition, weight transfer is required for failure recovery, as failures are common at large scale. Model weights are the heaviest state that a rollout worker must recover, and a restarted rollout worker cannot resume inference until it pulls the latest weights.

Production RL increasingly relies on diverse compute resources, posing new challenges for weight transfer. For example, the ByteDance production cluster has abundant spot GPU instances, but their frequent preemptions and restarts cause cluster membership churn, complicating their use. Multi-datacenter deployments provide access to geographically distributed compute capacity [73, 135], but slow inter-datacenter links make weight transfer a bottleneck.

Ideally, a weight transfer system should adapt to network topology and runtime load, accommodate cluster membership changes while masking failures, and avoid unnecessary control-flow coordination and data movement. It should also require little additional storage or memory.

However, existing weight transfer approaches fail to satisfy these goals. Collective communication (e.g., NCCL [69]) delivers high throughput, but its reliance on static communication groups makes it brittle under failures and ill-suited for dynamic clusters with membership changes. Point-to-point communication (e.g., UCX [98]) offers flexibility, yet suffers from network contention under fan-out. Moreover, both approaches couple trainer and rollout control flow, forcing the RL framework to coordinate workers during weight transfer.

Another weight transfer approach is through distributed storage (e.g., parameter servers [53, 59], Ray object store [67]). Unlike communication-based approaches, distributed storage offers a clean decoupling between trainers and rollouts: trainers *push* new weights to storage, while rollouts *pull* them on demand, all with minimal coordination; rollouts may scale up and down without trainers’ awareness. Yet, this push-then-pull pattern doubles data movement and incurs storage space or memory overhead, both of which are prohibitively expensive for TB-sized LLM weights.

We seek to retain the decoupling advantages of storage interfaces while eliminating the overhead of extra data movement and storage space. This is made possible by an important observation: each version of the model weights is *immutable* and *highly replicated* for data-parallel inference. Such redundancy naturally provides multiple equivalent data sources to serve read requests for that version.

Our key idea is to utilize highly replicated model weights *in place* and provide access to them efficiently via a storage interface, called **Reference-Oriented Storage** (ROS). ROS presents the illusion that certain versions of the model weights are stored and can be fetched on demand. Underneath, ROS does not physically store any copies of the weights; instead, it tracks the workers that hold these weights on GPUs for inference. Upon request, ROS directly uses them to serve reads. By avoiding explicit data ownership and extra copies, ROS combines the flexibility and simplicity of storage with the efficiency of direct transfer.

Building such a system is challenging due to *integrity* and *availability* requirements. ROS addresses these challenges with two novel mechanisms: a *mutability contract*

and a *retention protocol*. The mutability contract disciplines the mutation semantics that enable safe weight buffer reuse. The retention protocol defines which versions must be kept; in a corner case where a desired version may be lost, it offloads a copy to CPU memory to fulfill availability requirements.

We build **TensorHub**, which instantiates the ROS idea in a production system with scalable data transfer, model-parallel consistency, and fault tolerance. It schedules transfers with topology awareness and leverages pipeline replication to scale throughput. TensorHub enforces a consistent view among workers within each model-parallel group to ensure correctness. TensorHub proactively detects failures and orchestrates recovery, enabling transparent operation under dynamic cluster scaling.

We evaluate TensorHub on large-scale RL workloads spanning dozens to a thousand GPUs, elastic clusters with spot churn, and multi-datacenter topologies. In a 1024-GPU training task, TensorHub reduces total GPU stall time by up to 6.7× compared to NCCL. With an elastic cluster, TensorHub’s dynamic load balancing yields 4.8× faster updates compared to UCX. In a cross-datacenter setting, TensorHub cuts GPU stall time by 19× by avoiding redundant TCP transfers.

TensorHub has been deployed in ByteDance production to support cutting-edge RL training. With its strong performance and extensive functionality, TensorHub demonstrates a new way to build RL systems that operate effectively and efficiently at scale.

## 1.4 Contributions

This dissertation makes the following contributions.

- **HARE**: a cache-centric multi-resource allocation algorithm that exploits heterogeneous cache sensitivity across tenants to maximize each tenant’s throughput while preserving max-min throughput fairness.
- **HopperKV**: a cloud-native key-value cache that modifies Redis to cache data from a DynamoDB backend, leveraging HARE to holistically allocate cache size, network bandwidth, and DynamoDB read/write units across tenants.

- **BunnyFS**: a microkernel-style local filesystem for NVMe SSDs, utilizing HARE to allocate page cache, I/O bandwidth, and FS worker CPU cycles across tenants.
- **PrefixLoom**: a replica-side protocol that lazily reconstructs each reader’s prefix-consistent view from lightweight metadata, enabling scalable parallel replay over a single-version replica image.
- **MultiLog**: a high-performance log replication architecture for memory-centric storage systems built on PrefixLoom, scaling append and replay throughput on multi-core machines while preserving prefix consistency; production-ready in Microsoft’s open-source Garnet [20] key-value store.
- **ROS**: a storage abstraction tailored for model weights in LLM RL training, which exploits the immutability and high redundancy of weights to deliver storage-style flexibility without the data movement and memory overhead of conventional storage.
- **TensorHub**: a production-quality system based on ROS with user-friendly interfaces, topology-aware transfer, model-parallel consistency, and fault tolerance; deployed at ByteDance for cutting-edge reinforcement-learning training.

## 1.5 Overview

We briefly describe the contents of the chapters of this dissertation.

- **Resource Efficiency and Fairness Within a Node.** In Chapter 2, we develop HARE, a cache-centric multi-resource allocation algorithm that exploits cache sensitivity heterogeneity through a two-phase harvest/redistribute approach. We demonstrate HARE in two systems: HopperKV, a Redis-based key-value cache that fronts DynamoDB, and BunnyFS, a microkernel-style local filesystem for NVMe SSDs.
- **Scalability and Consistency Under Replication.** In Chapter 3, we present PrefixLoom, a replica-side protocol that lazily reconstructs each reader’s prefix-consistent view from lightweight metadata, enabling scalable parallel replay

without multi-version storage or snapshotting. Building on PrefixLoom, we develop MultiLog, a sharding-based log replication architecture for memory-centric storage systems that scales multi-core append and replay while preserving prefix consistency. MultiLog is production-ready in Microsoft’s open-source Garnet key-value store [20].

- **Memory-Centric Storage for LLM Reinforcement Learning.** In Chapter 4, we propose ROS, a storage abstraction for LLM RL weight transfer workloads that exploits weight immutability and redundancy. We build TensorHub, a production system based on ROS with user-friendly interfaces, topology-aware transfer, model-parallel consistency, and fault tolerance. TensorHub has been deployed at ByteDance for cutting-edge RL training.
- **Related Work.** In Chapter 5, we survey related work for all three contributions under a unified memory-centric storage lens. The discussion covers prior approaches to multi-tenant cache allocation, log-based replication and consistency protocols, distributed ML training systems, and peer-to-peer data systems.
- **Conclusions and Future Work.** In Chapter 6, we conclude the dissertation, distill lessons learned across the three works, and outline future directions for memory-centric storage.

## Chapter 2

# HARE: Resource Efficiency and Fairness in Memory-Centric Storage

We begin our study of memory-centric storage systems in a single-node setting. In production, memory-centric storage systems such as Redis [87] and Memcached [33] are widely deployed as caching layers for backend persistent databases [2–10, 90]. In this chapter, we focus on this important use case of memory-centric storage as caching systems.

These caching systems must manage not only precious memory capacity for caching, but also auxiliary resources such as network bandwidth and I/O for request serving and cache-miss handling. To improve resource utilization, these systems are commonly shared among multiple tenants [1, 41, 42, 103, 110].

The performance requirements and the sharing nature require these systems to pursue both *resource efficiency* and *fairness*. Resource efficiency aims to maximize performance under limited resource budgets. Meanwhile, fairness requires that every tenant benefit from such efficiency and no tenant be left starved.

However, managing multiple resources across multiple tenants is particularly challenging for these caching systems. Cache space is a special type of resource: it directly impacts a tenant's demand for other resources. For example, a tenant with a high cache miss ratio demands substantial I/O or backend service capacity. With a larger cache and, consequently, a lower miss ratio, most of the tenant's requests are served by the cache, reducing its demand for I/O and backend service capacity. Such

correlations between cache and other resources significantly expand the allocation space, introducing both complexity and opportunity.

Two orthogonal lines of work address parts of this joint problem, but neither captures the full picture. Existing multi-resource allocation frameworks, such as Dominant Resource Fairness (DRF) [38] and its variants [23, 37, 108], do not incorporate the complex correlations between cache space and other resources. Prior cache-allocation frameworks [1, 15, 25–27, 32, 49, 76, 80] focus solely on cache allocation and therefore miss opportunities to exploit these demand correlations.

This chapter presents HARE, a novel cache-centric multi-resource allocation algorithm that fully exploits the demand correlations between cache and other resources. Based on the observation that tenants do not benefit from cache equally, HARE allocates resources in two phases. In the *harvest* phase, HARE searches for a cache partition under which every tenant maintains its fair-share throughput while the aggregate consumption of other resources decreases. In the *redistribute* phase, HARE reallocates the harvested resources to improve the throughput of every tenant.

While HARE is motivated by the caching use case of memory-centric storage, it is also applicable to other persistent storage systems with caching components. We demonstrate HARE’s generality with two systems: HopperKV, a cloud-native key-value cache that modifies Redis to cache from a DynamoDB backend, and BunnyFS, a microkernel-style local filesystem for NVMe SSDs. HopperKV achieves up to 1.9× higher throughput, and BunnyFS achieves up to 1.4×.

The rest of the chapter is structured as follows. §2.1 covers background and motivation. §2.2 describes the HARE algorithm. §2.3 details the design, implementation, and evaluation of HopperKV. §2.4 presents and evaluates BunnyFS. §2.5 concludes the chapter.

## 2.1 Background and Motivation

In shared storage systems, multiple resources must be carefully allocated to achieve high throughput while respecting fairness goals. In this section, we describe how the types of shared resources are expanding, cover the background of existing cache-oblivious approaches, and discuss why caches bring new opportunities for more sophisticated allocation methods.

### 2.1.1 Diversified Resource Types in the Cloud

As data systems move from traditional on-premises infrastructure to cloud environments, the range of *resources* that are shared is expanding. With on-premises infrastructure, resources are directly associated only with hardware, such as network and I/O bandwidth. In contrast, in the cloud, resources are expanding to include software infra-services, each with its own manageable resources and pricing units; for example, DynamoDB [92] is provisioned and priced by *read units* and *write units*, where requests consume units in proportion to their size [93]. Thus, a system running inside an EC2 [94] instance and utilizing DynamoDB must manage both hardware resources (e.g., VM network bandwidth) and software resources (e.g., DynamoDB read/write units).

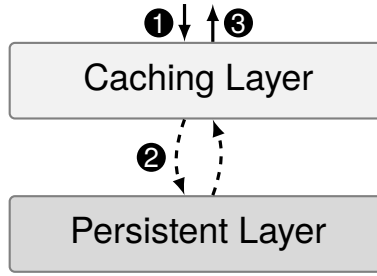
*Demand vectors* [23, 37, 38] are typically used to denote the average amount of each resource required for a tenant to make one unit of progress. Demand vectors compactly represent the ratio of different resources needed by a tenant in a closed-loop system. For example, if a tenant of DynamoDB was served 20K requests with 40K read units (RU) and 20K write units (WU), then its demand vector is  $\langle 2 \text{ RU}, 1 \text{ WU} \rangle$  (units are omitted when clear from context). Since the tenant consumes resources in the ratio of 2:1, if it is allocated 100 read units and 100 write units, it will eventually be throttled due to its shortage of read units, with 50 write units underutilized.

### 2.1.2 Multi-Resource Allocation

A key fairness property that multi-tenant systems must provide is *sharing incentive*: in a system shared by  $x$  tenants, every tenant must have throughput no lower than what it would achieve if it owned  $1/x$  of the total resources exclusively. Violating this property incentivizes tenants to utilize dedicated systems instead.

**Basic Approach: Equal Partition.** To ensure the sharing incentive, a common approach is to partition resources equally [13, 52]: each tenant gets exactly  $1/x$  of every resource. This approach is simple and provides performance isolation, but can be suboptimal; e.g., some tenants may demand more read units while others may demand more write units.

**Advanced Approach: Dominant Resource Fairness.** Dominant Resource Fairness (DRF) [38] is a widely used multi-resource allocation framework. DRF suggests that



**Figure 2.1: Generic Caching System Model.** A request ❶ arrives at the caching layer; ❷ on a miss, the cache fetches data from the persistent layer; ❸ the system then returns the response to the client.

each tenant eventually bottlenecks on a *dominant* resource, and the system should equalize each tenant’s share of the dominant resource. DRF captures heterogeneous demands and always produces a better (or equivalent) allocation than equal partitioning.

**Example.** Consider two tenants that share DynamoDB with a total capacity of 1200 read units/s and 1200 write units/s. Assume tenant A’s workload consists of 50% reads and 50% writes, where each read costs 2 RU and each write costs 4 WU; its demand vector is the weighted per-request cost  $\langle 1, 2 \rangle$ . Assume tenant B’s demand vector is  $\langle 4, 1 \rangle$ . With equal partitioning, each tenant is allocated  $\langle 600, 600 \rangle$ . However, for A, writing is the bottleneck, and it can only achieve 300 req/s, leaving 300 RU/s unutilized; for B, reading is the bottleneck, and it can only achieve 150 req/s, leaving 450 WU/s unutilized. With DRF, A is allocated  $\langle 400, 800 \rangle$  and B  $\langle 800, 200 \rangle$ , leaving only 200 WU/s underutilized; A gets 2/3 of the write units and B gets 2/3 of the read units. Both tenants achieve 33% higher throughput compared to equal partitioning.

### 2.1.3 Characterizing Caches as a Resource

We now consider a generic multi-tenant data system consisting of a caching layer on top of a persistent layer, as shown in Figure 2.1. Such systems are ubiquitous: in a filesystem, a page cache sits on top of an SSD; in the cloud, Redis [87] or Memcached [33] cache data for backend databases [92, 95]. In this model, cache space is a special type of resource that does not fit DRF’s model because of two properties: *non-linear performance* and *demand correlation*.

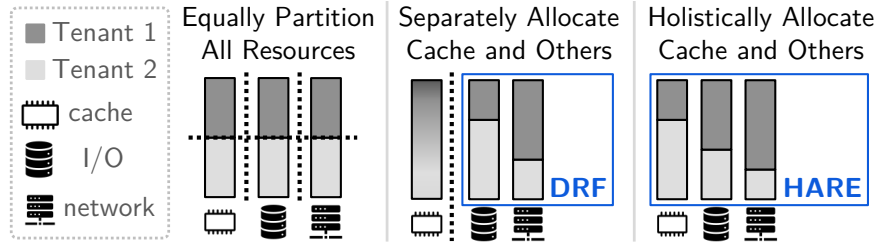
**Property 1: Non-Linear Performance.** In prior multi-resource allocation algorithms [23, 37, 38, 108], the number of requests that can be served in a time window is proportional to the amount of resources allocated (e.g., processing  $2\times$  network packets requires  $2\times$  network bandwidth). However, this linear relationship does not apply to caches:  $2\times$  cache size does not imply  $2\times$  throughput. Not only can the cache miss ratio exhibit a complex non-linear relationship with cache size, but the demand vector is also unsuitable as “the amount of cache needed to serve one request” is ill-defined.

**Property 2: Demand Correlation.** Similarly, in many previous multi-resource allocation algorithms [23, 37, 38], resource demands are mutually independent (e.g., a shortage of network bandwidth cannot be compensated by more I/O). However, this is not the case for caches. With more cache space, a tenant will have a lower miss ratio and thus have less demand (step ②) for all the resources below the cache (e.g., I/O, network, DB read units). We call these resources *cache-correlated* resources.

Cache correlation opens up a new dimension atop the existing multi-resource allocation problem. Each resource type benefits differently from caches. For example, I/O and DB read units are only consumed in step ② and fully skipped upon hits; network, if required in all three steps, is partially saved upon hits; DB write units do not benefit from hits.

**Existing Cache Partition Is Not Multi-Resource-Aware.** Most of the existing cache partition frameworks only tune cache size alone and leave other resources unmanaged [1, 15, 25–27, 32, 49, 76, 80]. CoPart [72] is a recent CPU last-level cache allocation algorithm that goes one step further by incorporating one cache-correlated resource (memory bandwidth). However, it is insufficient for a complex system with more resource types, where the allocation space significantly expands. For example, a high-miss-ratio workload may be disk I/O-bound; with more cache, its miss ratio decreases, and the bottleneck shifts to the network bandwidth of the caching system. An allocation framework must address the interaction among multiple cache-correlated resources.

**Integrating Caches Into Multi-Resource Allocation.** Given caches, Figure 2.2 compares existing approaches with the vision of this work. First, the most straightforward approach that equally partitions all resources (including the cache) achieves fairness



**Figure 2.2: Comparison of Existing Approaches and HARE.** Equal partition provides fairness at the cost of efficiency; DRF allocates multiple resource types, but does not apply to caches; HARE features a holistic view that includes caches.

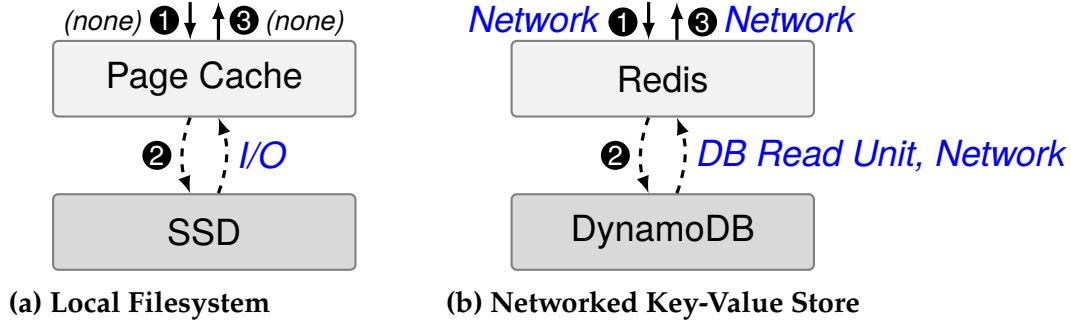
but at the cost of efficiency. Second, DRF improves on equal partitioning, but excludes caches since they do not fit into DRF’s model; as a result, the cache can be partitioned equally [13, 52], non-partitioned [16, 58], or delegated to some standalone multi-tenant cache allocation framework [1, 25–27, 49, 76]. Non-partitioned caches can improve performance by capturing globally hot data, but may not guarantee the sharing incentive: a tenant’s performance can suffer from the interference of noisy neighbors [76, 113]. Standalone cache frameworks miss opportunities to exploit the interaction among resources, leading to suboptimal allocations.

In contrast, we aim to integrate caches into multi-resource allocation and holistically allocate caches and cache-correlated resources to improve efficiency while maintaining fairness. To realize this vision, we first introduce an allocation algorithm, HARE, that captures cache-correlated demands. Second, we describe two systems, HopperKV and BunnyFS, that efficiently derive tenants’ miss ratio curves and demand vectors for HARE and adapt to workload changes at runtime.

## 2.2 Multi-Resource Allocation With Caches

We present Harvest-Redistribute (HARE), a cache-centric multi-resource allocation algorithm that extends DRF. The key idea is to exploit cache-correlated demands: HARE gives more cache to cache-sensitive tenants, whose demands for other resources may drop, and harvests the saved resources to benefit all tenants; among these resource types, the harvesting algorithm prioritizes the one that could maximize the gain.

In this section, we define the optimization goal, introduce HARE by progressing from a single cache-correlated resource to multiple ones, and follow with the



**Figure 2.3: Example Systems.** Each step is labeled with the consumed resources; “(none)” means negligible cost.

discussion; since the basic algorithm assumes a static system, we show how dynamic workloads are handled by real systems in §2.3 and §2.4.

### 2.2.1 Optimization Goal

In the cache-oblivious context, DRF [38] has shown that the system should pursue the max-min fairness of every tenant’s dominant resource share. We generalize this optimization goal to systems with caches. We define the *normalized throughput* of a tenant as its current throughput divided by its baseline (its throughput when all resources are equally partitioned), which quantifies how much a tenant benefits from an allocation. We propose the optimization goal of *maximizing the minimum normalized throughput* across tenants. To incentivize sharing, an allocation must ensure that the minimum normalized throughput across tenants is not lower than 1.

### 2.2.2 HARE With a Single Correlated Resource

We first present the HARE algorithm with only one cache-correlated resource, namely, I/O bandwidth. HARE consists of two phases: *harvest* and *redistribute*. The harvest phase optimizes the cache partition such that all tenants maintain the same throughput as the baseline but consume less I/O bandwidth in total; the saved I/O bandwidth is harvested. In the redistribute phase, the harvested I/O is reallocated to improve every tenant’s throughput equally.

### 2.2.2.1 HARE Algorithm in the Basic Setting

We illustrate HARE through a filesystem example shown in Figure 2.3(a), which only manages two resources: page cache and I/O bandwidth; upon a cache hit, no I/O is consumed.

**Inputs and Notation.** HARE takes two inputs from every tenant: the miss ratio curve (MRC) and per-request I/O cost (on cache misses). The MRC is a function that maps cache size to miss ratio, denoted as  $m$ . Per-request I/O bandwidth cost is analogous to the *demand* in DRF, denoted as  $d_b$ . Since this cost only occurs upon cache misses, the actual per-request cost is  $m(r_c) \cdot d_b$ . If a tenant is allocated  $r_c$  cache and  $r_b$  bandwidth, the throughput (unit: #req/s) is  $T(r_c, r_b) = \frac{r_b}{m(r_c) \cdot d_b}$ . Based on this model, we describe the algorithm procedures below.

❶ **Initialization.** Every tenant starts with the baseline allocation, i.e., all resources are equally partitioned, denoted as  $\langle r_c^{base}, r_b^{base} \rangle$ . We use  $r_b^{harvest}$  to denote the amount of I/O bandwidth harvested, initialized to zero.

❷ **Resource Harvest Phase.** The harvest phase iteratively optimizes the cache partition. In each iteration, HARE relocates a chunk of cache from a tenant  $t_1$  to another tenant  $t_2$ ; to maintain the same throughput as the baseline,  $t_1$  requires extra I/O as compensation, and  $t_2$  relinquishes some. HARE harvests the difference between these two. We call this a cache-I/O trading *deal* between  $t_1$  and  $t_2$ . The size of the relocated cache chunk  $\Delta r_c$  is a tunable parameter denoting a step size. After the harvest phase, all tenants have the same throughput as their baseline but consume less total I/O bandwidth.

#### Harvest Phase: Iteratively Trade Cache and I/O

Repeat the following iteration:

► Each tenant reports: if a chunk of cache  $\Delta r_c$  is taken away, how much I/O compensation does this tenant need to maintain the same throughput? Formally, suppose the tenant currently owns  $r_c$  cache and  $r_b$  I/O bandwidth with miss ratio  $m(r_c)$ ; reducing the cache size by  $\Delta r_c$  causes the miss ratio to increase by  $\Delta m(r_c)$ ; then the I/O compensation can be computed as  $\Delta r_b^{comp} = r_b \cdot \frac{\Delta m(r_c)}{m(r_c)}$ , where  $\Delta m(r_c) = m(r_c - \Delta r_c) - m(r_c)$ .

► Similarly, each tenant reports the amount of I/O bandwidth it can relinquish, if given additional cache  $\Delta r_c$ , to maintain the same throughput. This can be

computed using the same equation as above, except with the sign of  $\Delta r_c$  flipped. We denote its absolute value as  $\Delta r_b^{relinq}$ .

► HARE seeks to make a deal between two tenants  $t_1$  and  $t_2$  where the amount of I/O bandwidth that  $t_2$  would relinquish is greater than the compensation to  $t_1$ . Given such a deal, HARE relocates cache  $\Delta r_c$  from  $t_1$  to  $t_2$ ; meanwhile, it takes I/O bandwidth  $\Delta r_{b,t_2}^{relinq}$  from  $t_2$  and gives  $\Delta r_{b,t_1}^{compen}$  to  $t_1$ ; the difference  $(\Delta r_{b,t_2}^{relinq} - \Delta r_{b,t_1}^{compen})$  is harvested and added to  $r_b^{harvest}$ . If there are multiple deals, HARE picks the most profitable one.

► Repeat the trading until no profitable deal can be made.

**③ Resource Redistribute Phase.** The redistribute phase allocates the harvested I/O bandwidth to improve every tenant's throughput. With the cache miss ratio fixed, the throughput is proportional to the I/O bandwidth. To increase every tenant's throughput equally, HARE increases every tenant's I/O bandwidth by the same relative amount.

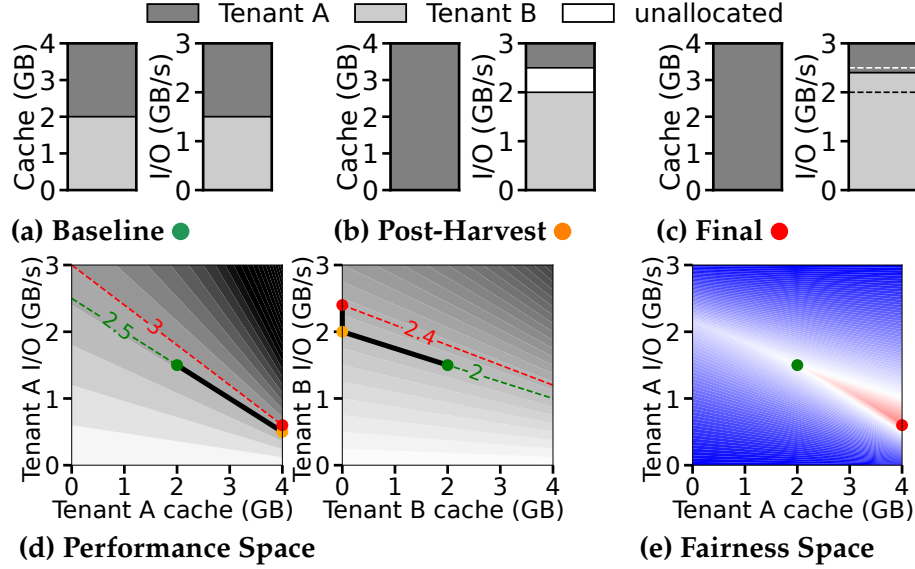
#### Redistribute Phase: Weighted-Partition the Harvest

► The harvested I/O bandwidth is allocated to each tenant weighted by the amount this tenant already owns. Suppose the sum of I/O bandwidth owned by all tenants (i.e., not harvested) is  $r_b^{sum}$ ; then every tenant  $t$  receives additional I/O bandwidth  $r_b^{harvest} \cdot \frac{r_{b,t}}{r_b^{sum}}$ .

### 2.2.2.2 An Illustrative Example With Visualization

Assume a filesystem manages 4 GB page cache and 3 GB/s I/O bandwidth, shared between two tenants A and B. A's working set is 5 GB; B's is 8 GB. Both tenants submit 1 MB read requests following a uniform hotness distribution.

Figures 2.4(a), 2.4(b), and 2.4(c) show each tenant's share of resources during the algorithm. Initially (2.4(a)), each tenant owns half of the cache and I/O. Given that A's miss ratio is 60%, with 1.5 GB/s I/O, its throughput is 2.5 GB/s (=2.5K req/s); similarly, B's throughput is 2K req/s. During the harvest phase, A trades I/O for cache; after this phase (2.4(b)), A has all 4 GB of cache with 0.5 GB/s I/O bandwidth, and B has 2 GB/s I/O bandwidth, delivering the same throughput for both tenants as the baseline but consuming only 2.5 GB/s I/O bandwidth in total.



**Figure 2.4: An Example of HARE With Visualization.**

The redistribute phase (2.4(c)) weighted-allocates the spare 0.5 GB/s I/O bandwidth, with A receiving 0.1 GB/s and B receiving 0.4 GB/s. As a result, A's throughput increases from 2.5K req/s to 3K, and B's increases from 2K to 2.4K; both tenants achieve 20% higher throughput.

The allocation process is detailed in Figure 2.4(d). The figure shows two tenants' *performance spaces*, where the shade at each point  $(x, y)$  denotes the throughput of this tenant with  $x$  GB cache and  $y$  GB/s I/O (darker for higher throughput). The allocation changes are plotted with solid black lines, and the three stages are marked with ●/●/●. The harvest phase (●  $\rightsquigarrow$  ●) essentially walks along a contour level without throughput change (e.g., increasing cache and decreasing I/O for A). The redistribute phase (●  $\rightsquigarrow$  ●) walks upwards along the I/O-axis to improve throughput for both tenants (e.g., increasing I/O bandwidth by 0.1 GB/s for A).

To further understand the merit of HARE, we visualize the *fairness space* in Figure 2.4(e). The color at  $(x, y)$  denotes the minimum normalized throughput between two tenants if A owns  $(x, y)$  and B owns the rest; blue means the minimum normalized throughput is below 1, violating the sharing incentive; red means it is larger than 1, showing improvements. The figure marks the baseline (●) and final allocation (●). As desired, HARE converges to the darkest red area, representing the maximum of the minimum normalized throughput.

### 2.2.2.3 What Has Been Simplified So Far?

The filesystem example in Figure 2.3(a) is simplified in two dimensions. Consider a more general example of a networked key-value store in Figure 2.3(b) where Redis caches data for DynamoDB; the system manages three shared resources: Redis cache size, DynamoDB read units, and network bandwidth.

First, in the local filesystem, I/O bandwidth is consumed only on misses; more generally, some resources (e.g., network bandwidth in a networked key-value store) are consumed on both hits and misses, with a lower consumption on hits. Our more general model captures this degree of cache saving for different resources in §2.2.3.

Second, our previous example only involves one cache-correlated resource (I/O bandwidth); when the miss ratio is zero, the modeled throughput becomes infinite. In practice, such a zero-miss system will eventually bottleneck on some other system resources (e.g., CPU cycles); this is not captured in the previous model, but is in §2.2.4.

### 2.2.3 Generalized Correlation

We generalize the cache-correlated demand by introducing another parameter  $\alpha_i$  for a resource type  $r_i$ . This parameter describes the ratio of resources saved upon a hit compared to a miss case. We call  $\alpha_i$  the *cache-saving constant* of this resource. I/O bandwidth in the filesystem example (Figure 2.3(a)) is a case with  $\alpha = 1$ . Network in the key-value store example (Figure 2.3(b)) has  $\alpha = 0.5$ , where a cache hit only saves the round trip between Redis and DynamoDB, but not the round trip between Redis and tenants.

Formally, a cache-miss request consumes  $d_i$  units of resource, and a hit request consumes  $(1 - \alpha_i) \cdot d_i$  units. Given the miss ratio  $m$ , one request consumes  $(m + (1 - m)(1 - \alpha_i)) \cdot d_i$  units on average. Therefore, the throughput (after reduction) is modeled as  $T_i(r_c, r_i) = \frac{r_i}{(1 - \alpha_i + \alpha_i \cdot m(r_c)) \cdot d_i}$ . Following this generalization, the compensated and relinquished resources can be computed as  $\Delta r_i = r_i \cdot \frac{\alpha_i \cdot \Delta m(r_c)}{1 - \alpha_i + \alpha_i \cdot m(r_c)}$ .

A special case is  $\alpha_i = 0$ , where cache hits do not save cost for this resource. We call it a *cache-independent* resource. Such a resource can still gracefully fit into our model; the compensated and relinquished resources are simply zero.

In practice,  $\alpha_i$  can be derived by maintaining two variables: the resource cost if all requests are hits and if all are misses.  $\alpha_i$  equals one minus the ratio between these two variables.

#### 2.2.4 HARE With Multiple Correlated Resources

We now present the complete HARE algorithm, where the system manages cache and  $n$  types of cache-correlated resources. The expanded allocation scope introduces complexity in the harvest phase: a deal may profit well on one resource type but poorly on others; without a careful deal selection strategy, the algorithm can easily converge to suboptimal states.

HARE's solution is to select the deal that is most profitable on the *system-wide dominant resource*. The observation is that resources are not equally valuable to harvest: the scarcest  $r_i^{harvest}$  limits the system-wide improvement during the redistribution and should be the primary target to harvest.

HARE takes three inputs from every tenant: MRC, the demand vector  $\langle d_1, d_2, \dots, d_n \rangle$ , and the cache-saving constants  $\langle \alpha_1, \alpha_2, \dots, \alpha_n \rangle$ . Every resource  $r_i$  supports a throughput  $T_i$  as modeled in §2.2.3; the end throughput bottlenecks on the *dominant* resource that supports the lowest throughput:  $T(r_c, r_1, r_2, \dots, r_n) = \min_{1 \leq i \leq n} T_i(r_c, r_i)$ . Other non-dominant resources are underutilized. Based on this model, we describe the algorithm procedure below.

**① Initialization.** Every tenant starts with the baseline allocation  $\langle r_c^{base}, r_1^{base}, r_2^{base}, \dots, r_n^{base} \rangle$ . With the baseline allocation, one resource is the dominant resource; others are underutilized. HARE collects underutilized resources across tenants as the initial harvests  $\langle r_1^{harvest}, r_2^{harvest}, \dots, r_n^{harvest} \rangle$ .

**② Resource Harvest Phase.** The harvest phase still follows a trading loop, with the trading resources represented in vectors. The major change is an additional step to determine the system-wide dominant resource before selecting a deal.

##### Harvest Phase: Trade for the Dominant Resource

Repeat the following iteration:

► For every tenant, compute the compensated and the relinquished resources to trade, denoted as vectors  $\vec{\Delta r}^{compen} = \langle \Delta r_1^{compen}, \dots, \Delta r_n^{compen} \rangle$  and  $\vec{\Delta r}^{relinq} = \langle \Delta r_1^{relinq}, \dots, \Delta r_n^{relinq} \rangle$ .

- Decide the system-wide dominant resource as the one relatively scarcest in the harvest:  $r_{dom} = \arg \min_{r_i} \frac{r_i^{harvest}}{r_i^{sum}}$
- Find a tenant  $t_1$  with the smallest  $\Delta r_{dom,t_1}^{compen}$  and another tenant  $t_2$  with the largest  $\Delta r_{dom,t_2}^{relinq}$ . Make a deal between the two tenants if  $\Delta r_{dom,t_2}^{relinq} > \Delta r_{dom,t_1}^{compen}$ : take the relinquished  $\overrightarrow{\Delta r}_{t_2}^{relinq}$  from  $t_2$  and give  $t_1$  the compensation  $\overrightarrow{\Delta r}_{t_1}^{compen}$ ; harvest the difference  $(\overrightarrow{\Delta r}_{t_2}^{relinq} - \overrightarrow{\Delta r}_{t_1}^{compen})$ .
- Repeat the trading until no profitable deal can be made.

③ **Resource Redistribute Phase.** The redistribute phase flows as in the basic setting but applies to multiple resources.

**Redistribute Phase: Weighted-Partition the Harvest**

- Allocate the harvest to every tenant weighted by the amount that tenant already owns, i.e., for every resource type  $r_i$ , every tenant  $t$  receives additional  $r_i^{harvest} \cdot \frac{r_{i,t}}{r_i^{sum}}$ .

## 2.2.5 Analysis and Discussion

We present a theoretical analysis, empirical insight, and an extension of HARE.

**Compatibility With DRF.** HARE gracefully degrades to DRF when all tenants are equally cache-sensitive. If no deal is made during the harvest phase, HARE produces the same allocation as DRF on the system-wide dominant resource. With non-dominant resources underutilized, DRF intentionally keeps them unallocated for strategy-proofness [38]; since strategy-proofness is not a focus of HARE, these idle resources can be weighted-allocated, which we call *conserving redistribution*.

Conserving redistribution provides tenants additional “margins” of non-dominant resources to tolerate temporal burstiness and inaccuracy in demand estimation. This generally does not improve tenants’ throughput, except for tenants with zero demand on the system-wide dominant resource; for example, in an I/O-bound system, if a tenant has a zero miss ratio and demands no I/O, it can get additional throughput with more non-dominant resources.

**Greediness.** HARE is a greedy algorithm and may not be optimal. It is computationally hard to find an optimal allocation due to the arbitrary shape of MRCs. HARE does, however, guarantee a better or equal throughput for every tenant compared to the baseline and DRF with an equal cache partition.

**Convergence Guarantee.** HARE guarantees convergence because every trading iteration requires a positive improvement of normalized throughput; since the normalized throughput cannot grow to infinity, the algorithm must terminate.

**Algorithm Complexity.** For a system with  $n$  tenants, each trading iteration takes  $O(n)$  to compute trading resources and  $O(n)$  to select the deal; the redistribute phase also takes  $O(n)$ . Therefore, the total complexity is  $O(n \cdot \text{trading\_iteration})$ . It is safe to terminate the harvest phase early when the number of trading iterations reaches a maximum threshold, but we never ran into a case needing that. Empirically, the algorithm running time is on the scale of microseconds; given that allocation frequency is at the scale of seconds, the cost is negligible.

**Weighted Baseline Extension.** The baselines as described thus far assume equal partitioning, but in practice, tenants may not be configured equally, e.g., tenant  $t_1$  pays  $2\times$  as much as tenant  $t_2$ . HARE can be generalized to handle such cases by using a weighted baseline, i.e., the baseline allocation is  $t_1$  owning  $2/3$  of all resources, and  $t_2$  owning  $1/3$ . The rest of the algorithm works without changes.

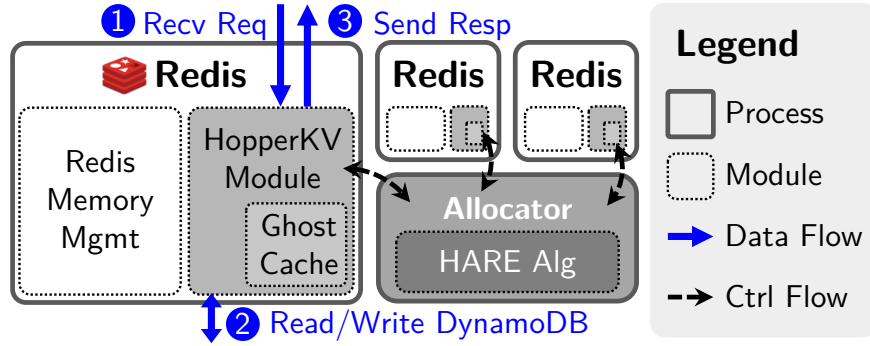
## 2.3 HopperKV

We developed HopperKV to demonstrate the utility of HARE in a real system. HopperKV is a cloud-native key-value store that modifies Redis [87] to cache data from DynamoDB [92]. HopperKV requires no prior knowledge of tenant demands and can dynamically adjust its allocation decisions when workloads change. HopperKV is implemented in 4K lines of C++, and the source code is publicly available [129].

We first describe the architecture and managed resources of HopperKV, how ghost caches are used to efficiently construct MRCs at runtime, and how HopperKV handles dynamic workloads. We then show that HopperKV outperforms alternatives across a range of benchmarks, including scenarios with up to 16 tenants, dynamic changes, and real-world traces.

### 2.3.1 HopperKV Architecture and Resources

Figure 2.5 shows the HopperKV architecture. HopperKV consists of two core components: a Redis Module and an allocator daemon. The Redis Module is



**Figure 2.5: HopperKV Architecture.** The new components introduced by HopperKV are in shade: a customized module is loaded into each tenant’s dedicated Redis instance; an allocator daemon controls the resource limits of each Redis instance.

loaded into every Redis instance; all Redis instances are co-located on a machine and managed by the allocator daemon; for better isolation, each tenant is assigned a dedicated Redis instance running LRU. HopperKV manages four resource types: Redis cache size, network bandwidth, DynamoDB read units (RU), and DynamoDB write units (WU). HopperKV does not incorporate CPU cycles into the allocation scope because Redis is rarely CPU-bound [83]. The allocator daemon determines each Redis instance’s resource budget.

Read request handling consists of three steps. ❶ The request is delivered to the tenant’s Redis instance. ❷ On a miss, the Redis server fetches data from DynamoDB. ❸ The server sends data back. The write path follows similar steps, except that step 2 occurs on both cache hits and misses; the current implementation uses write-through caching.

Different resources are consumed in these steps: network is consumed in all three steps; read units are consumed in step 2 for read misses; write units are consumed in step 2 for all writes. Thus, network and read units are both cache-correlated resources, while write units are cache-independent.

**Example.** Suppose each 2 KB read request costs  $\langle 4 \text{ KB}, 1 \text{ RU}, 0 \text{ WU} \rangle$  on a miss and  $\langle 2 \text{ KB}, 0 \text{ RU}, 0 \text{ WU} \rangle$  on a hit; each 1 KB write costs  $\langle 2 \text{ KB}, 0 \text{ RU}, 1 \text{ WU} \rangle$ .<sup>1</sup> For a workload

<sup>1</sup>DynamoDB imposes request size limits per unit: each read unit supports one read request up to 4 KB; each write unit supports one write request up to 1 KB; larger requests cost proportionally more units [93].

with 90% 2 KB reads and 10% 1 KB writes, the weighted-average per-request cost is  $\langle 3.8 \text{ KB}, 0.9 \text{ RU}, 0.1 \text{ WU} \rangle$  on a miss and  $\langle 2 \text{ KB}, 0 \text{ RU}, 0.1 \text{ WU} \rangle$  on a hit. This provides the demand vector  $\langle 3.8 \text{ KB}, 0.9 \text{ RU}, 0.1 \text{ WU} \rangle$  and the cache-saving constants 0.47 for network, 1 for read units, and 0 for write units.

**HopperKV Redis Module and APIs.** A core component of HopperKV is a Redis Module [84], which is a .so binary loaded into an unmodified Redis server at runtime via the Redis `MODULE LOAD` command. The module supports customized commands with native performance. HopperKV exposes `HOPPER.GET` and `HOPPER.SET` as the data APIs.

`HOPPER.GET` wraps on top of Redis native `GET`. If Redis reports a miss, the module fetches data from DynamoDB. During this process, the module records actual read unit and network usage and *hypothetical* usage if it was a hit/miss. The actual usage is useful for monitoring; the hypothetical usage provides the demand vectors and cache-saving constants. The module also updates the ghost cache for MRC (detailed in §2.3.2). `HOPPER.SET` inserts or updates key-value pairs in Redis and forwards the data to DynamoDB for persistence, also tracking network and write unit usage.

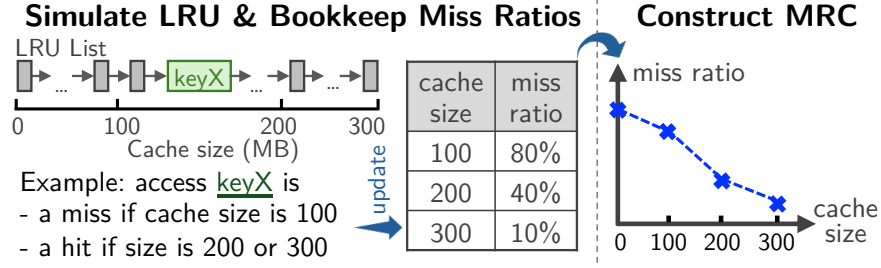
The module exposes control plane APIs to the allocator for extracting demand vectors and MRCs and setting resource limits. Internally, the module uses rate limiters to keep network and read/write unit consumption within the limit.

**HopperKV Allocator Daemon.** The allocator periodically collects demand vectors and MRCs from Redis instances, runs the HARE algorithm, and sets resource limits on each Redis instance based on algorithm outputs. §2.3.3 details its mechanisms built for dynamic and robust allocations.

HopperKV focuses on multi-tenancy on a single node. As future work, a multi-node cluster can be implemented with another placement allocator on top, similar to Pisces [103].

### 2.3.2 Spatially-Sampled Ghost Cache

To learn MRCs at runtime, HopperKV leverages ghost caches [18, 114, 115], as illustrated in Figure 2.6. The ghost cache is implemented as an LRU list, where each entry records the *hash* of the key and the *size* of its KV data (no actual KV data is stored). At runtime, every key access is forwarded to the ghost cache, which runs



**Figure 2.6: Construct Miss Ratio Curve With Ghost Cache.**

simulated cache replacement and maintains a miss ratio table for a range of cache sizes. Since LRU satisfies the inclusion property [63] (i.e., given two caches with sizes  $z_1 < z_2$ , the set of in-cache keys follows  $C(z_1) \subseteq C(z_2)$ ), the ghost cache only needs to maintain one LRU list of size  $z_n$ ; to determine whether an access is a hit in a cache of size  $z_i < z_n$ , the ghost cache checks whether the key is present in the  $z_i$  prefix of the LRU list. The MRC is constructed as a piecewise function based on the miss ratio table.

To reduce ghost cache overhead, HopperKV applies spatial sampling [115], which only forwards a key access to the ghost cache if it falls in the sampling space (e.g., leading hash bits are all zeros). With a  $1/s$  sample ratio, the ghost cache maintains an LRU list of  $1/s$  the size, substantially reducing CPU cost and memory footprint. Based on measurements, we use  $s = 32$  to balance accuracy ( $<1\%$  error) and overhead ( $<25$  ns/key). Note that even as the tenant count scales, the total overhead remains a small fraction of total resources, because the sampling ratio determines the fraction of total requests forwarded to the ghost cache, independent of the number of underlying tenants. If needed, more advanced MRC construction techniques can be applied, e.g., non-inclusive replacement policies [114], and bias-reduced sampling [18].

### 2.3.3 Dynamic and Robust Allocation

The HARE algorithm itself takes a static view of tenants' demands, but a real-world system must handle time-varying workloads. To adapt to workload changes, HopperKV runs an instance of HARE periodically (e.g., every 20 seconds) based on the statistics collected in a sliding window (e.g., past one minute). This frequency suffices because workload changes take seconds to become statistically significant,

only after which the system should react (e.g., Redis cache is usually a few GB while a tenant’s throughput is typically at the scale of 10K requests per second [126], accessing <1 MB of data).

HopperKV is built with robustness in mind. We implemented the following mechanisms to make the system resilient against noise and reduce tail latency.

**Selectively Apply New Allocations.** We observe that noise exists in the system even in a steady state; thus, the algorithm may converge to different (but close) resource allocations even when the workloads have not changed. To avoid oscillation between different, but similar, states, HopperKV only applies a new resource allocation if the new allocation is predicted to have a significant advantage (e.g., >5%) over the current one.

**Smooth Cache Migrations.** When applying a new allocation, the HopperKV allocator migrates memory quota across Redis instances. As a result, the instances that are receiving more cache will temporarily have some unpopulated cache space; since corresponding tenants may have had other resources decreased, these tenants may experience a temporary throughput degradation before the cache is warmed up.

To alleviate this degradation, the allocator divides cache space allocations (and other resources) into 16 MB chunks and migrates only one chunk at a time; the next chunk is only migrated after the previous one has been populated. The quota for other resources is synchronized with the migration of the individual cache chunks.

**Take MRC With a Grain of Salt.** In practice, the MRCs produced by the ghost caches are not perfectly accurate due to sampling and noise. These small errors are tolerable in most cases, but when the miss ratio is very low, a small absolute error leads to a large relative error. As an extreme example, consider a tenant with a very low miss ratio near 0.5%. If its miss ratio is incorrectly estimated as 0%, HARE will allocate it no read units, leaving that tenant’s (few) misses unhandled.

The simple and elegant solution is to add a small constant to MRCs. When computing  $\Delta r_i$  during trading,  $\Delta m$  is unaffected, and  $m$  is impacted insignificantly unless  $m$  is small. This method slightly favors low-miss-ratio tenants, but such benefits are well-bounded since the added constant is small. This constant is tuned based on the measured ghost cache error; our implementation uses 1%. Empirically, this method significantly improves system robustness against inaccuracy at a minor cost of fairness.

**Tail Latency.** While HopperKV primarily optimizes throughput, it also performs decently in tail latency. Since tail latency primarily arises from cache misses, we focus on the miss-handling path. Queueing theory suggests that tail latency increases when the cache-miss processing rate is low and utilization is high. In practice, this occurs when a low-miss-ratio tenant is allocated few read units (i.e., a low processing rate), but a burst of cache misses arrives within a short time window, triggering rate-limiter throttling. In HopperKV, the MRC salting technique provides additional capacity to low-miss-ratio tenants (i.e., reducing utilization), which can effectively absorb such temporal bursts and lower tail latency. §2.3.4.1 presents an empirical measurement of tail latency. As future work, the system can incorporate a queueing-theoretic model into the harvest phase and filter out the deals that may lead to tail-latency spikes.

### 2.3.4 HopperKV Evaluation

Through the evaluation, we answer the following questions: When does HopperKV provide benefits over alternatives? Can HopperKV scale to many tenants? When workloads vary over time, can HopperKV dynamically adjust allocation? How does HopperKV perform on real-world workloads?

**Experiment Settings.** We design the following sets of experiments to evaluate HopperKV: two-tenant *microbenchmarks* (§2.3.4.1) validate that HopperKV can handle a variety of sharing cases; the *scaling macrobenchmark* (§2.3.4.2) verifies that HopperKV can scale with the number of tenants; the *dynamic macrobenchmark* (§2.3.4.3) shows HopperKV can handle workloads varying over time; the *trace-replay macrobenchmark* (§2.3.4.4) demonstrates that HopperKV performs well on real-world workloads.

We compare HARE with four alternatives: 1. baseline where all resources are equally partitioned; 2. pure DRF with caches equal-partitioned (conserving redistribution enabled for a fair comparison); 3. Memshare [27] for the cache partition with DRF for other resources (*MS+DRF*); 4. *NonPart* where tenants share a global resource pool and all resources are non-partitioned. Memshare [27], a state-of-the-art multi-tenant key-value store, allocates more cache to the tenant with larger MRC gradients [26]; it does not consider other cache-correlated resources. The combination of Memshare and DRF represents the most sophisticated resource allocation prior

to our work. NonPart represents real-world scenarios where system administrators enforce no performance isolation, favoring efficiency by caching globally hot data.

All comparison policies are ported to the HopperKV codebase. In baseline, pure DRF, Memshare+DRF, and HARE, each tenant is assigned a dedicated Redis instance for the cache partition. For Memshare, we configure the minimum cache size of a tenant to 50% as in the original paper [27]. To handle the global LRU in NonPart and make it resemble the Redis cluster mode [85] used in practice, for an experiment with  $x$  tenants, we use  $x$  Redis instances, and every instance hosts  $1/x$  of the working set from each tenant.

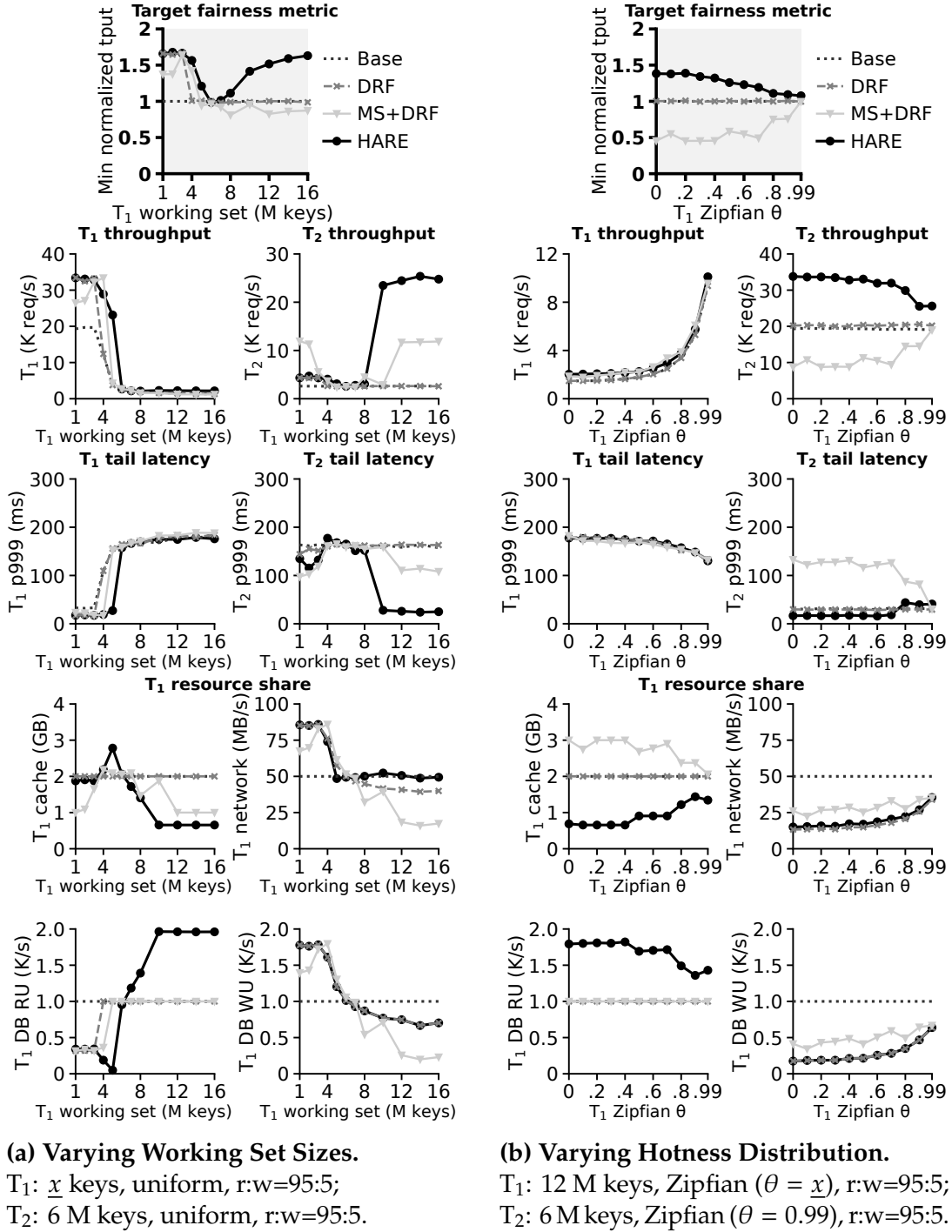
Unless otherwise specified, baseline resources are 2 GB cache, 50 MB/s network, 1K DynamoDB read units/s, and 1K write units/s; workloads use 16-byte keys and 500-byte values. Experiments run on AWS with a cluster of EC2 m6i instances; each tenant uses a 2xlarge instance (8 vCPUs, 32 GB memory) to submit requests for microbenchmarks, scaling, and dynamic experiments, and an 8xlarge instance (32 vCPUs, 128 GB memory) for the trace-replay experiment; the server uses a 16xlarge instance (64 vCPUs, 256 GB memory).

#### 2.3.4.1 HopperKV Microbenchmarks

Through two-tenant microbenchmarks, we demonstrate HARE’s merits over the baseline, DRF, and Memshare+DRF.

**Varying Working Set Sizes.** In the first experiment, we vary tenant  $T_1$ ’s working set from 1 million to 16 million keys (each million keys occupies 0.55 GB Redis memory) while keeping  $T_2$ ’s fixed at 6 million keys. Both workloads consist of 95% reads and 5% writes, with uniform hotness.

Figure 2.7(a) shows the result, with the leftmost column highlighting the minimum normalized throughput between the two tenants. In all cases, HARE performs as well as or better than pure DRF, which performs as well as or better than the baseline; Memshare+DRF performs worse than the baseline in some cases. Specifically, when  $T_1$  has  $\leq 3$  M keys, pure DRF and HARE produce the same allocation since  $T_1$ ’s working set fits within the baseline 2 GB cache, requiring no read units; meanwhile,  $T_2$  is read-unit-bound. DRF handles this case well; since HARE finds no profitable cache trading, it gracefully falls back to DRF. When the working set of  $T_1$  increases to



**Figure 2.7: HopperKV Microbenchmarks.** We report minimum normalized throughput (the target metric; highlighted on the top), two tenants' absolute throughput and p999 tail latency, and  $T_1$ 's resource share ( $T_2$ 's resource share is the remainder).

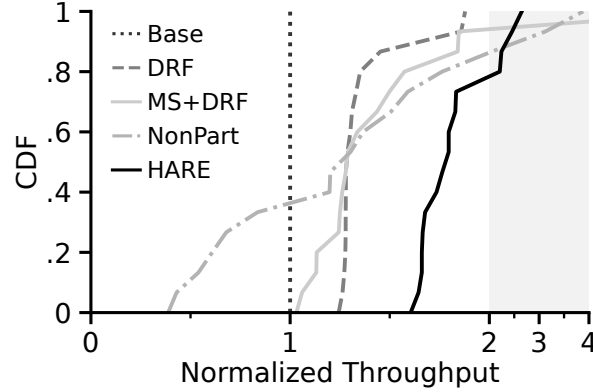
4 M keys,  $T_1$  is also bound by read units. Since both tenants have the same dominant resource, pure DRF does not improve the baseline; Memshare performs slightly better by optimizing the cache partition. HARE achieves a 56% improvement by trading more cache to  $T_1$ , allowing its working set to fit.

When  $T_1$  has 6 M keys, the two tenants have identical workloads, so the equal partition baseline is already optimal. As  $T_1$ 's working set grows,  $T_2$ 's workload is relatively more cache-friendly. Memshare and HARE react to this trend differently. Memshare prioritizes efficiency over fairness: since  $T_2$  utilizes cache better, it takes cache from  $T_1$  without compensation, degrading the performance of  $T_1$  and breaking the sharing incentive. Lack of fairness is a fundamental limitation of Memshare (and other algorithms that only manage cache): it must hurt at least one tenant to benefit others unless there is an MRC plateau. In contrast, HARE captures the interaction between cache and other resources and ensures the cache trading between tenants is fair:  $T_1$  donates more cache with additional read units compensated. In the end, HARE outperforms Memshare+DRF in both fairness and efficiency, with up to 63% higher throughput over the baseline.

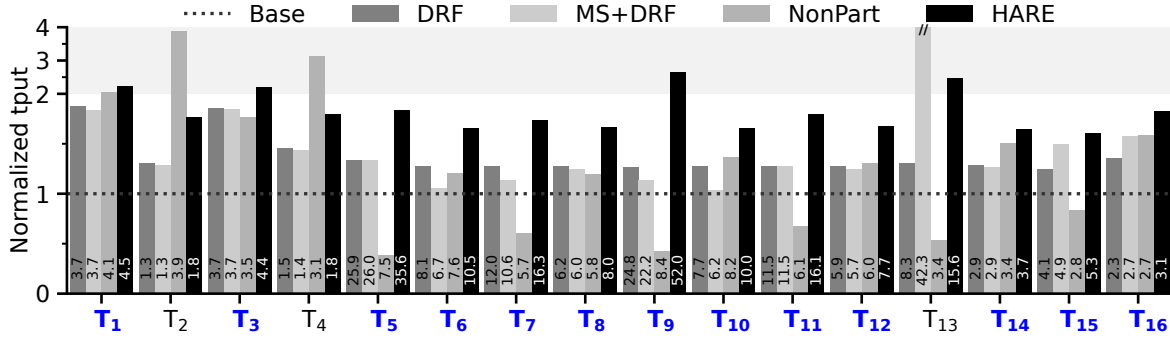
**Varying Hotness Distribution.** In the second microbenchmark, we vary the skewness of  $T_1$  from Zipfian  $\theta = 0$  to 0.99 (larger  $\theta$  means more skewed) with a fixed working set of 12 M keys;  $T_2$  accesses 6 M keys with Zipfian  $\theta = 0.99$ ; both workloads contain 95% reads and 5% writes.

Figure 2.7(b) shows that, when both tenants are bound by the same resource (read units), pure DRF and the baseline are equivalent, Memshare+DRF causes about 50% degradation, and HARE improves throughput by up to 38%. When  $T_1$ 's  $\theta$  is small, its MRC gradient is much larger than  $T_2$ 's, so Memshare only allocates the minimum cache (1 GB) to  $T_2$ , significantly degrading its performance. In contrast, HARE recognizes that allocating more cache to  $T_2$  is more resource-efficient; the saved resources eventually benefit both tenants.

**Tail Latency.** HARE primarily optimizes for throughput, but it still matches or outperforms other alternatives on p999 tail latency, because the MRC salting technique in HopperKV can absorb the temporal bursts of cache misses effectively.



(a) CDF of Normalized Throughput



(b) Individual Tenant's Throughput. Each bar is labeled with absolute throughput (unit: K req/s). 13 out of 16 tenants achieve the best throughput with HARE (highlighted in blue).

**Figure 2.8: HopperKV Scaling Macrobenchmark Result.** For better readability, the throughput axis is folded in the shaded area.

### 2.3.4.2 HopperKV Scaling Macrobenchmark

We show that HopperKV can scale through an experiment with 16 tenants running YCSB workloads [29]. These tenants are categorized into four groups:

- Group A:  $T_1$ - $T_4$  run YCSB-A (50% read, 50% write, Zipf)
- Group B:  $T_5$ - $T_8$  run YCSB-B (95% read, 5% write, Zipf)
- Group C:  $T_9$ - $T_{12}$  run YCSB-C (read-only, Zipf)
- Group E:  $T_{13}$ - $T_{16}$  run YCSB-E-like (95% scan, 5% write, Zipf, scan length between 1~100); to explicitly control the working set size, write ops are overwrites (instead of insertions as in the original YCSB-E)

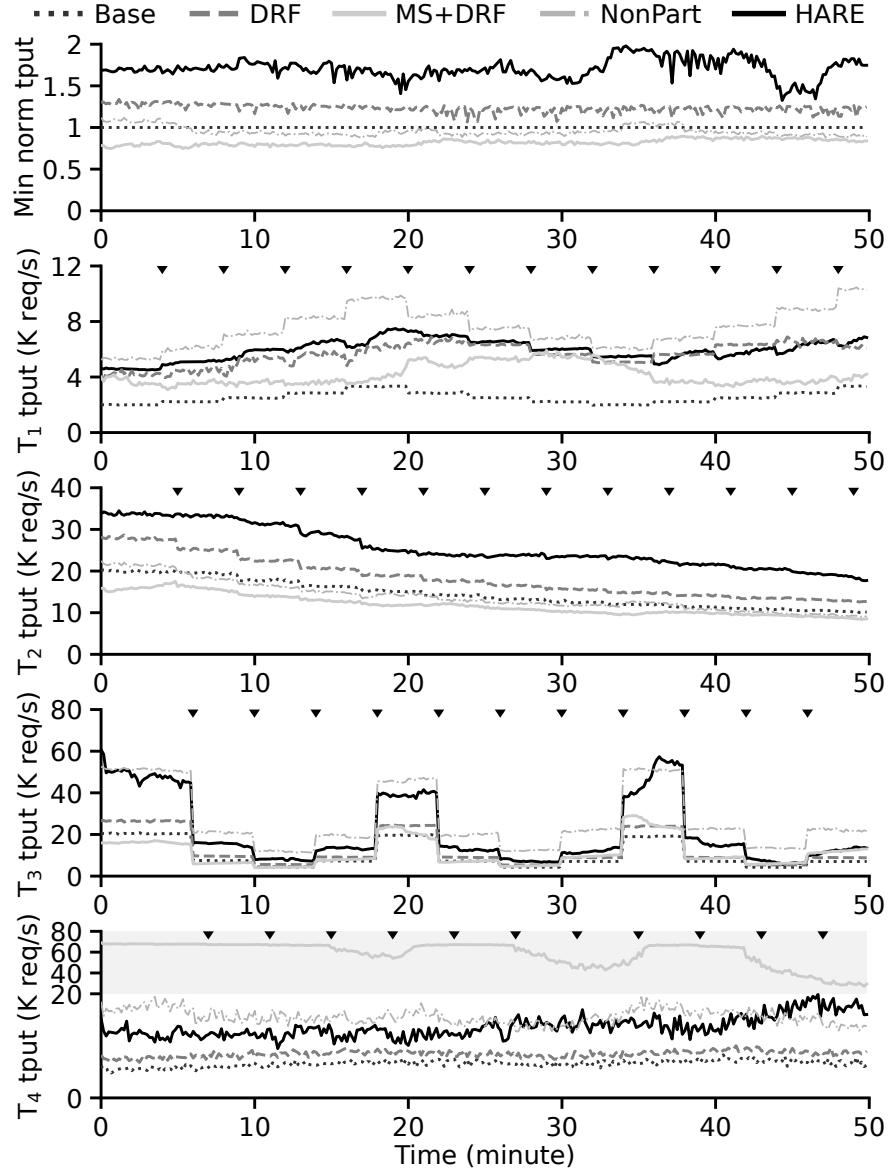
Within each group, the first two tenants have 6 million keys and the last two have 12 million keys; the first and the third tenants use 500-byte values, and the second and the fourth use 2000-byte values. Note that KV size impacts the ratio of read and write units: a 2000-byte value costs 1 RU or 2 WUs, while a 500-byte value only costs 1 RU or 1 WU.

Figure 2.8 shows normalized throughput with the baseline, DRF, Memshare+DRF, NonPart, and HARE. As expected, DRF and HARE guarantee the sharing incentive, and their normalized throughputs are always greater than 1. DRF achieves 1.2~1.9 $\times$  improvement over the baseline because Group A tenants are write-dominant and donate read units to others. Memshare+DRF performs similarly to pure DRF, but it favors  $T_{13}$  (due to its large MRC gradient) at the cost of lower improvement for others (e.g.,  $T_6$ ,  $T_7$ ,  $T_9$ , and  $T_{10}$ ). NonPart produces unfair allocations: 6 out of 16 tenants experience up to 3 $\times$  slowdowns. These slowdowns occur for two reasons. First, even though a 2 KB KV pair consumes 4 $\times$  more memory than a 0.5 KB pair, Redis’s LRU implementation evicts KV pairs only by recency, ignoring KV size; as a result, with a non-partitioned cache, tenants with larger value sizes take advantage of others. Second, NonPart does not provide isolation, so a cache-friendly workload can be dragged down by others:  $T_5$ ,  $T_7$ ,  $T_9$ , and  $T_{11}$  have high absolute throughput in the baseline and are significantly slowed down in NonPart.

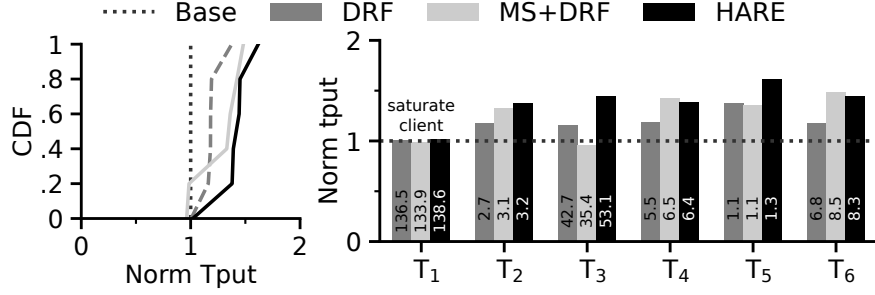
HARE significantly outperforms others in fairness: 13 out of 16 tenants achieve their best performance with HARE, with 1.6~2.7 $\times$  improvement. For Group A tenants, the cache yields less utility because half of their requests do not benefit from hits, making them less cache-sensitive; as a result, Group A becomes the major cache donor, despite the same hotness distribution as Groups B and C. Neither DRF nor Memshare exploits this important pattern.

### 2.3.4.3 HopperKV Dynamic Macrobenchmark

To evaluate the adaptivity of HopperKV, we conduct stress tests where every minute a tenant changes its workload pattern. Initially,  $T_1/T_2/T_3/T_4$  run YCSB-A/B/C/E-like workloads, each with 6 million keys. Starting at the 4<sup>th</sup> minute,  $T_1$ ’s write ratio oscillates between 50% and 30% (decreasing/increasing by 5% every 4 minutes); from the 5<sup>th</sup> minute,  $T_2$ ’s working set expands by 0.5 million keys every 4 minutes; from



**Figure 2.9: HopperKV Dynamic Macrobenchmark Result.** The top-most figure summarizes the minimum normalized throughput over time. Every "▼" denotes a workload change. For better readability, T<sub>4</sub>'s Y-axis is folded in the shaded area.



**Figure 2.10: HopperKV Twitter Trace Macrobenchmark.** Each bar is labeled with absolute throughput (unit: K req/s).

the 6<sup>th</sup> minute, T<sub>3</sub>’s Zipfian  $\theta$  value cycles among 0.99/0.8/0.6 to emulate periodic hot spots; from the 7<sup>th</sup> minute, T<sub>4</sub>’s max scan range cycles among 100/80/60/40 keys. This drastically changing workload stresses the adaptivity of HopperKV.

Figure 2.9 shows tenants’ throughput over time, with the minimum normalized throughput across tenants summarized at the top. Among these policies, only pure DRF and HARE preserve the sharing incentive; Memshare hurts T<sub>2</sub> and T<sub>3</sub> to disproportionately benefit T<sub>4</sub>, since T<sub>4</sub> has the highest MRC gradient; NonPart performs well for T<sub>1</sub>, T<sub>3</sub>, and T<sub>4</sub>, but degrades T<sub>2</sub>’s performance after the 5<sup>th</sup> minute.

Focusing on the two fair algorithms, DRF and HARE, we see that initially, DRF improves throughput by 1.3× relative to the baseline, while HARE improves throughput by 1.7×. When tenants change their workload, pure DRF reacts very quickly because it manages only stateless resources. HARE needs to migrate the cache quota, which takes time to converge; as shown, the migration mechanism of HopperKV (§2.3.3) performs this smoothly to ensure stable performance, achieving up to 1.9× improvement. Overall, HARE consistently outperforms other alternatives with the best fairness and comparable efficiency.

#### 2.3.4.4 Real-World Macrobenchmark With Twitter Traces

We use a macrobenchmark with Twitter production traces [126] to demonstrate that HopperKV handles realistic workloads. We sample six cluster traces from the dataset (cluster IDs: 2, 19, 33, 34, 40, 54), covering a variety of key-value pair sizes (average sizes in bytes: 89, 143, 1142, 355, 199, 265), write ratios (0%, 25%, 1%, 6%, 50%, 3%), and hotness distributions; more detailed trace statistics are available [125]. Since

these workloads are more skewed, we experiment with 1 GB baseline cache; other baseline resources remain the same.

The results are shown in Figure 2.10, with the CDF summary on the left. HARE generally achieves higher normalized throughput than pure DRF and Memshare+DRF. The performance of  $T_1$  is identical across approaches because  $T_1$  has a small working set ( $\sim 150$  MB) and 0% miss ratio; even the baseline achieves high throughput that saturates the client VM’s request submission capacity. Excluding  $T_1$ , HARE achieves at least 38% improvement, while pure DRF achieves only 16%, and Memshare+DRF degrades  $T_3$  by 4%.

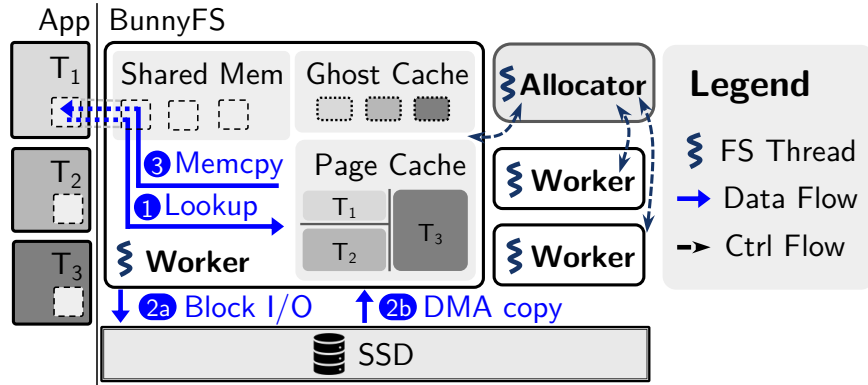
With DRF,  $T_5$  donates read units, as it is the only write-dominant workload. Memshare+DRF takes cache from  $T_1$  due to  $T_1$ ’s MRC plateau; however, it cannot take more than 500 MB because of Memshare’s 50% cache minimum limit, leaving 350 MB cache still wasted on  $T_1$ ; it also takes 200 MB cache from  $T_3$  due to its small MRC gradient, resulting in degradation. With HARE,  $T_1$  donates its unused 850 MB cache, and  $T_5$  trades 550 MB cache for read units. In summary, HARE handles real-world workloads with better or matching gains over the alternatives.

## 2.4 BunnyFS

To show the generality of HARE, we designed BunnyFS, a local filesystem based on the semi-microkernel uFS [60]. The uFS design instantiates the filesystem as a process that serves data through shared memory; the approach is tailored for high-performance NVMe SSDs with single-digit microsecond latency. Through such a performance-critical system, we demonstrate that HARE introduces minimal overhead. BunnyFS adds/modifies 4K lines of C++ code to uFS; the source code is publicly available [128]. We describe the BunnyFS design in §2.4.1 and present the evaluation in §2.4.2.

### 2.4.1 BunnyFS Design and Implementation

The BunnyFS architecture is shown in Figure 2.11. The filesystem process contains multiple worker threads and an allocator thread. A tenant submits requests to a worker thread via a shared memory region. Read request handling consists of the



**Figure 2.11: BunnyFS Architecture.**

following steps: **1** The worker polls the request and looks up the data in the tenant’s page cache. **2a** On a cache miss, the worker submits a block I/O request to the SSD via SPDK [106]. **2b** The NVMe controller of the SSD copies data to the page cache through direct memory access (DMA). For cache hits, step 2 is skipped. **3** The worker copies data from the page cache to the tenant’s shared memory region.

BunnyFS manages three types of resources: page cache, I/O bandwidth, and FS workers’ CPU cycles. The memory copy in step 3 is the major consumer of CPU cycles, while other steps are relatively CPU-light; since step 3 does not benefit from cache hits, we model CPU cycles as a cache-independent resource. Our implementation of BunnyFS is primarily optimized for reads. A write workload is treated like an uncacheable read workload: it consumes I/O bandwidth and does not benefit much from a large cache.

**BunnyFS Worker Threads.** Workers manage per-tenant LRU page caches and request queues; they use rate limiters to control every tenant’s CPU and I/O bandwidth consumption. Workers perform resource accounting and maintain ghost caches to construct demand vectors and MRCs. BunnyFS uses a ghost cache implementation similar to HopperKV’s.

**BunnyFS Allocator Thread.** The allocator thread wakes up periodically to adjust resource allocation. It polls demand vectors and MRCs from workers, runs the HARE algorithm, and notifies workers of the new allocation. By periodically adjusting resource allocations (every one or a few seconds), BunnyFS adapts to workload changes.

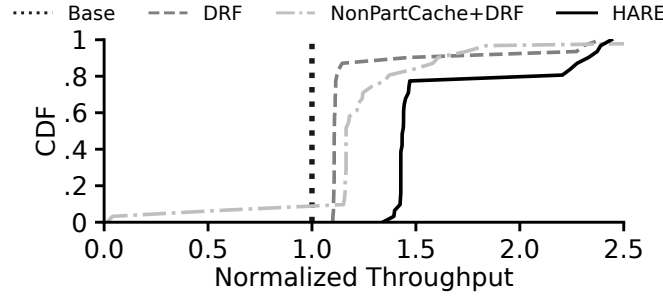


Figure 2.12: BunnyFS Scaling Experiment Results as CDF.

## 2.4.2 BunnyFS Evaluation

We have conducted a complete evaluation of BunnyFS, but include only the major results here.

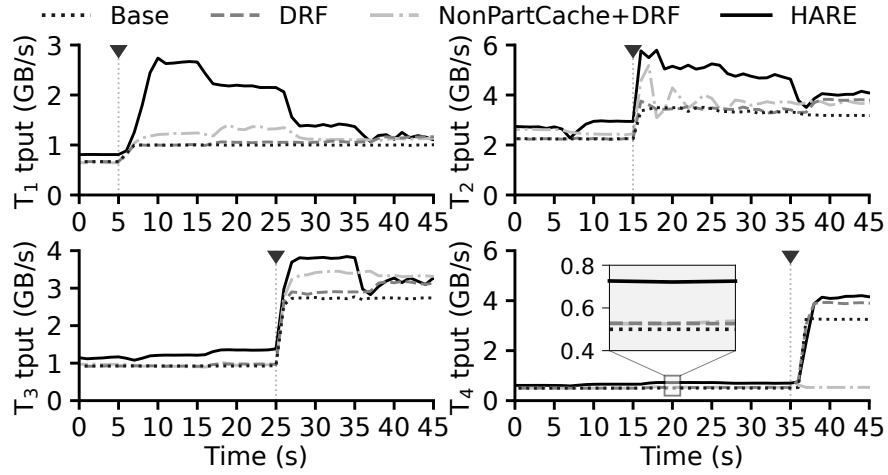
**Experiment Settings.** We show two sets of experiments to evaluate BunnyFS: the *scaling experiment* (§2.4.2.1) verifies that BunnyFS is scalable with 32 tenants; the *dynamic experiment* (§2.4.2.2) shows BunnyFS can adapt to workload changes.

We integrated three alternative policies into BunnyFS for comparison: the equal partitioning baseline, DRF with equal cache partitions (labeled as “DRF”), and DRF with a shared, non-partitioned cache (labeled as “NonPartCache+DRF”). As seen in §2.3.4, NonPart is not fair; NonPartCache+DRF is a middle ground between DRF and NonPart, where I/O bandwidth and CPU cycles are carefully allocated by DRF, but cache management is delegated to a global LRU pool.

In all experiments, BunnyFS manages a total of 2 GB page cache, four vCPUs for worker threads, and 2 GB/s I/O bandwidth, unless otherwise specified. To stress the filesystem, each tenant contains four threads submitting requests. Experiments are run on a machine with an Intel Xeon Gold 5218R 2.9 GHz processor, 64 GB memory, and an Intel Optane P4800X Series 375 GB SSD.

### 2.4.2.1 BunnyFS Scaling Experiment

We show that BunnyFS can manage workloads containing 32 tenants. There are four groups of tenants:  $T_1$ - $T_8$  read data with uniform distribution;  $T_9$ - $T_{16}$  read data with Zipfian distribution ( $\theta = 0.99$ );  $T_{17}$ - $T_{24}$  read data with less-skewed Zipfian distribution ( $\theta = 0.5$ );  $T_{25}$ - $T_{32}$  read data sequentially. Within each group, the working sets of tenants are 64 MB, 128 MB, 192 MB, ..., and 512 MB, respectively.



**Figure 2.13: BunnyFS Dynamic Experiment Results.**

Figure 2.12 compares the normalized throughput of four policies. As expected, both HARE and pure DRF guarantee the sharing incentive, but while DRF achieves only 10% improvement, HARE improves most tenants' throughput by 40%. HARE also outperforms NonPartCache+DRF in efficiency: HARE recognizes that the sequential workloads are uncacheable and does not allocate cache to them, while NonPartCache+DRF allows those tenants to continuously add low-utility data to the cache, evicting other tenants' data.

Importantly, NonPartCache+DRF violates the fairness goal: three tenants with low miss ratios (T<sub>1</sub>, T<sub>2</sub>, and T<sub>17</sub> with 1%, 12%, and 3%, respectively) experience a dramatic throughput drop compared to the baseline. This unfairness occurs because a tenant's demand vector depends on the miss ratios and is subject to other tenants' interference. These low-miss-ratio tenants are allocated little I/O bandwidth; after DRF allocation, some other tenants have their throughput increased and access the shared LRU more frequently, making their data appear hotter; those low-miss-ratio tenants consequently see higher miss ratios and suffer from insufficient I/O bandwidth. Moreover, once a tenant's throughput drops, it accesses the shared LRU cache less frequently and is more likely to have data evicted; this further exacerbates unfairness and makes it more difficult for the victim to recover. Thus, DRF fundamentally does not interact well with a non-partitioned cache.

### 2.4.2.2 BunnyFS Dynamic Experiment

We evaluate how well BunnyFS can handle tenants whose workloads change over time. Our workload contains four tenants whose overall trend is to become less I/O-bound over time:  $T_1$  performs uniform random reads over a working set that begins with 2 GB and shrinks to 1 GB at the 5<sup>th</sup> second;  $T_2$  reads with Zipfian distribution ( $\theta = 0.99$ ) beginning with 2 GB and shrinking to 1 GB at the 15<sup>th</sup> second;  $T_3$  reads a 1.5 GB working set following a Zipfian distribution, beginning with  $\theta = 0.5$  and turning more skewed to  $\theta = 0.99$  at the 25<sup>th</sup> second;  $T_4$  reads data sequentially beginning with 2 GB and shrinking to 0.5 GB at the 35<sup>th</sup> second. We set the allocation frequency in BunnyFS to be every one second.

Figure 2.13 compares the four tenants' throughput with the four allocation approaches over time. HARE consistently outperforms the other two fair alternatives (baseline and pure DRF). DRF does not achieve much improvement over the baseline in the first 35 seconds, because all tenants are I/O-bound; DRF only improves throughput when there is a mix of CPU- and I/O-bound tenants. NonPartCache+DRF violates the sharing incentive goal, performing worse than the baseline after the 35<sup>th</sup> second: at this point, the working set of  $T_4$  can fit into the baseline cache and therefore should have high throughput; however, with a non-partitioned cache,  $T_4$ 's data is evicted by others before being read again and its throughput remains the same as with 100% misses.

## 2.5 Conclusion

In this chapter, we study a critical use case of memory-centric storage systems: caches. We present HARE, a cache-centric multi-resource allocation algorithm. Using a novel two-phase harvest/redistribute approach, HARE holistically allocates cache space and other resources to fully exploit heterogeneous cache sensitivity across tenants, maximizing each tenant's throughput while maintaining fairness. Although motivated by memory-centric storage, HARE is a general algorithm applicable to many other caching systems.

We build HopperKV, a cloud-native key-value store, and BunnyFS, a high-performance local filesystem for NVMe SSDs. Both systems require no prior knowledge of tenants' demands; they can efficiently learn the demands and adapt to dynamic workloads on the fly. Evaluations show that HopperKV achieves up to 1.9× higher throughput, and BunnyFS achieves up to 1.4×.

More broadly, this work highlights the importance of treating caches as integral components of multi-resource allocation, rather than as isolated subsystems. As data systems grow increasingly complex, spanning both hardware resources (e.g., network bandwidth) and software resources (e.g., backend read units), there is a growing need for a unified framework that captures demand correlations and allocates all resources holistically. HARE represents a practical step toward such a vision, demonstrating how cache-centric allocation can effectively account for these interactions and improve both efficiency and fairness in modern storage systems.

## Chapter 3

# MultiLog: Scalability and Consistency Under Replication

In the previous chapter, we examined how memory-centric storage manages resources within a single node. Even with ideal resource management, one node can serve only as many reads as its cores and network bandwidth allow. In this chapter, we turn to how memory-centric storage scales beyond a single node.

Modern memory-centric stores [20, 31, 87, 105, 111] rely on replication to scale out read-dominant workloads. A primary instance is paired with one or more read-only replicas; the primary serializes every mutation into a write-ahead log, which is continuously shipped to each replica. Each replica replays the log to maintain a state that catches up with the primary.

However, the conventional single-log design introduces two serialization bottlenecks. At the primary, concurrent worker threads must coordinate on the log tail to allocate space, capping append throughput. At the replica, a single replay thread cannot match the primary's concurrent write rate; once replay can no longer keep up with the write rate, replication lag grows without bound and destabilizes the system.

Existing approaches force a trade-off. Single-log designs in Redis [87], Valkey [111], KeyDB [105], and earlier versions of Garnet [20] guarantee prefix consistency but fail to scale. Naive sharding in Dragonfly [31] replicates each shard independently and thus scales across cores, but a client reading across shards may observe time-travel anomalies. Parallel-replay protocols built for relational databases [46, 118, 127]

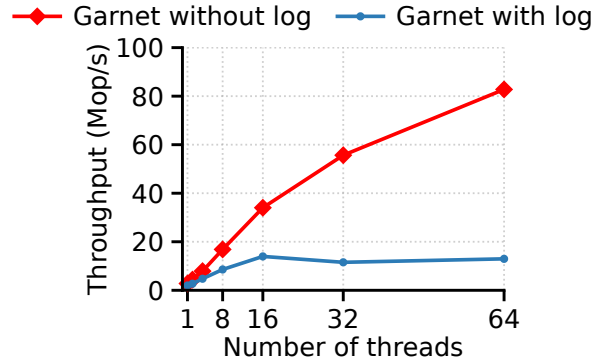
preserve consistency via MVCC or periodic snapshotting, neither of which fits the single-version design of memory-centric storage.

This chapter presents PrefixLoom, a lightweight replica-side protocol that serves prefix-consistent reads during parallel replay, and then MultiLog, a highly scalable replication architecture for memory-centric storage that resolves this trade-off. Unlike prior approaches relying on MVCC or snapshotting, PrefixLoom uses *lazy coordination* between replay and read threads to weave a prefix-consistent view out of a continuously updated, never-consistent replica image. Building on PrefixLoom, MultiLog further shards the log into multiple physical sublogs, each owning a disjoint keyspace and maintaining its own log-space allocator and replication stream; each append thread routes its own record to the corresponding physical sublog based on the key hash. Implemented in Garnet [20], MultiLog improves append throughput by up to 19× and replay throughput by up to 36× over a single-log baseline. Under a write-heavy replication workload, MultiLog sustains 3.8× higher primary throughput than the single-log baseline and 2× higher replica read throughput than C5 [46]. Its primary-replica lag stays within ~10 ms at p99.9, two orders of magnitude fresher than either baseline.

The rest of the chapter is structured as follows. §3.1 motivates the problem in detail. §3.2 describes the PrefixLoom read protocol. §3.3 presents the full MultiLog architecture. §3.4 evaluates the resulting system. §3.5 concludes the chapter.

### 3.1 Background and Motivation

Replication is critical for modern memory-centric stores [20, 31, 87, 105, 111] to scale beyond a single node. In a replicated architecture, a primary instance is paired with one or more read-only replicas. The primary serves read and write requests while serializing every mutation into a write-ahead log. This log is continuously shipped to all replicas, and each replica replays the log to update its local state. For high performance, replication in memory-centric stores is typically *asynchronous* [20, 31, 87, 105, 111]: the replica may serve data with minor staleness that reflects a prefix of writes on the primary. In this chapter, we focus on logical logging, the dominant logging scheme used by memory-centric stores [20, 31, 87, 105, 111], as its compact



**Figure 3.1: Garnet Throughput as the Thread Count Scales.** SET-only workload, uniform random, 8-byte keys and 8-byte values.

log records substantially reduce storage I/O and network cost compared to physical logging.

In what follows, we motivate our work by showing how the conventional single-log design becomes a scalability bottleneck on modern hardware (§3.1.1). We then elaborate on the four goals that a memory-centric storage replication system must satisfy (§3.1.2) and discuss why existing approaches are insufficient (§3.1.3).

### 3.1.1 Single-Log Scalability Bottleneck

A replicated memory-centric store that funnels all writes through one log introduces two serialization points that prevent it from scaling with CPU cores on modern hardware.

**Primary: Concurrent Appends Contend for the Log Tail.** Modern memory-centric stores utilize dozens of worker threads to sustain high throughput. However, this multi-threaded scalability collapses once logging is enabled. With logging, worker threads must coordinate to allocate space in the log, which typically translates to atomic fetch-and-add (FAA) or compare-and-swap (CAS) instructions on the log tail offset to serialize all appends. Such atomic instructions are expensive when contended across dozens of cores. Figure 3.1 compares Garnet [20] scalability with and without logging. Without logging, Garnet scales to 83 Mop/s on 64 threads. When logging is enabled, however, the system bottlenecks on log-tail contention. The throughput plateaus after 16 threads and is capped at 14 Mop/s, creating a 6× gap at 64 threads.

**Replica: Sequential Replay Fails to Catch Up.** A replica must apply log records as fast as the primary produces them. However, log replay is often sequential because later records may depend on earlier ones. As a result, the replica is limited by a single replay thread, which may not match the primary’s concurrent write rate. For example, our measurements show that in Garnet, a single thread can replay only  $\sim 2$  Mop/s, which is significantly lower than the 14 Mop/s write rate shown in Figure 3.1. Once the primary’s write rate exceeds a single replay thread’s capacity, replication lag grows without bound, potentially destabilizing the system and causing data loss. For instance, during GitLab’s 2017 outage, a load spike pushed the PostgreSQL replica so far behind that the replication pipeline broke; the manual rebuild cost 18 hours of downtime and 6 hours of permanent data loss [40].

Moreover, modern hardware makes the single-log bottleneck increasingly problematic for memory-centric stores. First, faster storage and networks expose single-log software overhead that was previously masked by hardware limitations. Second, in-memory replication [20] skips persistence, reducing replication latency and disk overhead while further shifting the bottleneck up the stack.

### 3.1.2 System Goals for Memory-Centric Storage Replication

A replication system for memory-centric stores must achieve four goals: scalability, consistency, memory efficiency, and data freshness.

- **G1: Scalable Multi-Threaded (a) Append and (b) Replay.** The replication pipeline must scale with thread count on both append and replay without a serial bottleneck. Modern memory-centric stores provision dozens of worker threads and target tens of millions of operations per second on commodity servers [20], demanding that log replication meet such throughput goals. On the primary, log append must sustain the full write rate from the workers. More critically, on the replica, replay must keep up with the primary’s aggregate write rate, since a throughput mismatch accumulates as unbounded replica lag.
- **G2: Prefix-Consistent Reads at Replicas.** Reads from a replica should expose a *monotonically progressing* view of the primary’s state. Formally, a read at the replica should reflect the primary’s state at some logical timestamp  $t_1$ , and each

subsequent read in the same client session should correspond to a timestamp  $t_2 \geq t_1$ . This guarantee simplifies applications' reasoning about time and invariants. For example, a web app may write a user's metadata before adding the user's ID to an active pool, with an invariant that every user in the pool has valid metadata written. Without prefix consistency, a reader on a replica may see a dangling user in the pool without valid metadata, causing confusion or crashes. This intuitive consistency guarantee is widely supported by memory-centric stores [20, 87, 105, 111] and databases [12, 50, 62, 70, 74, 109, 112].

- **G3: Single-Version Store for Memory Efficiency.** Memory-centric stores typically keep only a single version in memory and apply mutations in place. First, memory capacity is a dominant constraint for memory-centric stores, because they exist to keep hot data resident in DRAM and to serve it within microsecond latency. Retaining multiple versions of the data amplifies the memory footprint, which may trigger earlier cache eviction [88], overflow to disk [19, 20], or out-of-memory crashes [89]. Second, applying in-place mutation enables efficient space reuse, which not only preserves CPU cache locality but also avoids expensive memory allocation overhead [19, 20]. In contrast, keeping old versions prevents space reuse and loses the benefits of in-place mutation.
- **G4: High Data Freshness at Replicas.** Though staleness is unavoidable in asynchronous replication, replica reads should return data as fresh (i.e., close to the primary's latest state) as possible. Freshness is critical because many applications rely on the assumption that staleness is low enough that reading from replicas is acceptable. For example, an application may write a session token to the primary; downstream services reading from the replicas cannot proceed until they see it. Additional lag may add to the application's end-to-end latency and hurt the user experience.

### 3.1.3 Why Existing Approaches Fall Short

Unfortunately, no existing system satisfies all four goals.

**Existing Memory-Centric Stores Trade Off Scalability (G1) Against Consistency (G2).** Redis [87], Valkey [111], KeyDB [105], and earlier versions of Garnet [20] funnel

all writes through a single replication log, and the replica replays the log sequentially. This design keeps consistency simple: because the log establishes a total order of mutations, a continuous, sequential replay always keeps the replica in a prefix state that progresses monotonically. However, as discussed in §3.1.1, this single-log design becomes a scalability bottleneck. Such a limitation is acceptable for Redis [87] because Redis is single-threaded by design and therefore does not saturate the log. For highly scalable systems such as Garnet [20], this bottleneck negates their concurrency gains.

At the other end of the design spectrum, Dragonfly [31] shards the keyspace across threads, where each shard at the primary has an independent replication stream to a corresponding thread at the replica. This design scales performance at the cost of consistency: reading keys from different shards does not guarantee prefix consistency, and the client may observe time-travel anomalies. For example, consider an app that always writes a user’s metadata before adding the user’s ID to the active pool; if the pool and the user’s metadata are assigned to different shards, a read from a replica may see the user’s ID without the valid metadata available. Such anomalies hurt the overall usability of the system.

A similar trade-off also appears in relational databases. The parallel replay modes of PostgreSQL [74], MySQL [70], and MariaDB [62] serialize the commit phase by default, limiting scalability. MySQL [70] and MariaDB [62] offer out-of-order commit options that trade prefix consistency for better scalability.

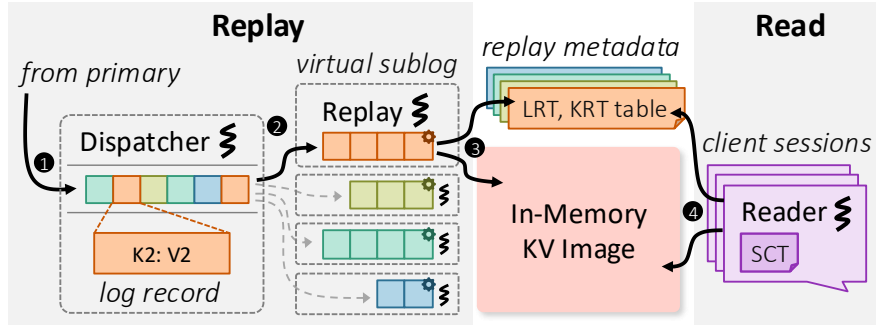
**Prior Scalable, Consistent Replay Fails on Single-Version (G3) or Freshness (G4) Goals.** Prior efforts on scaling replication focus on the replay path for relational databases, typically requiring MVCC [118, 127] and/or snapshotting [46] for consistent reads. For relational databases with orders-of-magnitude lower throughput, append contention is a less prominent concern than in memory-centric stores.

KuaFu [127] and Query Fresh [118] parallelize replay and require MVCC to serve prefix-consistent reads. The database stores multiple versions of each record; multiple replay threads concurrently create fuzzy new versions, and clients read stale, stabilized versions for prefix consistency. While this achieves scalability and prefix consistency, its multi-version design is ill-suited for memory-centric stores. For memory-centric stores, memory is a key constraint, and in-place mutation is required for high performance.

C5 [46] is a state-of-the-art parallel replay protocol with prefix consistency. Although implemented on MVCC engines, C5 can, in principle, be applied to single-version stores by taking periodic snapshots to serve prefix-consistent reads. However, snapshotting is too costly for memory-centric stores for two reasons. First, capturing a snapshot requires stalling replay threads to ensure that every mutation before a cut has been fully applied and none after; replay does not resume until the snapshot is completed. Second, keeping a snapshot requires tracking and maintaining captured stale versions, which still increases memory footprint and blocks in-place mutation. Frequent snapshots amplify these costs, whereas infrequent snapshots degrade data freshness: even after a mutation has been applied, it remains invisible to clients until the next snapshot.

**Why Is It Challenging?** A single-version memory-centric store faces an inherent tension between scalable replay and prefix-consistent reads. Scalable replay requires minimizing coordination across replay threads, ideally letting each thread work independently on a disjoint partition of the keyspace. Without coordination, threads progress at different rates, and the replica’s in-memory state mixes records replayed to different timestamps. The resulting image never corresponds to a single timestamp at the primary.

To resolve this tension, prior work [46, 118, 127] relies on capturing a *physically consistent image* during replay, in which every key’s value reflects a single timestamp  $T$ , and serves reads from it. Once an image is captured, reading any keys from it is automatically consistent, and monotonic progress reduces to capturing another image. MVCC and snapshotting are two implementations of this idea: MVCC retains stale versions so that a per-timestamp view can be assembled on demand; periodic snapshotting stalls all replay threads with a global barrier to capture one explicitly. However, capturing such a physically consistent image is too costly for performance-critical memory-centric stores, which typically operate on a single-version, in-place-updated image.



**Figure 3.2: Overview of the PrefixLoom Protocol on a Replica.** ① The replica receives log records streamed from the primary. ② The dispatcher thread routes log records to replay threads based on key hashes. ③ Each replay thread applies the records to the in-memory KV image and updates replay metadata (§3.2.2). ④ Reader threads query metadata and serve prefix-consistent reads from the KV image (§3.2.3).

## 3.2 PrefixLoom: Prefix-Consistent Reads During Parallel Replay

We present *PrefixLoom*, a lightweight replica-side protocol that serves prefix-consistent reads during parallel replay. Unlike prior approaches [46, 118, 127], PrefixLoom does not rely on MVCC or snapshotting to capture a physically consistent image. Instead, it uses *lazy coordination* between replay and reads to weave a prefix-consistent view from a continuously updated replica image that may never be physically consistent. Parallel replay therefore proceeds without cross-thread coordination, while reads block only when they might otherwise violate prefix consistency.

In what follows, we first present an overview of PrefixLoom (§3.2.1), then describe the replay protocol (§3.2.2) and the read protocol (§3.2.3). We then illustrate the protocol with an example (§3.2.4) and discuss its scalability limitation (§3.2.5).

### 3.2.1 PrefixLoom Overview

We illustrate PrefixLoom by first elaborating on the assumed system model and then describing the newly introduced components.

**System Model.** We model a memory-centric store as two logical subsystems: an in-memory *key-value (KV) image* and a *log*. The KV image maintains a single version of the value for each key and supports linearizable point reads and writes. The log

consists of a sequence of records, each of which contains a key and a mutation applied to the corresponding value, such as an upsert, read-modify-write, or delete. The order of log records reflects the serial execution order of mutations at the primary.

Because a record’s position in the log induces a logical notion of time, we use the log offset of a record as its *logical timestamp*: replaying a prefix of the log up to the offset  $T$  restores the replica’s KV image to the state of the primary at logical time  $T$ . Since each log offset uniquely identifies a record, we assume throughout this section that record timestamps are unique.

**Architecture.** PrefixLoom parallelizes replay by hash-partitioning the log into *virtual sublogs*, each of which owns a disjoint keyspace (Figure 3.2). Because each key’s mutation history is serialized exclusively within one sublog, each sublog can be replayed independently to restore values without cross-sublog coordination. Prefix consistency across sublogs is restored lazily at read time. PrefixLoom introduces three classes of threads: a single *dispatcher thread*, per-sublog *replay threads*, and per-client-session *reader threads*.

The dispatcher thread receives a continuous log stream from the primary. It scans incoming log records and routes each record to a sublog based on its key hash.

Each virtual sublog is implemented as a queue owned by a replay thread. The replay thread dequeues log records and applies them to the KV image in order. While applying records, the replay thread also maintains lightweight metadata that captures a logical-time validity window for each key (more details in §3.2.2). Throughout the process, replay threads do not coordinate with each other, allowing replay to proceed at full speed.

Each client connection to the replica is managed as a stateful *session*, served by a dedicated reader thread. The reader thread consults the per-sublog metadata to keep the session’s reads on a monotonically advancing prefix of the primary’s timeline (more details in §3.2.3). A reader waits only when it detects a potential consistency violation.

### 3.2.2 Replay Protocol

PrefixLoom requires only minor changes to single-log replay, as shown in Algorithm 1. The replay thread sequentially iterates through the records in its sublog. For

---

**Algorithm 1:** Replay Protocol on Sublog  $L$ .

---

**Data:** sublog  $L$  with the *log replayed timestamp* ( $LRT$ ) and *key replayed timestamp table* ( $KRT\_table$ )

**Input:** a log record with key  $K$  and logical timestamp  $T$

- 1  $L.KRT\_table[K] \leftarrow T$ ;
  - 2 apply the record to the KV image;
  - 3  $L.LRT \leftarrow T$ ;
- 

each record, the replay thread applies the mutation and maintains two watermark timestamps: the *Log Replayed Timestamp* ( $LRT$ ) and the *Key Replayed Timestamp* ( $KRT$ ).  $LRT$  and  $KRT$  denote the maximum timestamp of the records that have been applied to the sublog and to the key, respectively; both are initialized to zero. The  $KRT$  of every key in a sublog is compactly stored in a  $KRT$  table.

The two watermarks bound the *validity window* of every key-value pair in logical time. For every key  $K$  and its current value  $V$  in the KV image:

- Every record with timestamp  $T \leq LRT$  has been applied to the KV image.  $\Rightarrow$  The current value  $V$  is valid at least until  $LRT$  (i.e., the next mutation must be after  $LRT$ ).
- The key-value pair was last created/modified at  $KRT$ .  $\Rightarrow$  The current value  $V$  has been valid since  $KRT$ .

Together, they enforce the following invariant: For every key  $K$ , the current value  $V$  in the KV image is valid within the logical-time window  $[KRT, LRT]$ . The bounds are inclusive because we use the log offset as the logical timestamp, which is unique for each record.

Note that, with proper operation ordering, advancing the two watermarks and applying the mutation require no lock for atomicity. Since  $KRT/LRT$  represents a lower/upper bound, respectively, the validity window can be safely tightened by allowing  $KRT/LRT$  to be conservatively larger/smaller than their exact values. Therefore, as shown in Algorithm 1,  $KRT$  is updated before applying the mutation, while  $LRT$  is updated afterward.

---

**Algorithm 2:** Prefix-Consistent Read of Key  $K$  at Session  $S$ .

---

**Data:** session  $S$  with *session current timestamp* ( $SCT$ ) and the sublog read by the last request ( $L_{prev}$ )

**Input:** key  $K$

**Output:** value  $V$

// check if the read falls within the validity window

```

1  $L \leftarrow$  get sublog of  $K$ ;
2 if  $L \neq L_{prev}$  then
3   | if  $L.LRT < S.SCT$  then
4   |   | wait for more replay until  $L.LRT \geq S.SCT$ ;
// consistency check passed; now safe to read
5  $V \leftarrow$  read  $K$  from the KV image;
// update the session states
6  $KRT \leftarrow L.KRT\_table[K]$ ;
7  $S.SCT \leftarrow \max(S.SCT, KRT)$ ;
8  $S.L_{prev} \leftarrow L$ ;
9 return  $V$ ;
```

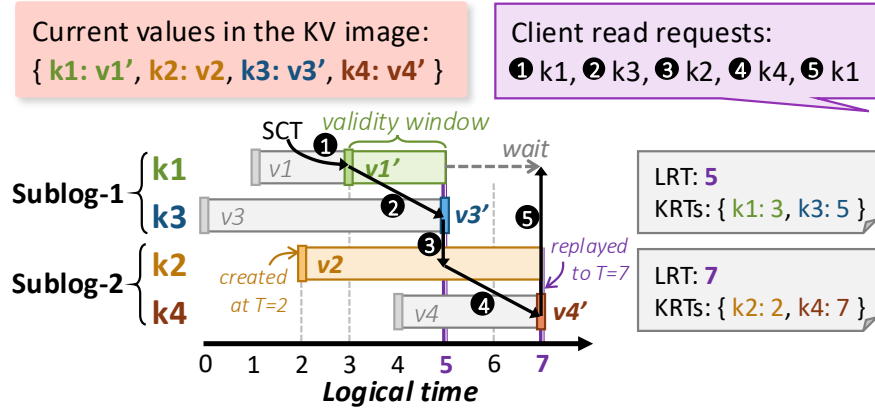
---

### 3.2.3 Prefix-Consistent Read Protocol

A replica serves prefix-consistent reads using the protocol shown in Algorithm 2. Each client connection is managed as a stateful session that tracks the logical timestamp at which reads are executed. PrefixLoom enforces that this timestamp falls within the validity window (defined in §3.2.2) and advances monotonically to ensure prefix consistency.

Each session maintains two pieces of metadata: the *Session Current Timestamp* ( $SCT$ ), the timestamp at which the session executes its reads, and  $L_{prev}$ , the sublog touched by its previous read request.  $SCT$  is initialized to zero, and  $L_{prev}$  to a null sentinel.

When the current key maps to the same sublog as the previous read, the session takes a fast path by skipping the consistency check. This is safe because each sublog's records are applied sequentially by a single replay thread, so consecutive reads on the same sublog always observe a monotonically growing prefix and are prefix-consistent by construction.



**Figure 3.3: PrefixLoom Read Example With a Visualized Logical Timeline.** A client session issues five reads. Requests ①–④ execute without blocking. For request ⑤, since it is unclear whether a new version of  $k1$  was created during the logical-time interval  $(5, 7)$ , the session waits for sublog-1.

Otherwise, the session must check whether the read can be executed within the key’s validity window. The check may fail if the key’s sublog has not yet replayed enough records ( $LRT < SCT$ ). In that case, a mutation with timestamp  $T \in (LRT, SCT)$  may not yet have been applied to the KV image; executing the read now could therefore observe a stale value that violates consistency. As a result, the reader thread must wait until additional records have been replayed before proceeding.

After completing the read, the session updates  $SCT$ , which must be monotonically non-decreasing to preserve prefix consistency. Although any timestamp within  $[\max(KRT, SCT), LRT]$  is valid for executing the read, choosing a smaller value for  $SCT$  provides greater flexibility for future reads. Therefore,  $SCT$  is updated to the lower bound  $\max(KRT, SCT)$ ;  $L_{prev}$  is updated accordingly to remember the sublog touched by this read.

The protocol enforces prefix consistency at the session granularity by default. This fine-grained management enables more permissive read execution without blocking. The protocol can be generalized for global prefix consistency by simply sharing a common  $SCT$  across all clients.

### 3.2.4 PrefixLoom Example

Figure 3.3 illustrates a client session executing the PrefixLoom read protocol. In this example, sublog-1 has been replayed up to logical timestamp 5, and sublog-2 up to 7. The session starts with  $SCT = 0$ .

❶ The session first reads  $k_1$  and advances  $SCT$  to  $k_1$ 's  $KRT = 3$ . ❷ It then reads  $k_3$ , whose validity window is  $[5, 5]$ , and advances  $SCT$  to 5. ❸ Next, it reads  $k_2$ , whose validity window is  $[2, 7]$ . Since the current  $SCT$  already falls within this validity window,  $SCT$  remains unchanged. ❹ The session then reads  $k_4$  and advances  $SCT$  to 7. All of these reads are non-blocking, and  $SCT$  grows monotonically to ensure prefix consistency.

❺ Finally, the session attempts to read  $k_1$  again. However,  $SCT$  has advanced to 7, while sublog-1 has only been replayed up to timestamp 5. Thus, it is unclear whether another version of  $k_1$  was created during the logical-time interval  $(5, 7)$ . To avoid a prefix-consistency violation, PrefixLoom requires the session to wait until additional records from sublog-1 have been replayed.

The example above assumes that no replay progress is made during the sequence of reads. In practice, replay threads progress concurrently with client reads. Therefore, by the time the fifth request is executed, sublog-1 may have already been replayed beyond timestamp 5, further reducing the wait time.

### 3.2.5 Scalability Limitation

While PrefixLoom demonstrates how replay can be parallelized, it does not fully address the scalability goals. On the primary side, PrefixLoom does not improve append scalability (G1a): all log appends still contend at the log tail, limiting append throughput under high concurrency. On the replica side, replay scalability (G1b) is constrained by the centralized dispatcher. Although dispatching is lightweight compared with replay execution, every replay task must first pass through the dispatcher. As the number of replay threads increases, this single dispatcher eventually becomes overloaded. Consequently, end-to-end replay throughput remains capped by the capacity of a single dispatcher thread.

### 3.3 MultiLog: Bottleneck-Free Scaling

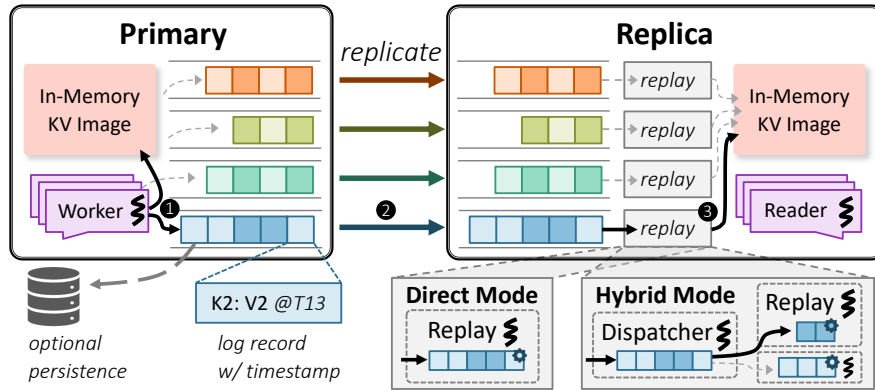
We now present MultiLog, a scalable replication architecture that builds on PrefixLoom to remove the append and dispatcher bottlenecks identified in §3.2.5. The core idea is *physical sharding*: the log is partitioned into multiple *physical sublogs*, each owning a disjoint keyspace by key hash. Unlike virtual sublogs, each physical sublog maintains its own log-space allocator and replication stream. Append threads route records directly to physical sublogs, alleviating append-tail contention and eliminating centralized dispatch.

In MultiLog, log records can either be kept volatile (i.e., purely in memory) or persisted to storage, depending on the purpose they serve. In this work, we focus on *replication*, in which the primary can keep log records in memory for performance and discard them once replication completes. MultiLog can also support *recovery* if records are persisted to durable storage. Upon failure, a restarted memory-centric store instance replays the durable logs to recover its KV image state.

Figure 3.4 illustrates the overall architecture of MultiLog. In what follows, we describe the design on the primary (§3.3.1) and on replicas (§3.3.2). We then discuss several challenges that MultiLog must address. First, replay threads on different sublogs may progress at uneven rates, increasing the wait time of cross-sublog reads; we bound this divergence with a barrier mechanism (§3.3.3). Second, a sublog may remain idle for extended periods, stalling reads that depend on it; we preserve liveness with periodic no-op records (§3.3.4). Finally, we describe how MultiLog supports recovery (§3.3.5) and present a memory-saving optimization (§3.3.6).

#### 3.3.1 MultiLog on the Primary

MultiLog exposes the abstraction of a single logical log; internally, it shards the logical log into  $m$  independent *physical sublogs* (Figure 3.4). On the primary, each client session is served by a memory-centric store worker thread for reads and writes. For each write operation, the worker thread calls into MultiLog to append a log record. MultiLog computes the hash of the key to determine the target physical sublog and appends the record accordingly. Each physical sublog asynchronously ships new records to replicas.



**Figure 3.4: MultiLog Architecture.** ❶ A worker thread serves read/write requests from clients and appends a log record for each write; the record is routed to a physical sublog based on the key hash. ❷ The log record is shipped from the primary to the replica. ❸ At the replica, the log record is replayed in either direct replay mode or hybrid replay mode. On both the primary and the replica, the log can be kept only in memory for performance or optionally persisted for recovery.

The key challenge introduced by physical log sharding is how to preserve the logical timeline required by PrefixLoom across multiple physical sublogs. In the single-log setting of §3.2, log offsets directly provide logical timestamps. With physical log sharding, however, the  $m$  independent offset spaces do not naturally compose into a single timeline: an offset in one sublog is not comparable to an offset in another.

To preserve such a timeline, MultiLog explicitly acquires a logical timestamp at append time and attaches it to the log record. Timestamp acquisition must be both lightweight and scalable, because it is on the critical path for every record append, and many worker threads append concurrently. A naive shared fetch-and-add counter would recreate the serial bottleneck because of its atomic instruction. We present two implementations.

- *TSC Register.* Modern x86 CPUs commonly provide a timestamp counter (TSC) register synchronized across cores. A worker obtains a timestamp using a single `rdtsc` instruction without cross-thread synchronization.
- *Batched Counter.* Without a synchronized hardware counter, MultiLog can use an epoch-like counter that is read on each append and advanced every few

microseconds. It batches records under the same timestamp, amortizing the update cost.

The read path handles non-unique timestamps as described in §3.3.2.

Note that the acquired timestamps must order log records consistently with the serial execution order at the primary, as determined by its concurrency-control protocol. In practice, this requirement is usually satisfied naturally because the log append is a valid serialization point. For example, under two-phase locking, locks are released only after the log record is appended, so the timestamp acquired during log append reflects the serial execution order.

### 3.3.2 MultiLog on the Replica

The replica receives log records over the network and applies them following PrefixLoom. In what follows, we describe MultiLog’s modifications to the replay and read paths.

**Replay.** MultiLog supports two replay modes: *direct* and *hybrid*, as shown on the right side of Figure 3.4. In *direct* replay, each physical sublog maps to one virtual sublog in PrefixLoom. Each physical sublog is therefore scanned and replayed directly by a replay thread, without a dispatcher.

In *hybrid* replay, each physical sublog is further partitioned into  $n$  virtual sublogs, resulting in a total of  $m \times n$  virtual sublogs in PrefixLoom. For each physical sublog, a dispatcher routes records to  $n$  replay threads, as in §3.2.2. Unlike the single dispatcher in §3.2, MultiLog uses  $m$  dispatchers to share the routing load, each handling only  $1/m$  of the total traffic and thereby improving scalability.

The two replay modes provide different ways to compose replay parallelism. For example, given a budget of 16 replay threads, direct replay uses  $m = 16$  physical sublogs, whereas hybrid replay can use  $m = 4$  physical sublogs, each fanning out to  $n = 4$  virtual sublogs.

These two modes present a trade-off between efficiency and flexibility. Direct replay avoids dispatcher overhead and therefore provides higher performance. However, the number of physical sublogs  $m$  is a contract between the primary and all replicas and is difficult to change at runtime. Hybrid replay instead lets each

replica adjust  $n$  independently in response to load. We evaluate and compare the performance of these two modes in §3.4.3.2.

**Read.** Unlike PrefixLoom’s log offsets (§3.2.1), MultiLog’s timestamps (§3.3.1) are not necessarily unique across records. When log records in different sublogs share a timestamp, their relative order is ambiguous. The read protocol therefore tightens the validity window to the open interval  $(KRT, LRT)$ , excluding both bounds, and modifies the consistency check in Algorithm 2 (lines 3–4) to require  $LRT$  strictly greater than  $SCT$ .

### 3.3.3 Bounded Replay Progress Divergence

The PrefixLoom protocol has a nice performance property: although consistency is enforced at read time, readers are not directly penalized by fast replay. Their wait time depends on the *difference* in replay progress across sublogs, rather than on the absolute replay throughput. If all virtual sublogs advance their logical time at the same high rate, all  $LRT$  values advance together and remain no smaller than any  $KRT$  value and, by induction, any  $SCT$  value. As a result, readers rarely encounter waits.

In real systems, however, replay progress is often uneven across sublogs. Even under uniformly distributed writes, replay threads may run at slightly different rates. Such minor progress divergence can accumulate over time and eventually cause long reader waits and hurt tail latency. This divergence can grow even faster under skewed writes, where some sublogs have more replay work to process than others.

MultiLog introduces the *replay barrier* mechanism to bound such progress divergence. A replay barrier is activated when a replay or reader thread observes that the difference between the maximum and minimum  $LRT$  values across sublogs exceeds a predefined threshold. The activated barrier is stamped with the maximum  $LRT$ , denoted  $T_b$ . Each replay thread periodically checks the barrier and pauses once  $LRT \geq T_b$ . Once all replay threads have arrived, they resume together, with their  $LRT$  values roughly aligned.

With replay barriers, each sublog’s replay throughput is bounded by the slowest sublog. With hash partitioning, each sublog is usually assigned some hot keys, so load is relatively evenly distributed across sublogs, and replay throughput does not degrade substantially. For example, consider a 10M-key working set with a Zipfian

distribution ( $\theta = 0.99$ ). With four sublogs, the most heavily loaded sublog receives 29% of records, only slightly above the ideal 25% with even distribution. Admittedly, extreme skew can impose an upper bound on scaling to more sublogs. For the above Zipfian distribution, the hottest key alone accounts for 5.5% of records. With  $\geq 32$  sublogs, the sublog containing the hottest key becomes overloaded (9.0%, compared to the ideal 3.1%), and each replay thread's throughput is eventually bounded by the capacity to process writes to this single hottest key. In §3.4.4, we evaluate the performance impact of the replay barrier and its threshold.

### 3.3.4 Liveness Upon Quiescence

In practice, a physical sublog may produce no records for an extended period (e.g., no writes to the primary). We call such a sublog *quiescent*. Recall from §3.2.3 that a session blocks whenever its *SCT* outruns the target sublog's *LRT*; on a quiescent sublog, *LRT* never advances, so any dependent session waits indefinitely.

To address this challenge, the primary periodically appends a special no-op record on each physical sublog that has produced no record for a configured timeout. The no-op record carries only a timestamp  $T$  without a key or a value. Unlike regular records, such a record need not be persisted. It signals a promise that the primary will not subsequently produce any record on this sublog with a timestamp below  $T$ . On the replica, replaying a no-op record advances the sublog's *LRT* to  $T$  (in hybrid mode, the dispatcher fans the no-op record out to all  $n$  replay threads). The blocked sessions on these virtual sublogs can then proceed. This *LRT* advance also lets the sublog's replay threads arrive at any active replay barrier stamped with  $T_b \leq T$  (§3.3.3), so a quiescent sublog does not stall a barrier indefinitely.

Upon full-system quiescence, such no-op appends do not need to continue, imposing bounded overhead. When the primary receives no write requests and produces no regular records across all sublogs, a single round of no-op emission suffices to advance every sublog's *LRT* beyond every key's *KRT* in the system. Consequently, no session's *SCT* can exceed any sublog's *LRT*, and none is blocked. The primary therefore need not emit additional no-op records.

### 3.3.5 Recovery

Beyond replication, the log also supports recovery. Upon restart, a memory-centric store reconstructs its in-memory state by replaying recent log records. To reduce recovery time, it may optionally load a checkpoint and skip replaying a prefix of the log. MultiLog adopts a recovery protocol that generalizes standard WAL recovery.

#### 3.3.5.1 Recovery to a Prefix

MultiLog’s recovery protocol reconstructs the in-memory state to a consistent point in time. Physical log sharding complicates standard WAL recovery: physical sublogs may be persisted up to different logical times, and naively replaying each sublog to its durable tail can lead to an inconsistent state.

MultiLog determines the recovery timestamp  $T_r$  as the minimum last-durable record timestamp across all physical sublogs. During recovery, only records with timestamp  $T < T_r$  are replayed, and others are discarded. This recovers the in-memory state to the moment right before  $T_r$ .

If the recovering memory-centric store is a replica, it does not need an explicit recovery phase. Instead, it simply executes the replication replay protocol (§3.2.2) to continuously update its state. Reading the log from local storage is only a performance optimization compared to streaming from the primary.

#### 3.3.5.2 Checkpointing and Log Truncation

Checkpointing reduces recovery time by skipping replay of records already incorporated in a persisted image. For each sublog, the checkpoint also records the offset and timestamp of the last record reflected in the image. Critically, sublogs may be checkpointed at different logical times. A checkpoint becomes usable once  $T_r$  exceeds every sublog’s checkpoint timestamp. Once usable, all records reflected in it can be safely truncated. During recovery, the system loads the image and replays each sublog from its checkpointed offset through the last record with timestamp  $T < T_r$ .

The checkpoint does not persist PrefixLoom state; the  $LRT$  and  $KRT$  are simply initialized to the recovery timestamp  $T_r$ . The value  $T_r$  is a tight bound for  $LRT$  and a conservative estimate for  $KRT$ , which is safe as discussed in §3.2.2.

### 3.3.6 Optimization: KRT Table Compression

Maintaining one *KRT* entry for every key incurs substantial memory overhead. We therefore introduce an optimization that compresses the *KRT* table and bounds its memory footprint. This optimization exploits the slack described in §3.2.2: because *KRT* serves only as a lower bound, replacing it with a conservatively larger value still preserves the validity-window invariant.

MultiLog implements the *KRT* table as a fixed-size array with  $k$  slots indexed by the key’s hash, where  $k$  is a configurable parameter, typically chosen to be much smaller than the keyspace size. When multiple keys map to the same slot, collisions are resolved using max: when a record with timestamp  $T$  is replayed, the corresponding slot is updated as  $KRT_{slot} \leftarrow \max(KRT_{slot}, T)$ .

This optimization bounds the memory footprint at  $O(k)$ , independent of the keyspace size. The parameter  $k$  introduces a trade-off between validity-window tightness and memory efficiency. A smaller  $k$  reduces memory overhead but increases the likelihood of collisions. In turn, more keys may be assigned conservatively larger *KRT* values, increasing the probability that reader threads are blocked. Empirically, a total budget of 256K slots, aggregated across sublogs, delivers good performance with only 2 MB of memory overhead.

## 3.4 Evaluation

Our evaluation aims to answer the following questions: (1) *Can MultiLog scale append throughput?* (2) *Can MultiLog scale replay throughput?* (3) *Can MultiLog deliver low-latency, fresh reads on the replica?* (4) *How does MultiLog perform under realistic, end-to-end replication workloads over the network?*

### 3.4.1 Implementation and Experiment Setup

**Implementation.** We implement MultiLog in Garnet [20], a state-of-the-art multi-threaded memory-centric store. Garnet consists of a hash-indexed, log-structured KV layer and a write-ahead logging layer; we replace the latter with MultiLog. Garnet is a suitable testbed because its multi-threaded KV layer scales to many cores, exposing log replication as the primary bottleneck (as shown in Figure 3.1).

**Baselines.** We compare MultiLog against three baselines, all ported to the Garnet codebase [65] for fair comparison.

- *SingleLog*: Standard single-log replication. On the primary, all writes are appended to a single log. On the replica, the log is replayed sequentially by one replay thread. This architecture is widely adopted in Redis [87], Valkey [111], and KeyDB [105], and was used in Garnet [20] prior to our work.
- *NoPrefix*: A sharded log scheme without prefix-consistency guarantees. The log is partitioned into multiple physical sublogs, each replayed by a dedicated thread. Unlike MultiLog, it does not maintain the data structures required by the PrefixLoom protocol. This baseline serves as a reference for quantifying the overhead introduced by PrefixLoom. Dragonfly [31] adopts a similar design for replication.
- C5 [46]: A state-of-the-art parallel-replay protocol that serves prefix-consistent reads from periodic snapshots. We implement C5 in Garnet using its native incremental-snapshotting support, which creates a snapshot by simply marking a prefix of the log-structured KV layer as immutable [19, 75]. This provides C5 with a favorable setting relative to systems without efficient incremental snapshotting.

Throughout the evaluation,  $\text{MultiLog}(m)$  denotes direct replay with  $m$  physical sublogs, whereas  $\text{MultiLog}(m, n)$  denotes hybrid replay with  $m$  physical sublogs and  $n$  virtual sublogs per physical sublog. Unless otherwise specified, workloads use 8-byte keys, 8-byte values, and a working set of 10 million keys. All experiments use in-memory logging.

**Hardware.** We run experiments on CloudLab r650 instances, each with 144 logical cores across two NUMA nodes, 256 GB DRAM, and a Mellanox ConnectX-6 100 GbE NIC. The machines run Ubuntu 24.04. We pin every Garnet server instance to a single NUMA node.

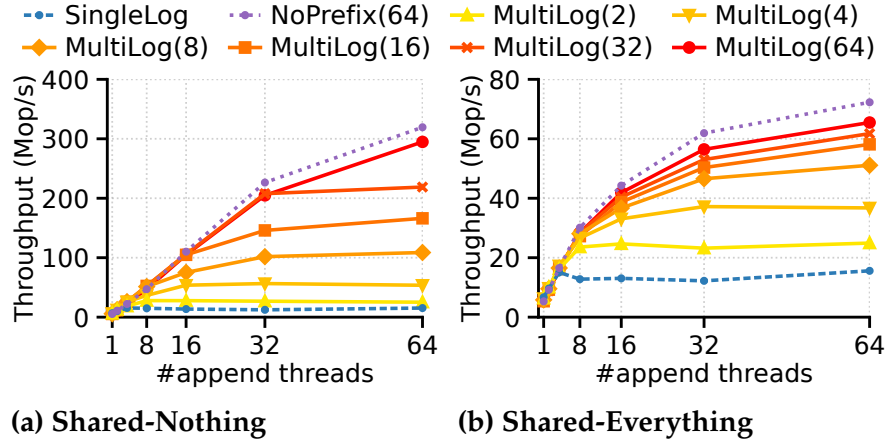


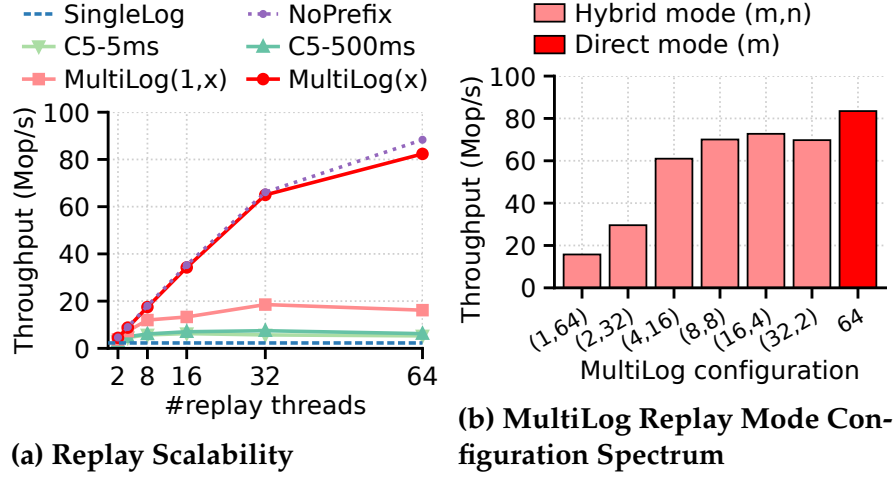
Figure 3.5: Append Throughput as the Thread Count Scales.

### 3.4.2 Append Microbenchmarks

We first evaluate the append scalability. In this microbenchmark, we directly use multiple worker threads to append records to the log concurrently, bypassing the KV layer. We vary the number of appending threads and report throughput. The appended keys follow a uniform random distribution. We compare MultiLog( $m$ ) for  $m \in \{2, 4, 8, 16, 32, 64\}$  against the SingleLog and NoPrefix(64) baselines. We do not include C5 here because C5 only optimizes replay, and its append path is the same as SingleLog’s.

We evaluate two threading architectures common in memory-centric stores: shared-nothing and shared-everything. To emulate logging in the shared-nothing architecture, as used by Dragonfly [31] and Redis cluster mode [87], the keyspace is partitioned among worker threads. We use the same hash for MultiLog sharding and worker keyspace sharding, aligning worker keyspace partitions with physical sublogs and minimizing log-tail contention. To emulate logging in the shared-everything architecture, as used by KeyDB [105] and Garnet [20], every worker can operate on any key.

**Results.** Figure 3.5(a) shows that, in the shared-nothing setting, each MultiLog( $m$ ) configuration scales linearly until the thread count reaches  $m$ . At 64 threads, MultiLog(64) sustains 297 Mop/s, 19 $\times$  higher than SingleLog. When the number of threads is less than or equal to  $m$ , no physical sublog is shared among workers. When the thread count exceeds  $m$ , multiple workers begin to share physical sublogs.



**Figure 3.6: Replay Throughput Over Various Configurations.**

Contention on the log tail then causes sub-linear scaling and eventually a throughput plateau. Larger  $m$  leads to higher saturation throughput. NoPrefix(64) uses similar log sharding but does not maintain metadata for restoring prefix consistency. Its throughput is only 9% higher than MultiLog(64), showing that MultiLog’s per-record timestamp acquisition introduces only modest overhead.

Figure 3.5(b) shows that, in the shared-everything setting, MultiLog(64) sustains 65 Mop/s at 64 threads, 4.1 $\times$  higher than SingleLog. Because every worker may append to any sublog, concurrent appends can contend at individual log tails. More physical sublogs alleviate this contention, but the marginal gain diminishes: with random key assignment, worker threads still collide on the same sublog even with large  $m$ , due to the birthday paradox. NoPrefix(64) delivers slightly higher throughput than MultiLog(64), but the gap is within 12%.

### 3.4.3 Replay Microbenchmarks

We then evaluate MultiLog’s replay performance and scalability in both direct and hybrid modes.

#### 3.4.3.1 Replay Scalability

To stress the replay path, we pre-generate a batch of log records and measure the throughput of applying them to the KV image while varying the replay thread count.

This setup also emulates recovery, where a restarted instance applies a batch of in-memory records loaded from durable storage. We evaluate two representative MultiLog configurations:  $\text{MultiLog}(x)$  and  $\text{MultiLog}(1, x)$ . With  $x$  replay threads,  $\text{MultiLog}(x)$  uses  $x$  physical sublogs with direct replay, whereas  $\text{MultiLog}(1, x)$  uses one physical sublog with hybrid replay plus a dispatcher thread that fans out records to  $x$  virtual sublogs.

We compare MultiLog against SingleLog, NoPrefix, and C5. SingleLog is a single-threaded replay baseline. NoPrefix provides a reference for quantifying the PrefixLoom overhead of maintaining the *LRT* and *KRT* metadata. C5 relies on periodic snapshots to serve prefix-consistent reads, so we evaluate high- and low-frequency snapshotting with intervals of 5 ms and 500 ms, respectively. Replayed keys are drawn uniformly at random. In this experiment, we disable MultiLog’s replay barrier to isolate raw replay performance, thereby establishing an upper bound on its replay throughput.

**Results.** Figure 3.6(a) shows that  $\text{MultiLog}(x)$  scales linearly from 4.4 Mop/s at  $x = 2$  to 65 Mop/s at  $x = 32$ ; it peaks at 83 Mop/s with  $x = 64$  and outperforms SingleLog by 36 $\times$ . NoPrefix(64) reaches 88 Mop/s, within 8% of  $\text{MultiLog}(64)$ , showing that maintaining the *LRT* and *KRT* metadata imposes only a modest cost.

Both  $\text{MultiLog}(1, x)$  and C5 use a single dispatcher thread to fan out log records to replay threads. Consequently, they scale linearly only through  $x = 8$ ; beyond that, the saturated dispatcher caps throughput (§3.2.5). Despite this bottleneck,  $\text{MultiLog}(1, x)$  peaks at 18 Mop/s and outperforms the best C5 configuration by 1.9 $\times$ . C5’s lower performance stems from two sources: snapshotting overhead and the lack of in-place mutation. Garnet’s log-structured KV layer supports efficient incremental snapshotting, making the first source of overhead relatively small. However, each snapshot freezes the existing KV image, so subsequent mutations cannot be in-place. This leads to costly contention on memory allocations when applying log records and enlarges the CPU cache footprint. Lowering the snapshot frequency therefore provides limited improvement because it mitigates snapshotting overhead but only modestly reduces out-of-place mutation costs. For other memory-centric stores that lack efficient incremental snapshotting, we expect C5 to experience greater performance degradation.

### 3.4.3.2 Replay Mode Configuration Spectrum

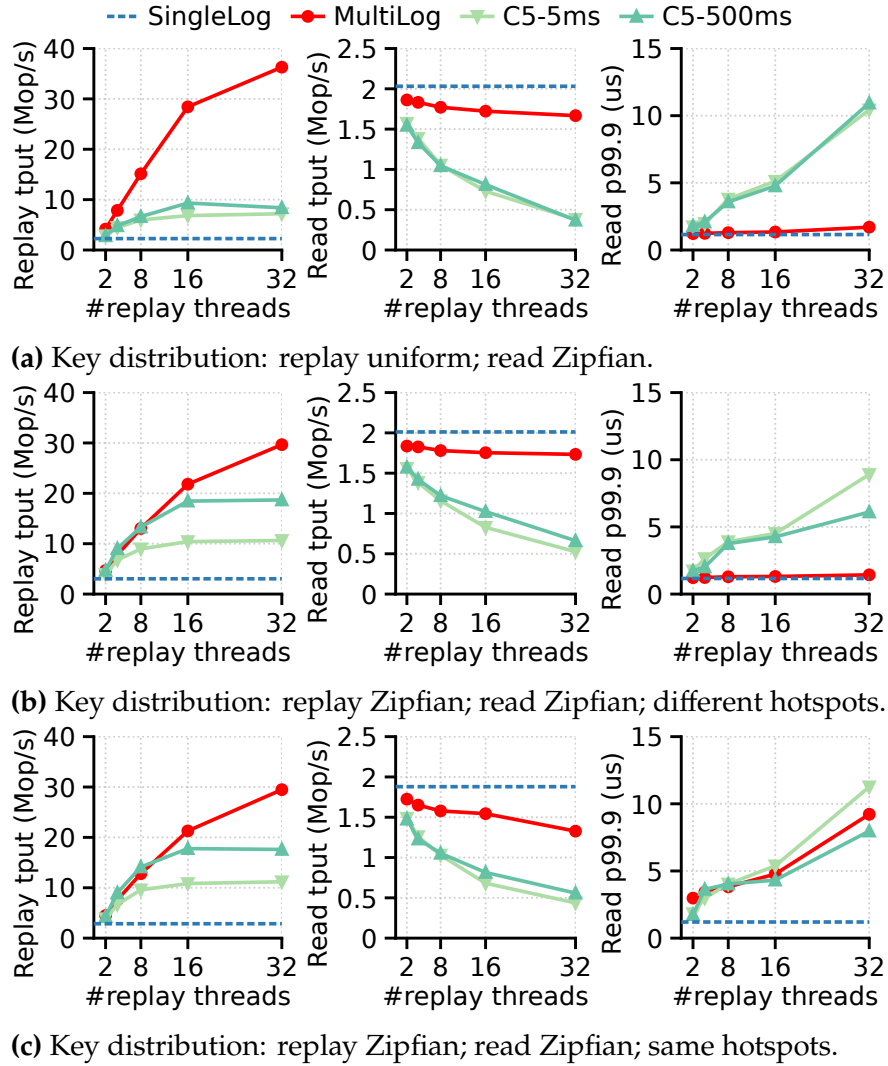
We next fix the number of replay threads at 64, excluding dispatcher threads, and measure throughput across direct and hybrid configurations with different ratios of physical to virtual sublogs. This experiment uses the same workload as §3.4.3.1.

**Results.** Figure 3.6(b) reveals two trends: within hybrid replay, increasing the number of physical sublogs raises routing capacity until the dispatcher is no longer the bottleneck; across modes, direct replay remains faster because it avoids communication between dispatchers and replay threads. Each physical sublog adds a dispatcher thread, producing nearly linear throughput growth from 16 Mop/s with MultiLog(1,64) to 61 Mop/s with MultiLog(4,16). Starting with MultiLog(8,8), dispatcher threads are no longer the bottleneck, so throughput is bounded by the replay threads and remains between 70 and 73 Mop/s. Nevertheless, hybrid configurations remain below direct replay, which reaches 83 Mop/s.

### 3.4.4 Prefix-Consistent Read Microbenchmarks

Because MultiLog enforces prefix consistency at read time, we evaluate read performance as replay scales. To do so, we extend the replay microbenchmark from §3.4.3 with a reader thread. The reader executes read operations directly in the server process, bypassing the network stack. This setup exposes the full read-side performance penalty without amortizing it over network overhead, yielding an upper bound on the relative performance loss for networked readers.

We evaluate the system under three workloads. In all three workloads, reads follow a Zipfian distribution ( $\theta = 0.99$ ). Replayed keys follow (1) a uniform distribution, (2) a Zipfian distribution whose hotspots differ from the reader's, or (3) a Zipfian distribution whose hotspots match the reader's. The second workload represents a common pattern, such as in a producer-consumer queue, where writers access keys at the tail of the queue while readers access keys at the head, resulting in distinct read and write hotspots. The third workload represents a pattern where some keys are hot for both reads and writes, which is also one of the most adversarial workloads for MultiLog. We set the replay barrier divergence threshold to the time required to replay  $\sim 5$  records.



**Figure 3.7: Replay and Read With Various Key Distributions.**

**Results.** Figure 3.7(a) shows that scaling MultiLog from 2 to 32 sublogs raises replay throughput from 4.2 to 36 Mop/s while read throughput decreases only slightly (10%). Read p99.9 latency remains at most 1.7 us. MultiLog consistently outperforms C5, reaching 4.3× higher replay throughput and 4.5× higher read throughput at 32 sublogs. Under uniform replay, sublogs advance at similar rates, so reader waits remain rare. Even at 32 sublogs, where accumulated divergence across more replay threads triggers the barrier more frequently, only 0.0075% of reads wait, up from 0.0004% at 2 sublogs.

Beyond the replay penalty discussed in §3.4.3, three factors further reduce C5’s read performance. First, reading a snapshotted version requires an additional version-chain traversal to locate the correct version. Second, C5’s out-of-place replay mutations enlarge the cache footprint and cause more misses for readers. Third, allocation and version writes consume cache and memory bandwidth, increasing reader access latency. Together, these costs make frequent snapshotting a poor fit for performance-critical memory-centric stores designed around in-place mutation.

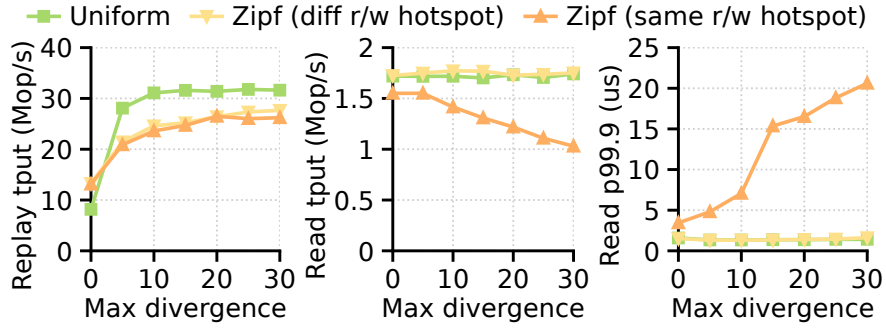
Figure 3.7(b) shows that distinct read and replay hotspots improve read performance, while replay skew limits replay scalability. Hot keys concentrate records in a few sublogs, and the resulting relative imbalance grows with the sublog count. At 32 sublogs, the hottest sublog accounts for 9.0% of records (compared with the ideal 3.1%) and becomes the replay bottleneck. C5-500ms achieves 1.8× higher replay throughput than C5-5ms because its longer snapshot interval allows repeated hot-key updates to be applied in place. Both variants plateau because of the single dispatcher.

Read performance in this workload is better than under uniform replay: p99.9 latency is 16% lower at 32 sublogs. Because the reader mostly accesses infrequently updated keys, its *SCT* usually remains below *LRT*, widening the validity window for non-blocking reads.

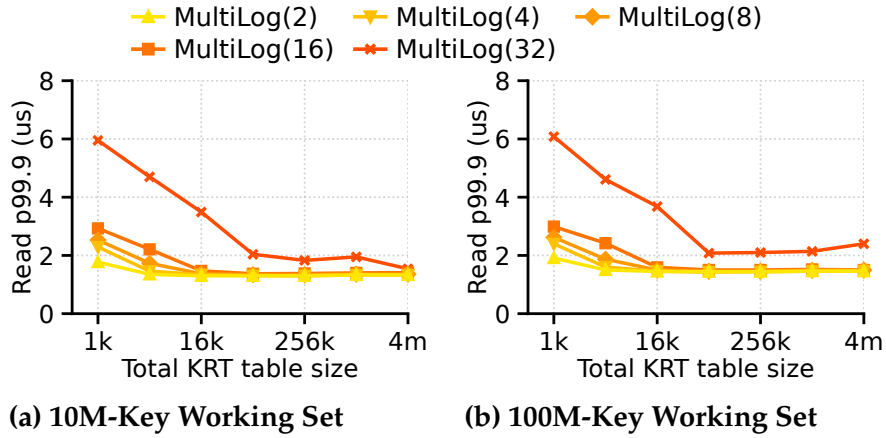
Figure 3.7(c) shows the results when reads and replay share the same hotspots. Frequent replay keeps a hot key’s *KRT* close to *LRT*, narrowing the validity window. Reading such a key therefore often advances the reader’s *SCT*, making subsequent reads more likely to block. Through 16 sublogs, p99.9 read latency remains below 5 us. At 32 sublogs, more frequent progress divergence causes 4.2% of reads to wait, raising p99.9 latency to 9 us.

#### 3.4.4.1 Sensitivity Analysis of Replay Divergence Threshold

The replay barrier bounds replay divergence to reduce reader blocking at the potential cost of replay throughput. To understand this trade-off, we vary the divergence threshold at which the barrier triggers. We focus on MultiLog(16), which combines high replay throughput with low read tail latency in the experiments above.



**Figure 3.8: Sensitivity Analysis of Replay Progress Divergence at MultiLog(16).** The x-axis shows the progress threshold to trigger the barrier, normalized by the time to replay a record.



**Figure 3.9: Effect of KRT Table Size on Read Tail Latency.**

**Results.** Figure 3.8 shows that replay throughput remains stable unless the threshold is so tight that it forces sublogs into lockstep, while read tail latency is threshold-sensitive when reads and replay share hotspots. For the uniform and different-hotspots workloads, read performance is largely insensitive to the threshold because wide validity windows keep the wait ratio low. With shared hotspots, however, a relaxed threshold permits greater replay divergence, increasing both the frequency and duration of reader waits and thereby raising p99.9 latency. Overall, setting the replay divergence threshold to the time it takes to replay 5–10 records offers a good balance between replay and read performance.

#### 3.4.4.2 Sensitivity Analysis of KRT Table Size

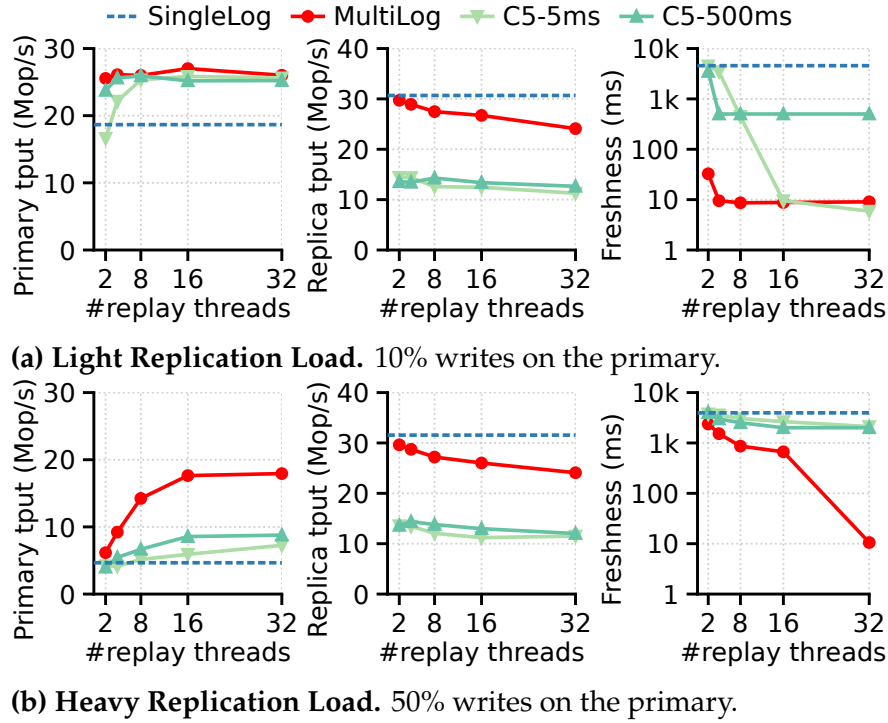
The *KRT* compression optimization (§3.3.6) trades off validity-window tightness against memory efficiency. To quantify the reader’s sensitivity to the *KRT* table size, we sweep the total table size (summed over all sublogs) across configurations from MultiLog(2) to MultiLog(32). We use 10M-key and 100M-key working sets to evaluate different table-to-working-set size ratios. Replay follows a uniform distribution, while reads follow a Zipfian distribution. We use uniform replay because it is more sensitive to the *KRT* table size: Zipfian replay concentrates recent updates on a small set of hot keys. We report p99.9 read latency because it is the metric most sensitive to the *KRT* table size.

**Results.** Figure 3.9 shows that 64K *KRT* slots suffice to stabilize p99.9 read latency even for MultiLog(32), largely independent of the working-set size. At smaller table sizes, MultiLog(32) is the most sensitive configuration, but larger tables provide little benefit beyond 64K slots. The knee changes little between the 10M-key and 100M-key working sets, even though the latter maps more keys to each *KRT* slot. This insensitivity to working-set size arises because only *KRT* values for recently updated keys affect the PrefixLoom wait path: a key not updated recently has a *KRT* well below every *SCT*, so its exact value does not matter (*SCT* remains unchanged in Algorithm 2 line 7). The relevant recency window depends primarily on system throughput rather than the total working-set size. Our implementation uses 256K slots for more headroom, which costs only 2 MB of memory.

#### 3.4.5 Replication Macrobenchmarks

We evaluate MultiLog end to end over the network, deploying the primary, replica, and clients on three separate machines. Both log replication and client-server traffic traverse the network.

The client machine runs 32 clients issuing mixed reads and writes to the primary, another 32 clients issuing read-only requests to the replica, and a *freshness watcher*. The freshness watcher periodically writes a special key to the primary and measures the time until the key becomes visible at the replica. We vary the primary’s write fraction to control replication load: 10% for light load and 50% for heavy load. Requests



**Figure 3.10: Replication Macrobenchmark Results.**

to both the primary and the replica follow Zipfian distributions but with different hotspots. We report primary and replica throughput and the p99.9 freshness delay.

To keep the system stable, replication uses backpressure to bound its lag. If the replica replays entries more slowly than the primary generates them, unreplayed entries accumulate and the lag grows. When the lag reaches the configured limit, the replica applies backpressure to throttle primary appends. We set this limit to 512 MB.

**Results.** Figure 3.10(a) shows the results under the light replication load. At the primary, MultiLog matches C5 and outperforms SingleLog by 1.4 $\times$ . At the replica, MultiLog outperforms both C5 variants by at least 1.9 $\times$  and stays within 22% of SingleLog. For freshness, MultiLog keeps the p99.9 delay within 10 ms, orders of magnitude better than SingleLog and C5-500ms.

SingleLog cannot sustain even this light replication load, so its replication lag reaches the 512 MB limit, triggering backpressure that throttles the primary. A newly written key must then wait for the replica to replay the backlogged records, raising freshness delay to 4.5 seconds. C5-5ms and C5-500ms achieve similar primary and

replica throughput, but their snapshot intervals produce very different freshness. With C5-500ms, a replayed update remains invisible until the next snapshot, so its freshness delay reaches 0.5 seconds. By contrast, MultiLog’s replay capacity exceeds the write rate, keeping replication lag minimal and newly written keys visible within 10 ms.

Figure 3.10(b) shows that under the heavy replication load, only MultiLog(32) sustains the incoming log rate. It delivers  $3.8\times$  higher primary throughput than SingleLog and at least  $2\times$  higher replica read throughput than either C5 variant. SingleLog and C5 both reach the 512 MB lag limit, triggering backpressure that throttles the primary; their freshness delays also reach multiple seconds. Freshness falls into two regimes: when replay keeps up, it reflects replication latency ( $\sim 10$  ms); otherwise, it is dominated by the time to replay the backlogged records. Even when MultiLog falls behind, scaling from 2 to 8 sublogs raises replay throughput and reduces p99.9 freshness delay from 2.4 seconds to 0.9 seconds.

### 3.5 Conclusion

In this chapter, we present MultiLog, a replication architecture for memory-centric storage that scales to multiple threads by sharding the log into multiple sublogs that can be appended and replayed independently. At its core lies PrefixLoom, a replica-side protocol that lazily restores prefix consistency through lightweight metadata, avoiding the MVCC and snapshotting machinery prior parallel-replay systems rely on. Implemented in Garnet, MultiLog delivers up to  $19\times$  higher append throughput and  $36\times$  higher replay throughput over the single-log baseline. Under a write-heavy replication workload, MultiLog sustains  $3.8\times$  higher primary throughput than the single-log baseline and  $2\times$  higher replica read throughput than C5. Its replica stays within  $\sim 10$  ms of the primary at p99.9, two orders of magnitude fresher than either baseline. More broadly, this work shows that the long-standing tension between scalability and prefix consistency for log-replicated stores can be resolved with a lightweight protocol rather than heavyweight physical versioning.

## Chapter 4

# ROS: Memory-Centric Storage for LLM Reinforcement Learning

Earlier chapters discuss memory-centric storage in the classic setting for generic workloads. Recently, large language models (LLMs) have emerged as a dominant workload for modern computing infrastructure. In particular, reinforcement learning (RL) for LLM post-training introduces highly dynamic and data-intensive execution patterns. In each training step, the trainer workers commonly transfer terabytes of model weights to thousands of rollout workers. In the last part of this dissertation, we turn to this emerging workload and explore how to simplify its data flows through the lens of memory-centric storage.

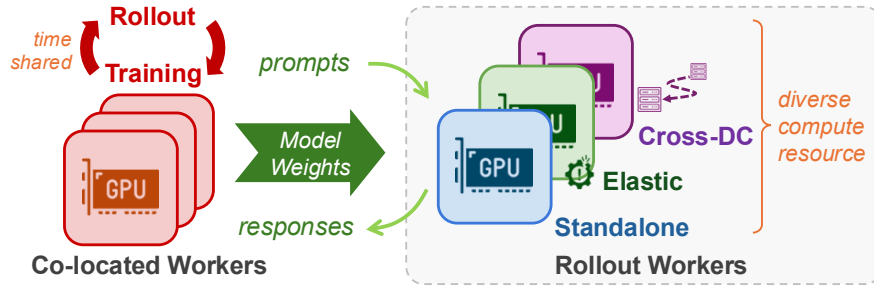
Existing LLM RL frameworks [34, 47, 102, 139] typically build weight transfer directly on top of communication primitives, using either collectives (e.g., NCCL [69]) or point-to-point communication (e.g., UCX [98]). However, both approaches have limitations for RL weight transfer at scale. NCCL relies on a fixed communication group, and a single worker failure aborts the entire group; this rigidity makes NCCL ill-suited to clusters with failures and restarts. Point-to-point communication naturally supports membership changes, but each trainer serves weights independently without a global view of network topology or runtime load, causing congestion under fan-out traffic. Both approaches also require RL frameworks to coordinate transfers across GPU workers. This temporal coupling leaves GPUs idle while they wait for other workers.

Another weight transfer approach is through distributed storage (e.g., parameter servers [53, 59], Ray object store [67]). Unlike communication-based approaches, distributed storage offers a clean decoupling between trainers and rollouts: trainers *push* new weights to storage, while rollouts *pull* them on demand, all with minimal coordination; rollouts may scale up and down without trainers’ awareness. Yet, this push-then-pull pattern doubles data movement and incurs storage space or memory overhead, both of which are prohibitively expensive for TB-sized LLM weights.

We seek to retain the decoupling advantages of storage interfaces while eliminating the overhead of extra memory space and data movement. This is enabled by a key observation: each version of the model weights is *immutable* and *highly replicated* for data-parallel inference. Such redundancy naturally provides multiple equivalent data sources to serve read requests for that version.

Our core idea is to utilize those highly replicated model weights *in place* to build a new memory-centric storage system specialized for RL training workloads. We propose **Reference-Oriented Storage** (ROS). ROS presents the illusion that certain versions of the model weights are stored and can be fetched on demand. Underneath, ROS does not physically store any copies of the weights; instead, it tracks the workers that hold these weights on GPUs for inference. Upon request, ROS directly uses them to serve reads. Based on the ROS idea, we built a production-quality system called **TensorHub**, which provides rich features like user-friendly interfaces, topology-aware transfer, model-parallel consistency, and fault tolerance. TensorHub reduces total GPU stall time by up to 6.7× for standalone rollouts, accelerates weight updates for elastic rollouts by 4.8×, and cuts cross-datacenter rollout stall time by 19×. TensorHub has been deployed at ByteDance for cutting-edge reinforcement-learning training.

The remainder of this chapter is organized as follows. §4.1 lays out the workload and existing approaches in detail. §4.2 introduces the ROS abstraction along with its two key mechanisms, the mutability contract and the retention protocol. §4.3 describes TensorHub’s interfaces, topology-aware transfer, model-parallel consistency, and fault tolerance. §4.4 evaluates TensorHub with microbenchmarks and three rollout case studies. §4.5 shares lessons from operating weight transfer in production. §4.6 concludes this chapter.



**Figure 4.1: RL Workload With Diverse Rollout Patterns.** Rollout workers stream in prompts and model weights and stream out responses. While prompts and responses are lightweight, the model weights can be several terabytes in size, necessitating an efficient transfer system.

## 4.1 Background and Motivation

In modern LLM reinforcement learning (RL) workloads, weight transfer is essential for fully leveraging heterogeneous compute resources. This section covers the background of RL workloads, the goals of a weight transfer system, and the limitations of existing approaches.

### 4.1.1 Reinforcement Learning Workload Background

A typical RL algorithm [91, 99] proceeds through three stages: (1) *rollout*, which runs inference and generates responses for given prompts; (2) *scoring*, which judges the responses with a reward module (e.g., human ratings [14, 24, 71], a reward model [24, 71], or rule-based rewards [99]); and (3) *training*, which computes gradients and updates the model weights. The updated weights are used in the next rollout, forming a continuous feedback loop.

RL frameworks implement the algorithms by mapping rollout and training onto GPU workers. Figure 4.1 illustrates the rollout and training patterns used by representative RL frameworks. In a basic *co-located* architecture, rollout and training time-share the same set of physical GPUs. We refer to these co-located workers as *trainers*. This architecture is widely adopted by veRL [102], ReaL [64], and RLHFuse [136] for its simplicity and high resource utilization.

However, rollout time can account for more than 90% of training time [45, 47, 102], thereby becoming the bottleneck. This motivates many new patterns to scale out rollout by utilizing additional GPUs:

- *Standalone rollouts* are GPU workers dedicated to rollout tasks. This design is widely adopted by many frameworks [34, 47, 101, 121].
- *Elastic rollouts* run on spot GPU instances that can be preempted or restarted at any time. Their lower price enables greater cost efficiency. RLBoost [122] reports up to 1.97× higher training throughput with this approach.
- *Cross-datacenter rollouts* run on GPUs located in a datacenter separate from the trainers. They enable hyperscale training [73] and leverage heterogeneous resource availability [135]. Because GPUs are often procured and deployed in different batches, inference-optimized GPUs may not be available in the trainers’ datacenter [135].

With rollouts running on dedicated GPUs, model weights must be transferred from trainers after each training step, typically every few to tens of minutes. At scale, each update disseminates terabyte-sized model weights to thousands of GPUs. This transfer lies on the critical path: rollout workers cannot perform inference using partially updated weights, leaving their GPUs idle until the weight transfer is complete.

In addition, rollout workers are organized into model-parallel groups, each holding a full model copy sharded across its workers. These groups are loosely coupled with one another, progress at different paces as they generate variable-length outputs, and become ready for new weights at different times. The weight transfer system must coordinate these transfers dynamically across groups.

#### 4.1.2 Weight Transfer System Goals

To serve these diverse rollout patterns, an ideal weight transfer system must satisfy five goals:

- **G1: Topology Awareness.** To fully utilize network bandwidth, the weight transfer system must incorporate network topology and runtime load knowledge.

| Approaches           | Example                        | G1 | G2 | G3 | G4 | G5 |
|----------------------|--------------------------------|----|----|----|----|----|
| Collective Comm.     | NCCL [69]                      | ✓* |    |    | ✓  | ✓  |
| Point-to-Point Comm. | UCX [98]                       |    | ✓  |    | ✓  | ✓  |
| Distributed Storage  | PS [59], Ray object store [67] | ✓  | ✓  | ✓  |    |    |
| TensorHub            |                                | ✓  | ✓  | ✓  | ✓  | ✓  |

**Table 4.1: Comparison of Different Weight Transfer Approaches Against Five Goals.** G1: Topology Awareness; G2: Elasticity; G3: Minimal Coordination; G4: Minimum Data Movement; G5: No Extra Storage or Memory Space. \*only if within a single datacenter.

For cross-datacenter training, it must minimize the network traffic over slow inter-datacenter links. Within the same datacenter, it should avoid single-node bandwidth congestion.

- **G2: Elasticity.** The weight transfer system must support a dynamic cluster that scales up and down while transparently masking failures. If a worker fails or is preempted, the system must safely evict it without disrupting healthy workers. If a worker restarts or joins the cluster, the system must promptly serve weights on demand.
- **G3: Minimal Coordination.** Weight transfer should minimize coordination among workers. Coordination not only complicates programming but also introduces additional waiting and amplifies stragglers at scale.
- **G4: Minimum Data Movement.** The optimal data transfer must occur directly between the source and destination GPUs, with no intermediate hop. An extra hop implies additional network or PCIe costs, hurting overall efficiency.
- **G5: No Extra Storage or Memory Space.** As LLM weights can be several terabytes, storing extra copies requires massive storage or memory space. Such extra resource requirements should be minimized for cost efficiency.

### 4.1.3 Limitations of Existing Approaches

Existing weight transfer approaches fall into three main categories: *collective communication*, *point-to-point communication*, and *distributed storage*. Table 4.1 summarizes how well each meets the above goals.

**Collective Communication.** The NVIDIA Collective Communication Library (NCCL) [69] is widely used for weight transfer, adopted by veRL [102], OpenRLHF [47], AReaL [34], vLLM [56], SGLang [132], Slime [139], among others. In these frameworks, trainers broadcast weights to rollouts via NCCL collectives, achieving high throughput with topology-aware transfer that saturates RDMA bandwidth.

However, collective communication fails on the elasticity and minimal coordination goals. First, collective communication requires a pre-defined communication group that cannot add or remove nodes afterward. This prevents it from being applied to a dynamic cluster where nodes may fail, be preempted, or be restarted; any failure will cause the entire cluster to crash. Second, broadcast requires all nodes to cooperate, which in practice translates to a global barrier in RL frameworks that interrupts all workers and does not resume execution until all transfers have finished [34, 47, 102]. This amplifies stragglers, which are common in RL workloads. For example, a rollout may delay responding to an interrupt while generating a long sequence, or network contention may slow a subset of nodes during transfer.

**Point-to-Point Communication.** Point-to-point communication (e.g., UCX [98]) enables flexible data transfer between workers without a communication group, and thus naturally supports dynamic clusters. In an ideal case, such direct transfer can saturate the RDMA bandwidth between two endpoints, delivering high throughput.

However, point-to-point communication is less commonly used in RL because it has no global view of the runtime topology and therefore cannot balance load effectively. Senders serve requests independently, making their outbound bandwidth the bottleneck under fan-out. For example, when multiple rollouts pull weights from a trainer, they contend for the trainer’s uplink bandwidth, degrading performance. Moreover, point-to-point communication still requires coordination at the framework level. When a trainer issues a `send()`, the corresponding rollout must be interrupted to invoke `recv()`; the `send()` blocks until the receiver cooperates.

**Distributed Storage.** Distributed storage systems, e.g., parameter servers (PS) [53, 59] and the Ray object store [67], provide a compelling *decoupled* interface for weight transfer: trainers *push* updated weights to storage, and rollouts *pull* weights on demand. By replicating weights across storage nodes, such systems can exploit topology and locality, serving requests from nearby or less-loaded replicas. Moreover, this decoupling also naturally supports elasticity, since rollouts can join, leave, or restart independently without trainers’ awareness. Finally, trainers can proceed immediately after pushing weights, while rollouts fetch weights on demand, minimizing coordination between them.

Despite these compelling properties, storage has not been adopted as the primary weight transfer mechanism in major LLM RL frameworks due to poor performance. For parameter servers [53, 59], the push-then-pull workflow amplifies network traffic, and the storage server’s network bandwidth becomes a potential bottleneck at scale. The Ray object store [67] mitigates the network bottleneck by co-locating storage with workers, but still incurs extra data movement, requiring GPU–CPU copies and (de)serialization before transferring and reconstructing weights. For example, transferring 40 GB of data between two 8-GPU machines via the Ray object store takes 32 s in our measurement, whereas GPU-direct RDMA completes the same transfer in  $\sim 0.2$  s. Such orders-of-magnitude performance losses can hardly be justified by functionality benefits. Lastly, storing TB-sized LLM weights is too expensive in either remote or co-located storage. In our experiment, the Ray object store commonly crashes due to out-of-memory errors when transferring more than 300 GB of data at once.

More fundamentally, data *ownership* is the root cause of storage’s extra movement and space overhead. Each write operation copies weights into storage-managed space (either a remote server or co-located CPU memory), allowing the system to manage data independently of workers. This ownership enables trainer-rollout decoupling but also introduces additional serialization, copying, network traffic, and storage overhead that ultimately degrade performance.

**Summary.** Existing approaches present a trade-off. Collective communication achieves high throughput with topology-aware transfer, but requires rigid membership and global coordination. Point-to-point communication supports dynamic

clusters and direct transfer, but struggles with load balancing and still incurs coordination at runtime. Distributed storage offers clean decoupling and flexibility, but pays the price of additional data movement and storage overhead due to full data ownership. As a result, none of these approaches satisfies all five goals—topology awareness, elasticity, minimal coordination, minimum data movement, and no extra storage or memory space.

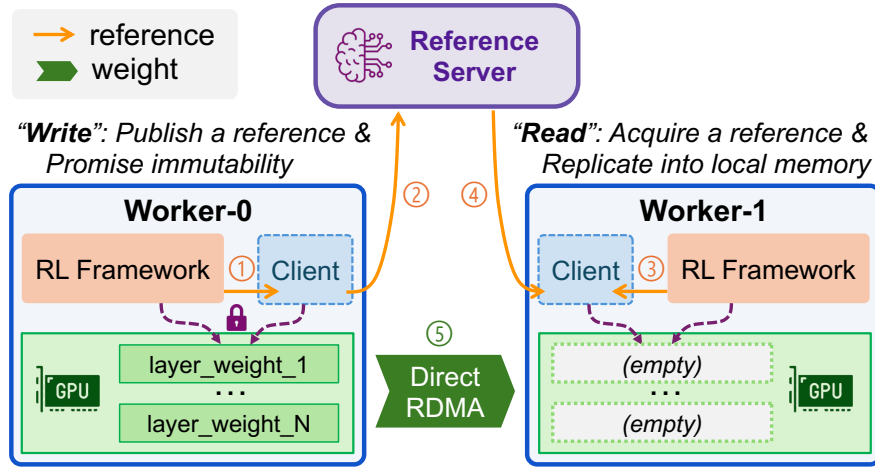
## 4.2 Reference-Oriented Storage

Distributed storage offers a clean and flexible interface for weight transfer, but its reliance on data ownership introduces fundamental inefficiencies. To address this limitation, we present *Reference-Oriented Storage* (ROS), a new storage abstraction that does not own data.

ROS is enabled by three important properties of model weights in RL. First, weights are *highly replicated* due to data parallelism. This redundancy means there are multiple symmetric sources available for transfer. Second, weights used in inference are *immutable*. As a result, reading them from a remote worker’s GPU memory is safe and requires minimal coordination. Third, weight versions are *short-lived*. Training constantly generates new versions, and only the most recent few are relevant, so long-term durability is unnecessary.

ROS directly repurposes those model weights *in place* to provide the illusion that certain versions of the weights are stored and can be fetched on demand. Under the hood, ROS does not physically store any copies of the weights. Instead, it tracks the workers with these weights on GPUs for inference; upon request, ROS directly uses them to serve reads. By avoiding explicit data ownership and extra copies, ROS combines the flexibility of storage with the efficiency of direct transfer.

This section covers the elemental primitives required for a minimal ROS system. We first illustrate the overview of ROS and then elaborate on two key mechanisms required for integrity and availability. A more sophisticated system with rich features will be covered in §4.3.



**Figure 4.2: Reference-Oriented Storage Workflow.** The server only operates on lightweight references. The bulk weight transfer is directly between clients.

#### 4.2.1 ROS Overview

ROS consists of a centralized reference server and a client library imported by each worker. Figure 4.2 illustrates an example where Worker-0 holds a version of model weights in GPU memory for inference, and Worker-1 needs that version.

ROS exposes `publish` and `replicate` primitives for write and read operations. ① To write weights into ROS, Worker-0 calls `publish` and passes a reference to its in-GPU weights to its client. ② The client forwards this reference to the reference server. ③ To read weights, Worker-1 invokes `replicate` with a reference to its local weight buffers. ④ Its client queries the server, which returns a source reference pointing to Worker-0's GPU memory. ⑤ Worker-1's client then issues RDMA to read the weights directly from Worker-0 into its local buffers. Upon completion, Worker-1's copy becomes another replica of the model weights. Future replication requests can then be served from either worker.

Critically, these data flows require no additional data movement or storage overhead. Unlike traditional distributed storage, the ROS server never stores or transfers weight data. It only operates on the lightweight references. The bulk data transfer is directly between the source and destination memory without intermediate hops or staging space.

ROS treats all copies of the same weights as symmetric replicas, whether created via `publish` or `replicate`. This means a rollout need not fetch weights from a trainer; it can retrieve them from any peer. This design enables efficient and resilient peer-to-peer data dissemination. In the presence of failures, as long as at least one replica remains, a restarted worker can recover from the live peer, effectively making the cluster self-healing.

**Challenges.** Since ROS only *references* weights but does not *own* them, it must address two challenges to be functional.

- *Integrity:* Weights require the flexibility for occasional mutations (e.g., buffer reuse for a new version). ROS must define clear mutation semantics to avoid data corruption.
- *Availability:* Because ROS does not own the data, a weight version disappears if no worker holds a copy. The system must define which weights should remain accessible and introduce protocols to preserve that.

ROS introduces two key mechanisms to address the challenges: the *mutability contract* and the *retention protocol*.

#### 4.2.2 Integrity via Mutability Contract

ROS relies on weight immutability for safe sharing, but RL training requires the flexibility for mutability. For example, as weights are too large to be duplicated on GPUs, a rollout may want to reuse the weight buffers in place when pulling the new weights. Similarly, a trainer must be able to mutate the weights to produce a new version.

ROS addresses this demand with the `unpublish` primitive and the *mutability contract*. When holding a replica, a worker commits not to mutate the weight data. When mutation is required, the worker must call `unpublish` to explicitly revoke the immutability commitment. Upon request, the reference server ensures this worker's weights are no longer visible to others; if there is an ongoing transfer from this worker, the server waits for the transfer to drain and only acknowledges `unpublish` success afterward. This ensures every concurrent transfer still receives uncorrupted data.

Fortunately, adhering to the mutability contract is easy in practice. An RL worker’s execution flow is typically organized into stages, in which the immutability boundary is straightforward to identify. In addition, RL frameworks are often implemented in Python, whose memory safety avoids many dangling-pointer bugs.

### 4.2.3 Availability via Retention Protocol

ROS stores no copies, so if all workers unpublish a version, that version is unavailable. Such behavior is appropriate for overly stale weights, as they are not needed by any workers and thus safe to drop. However, losing the latest (few) versions of the weights is unacceptable. This scenario can occur in a corner case if all trainers unpublish the latest version before it is replicated by any rollout. To avoid such unavailability, the system requires clear semantics for “what data must be available” and a protocol to enforce that.

ROS fulfills this requirement with the *retention protocol*. ROS exposes *retain* semantics under which a worker can declare the versions to keep available (e.g., the latest  $k$  versions). When processing *unpublish* requests, the server checks if the requesting replica is the last copy of a retained version; if so, it notifies the client to offload a copy of the weights to CPU memory and publish that as an *offload* replica.

Offloading is a safeguard for corner cases and is rarely triggered in practice. Rollouts typically see and replicate the new version promptly; as long as one copy is replicated, trainers can unpublish without triggering offload. When offloading does occur, only a single copy across the entire cluster is kept, minimizing memory overhead. Measurements show that offloading over the PCIe bus is fast ( $\sim 48$  GB/s in our setup), resulting in a minor delay for unpublish ( $< 2$  s even when offloading the entire GPU’s weights). The offload replica is automatically released once it has been replicated by another worker (and is thus no longer the last copy) or once a newer version’s publish makes it no longer retained.

The elegance of the retention protocol is that it provides similar availability to traditional ownership-based storage, while eliminating the ownership overhead in common cases.

#### 4.2.4 ROS Summary

ROS achieves Goals 3, 4, and 5 (§4.1.2) by design. By providing a storage interface, ROS enables rollouts to pull weights on demand without trainers’ explicit coordination (G3). By not taking ownership of weight data, ROS avoids the overhead of data movement (G4) and extra storage space (G5) in the common case.

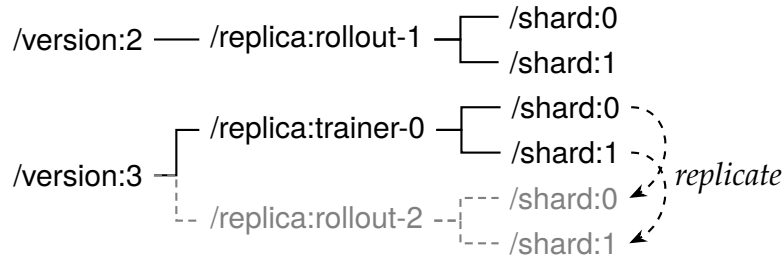
ROS also establishes a good foundation for Goals 1 and 2. In principle, its centralized server architecture offers a global view of network topology and runtime load, making topology-aware transfer easier to implement (G1). Elasticity can be achieved as long as the ROS system itself is fault-tolerant (G2). In summary, both goals fit into ROS’s design but require a well-optimized implementation.

### 4.3 TensorHub Design and Implementation

We build TensorHub, a production-quality system that instantiates the ROS idea with rich features required by real-world use cases. As LLM sizes increase, model parallelism is required, with each GPU worker holding only one shard of weights. Therefore, sharding must be natively incorporated in addition to the previous ROS requirements.

Overall, a production-quality system must satisfy four practical requirements. First, it must provide concise and expressive interfaces for easy integration into existing RL frameworks. Second, for efficient transfer at scale, the system must schedule transfer based on the network topology and runtime loads. Third, for correctness, the system must guarantee that workers in the same model-parallel group see a consistent view of the system state, preventing control-flow divergence. Finally, to support dynamic cluster scaling up and down, it must gracefully handle failures.

This section first describes the naming scheme (§4.3.1) and APIs (§4.3.2) and then introduces topology-aware transfer (§4.3.3), model-parallel consistency (§4.3.4), fault tolerance (§4.3.5), and implementation (§4.3.6).



**Figure 4.3: TensorHub Naming Scheme.** Rollout-2 fetches version 3 by replicating a copy; after doing so, it also becomes a replica capable of serving subsequent replication requests.

### 4.3.1 Naming Scheme

We begin with a hierarchical naming scheme to establish a precise vocabulary. In TensorHub, each model forms an independent domain, managed by a single reference server. Each model evolves over *versions*, each produced by one step of training. A version consists of one or more *replicas*, each representing a full copy owned by a model-parallel worker group. Finally, each replica is divided into *shards*, with each shard owned by a single worker. Figure 4.3 illustrates an example with three replicas, each composed of two shards.

Each version is uniquely identified by an integer, called the *absolute* version. TensorHub also supports *relative* versions, where the strings “latest” or “latest- $k$ ” are resolved to the latest version or the latest minus  $k$ , respectively. Relative versions are useful because RL typically only cares about freshness relative to the latest one. Off-by- $k$ -step RL training workloads [34, 101, 121, 124, 138] may require multiple versions to co-exist in the system, where “latest- $k$ ” is useful.

### 4.3.2 TensorHub APIs

TensorHub APIs are summarized in Table 4.2. As the same set of tensor buffers is commonly used across versions, TensorHub exposes `ShardHandle` to operate on them.

- **Open.** A worker begins by calling `tensorhub.open()` to acquire a handle for a shard within a replica. If it wishes to retain certain versions, it specifies them upon open.

| Name                            | Description                                                                                                                                                                                                          |
|---------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>open(...)</b> -> ShardHandle | Acquire a shard handle. Optionally, declare the desired versions to retain for the retention protocol.                                                                                                               |
| <b>register(named_tensors)</b>  | Register weight tensors.                                                                                                                                                                                             |
| <b>unregister()</b>             | Unregister weight tensors.                                                                                                                                                                                           |
| <b>publish(version)</b>         | Publish a reference to the registered tensors under the given version.                                                                                                                                               |
| <b>unpublish()</b>              | Revoke the previously published reference.                                                                                                                                                                           |
| <b>replicate(version)</b>       | Replicate the given version of weights into registered tensors. <i>Supports relative versions.</i>                                                                                                                   |
| <b>update(version)</b> -> bool  | Atomically check whether the given version is available and differs from the current one; if so, unpublish the current version and replicate the new one; return true if updated. <i>Supports relative versions.</i> |
| <b>list()</b> -> Dict           | Return a dictionary mapping each available version to its set of replica names.                                                                                                                                      |
| <b>wait(predicate)</b>          | Wait until <code>predicate(list())</code> evaluates to true.                                                                                                                                                         |
| <b>close()</b>                  | Destroy the handle; automatically unpublish and unregister if applicable.                                                                                                                                            |

**Table 4.2: TensorHub APIs.** `open()` returns a handle, and the rest of the APIs are methods of this handle.

- **Registration.** Before publishing or replicating, a worker must register weight tensors with the handle; before deallocating these tensors, it must unregister. These APIs support both CPU and GPU tensors, offering flexibility in memory management. For example, trainers in fully disaggregated architectures [121, 135] execute training in a tight loop without co-located rollout and thus constantly mutate their on-GPU weights; they can publish weights from CPU memory.
- **Publish.** To make its registered tensors visible under version  $v$ , the worker calls `publish(v)` with an immutability commitment. When the worker desires mutation, it must invoke `unpublish()` first. These two calls form the core mutability discipline for reference safety.

- **Replicate.** A worker obtains weights by naming the desired version (absolute or relative). `replicate(v)` materializes version  $v$  into the handle's registered tensors, blocking if necessary until the version exists.
- **Update.** With a version held, a worker can call `update(v)` to atomically transition to a newer version if available. Atomicity avoids races between a version check and replication. This API is useful for a rollout to poll whether a new version of the weights is available; if not, it continues inference with the current weights.
- **Query.** TensorHub provides lightweight introspection through `list()` and `wait(predicate)`, which expose global metadata (such as available versions or replicas) for implementing custom policies. For example, rollouts can discard excessively stale responses, or trainers can wait for lagging rollouts to catch up [34, 101, 121, 124, 138].
- **Close.** When a worker finishes using the handle, it calls `close()` to release resources and clean up state.

**Example.** Figure 4.4(a) shows the workflow of a trainer that runs both training and rollout tasks. At the beginning of each step, it publishes the current weights and starts rollout. After completing a configured amount of rollout work, it unpublishes the weights and switches to training. Once training completes, it converts the updated weights to an inference-ready format (e.g., resharding and quantization), publishes the new version, and begins the next rollout step.

Figure 4.4(b) shows an example of a standalone rollout. Upon initialization, it calls `replicate()` to fetch a copy of weights before starting inference. Then, after finishing a batch of inference or receiving a control message, it calls `update()` to fetch a new version if available; otherwise, it continues inference with the existing weights.

### 4.3.3 Topology-Aware Replication at Scale

In RL training, every newly published version often needs to be fetched by multiple rollouts. A system that funnels all replication through a single node will collapse under network contention. TensorHub is therefore architected with a centralized

---

```

import tensorhub
handle = tensorhub.open(
    model_name="actor",
    replica_name="trainer-0",
    num_shards=WORLD_SIZE,
    shard_idx=RANK,
)

tensors = get_weights()
handle.register(tensors)

for step in range(NUM_STEPS):
    handle.publish(version=step)
    do_rollout()

    handle.unpublish()
    do_train()
    # convert to the
    # inference-ready format
    post_process_weights()

handle.close()

```

---

(a) Trainer

---

```

import tensorhub
handle = tensorhub.open(
    model_name="actor",
    replica_name="rollout-1",
    num_shards=WORLD_SIZE,
    shard_idx=RANK,
    retain="latest",
)

tensors = get_weights()
handle.register(tensors)

# fetch the initial weights
handle.replicate("latest")

while has_work():
    is_updated = \
        handle.update("latest")
    if is_updated:
        clear_kv_cache()
        do_rollout()

handle.close()

```

---

(b) Standalone Rollout

**Figure 4.4: TensorHub Examples.**

reference server to schedule transfer based on network topology and runtime loads, while the clients remain decentralized for scalable data movement.

#### 4.3.3.1 Load-Balanced Scheduling at the Server

Efficient transfer begins with choosing the right source for each request. TensorHub centralizes scheduling on the server, exploiting its global view of the system for informed decisions.

The server always prioritizes a source replica located in the same datacenter. Within each datacenter, the server maintains a reference count for each replica—representing the number of replication requests it is currently serving—and always selects the least-loaded replica. Under pipeline replication (§4.3.3.3), each replica

serves at most one request at a time, barring failures. This load-balancing policy also keeps unpublish latency low: if a source replica requests to unpublish, it needs to wait for in-flight replication to drain, which is brief and bounded by a single request. If no source replica is available within the same datacenter, the server falls back to cross-datacenter replication (§4.3.3.4).

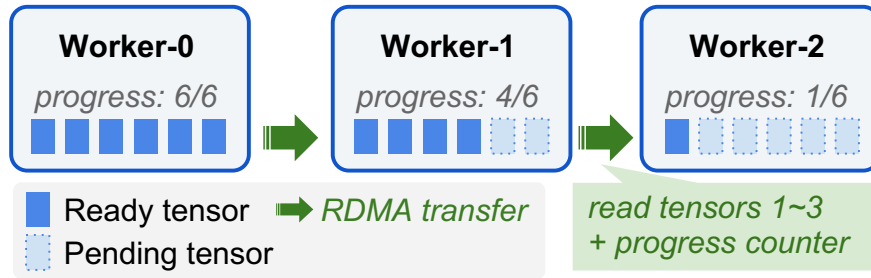
In practice, a single server is sufficient to scale a training task to thousands of workers (more measurements in §4.4.1.4). As future work, if the server becomes a scalability bottleneck at even larger scales, TensorHub could partition workers into sub-clusters, each managed independently by a server. These servers would only need to lazily synchronize on critical events (e.g., a new version is published).

#### 4.3.3.2 High-Performance Transfer Between Clients

TensorHub’s data-plane efficiency relies on a high-performance transfer method between clients. To achieve this, each client embeds a hardware-affinity-aware transfer engine that uniformly supports RDMA and TCP transfers over both GPU and CPU memory. More specifically, the transfer engine offers three transfer modes, customized for workloads and available transport methods:

- *RDMA Direct*: In workloads where tensors remain allocated for long periods, the transfer engine registers tensor buffers directly with the RDMA NIC. Remote clients can then issue one-sided RDMA reads to pull data directly. This zero-copy path bypasses CPUs entirely and achieves the best performance.
- *RDMA Copy*: When users frequently reallocate tensors, repeated RDMA NIC registration is too costly. In this case, the transfer engine employs background threads to serve remote requests by copying the tensor data slice-by-slice into pre-registered buffers and then issuing RDMA writes. This approach eliminates re-registration overhead while still preserving RDMA’s high throughput.
- *TCP*: In environments where RDMA is unavailable (e.g., across datacenters), the transfer engine uses TCP.

The transfer engine is hardware-affinity-aware and optimized for modern multi-GPU, multi-NIC servers. For GPU-resident tensors, the engine selects the NIC closest



**Figure 4.5: Pipeline Replication.** Worker-0 is the only source, while both Worker-1 and Worker-2 are requesting data. To scale throughput, TensorHub schedules a pipeline where Worker-2 reads partially replicated data on Worker-1.

to the tensor’s GPU; in RDMA copy mode, it uses the pre-registered RDMA regions on the same GPU devices.

**Tiny-Tensor Optimization.** LLMs contain a large number of tiny tensors, which are inefficient to register and transfer. To address this, TensorHub compacts all tiny tensors (<2 MB) into contiguous buffers; only these compacted buffers are registered and transferred. During replication, the receiver unpacks them into their corresponding slices.

This optimization reduces registration costs and improves bandwidth utilization, at the cost of very minor memory overhead. For example, a 36B model has approximately five hundred tensors, half of which are tiny. When compacted, they require only about 3 MB of extra memory in total, which is negligible for a 19 GB shard.

#### 4.3.3.3 Pipeline Replication

Pipeline replication enables TensorHub to scale throughput beyond what a single source could provide. A key challenge in RL training is that large bursts of rollouts may simultaneously request the latest version, creating extreme fan-out. Pipeline replication allows partially replicated workers to begin serving others immediately, turning replication into an expanding pipeline rather than a single-rooted broadcast (as shown in Figure 4.5). Pipeline replication builds on the observation that most RDMA NICs are full-duplex, and therefore, when a NIC is receiving data, its uplink is idle and can serve data.

Pipeline replication uses the metadata and transfer mechanisms described below. Each replicating worker tracks a *progress counter* that indicates how many tensors of

the target version it has received locally. When a new worker requests a version, the reference server selects the least-loaded source, which can be either fully replicated or still in progress. If the source is only partially replicated, the requester reads the source’s progress counter and then fetches the prefix that is available. It then re-reads the progress counter and repeats until all data is received.

Pipeline replication transforms the topology from a fixed N-way fan-out into a bandwidth-amplifying DAG, where each replicating worker increases the system’s bandwidth. Because transfers are decentralized and occur directly between peers, throughput grows with the number of active clients, and latency remains low even under heavy demand.

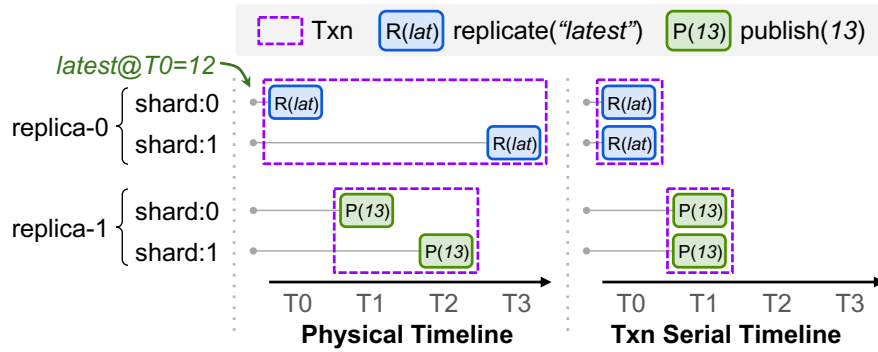
#### 4.3.3.4 Cross-Datacenter Replication

TensorHub’s replication model generalizes to clusters spanning multiple datacenters, with a two-stage replication pattern.

When a new version is published, the first replica located in a different datacenter must fetch the data over TCP; we refer to this replica as the *seeding replica*. Once seeding completes, subsequent replicas in that datacenter can fetch the data over RDMA, enabling full pipeline replication locally without repeatedly invoking cross-datacenter transfer.

TCP-based seeding has substantially lower bandwidth than RDMA and can therefore stall pipeline replication if other replicas immediately try to update and fetch from the partially filled seeding replica. To avoid this stall, TensorHub augments the `update()` API with *smart skipping*: if a polling replica observes that a seeding replica is actively fetching over TCP, it treats the new version as temporarily unavailable and retries later, allowing seeding to complete before datacenter-local replication begins. This avoids serializing the entire pipeline behind slow cross-datacenter transfer.

To further reduce GPU stall time for seeding replicas, TensorHub offers an *offload seeding* option. When seeding is required, the client library creates a temporary offload buffer in CPU memory and initiates cross-datacenter transfer into this buffer in the background. The `update()` call returns immediately, and subsequent polling prioritizes consuming the offloaded copy if completed. This approach trades a



**Figure 4.6: Sharding Consistency Example.** Left: the physical request order lets replica-0’s shards observe different latest versions: shard:0 sees version 12 at  $T_0$ , while shard:1 sees version 13 after replica-1’s two shards publish it at  $T_1$  and  $T_2$ . Without coordination, this difference can cause SPMD control-flow divergence. Right: TensorHub serializes replica-0’s replicate requests as one transaction, so both shards observe version 12.

bounded amount of CPU memory for reduced GPU stall time by creating at most one offload seeding replica per datacenter.

#### 4.3.4 Consistency for Model-Parallel Sharding

LLM weights often exceed the memory of a single GPU, so model parallelism is required. In this setup, weights are sharded across a group of workers that must execute identical code, following an SPMD (single-program, multiple-data) paradigm. Divergence in a group risks hanging or corruption.

For correctness, all workers in a model-parallel group should ideally observe a consistent view of system states, such as whether version  $N$  is available or which version is the latest. In practice, however, requests from different workers may reach the server in an interleaved order and observe inconsistent states. Figure 4.6 shows one such example.

TensorHub guarantees that all workers in a model-parallel group (i.e., replica) observe the same control-plane states, thereby avoiding inconsistent decisions and SPMD control-flow divergence. The server implements this guarantee with transactions at replica granularity. When the first request from a group arrives, the server starts a transaction, evaluates the operation for the entire group, and stores the resulting information (e.g., the latest version) in a transaction-local buffer. Subsequent

requests from that group consume this information. Transactions are serialized in their start order.

TensorHub does not enforce control-plane consistency across replicas. Each replica polls TensorHub independently and may observe a newer version at a different time. This behavior is acceptable for asynchronous, off-policy RL workloads, which allow rollouts to use slightly stale weights [68]. If cross-replica agreement is required, the RL framework can simply broadcast a message to all workers to pull a specified version from TensorHub.

### 4.3.5 Robustness to Churn and Failures

Large-scale training runs on clusters with highly variable reliability. Rollout workers on spot instances can be preempted at any time, and even non-spot nodes occasionally fail at scale. Our storage abstraction must therefore assume frequent replica churn and provide correctness without manual recovery. TensorHub is designed for this failure model: it detects failed replicas, safely re-routes ongoing replication, and integrates cleanly with our retention semantics.

**General Failure Handling.** TensorHub considers a client failed if no heartbeat arrives within the timeout window. Failure is handled at the replica granularity: the server removes the entire failed replica and releases associated resources.

Failures can occur during active replication. If a replica fails while fetching data, the timeout causes the server to invalidate the incomplete replica and release the reference count it acquired. Conversely, if a replica fails while acting as a source, the receiving replica detects the transfer failure and reports it to the server. The server marks the source as failed, preventing future replication from it, and selects an alternate source to recover the transfer.

**Retention Under Frequent Churn.** Spot instances change the failure model: failures are not rare events but the norm. To ensure retention with spot instances, when counting the replicas that satisfy a retention rule, TensorHub excludes replicas hosted on spot instances.

In the pathological case where the last non-spot replica of a retained version fails, the system can no longer serve that version. In that event, TensorHub returns a graceful error indicating that the version is unavailable, and the client can retry later

on another version. This behavior is acceptable in RL training because a new version will be trained and published shortly. The consequence of this corner case is a brief delay, not loss of training progress.

**Reference Server Failure.** Because the reference server maintains only soft state (e.g., references), recovery is simple and lightweight. Upon detecting a server failure, a client resets its state to unpublished and connects to a preconfigured backup server. Critically, this new server does not need to recover any state from its predecessor; it simply waits to be populated by the next round of publishing from trainers. Before the new server is populated, rollouts continue using the existing weights without interruption. Eventually, all clients notice the failure and switch to the new server.

A special case arises when the server fails in the middle of a replica transaction, so some shards complete the operation while others observe the failure. To avoid divergence, the backup server uses a per-replica barrier: it waits for all shards to switch before allowing the replica to proceed. If some shards do not switch within a timeout, it returns an error to the RL framework, which can then restart this replica.

### 4.3.6 Implementation

TensorHub is implemented with 3.7 K lines of Python and 2.4 K lines of C++ code. The reference server is implemented as a Ray [67] named actor, instantiated lazily upon the first client call to `open()`. The TensorHub transfer engine is built on the Mooncake transfer engine [78] for RDMA.

**Consistency Testing.** To validate the server’s consistency guarantees, we construct unit tests in which a single test process issues requests on behalf of multiple clients to deterministically simulate arbitrary request interleavings. Failures are injected by simply halting requests for a given client. These tests operate solely at the reference-request layer, without involving data transfer, and therefore do not require RDMA NICs or GPUs. Because all requests originate from a single process, the resulting executions are deterministic and reproducible, greatly simplifying debugging. This approach to simulated concurrency testing is inspired by FoundationDB [137], which has demonstrated in production its effectiveness at uncovering subtle concurrency bugs in distributed systems.

**End-to-End Checksum.** To verify end-to-end correctness, we further implement checksums. Upon publishing, the client computes the checksum of each tensor and attaches it to the reference. When another client acquires this reference as a data source, it validates the checksum after transfer. In addition, these checksums help confirm correct enforcement of the mutability contract without corruption. Since tensors typically reside on GPUs, checksum computation is fast and can often be overlapped with RDMA transfer.

## 4.4 Evaluation

Our evaluation answers five questions: (1) *Can TensorHub transfer weight tensors efficiently and fully utilize RDMA bandwidth?* (2) *Can TensorHub scale replication throughput under bursty requests?* (3) *Can TensorHub mask failure transparently?* (4) *Can a single reference server sustain control-plane operations at scale?* (5) *Does TensorHub uniformly handle a variety of realistic RL workloads?*

We begin with microbenchmarks to evaluate TensorHub’s data-plane efficiency, scalability under bursts, resilience to failure, and reference-server capacity. We then present three case studies of distinct rollout workloads, focusing on (i) standalone rollouts, (ii) elastic rollouts on spot instances, and (iii) cross-datacenter rollouts, showing how TensorHub optimizes transfer with topology, supports dynamic cluster membership, and minimizes worker coordination, all without extra data movement or space.

**Correctness and Validation.** We validate the correctness of all experiments using the mechanisms described in §4.3.6. We implemented 1.3 K lines of concurrency tests that cover model-parallel consistency and failure cases. Each experiment runs a separate pass to verify end-to-end checksums on weights, and we trained real models to confirm that loss and accuracy match those from ByteDance production.

**Hardware Specification.** Unless otherwise specified, our experiments use machines with 160 CPU cores, 8 NVIDIA Hopper GPUs, 1800 GB of DRAM, 4 Mellanox InfiniBand RDMA NICs (400 Gbps), and 1 VPC NIC (200 Gbps). The reference-server microbenchmark (§4.4.1.4) uses a CPU-only cluster with 32 CPU cores and 32 GB of DRAM per machine.

### 4.4.1 Microbenchmarks

We use four microbenchmarks to evaluate TensorHub’s data-plane efficiency, scalability under bursts, failure resilience, and reference-server capacity. Unless otherwise specified, each worker group consists of 8 shards on an 8-GPU machine.

#### 4.4.1.1 Data-Plane Bandwidth Efficiency

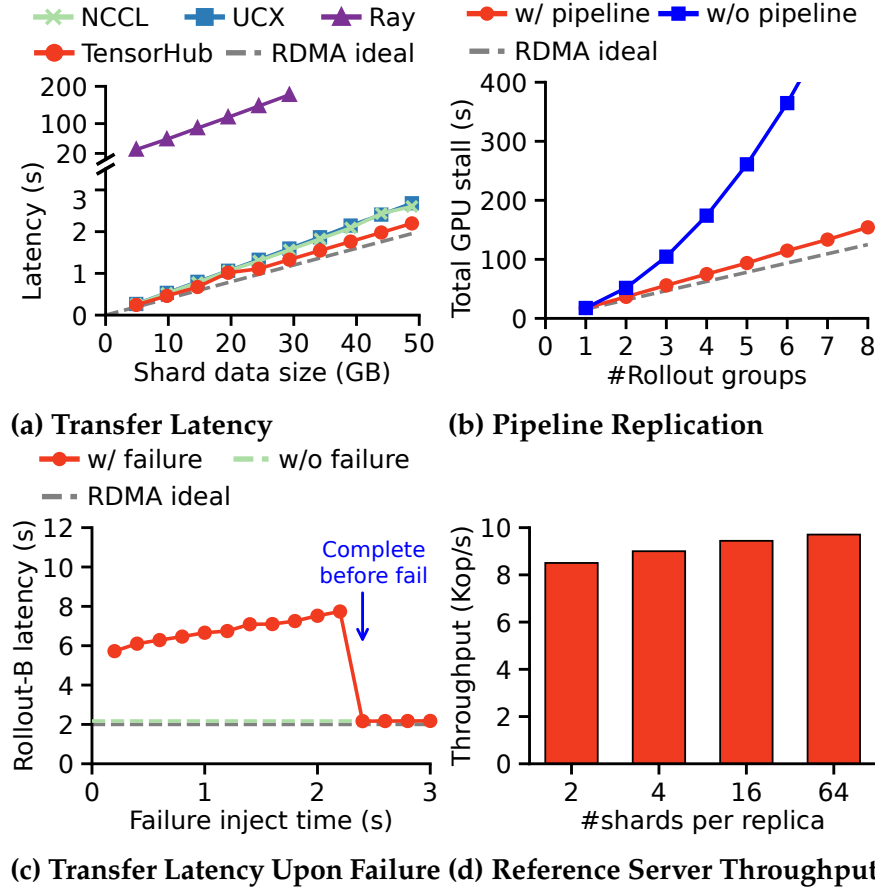
In the first microbenchmark, one machine serves as the trainer group and sends tensors, while another serves as the rollout group and receives tensors from the corresponding shards. We vary the number of tensors per shard to control shard size (50 MB per tensor) and measure the latency required to complete the transfer. We compare with NCCL [69], UCX [98], and the Ray object store [67], and show an additional *RDMA ideal* curve that denotes the theoretical number if the RDMA bandwidth is fully saturated (25 GB/s per shard).

Figure 4.7(a) shows the results, where TensorHub consistently delivers the lowest latency. The Ray object store is two orders of magnitude slower than other approaches because it is primarily optimized for CPU memory and TCP, incurring prohibitive costs for GPU–CPU movement and serialization. In addition, Ray crashes when the per-shard size exceeds 35 GB. Focusing on the three direct transfer approaches, TensorHub transfers 50 GB of data in 2.2 s with 22 GB/s throughput (88% of the theoretical), while NCCL and UCX only achieve 18.8 GB/s and 18.1 GB/s. These results show TensorHub’s data-plane implementation is highly efficient and can saturate RDMA bandwidth.

#### 4.4.1.2 Scaling With Bursts

In the second microbenchmark, we evaluate TensorHub’s scalability under bursty request arrival. We use one machine for a trainer group and 1–8 machines for rollout groups; each shard contains 50 GB of data. All rollouts simultaneously request replication.

We define the *total GPU stall time* as the sum of stall times over all GPUs involved in the replication. This metric reflects the aggregated compute resource loss across the cluster. We report the total GPU stall time for both the default TensorHub and



**Figure 4.7: Microbenchmark Results.**

the one with pipeline replication disabled, together with the RDMA ideal number for reference.

Figure 4.7(b) shows total GPU stall time. With pipeline replication, the trainer group serves only one rollout group, and the remaining rollouts fetch peer-to-peer. Each shard transfers 50 GB in 2.2 s, independent of replica count. Total stall time scales linearly with the group count and remains close to the RDMA roofline. Without pipeline replication, all replicas fetch from the trainer, causing bandwidth contention. Stall time grows quadratically with group count. This scaling gap demonstrates that pipeline replication is necessary for efficient weight dissemination at scale.

#### 4.4.1.3 Transparent Failure Masking

We evaluate the system’s resilience to failure with the third microbenchmark. Since failure at rest does not impact performance, we focus on failure during a transfer. A resilient system should mask failure without disrupting other healthy workers.

To evaluate this, we construct a pipeline where a trainer group sends data to rollout group A, which then forwards data to rollout group B. Both rollout groups start simultaneously. Each shard has 50 GB of data. We inject a failure into rollout-A sometime after the transfer has begun, and then measure whether rollout-B can complete the transfer without interruption, as well as the total time required.

Figure 4.7(c) presents the results, where rollout-B always completes the transfer. When the failure occurs immediately after the transfer starts, rollout-B quickly detects it and retries. The RDMA layer takes a conservative timeout of  $\sim 4$  s before B confirms A’s failure and requests recovery from the server. The server schedules B to directly read from the trainer to complete the transfer. For failures injected at later points, B takes longer to complete because retransmission is delayed. If the failure is introduced after 2.2 s, rollout-B has already finished the transfer and is therefore unaffected. These results demonstrate that TensorHub can transparently mask failure with only minor delay.

#### 4.4.1.4 Reference-Server Capacity

In the last microbenchmark, we measure the sustained throughput of a single reference server. This benchmark isolates the control plane: clients issue requests but perform no RDMA transfers. We use one machine to run the server and 64 machines to run a total of 8192 clients. Each client submits publish, update, and unpublish requests in a closed loop. We group clients into replicas and vary the number of shards per replica.

Figure 4.7(d) shows that a single server sustains 8.5–9.7 Kop/s. This capacity is sufficient for the control-plane load of RL clusters with thousands of workers. Throughput does not degrade as replica size increases, showing that model-parallel transactions add little overhead. Larger replicas reduce the number of replicas and the associated per-replica state, yielding slightly higher throughput.

| Model size       | 9B | 36B | 260B | <i>mocked</i> 1T |
|------------------|----|-----|------|------------------|
| # shards         | 2  | 4   | 8    | 16               |
| Shard size (GB)  | 10 | 19  | 34   | 66               |
| Trainer #GPUs    | 16 | 16  | 64   | 768              |
| Standalone #GPUs | 8  | 8   | 16   | 256              |

Table 4.3: Training Workload Parameters.

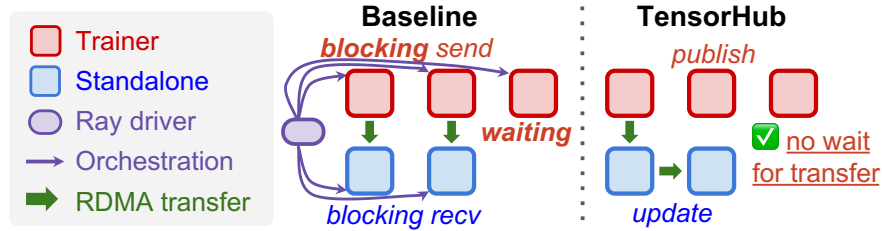
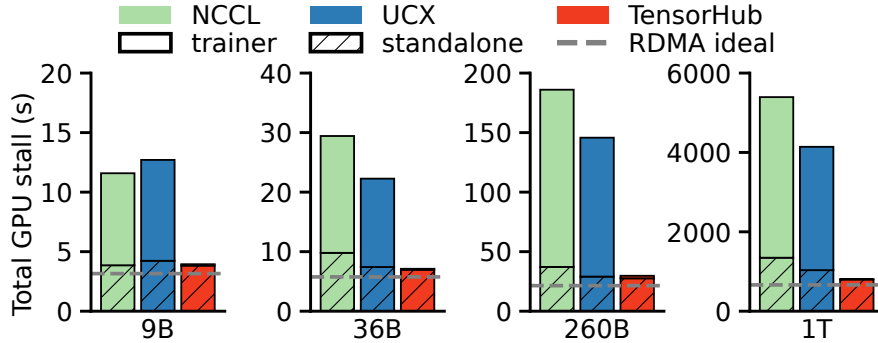


Figure 4.8: Weight Transfer Flows With Standalone. TensorHub does not require the Ray driver to orchestrate weight transfer. Each standalone rollout pulls weights on demand.

#### 4.4.2 RL Training With Standalone Rollouts

We now evaluate TensorHub in realistic RL settings. We use the ByteDance production framework, which extends veRL’s [102] co-located architecture with standalone rollouts. The framework relies on a Ray driver to orchestrate weight transfer across workers, which is a common and representative design of RL frameworks [34, 47, 102].

**Baseline: NCCL and UCX.** We compare against two weight transfer systems used in the ByteDance production stack: one based on NCCL [69] and another based on UCX [98]. As discussed in §4.1.3, the communication-based weight transfer systems require the RL framework to coordinate workers. In practice, it is implemented as an RPC broadcast that interrupts all workers for a global weight transfer stage. More fine-grained coordination is theoretically possible, but it requires a major refactor of the framework. Though we are unable to compare with every possible implementation, we again include a roofline curve for reference (labeled “RDMA ideal”), denoting an ideal case with zero coordination cost or contention, bottlenecked only by the rollouts’ RDMA bandwidth. Performance close to this reference implies near-optimal results. We did not include a comparison with other storage-based approaches because we have shown in §4.4.1 that they are orders of magnitude slower.

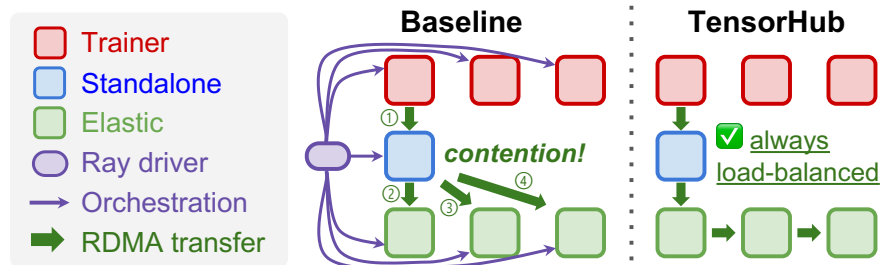


**Figure 4.9: Standalone Rollout Results.** Ideally, only standalone rollouts need to stall due to their dependence on weights; trainers can proceed without waiting. TensorHub not only eliminates trainer stalls, but also keeps standalone stall time consistently lower.

**Engineering Effort for Integration.** NCCL requires ~450 lines of code changes to the ByteDance production RL framework, mostly for coordination. UCX, with more low-level control, requires ~1200 lines, trading simplicity for flexibility. TensorHub requires only ~40 lines of change (similar to the example in Figure 4.4). This highlights that TensorHub offers a simple and user-friendly interface.

**Workloads.** We train four models for evaluation (Table 4.3), covering various sharding and GPU count setups. The shard count is the number of GPUs in one model-parallel replica, and the total GPU counts determine the number of trainer and standalone replicas. For example, the 260B configuration uses 8-shard replicas, yielding 8 trainer replicas and 2 standalone replicas. The 9B, 36B, and 260B models are initialized from pretrained weights. To stress the system at an extreme scale, we construct a 1T-parameter model with mocked weight data by duplicating the 260B-parameter model’s layer weights four times: training such a huge model requires 768 GPUs for trainers and 256 GPUs for standalone rollouts (1024 GPUs in total).

**Results.** Figure 4.8 compares the workflows of the baselines and TensorHub, and Figure 4.9 shows that TensorHub reduces total GPU stall time by up to 6.7× compared with NCCL and by 5.2× compared with UCX. These reductions stem from TensorHub’s minimal-coordination design. First, `publish` is a lightweight reference-passing request that does not wait for actual data transfer; trainers immediately resume for their own co-located rollout tasks. In contrast, NCCL and UCX require



**Figure 4.10: Weight Transfer Flows With Standalone and Elastic.** Note that TensorHub only shows one possible data flow; an elastic rollout can also fetch weights from a trainer.

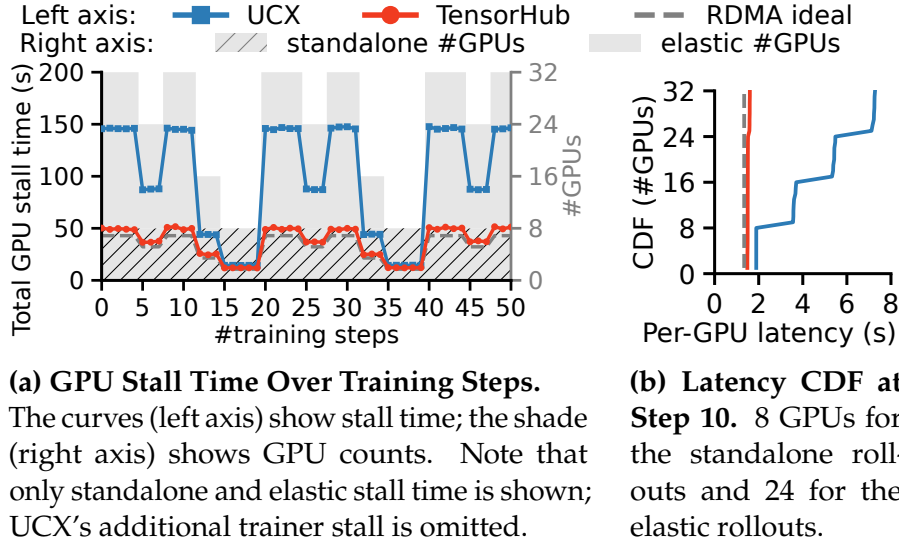
coordination by the framework’s Ray driver, and the driver does not move forward to its next stage until the weight transfer stage is completed.

Second, for large models, stragglers are amplified at scale: the weight transfer stage completes only when the slowest worker finishes. For the 1T model, transferring 66 GB should ideally take 2.6 s at full bandwidth, but NCCL and UCX stall all 1024 GPUs for 5.3 s and 4.0 s, respectively. TensorHub localizes the stall: mean latency is 3.1 s, and only the standalones (256 GPUs) need to wait. This demonstrates the benefit of decoupling trainer and rollout control flow.

### 4.4.3 Elastic Rollout on Spot Instances

In the second case study, we evaluate TensorHub when rollout workers run on a hybrid of stable GPUs (for standalone) and preemptible spot GPUs (for elastic). In the ByteDance production environment, an autoscaler monitors training progress and instantiates new elastic rollouts when rollout tasks are backlogged and spare spot GPUs are available. When their GPU resources are preempted, elastic rollouts are killed immediately without a grace period.

**Baseline.** Only UCX-based weight transfer supports elastic rollout, because NCCL assumes static membership. Supporting elastic rollouts took substantial engineering effort for the ByteDance production UCX-based implementation. When a new elastic worker appears, trainers may be mid-training and unable to serve weights, so standalone rollouts must also act as data senders. This introduces a trainer-to-standalone-to-elastic transfer topology.



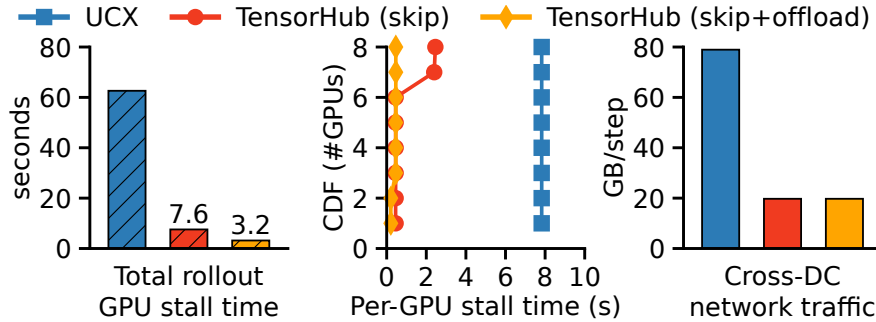
**Figure 4.11: Elastic Rollout Results.**

**Workloads.** We use the 260B model described in Table 4.3, but with one standalone machine (8 GPUs) and up to three elastic machines (24 GPUs). To keep experiments reproducible, we intercept the autoscaler and deterministically issue scale-up and scale-down events. Randomly chosen elastic replicas are terminated during scale-down.

**Results.** Figure 4.10 compares the workflows, and Figure 4.11(a) shows total stall time over training steps with GPU count varied. To isolate rollout performance, we report stall time only across standalone and elastic rollout GPUs, omitting trainer stalls for both methods.

In the UCX baseline, elastic rollouts must wait for standalone rollouts to pull weights from trainers first, introducing a delay. When elastic workers outnumber standalones, bandwidth contention further amplifies tail latency. Figure 4.11(b) shows per-GPU stall time at step 10: UCX produces a stair-shaped CDF, with the last batch of elastic GPUs waiting up to 7.2 s for weights.

With TensorHub, the reference server balances load with topology awareness at runtime, so elastic rollouts can fetch weights directly from trainers, standalones, or peer elastic rollouts, and pipeline replication spreads load across all available replicas. As a result, stall time remains near-constant ( $\sim 1.5$  s) independent of the elastic GPU count, approaching the RDMA ideal. TensorHub reduces transfer latency by up



**Figure 4.12: Cross-Datacenter Rollout Results.** Note that the left figure shows only the rollout stall time; UCX incurs additional trainer stall time that is omitted here.

to 4.8 $\times$  compared with UCX. This result demonstrates that TensorHub naturally accommodates dynamic membership and runtime topology.

#### 4.4.4 Cross-Datacenter RL Training

The final case study evaluates TensorHub in a multi-datacenter deployment. This setting reflects production scenarios where spare inference resources are available in a separate datacenter, and utilizing them requires weight transfers over slower inter-datacenter networks.

**Baseline.** We use a UCX-based implementation with TCP transport as the baseline. The transfer topology is similar to that of standalone rollouts (Figure 4.8), except the data path is now cross-datacenter over TCP.

**Workloads.** We use the 9B model from Table 4.3, with eight 2-shard trainer replicas (16 GPUs) and four 2-shard rollout replicas (8 GPUs). Trainers are located in a datacenter with training-optimized Hopper GPUs; rollouts run in a second datacenter with inference-optimized Hopper GPUs. The two datacenters are connected only via the 200 Gbps VPC NIC (one per machine) with TCP.

**Results.** Figure 4.12 reports total stall time across rollout GPUs, per-GPU latency distribution, and total cross-datacenter TCP traffic. TensorHub reduces total rollout GPU stall time by up to 19 $\times$  compared with UCX. With the UCX baseline, every rollout replica fetches weights directly over TCP. Since all flows contend for the single VPC NIC per machine, stall time is dominated by cross-datacenter bandwidth: each

GPU waits 7.8 s for weights. Such waiting is costly, given that an end-to-end training step takes only 68 s.

With TensorHub and smart skipping, only the first rollout replica (2 GPUs) waits for cross-datacenter TCP transfer (2.5 s), and the contention on the VPC NIC is reduced. Other rollout replicas can fetch data via high-throughput RDMA with merely 0.45 s latency. The result is a latency distribution with a single long tail for the seeding replica and near-ideal RDMA latency for all others. The end-to-end training step time is reduced to 60 s, 13% faster than UCX.

With offload seeding enabled, TensorHub further hides the cross-datacenter transfer latency by using an offload replica. The seeding replica stalls only for the 0.21 s PCIe transfer. In summary, TensorHub exploits datacenter locality and achieves near-local performance after a single cross-datacenter seeding transfer.

## 4.5 Experience and Lessons

We share our experience and lessons learned when operating the weight transfer systems in production.

**Clean Interfaces Avoid Deadlocks.** Weight transfer introduces synchronization across worker groups and hence is vulnerable to deadlock. In earlier NCCL/UCX-based weight transfer systems, we observed a series of related bugs. For example, one deadlock arose from a circular dependency between the Ray driver and workers: the driver asked all rollouts to stop; a standalone rollout acquired a write lock on its weights, while a bootstrapping elastic rollout blocked trying to read those weights from the standalone and could not respond to the driver; the driver then waited indefinitely. Such bugs arise because weight transfer creates a large critical section, and each worker runs customized logic to enter it, potentially introducing more dependencies.

In contrast, TensorHub APIs avoid this dependency pattern. `publish` is non-blocking; `update` returns once the transfer is complete; and `unpublish` waits only for ongoing readers to drain. Each API has fixed blocking behavior to minimize the critical section. No external, framework-specific coordination logic enters the critical section, preventing circular dependencies by design.

**Decoupling Isolates Delays.** TensorHub’s decoupled interfaces also help to isolate the impact of suboptimal implementations. For example, in earlier systems we noticed cases where a trainer incorrectly ran manual garbage collection or expensive weight preprocessing in the middle of a weight transfer stage, causing unnecessary stalls for rollouts. TensorHub APIs block a rollout only for its transfer time, preventing unrelated preprocessing from ever entering the critical section.

## 4.6 Conclusion

In this chapter, we present Reference-Oriented Storage (ROS), a new approach to weight transfer in large-scale LLM reinforcement learning systems. By eliminating explicit data ownership and instead leveraging the inherent redundancy and immutability of model weights, ROS reconciles a long-standing trade-off between flexibility and efficiency in RL weight transfer systems. Building on this abstraction, TensorHub enables topology-aware, low-overhead weight transfer with minimal coordination, while preserving model-parallel consistency and fault tolerance in dynamic environments. Our evaluation shows that TensorHub can saturate network bandwidth and deliver near-optimal end-to-end performance. TensorHub has been deployed in ByteDance production for cutting-edge RL training.

# Chapter 5

## Related Work

In this chapter, we discuss how previous work relates to the contributions of this dissertation. First, we cover prior approaches to multi-tenant resource sharing and cache allocation (§5.1). Next, we discuss existing scalable logging and replication systems (§5.2). Finally, we describe related work in machine learning systems and peer-to-peer data dissemination (§5.3).

### 5.1 Resource Allocation

Multi-resource allocation and cache allocation have long developed as separate research lines: the former does not capture the complex correlations between cache space and other resources, while the latter optimizes the cache in isolation. HARE bridges the two by exploiting this very correlation across cache and other resources.

**Multi-Resource Allocation.** DRF [38] is one of the earliest studies addressing fair allocation across multiple resource types. A series of variants have since been proposed, each tailored to different system settings. DRFQ [37] applies DRF to network packet queueing. HUG [23] optimizes utilization when demands are elastic on non-dominant resources. Choosy [39] presents a fair cluster job scheduler that incorporates placement constraints. Choi et al. [22] propose a lifetime-aware DRF for flash-based caching systems. Synergy [66] jointly allocates GPUs, CPUs, and memory across deep-learning jobs. All of these frameworks focus on resources without negative correlation.

MIRA [108] extends DRF to model interchangeable resources (e.g., CPUs and GPUs), a case of linear correlation where one resource substitutes for another by a fixed factor. In contrast, HARE focuses on cache correlation, which is non-linear: a change in cache size shifts the demands on every other resource through the miss ratio curve.

**Cache Allocation.** Multi-tenant caches have been extensively studied. A long line of work allocates cache as a standalone resource and optimizes only for miss ratios [1, 25–27, 49, 76]. Each of these systems improves the cache’s own efficiency, but misses the opportunity to exploit the correlation between caches and other resources.

Several systems capture cache correlation, but only with one correlated resource. Moirai [107] and Centaur [55] treat cache as the primary knob and passively report the resulting backend I/O. Argon [113] takes the opposite view, allocating I/O explicitly and passively reporting the cache the workload requires. As a result, these systems miss the opportunity to jointly allocate both cache and I/O. CoPart [72] shares HARE’s vision of joint allocation, but is limited to the single correlation between last-level cache and memory bandwidth. The concurrent work Spirit [57] jointly allocates cache and network bandwidth, again with only one cache-correlated resource in scope. In contrast, HARE allocates cache jointly with multiple correlated resources and captures their non-linear interactions through tenant-specific miss-ratio curves.

## 5.2 Scalable Logging and Replication

A few lines of research have been devoted to improving the scalability of logging and replication prior to MultiLog.

**Replication for Memory-Centric Stores.** Replication is widely used in memory-centric stores [20, 31, 87, 105, 111] to scale out throughput of read-dominant workloads. Existing replication schemes fall into two categories that trade off multi-core scalability against consistency guarantees. Redis [87], Valkey [111], and KeyDB [105] funnel all write traffic through a single totally ordered replication log. This design simplifies prefix consistency for replica reads, but scales poorly on multi-core machines. At the other end of the spectrum, Dragonfly [31] partitions the keyspace across threads, each with an independent replication stream. This approach requires more complex client request routing and does not provide cross-thread prefix consistency.

MultiLog reconciles these trade-offs. By sharding replication logs, MultiLog scales up both log append and replay operations. Through lazy coordination between replay and read operations, MultiLog preserves prefix consistency with minimal overhead.

**Parallel Logging for Recovery.** Parallelizing recovery-oriented write-ahead logging has been studied extensively. Aether [54] improves logging performance through early lock release, flush pipelining, and consolidation buffers. SiloR [133] combines per-thread loggers with an epoch-based commit protocol to improve scalability on multi-core machines. Taurus [123] parallelizes both logging and recovery using LSN vectors for dependency tracking.

MultiLog focuses on replication logs, which impose more complex requirements than recovery-oriented logs. Recovery only needs to reconstruct the correct system state by the end of recovery, without any consistency guarantee during replay. In contrast, replication is a continuous replay process that must preserve a consistent view throughout execution.

**Parallel Replication.** Scalable replication has been explored in both industry and academia, but no existing system satisfies the desired goals of memory-centric stores described in §3.1.2. PostgreSQL [74], MySQL [70], and MariaDB [62] support multi-threaded log replay. However, replay threads still serialize during their commit phases, which limits scalability. MySQL and MariaDB additionally support an out-of-order commit option, which improves scalability at the cost of prefix consistency guarantees.

KuaFu [127] and Query Fresh [118] parallelize replay without the commit-phase serialization bottleneck. KuaFu tracks transaction dependencies by analyzing write sets and replays independent transactions concurrently. Query Fresh directly uses the log stored in non-volatile memory as the database itself and maintains lightweight metadata to chain versions. Both systems rely on MVCC to serve prefix-consistent reads by retaining stale but stable versions. Memory-centric stores, however, are typically single-versioned and rely on in-place updates for high performance, so these approaches are not directly applicable.

C5 [46] is a state-of-the-art parallel replay protocol. It employs a centralized scheduler thread to dispatch log records to replay threads. As the number of replay threads increases, the scheduler becomes a scalability bottleneck. In addition,

C5 requires periodic snapshotting, which is costly given the high performance and freshness requirements of memory-centric storage. MultiLog fully eliminates the centralized scheduling bottleneck to improve scalability. With the lightweight PrefixLoom protocol, MultiLog does not require expensive snapshotting.

MultiLog focuses on asynchronous replication, which is the dominant replication model for memory-centric storage [20, 31, 87, 105, 111]. A separate line of work targets scalable synchronous replication, commonly in the context of consensus and state-machine replication, including Rolis [100], Rex [44], and Qin et al.’s replay-based replication [77]. These synchronous systems primarily use replication for durability rather than for scaling out read throughput. As a result, they either do not support replica reads [44, 77, 100] or provide no consistency guarantees across multiple replication streams.

### 5.3 Machine Learning Systems and Data Dissemination

In this section, we cover LLM reinforcement learning frameworks, which require highly efficient weight transfer systems and directly benefit from TensorHub. We then describe prior storage systems customized for machine-learning workloads. Finally, we discuss peer-to-peer data dissemination, a line of work that shares ROS’s approach of serving reads from peers rather than from dedicated servers.

**Reinforcement Learning for LLMs.** Many frameworks [47, 102, 136] have been proposed to accelerate RL training. veRL [102] combines single-controller and multi-controller paradigms to drive stage execution efficiently. RLHFuse [136] further improves training compute efficiency with stage fusion. RollPacker [36] optimizes the long-tail decoding in rollout, while SpecActor [21] introduces decoupled and Best-of-N speculation to accelerate the rollout phase, ensuring on-policy model training. RLBoost [122] harvests preemptible GPUs to speed up the rollout phase with resource elasticity. Recent work also explores off-policy approaches for training acceleration. AsyncRLHF [68], AReaL [34], StreamRL [135], and Laminar [101] introduce asynchronous RL post-training with tailored optimizations to increase job throughput. TLT [48] trains a draft model in parallel for speculation when RL model training is performed; thus, the draft model also requires weight synchronization.

These frameworks optimize RL scheduling, which increasingly requires efficient weight transfer; they can thus benefit from TensorHub’s clean weight transfer abstraction.

**Storage Systems for Machine Learning.** Prior studies have built storage systems for machine learning workloads. Parameter Server [59] stores model weights across a cluster of servers. Ray [67] provides an object store for RL workloads, but is primarily optimized for CPU workloads. Hoplite [140] extends Ray with fault-tolerant collective primitives. Unlike TensorHub, these systems require a system-owned copy of model weights, which is prohibitively expensive for LLM weights at TB scale.

Checkpointing systems are another class of storage for weights. Unlike RL weight transfer, they retain model states for recovery and are less latency-sensitive. GeminiFS [79] accelerates checkpointing and activation offloading by optimizing the metadata through a POSIX-like interface. ByteCheckpoint [117] is designed for efficient model weight storage and supports dynamic resharding across different training parallelism configurations. Gemini [119] stores in-memory checkpoint replicas for faster failure recovery.

**Peer-to-Peer Data Dissemination.** The idea of serving data requests from peer clients instead of servers dates back to cooperative caching [30] and xFS [11]. However, retaining additional copies on clients is too costly for LLM weights. TensorHub avoids this space overhead by borrowing the application-owned copies via a mutability contract and a retention protocol. Ekko [104] disseminates recommender-model updates peer-to-peer across WANs, exploiting their sparsity and locality with eventual consistency. In contrast, TensorHub focuses on transferring dense LLM weights on the performance-critical path of RL workloads.

## Chapter 6

# Conclusions and Future Work

In this chapter, we first summarize each part of the dissertation (§6.1). Then we share the lessons learned throughout the journey (§6.2), discuss the future work (§6.3), and finally conclude (§6.4).

### 6.1 Summary

This dissertation has investigated how to improve the *scale* and *performance* of memory-centric storage systems across three contexts: multi-tenant resource sharing within a node, primary-replica replication across nodes, and emerging LLM reinforcement learning workloads. We summarize each contribution in turn.

#### 6.1.1 Resource Efficiency and Fairness Within a Node

In the first part of this dissertation, we studied how memory-centric storage manages resources within a single node, focusing on the caching use case. Memory-centric storage systems such as Redis [87] and Memcached [33] are widely deployed as caching layers for backend persistent databases, and must manage not only memory capacity for caching but also auxiliary resources such as network bandwidth and I/O for request serving and cache-miss handling. To improve resource utilization, these systems are commonly shared among multiple tenants, and must therefore pursue both *resource efficiency* and *fairness*. Cache space is a special type of resource: it directly impacts a tenant’s demand for other resources, since a tenant with a

larger cache and a lower miss ratio reduces its demand for I/O and backend service capacity. Existing multi-resource allocation frameworks such as Dominant Resource Fairness [38] and its variants [23, 37, 108] do not incorporate these correlations, and prior cache-allocation frameworks [1, 15, 25–27, 32, 49, 76, 80] focus solely on cache allocation and therefore miss opportunities to exploit the demand correlations.

We presented HARE, a cache-centric multi-resource allocation algorithm that fully exploits the demand correlations between cache and other resources. Based on the observation that tenants do not benefit from cache equally, HARE allocates resources in two phases. In the *harvest* phase, HARE searches for a cache partition under which every tenant maintains its fair-share throughput while the aggregate consumption of other resources decreases. In the *redistribute* phase, HARE reallocates the harvested resources to improve the throughput of every tenant. We demonstrated HARE’s generality with two systems: HopperKV, a cloud-native key-value cache that modifies Redis to cache data from backend DynamoDB, and BunnyFS, a microkernel-style local filesystem for NVMe SSDs. Both systems require no prior knowledge of tenants’ demands and can efficiently learn the demands and adapt to dynamic workloads on the fly. HopperKV achieves up to 1.9× higher throughput, and BunnyFS achieves up to 1.4×.

### 6.1.2 Scalability and Consistency Under Replication

In the second part of this dissertation, we studied how memory-centric storage scales out under replication while keeping replicas consistent with the primary. In the context of a replicated memory-centric store, a primary instance serializes every mutation into a write-ahead log. The log is continuously shipped to read-only replicas, which replay the log to catch up. The conventional single-log design introduces two serialization bottlenecks. At the primary, concurrent worker threads must coordinate on the log tail to allocate space, capping append throughput. At the replica, a single replay thread cannot match the primary’s concurrent write rate; under high write loads, replication lag grows without bound and destabilizes the system. Existing approaches force a trade-off: single-log designs [20, 87, 105] preserve prefix consistency but fail to scale; naive sharding-based designs [31] scale but break prefix consistency. Parallel-replay protocols built for relational databases [46, 118, 127]

preserve consistency via MVCC or periodic snapshotting, neither of which fits the single-version design of memory-centric storage.

We presented MultiLog, a highly scalable replication architecture for memory-centric storage that resolves this trade-off. At its core lies PrefixLoom, a lightweight replica-side protocol that serves prefix-consistent reads during parallel replay. Rather than relying on MVCC or snapshotting, PrefixLoom uses *lazy coordination* between replay and read threads to weave a prefix-consistent view out of a continuously updated, never-consistent replica image. Building on PrefixLoom, MultiLog further shards the log into multiple physical sublogs, each owning a disjoint keyspace and maintaining its own log-space allocator and replication stream; each append thread routes its own record to the corresponding physical sublog based on the key hash. Implemented in Garnet [20], MultiLog improves append throughput by up to 19× and replay throughput by up to 36× over a single-log baseline. Under a write-heavy replication workload, MultiLog sustains 3.8× higher primary throughput than the single-log baseline and 2× higher replica read throughput than C5 [46]. Its replica stays within ~10 ms of the primary at p99.9, two orders of magnitude fresher than either baseline.

### 6.1.3 Memory-Centric Storage for LLM Reinforcement Learning

In the last part of this dissertation, we examined memory-centric storage in the context of LLM RL workloads. RL training introduces highly dynamic and data-intensive execution patterns: in each training step, trainer workers commonly transfer terabytes of model weights to thousands of rollout workers. Existing LLM RL frameworks typically transfer weights using NCCL collectives [69] or UCX point-to-point communication [98]. However, NCCL is brittle under worker failures, whereas UCX suffers from fan-out contention; both require temporally coupled coordination at the framework level. Distributed storage offers a cleanly decoupled alternative: trainers *push* weights to storage, while rollout workers *pull* them on demand. Yet existing systems such as parameter servers [59] and the Ray object store [67] incur prohibitive memory overhead and redundant data movement from push-then-pull transfers.

We presented *Reference-Oriented Storage* (ROS), a new approach that retains the decoupling advantages of storage interfaces while eliminating the overhead of extra memory space and data movement. ROS presents the illusion that certain weight versions are stored and can be fetched on demand. Underneath, it stores no copies; instead, it tracks workers that hold the weights in GPU memory for inference. Building on the ROS idea, we built TensorHub, a production-quality system providing user-friendly interfaces, topology-aware transfer, model-parallel consistency, and fault tolerance. TensorHub reduces total GPU stall time by up to  $6.7\times$  for standalone rollouts, accelerates weight updates for elastic rollouts by  $4.8\times$ , and cuts cross-datacenter rollout stall time by  $19\times$ . TensorHub has been deployed at ByteDance for cutting-edge reinforcement-learning training.

## 6.2 Lessons Learned

In this section, we reflect on a few broader lessons that emerged from the systems built in this dissertation.

### 6.2.1 The Elegance of Storage Abstraction Is Under-Appreciated

The ROS project in this dissertation started from a simple intuition: when transferring model weights in RL, a rollout does not necessarily need to fetch weights from a trainer. If a peer rollout already holds the same weights, fetching from that peer may be more efficient. The challenge is how to support such peer-to-peer sharing without exposing the resulting complexity to the application. One approach would be to add more control flow to the Ray driver [67] to coordinate among workers. However, this approach bloats system complexity, making the system harder to reason about and more prone to bugs.

We found that peer-to-peer data sharing can be implemented naturally behind a storage abstraction. We proposed Reference-Oriented Storage (ROS) and built the TensorHub system. With TensorHub, a rollout simply names the weight version it wants. TensorHub is responsible for locating a copy to serve, whether from the trainer or from another rollout. Critically, unlike traditional storage systems, TensorHub does not store its own copy of the weights. Instead, it tracks the workers that already

hold those weights in GPU memory for inference and uses those workers as data sources. In this way, TensorHub provides a storage interface without maintaining physical copies, which would be prohibitively expensive for terabyte-scale LLM weights.

Throughout the development of TensorHub, we repeatedly observed useful properties of a storage abstraction that are often under-appreciated. First, a storage interface *decouples* data producers from data consumers. Producers can write data into storage and move on. They do not need to know when consumers will read the data or how many consumers there are. In RL training, this means trainers do not need to wait for all rollouts to finish weight transfers. Meanwhile, rollouts can scale up or down dynamically without the trainers’ awareness, which is critical for utilizing spot GPU instances.

Second, a storage abstraction provides a clean scope for optimization. In RL training, workers only need to read from or write to TensorHub, rather than explicitly programming data-transfer flows. Underneath this interface, TensorHub has global knowledge of runtime load and network topology, allowing it to make more informed transfer-scheduling decisions. As we expanded the system to support more use cases, we found that many new workloads could be integrated seamlessly and automatically benefit from earlier optimizations.

Lastly, an easy-to-use interface can produce surprising performance gains. When designing TensorHub, we primarily aimed for concise APIs that were easy to reason about. We did not expect large performance improvements, because we assumed that existing NCCL- and UCX-based weight transfer systems in production already saturated RDMA bandwidth. However, after deploying TensorHub in production, we found that prior implementations were often too complicated to reach their ideal performance: tangled control flows introduced unexpected stalls, buried deeply enough that even the original developers were often unaware of them (§4.5). In contrast, TensorHub APIs eliminate this complexity, making ideal performance easier to achieve.

Together, these experiences suggest that the storage interface is valuable not merely because it stores data, but because it gives distributed systems a simple contract for naming data, locating data, and optimizing access to it.

### 6.2.2 Design for Performance, Then Repair Violated Constraints

Many systems design problems can be framed as optimizing performance under a set of constraints. When building clean-slate systems, we found that some performance loss in prior work came from defensive designs around constraints that were treated as unavoidable, but were in fact overly conservative. For example, in the MultiLog project, prior parallel-replay systems [46, 118, 127] often assumed that MVCC or snapshotting was necessary to provide prefix-consistent reads. In the ROS project, existing storage systems [59, 67] assumed that a storage system must own the data in order to control its lifecycle.

We found a different design methodology useful: start by asking how to achieve ideal performance if those constraints did not exist, and then identify which constraints this ideal design violates. The next step is to repair these violated constraints, making explicit trade-offs where necessary. This methodology reveals how far the system is from ideal performance and forces us to revisit each inherited constraint, rather than accepting overly defensive designs upfront.

In the MultiLog project, ideal replay throughput would come from sharding the log into multiple sublogs, each replayed by a dedicated thread without coordination. However, removing coordination would violate the goal of prefix-consistent reads. We therefore designed the PrefixLoom protocol to restore consistency lazily. The result is that PrefixLoom largely preserves the throughput benefits of independent replay while still providing the desired consistency guarantee.

Another example comes from the ROS project. The ideal weight transfer path should move data directly from the source GPU to the destination GPU, without extra PCIe movement or intermediate network hops through storage servers. However, existing storage systems typically require data to move into storage-managed space, which introduces fundamental overhead. This observation led us to realize that the true cost of conventional storage, in this setting, is data ownership. It eventually inspired us to build Reference-Oriented Storage, which provides a storage interface without owning any physical copy of the data. By doing so, ROS retains the benefits of storage interfaces while preserving ideal data-movement performance.

## 6.3 Future Work

This section outlines several directions for extending the systems and abstractions developed in this dissertation.

### 6.3.1 Distributed Cache-Centric Resource Allocation

In Chapter 2, we presented HARE in a single-node setting. As memory-centric stores are increasingly deployed in cluster mode, it would be valuable to generalize HARE to distributed settings.

In a distributed cluster, the resource allocation framework must decide not only how many resources each tenant should receive, but also where each tenant should be placed. Two challenges must be addressed. First, each node has limited resource budgets, so the framework must resolve placement decisions under per-node budget constraints. When a tenant’s demand cannot be satisfied by a single node, how to split its workload across multiple nodes and properly account for cross-node overhead remains an important question. Second, migration across nodes is more expensive than migration within a single node, as it involves additional network routing and potentially transferring hot cache state. As a result, the framework requires a sophisticated migration mechanism that can smooth the migration process while minimizing degradation of system performance.

### 6.3.2 Fair Resource Allocation for Computational Caches

HARE is designed for fair sharing of *data caches*, where a cache hit avoids data movement from the persistent layer. A natural next step is to explore fair sharing of *computational caches*, where a cache hit saves recomputation. The LLM KV cache used in inference is one of the most prominent examples of such computational caches.

Designing a fair and efficient allocation framework for LLM KV caches is challenging. First, the cost of a cache miss is complex, as it depends on factors such as batching, GPU kernel implementation, and parallelism strategy. Second, the prefix structure of KV caches introduces unique miss-ratio patterns, and shared prefixes across tenants further complicate these patterns. Third, with the increasing adoption of offloading, KV caching effectively becomes a multi-tier caching problem, where

CPU memory and SSDs serve as slower tiers beneath GPU memory. Together, these challenges require a new performance model that can properly account for cache utility and a holistic methodology for trading off offloading against recomputation.

### 6.3.3 Multi-Key Transactions on MultiLog

Chapter 3 presented MultiLog with a focus on single-key reads and writes, which dominate memory-centric store workloads. In practice, however, multi-key transactions are useful for implementing atomic operations without relying on extra coordination through distributed locking services. Therefore, a natural extension is to apply MultiLog to multi-key transactions.

Such an extension could build on MultiLog’s timestamp-based protocol. A write transaction could stamp all of its log records with the same timestamp, while a read transaction on a replica would ensure that all reads observe a single timestamp. The former should be relatively straightforward to implement. The latter requires a careful exploration of the design space for coordination between replay threads and reader threads. Ideally, replay threads should avoid blocking in order to maintain high replay throughput. However, a high replay rate may cause a reader to miss the values needed at a desired timestamp, causing the reader to repeatedly abort and eventually starve. The system therefore needs a careful design that balances replay throughput against reader progress.

### 6.3.4 Unifying Weight Transfer With Checkpointing

In Chapter 4, we presented ROS, a storage abstraction for model weights without persistence, and TensorHub, the system that instantiates it. In LLM training, another important form of weight storage is checkpointing. A checkpointing system selectively persists some versions of model weights to durable storage [17], so that failed nodes or an entire restarted training job can recover from previous training steps.

We believe the clean abstraction of TensorHub can be extended to support checkpointing. When a trainer publishes a new version of weights, it could simply specify whether this version should be persisted. TensorHub could then schedule background jobs to serialize the weights and save them to durable storage. Meanwhile, when a restarted worker requests a desired weight version, it does not need to know

whether the weights come from a peer worker’s GPU memory or from durable storage. TensorHub can choose the data source based on data availability and transfer cost. As a result, such a unified weight-storage system would not only simplify checkpointing logic, but also accelerate recovery when the desired weights are already available in peer workers’ GPU memory.

### 6.3.5 Co-Designed Data Systems for Emerging Data Types

TensorHub is a storage system specialized for one data type: model weights. TensorHub achieves its performance benefits by co-designing the system around the usage patterns of this data type. In RL training, model weights are highly replicated and immutable because of data-parallel inference. This makes it possible to build storage that references existing copies rather than owning its own copy.

As LLMs become increasingly dominant workloads in modern datacenters, several new data types have emerged, such as KV caches, tokens, and logits. These data types flow across GPU clusters and require unified data systems that can transfer them efficiently and robustly through easy-to-use APIs. Since each type has its own production and consumption patterns, it is worth rethinking how to co-design storage abstractions around these patterns. For example, some KV caches may become popular because they are derived from common system prompts, and KV caches may also require large transfers due to prefill-decoding disaggregation [134]. These usage patterns open opportunities for high-performance and scalable data systems specialized for emerging LLM data types.

## 6.4 Closing Words

Data systems have been a defining problem of computing systems for decades, and memory-centric storage has become a critical layer on which modern applications depend. This dissertation has presented three contributions toward improving its scale and performance: HARE for multi-resource efficiency and fairness within a node, MultiLog for scalability and consistency under replication, and ROS for memory-centric storage in LLM reinforcement learning.

This dissertation reflects not only how to improve existing memory-centric storage systems, but also how to build new forms of such systems for emerging workloads. In this journey, we found that the principles of memory-centric storage apply beyond their conventional settings. A workload that initially appears unrelated to storage can often be reframed as a storage problem, opening opportunities for significant performance and scalability gains. We hope these perspectives provide useful starting points for future systems in the era of increasingly data- and computation-intensive applications.

# Bibliography

- [1] Christoph Albrecht, Arif Merchant, Murray Stokely, Muhammad Waliji, François Labelle, Nate Coehlo, Xudong Shi, and C. Eric Schrock. Janus: Optimal flash provisioning for cloud storage workloads. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 91–102, San Jose, CA, June 2013. USENIX Association.
- [2] Amazon Web Services. MailUp Group: Marketing campaign planning and management (Italian). <https://www.youtube.com/watch?v=HwHVFwDczVw>, 2019.
- [3] Amazon Web Services. Mission: Gamified platform scaling with Amazon EKS. [https://www.youtube.com/watch?v=xKaPAihW\\_gE](https://www.youtube.com/watch?v=xKaPAihW_gE), 2019.
- [4] Amazon Web Services. Heimdall Data: Query caching without code changes. <https://www.youtube.com/watch?v=0WLGK-eVrTw>, 2020.
- [5] Amazon Web Services. Ânima Educação: Digitizing student experience for colleges in Brazil. <https://www.youtube.com/watch?v=9yziTe6lBwk>, 2020.
- [6] Amazon Web Services. Love, Bonito: Achieving scalability using Magento with Kubernetes. <https://www.youtube.com/watch?v=BX1K8x1lVLc>, 2021.
- [7] Amazon Web Services. Social Quantum: Using serverless in gaming. <https://www.youtube.com/watch?v=fSV0u48sEVg>, 2021.
- [8] Amazon Web Services. ADP: Unmatched people data for extraordinary outcomes. <https://www.youtube.com/watch?v=KiH7hVJKzns>, 2022.
- [9] Amazon Web Services. Carbon by Indigo: Carbon credits for regenerative agriculture. <https://www.youtube.com/watch?v=FfSNnH2bbNc>, 2022.

- [10] Amazon Web Services. Salesflo: Transforming field-force operations using event-driven architecture. <https://www.youtube.com/watch?v=qi017F1UwvM>, 2023.
- [11] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles, SOSP '95*, pages 109–126, New York, NY, USA, 1995. Association for Computing Machinery.
- [12] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, Vijendra Purohit, Hugh Qu, Chaitanya Sreenivas Ravella, Krystyna Reisteter, Sheetal Shrotri, Dixin Tang, and Vikram Wakade. Socrates: The new SQL server in the cloud. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD '19*, page 1743–1756, New York, NY, USA, 2019. Association for Computing Machinery.
- [13] Apache Hadoop. Hadoop: Fair Scheduler. <https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/FairScheduler.html>.
- [14] Yuntao Bai, Andy Jones, Kamal Ndousse, Amanda Askell, Anna Chen, Nova DasSarma, Dawn Drain, Stanislav Fort, Deep Ganguli, Tom Henighan, Nicholas Joseph, Saurav Kadavath, Jackson Kernion, Tom Conerly, Sheer El-Showk, Nelson Elhage, Zac Hatfield-Dodds, Danny Hernandez, Tristan Hume, Scott Johnston, Shauna Kravec, Liane Lovitt, Neel Nanda, Catherine Olsson, Dario Amodei, Tom Brown, Jack Clark, Sam McCandlish, Chris Olah, Ben Mann, and Jared Kaplan. Training a helpful and harmless assistant with reinforcement learning from human feedback, 2022.
- [15] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. RobinHood: Tail latency aware caching – dynamic reallocation from Cache-Rich to Cache-Poor. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 195–212, Carlsbad, CA, October 2018. USENIX Association.
- [16] Daniel S. Berger, Ramesh K. Sitaraman, and Mor Harchol-Balter. AdaptSize: Orchestrating the hot object memory cache in a content delivery network. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 483–498, Boston, MA, March 2017. USENIX Association.
- [17] Dhruba Borthakur. Hdfs architecture guide. [https://hadoop.apache.org/docs/r1.2.1/hdfs\\_design.html](https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html).

- [18] Damiano Carra and Giovanni Neglia. Efficient miss ratio curve computation for heterogeneous content popularity. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 741–751. USENIX Association, July 2020.
- [19] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. Faster: A concurrent key-value store with in-place updates. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD '18*, page 275–290, New York, NY, USA, 2018. Association for Computing Machinery.
- [20] Badrish Chandramouli, Vasileios Zois, Ted Hart, Tal Zaccai, Lukas M. Maas, Yoganand Rajasekaran, and Darren Gehring. Garnet: A next-generation cache-store for accelerating applications and services. *Proc. VLDB Endow.*, 19(2):224–237, October 2025.
- [21] Rongxin Cheng, Kai Zhou, Xingda Wei, Siyuan Liu, Mingcong Han, Mingjing Ai, Yeju Zhou, Baoquan Zhong, Wencong Xiao, Rong Chen, and Haibo Chen. Fast LLM post-training via decoupled and Best-of-N speculation, 2025.
- [22] Wonil Choi, Bhuvan Urgaonkar, Mahmut Taylan Kandemir, and George Kesidis. Multi-resource fair allocation for consolidated flash-based caching systems. In *Proceedings of the 23rd ACM/IFIP International Middleware Conference, Middleware '22*, page 202–215, New York, NY, USA, 2022. Association for Computing Machinery.
- [23] Mosharaf Chowdhury, Zhenhua Liu, Ali Ghodsi, and Ion Stoica. HUG: Multi-Resource fairness for correlated and elastic demands. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI'16)*, pages 407–424, Santa Clara, CA, March 2016. USENIX Association.
- [24] Paul F. Christiano, Jan Leike, Tom B. Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. In *Proceedings of the 31st International Conference on Neural Information Processing Systems, NIPS'17*, pages 4302–4310, Red Hook, NY, USA, 2017. Curran Associates Inc.
- [25] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Dynacache: Dynamic cloud caching. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, Santa Clara, CA, July 2015. USENIX Association.
- [26] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 379–392, Santa Clara, CA, March 2016. USENIX Association.

- [27] Asaf Cidon, Daniel Rushton, Stephen M. Rumble, and Ryan Stutsman. Memshare: a dynamic multi-tenant key-value cache. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 321–334, Santa Clara, CA, July 2017. USENIX Association.
- [28] Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multi-modality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.
- [29] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.
- [30] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *First Symposium on Operating Systems Design and Implementation (OSDI 94)*, Monterey, CA, November 1994. USENIX Association.
- [31] Dragonfly. Dragonfly: An in-memory data store without limits. <https://www.dragonflydb.io/>, 2022. Accessed: 2026-04-21.
- [32] Nosayba El-Sayed, Anurag Mukkara, Po-An Tsai, Harshad Kasture, Xiaosong Ma, and Daniel Sanchez. Kpart: A hybrid cache partitioning-sharing technique for commodity multicores. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 104–117, 2018.
- [33] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.
- [34] Wei Fu, Jiaxuan Gao, Xujie Shen, Chen Zhu, Zhiyu Mei, Chuyi He, Shusheng Xu, Guo Wei, Jun Mei, Jiashu Wang, Tongkai Yang, Binhang Yuan, and Yi Wu. AReaL: A large-scale asynchronous reinforcement learning system for language reasoning, 2025.
- [35] Tharanga Gamaethige and Prudhviraaj Karumanchi. How netflix operates mission-critical data stores on aws. AWS re:Invent 2021 Conference Presentation (NFX203), 2021.
- [36] Wei Gao, Yuheng Zhao, Dakai An, Tianyuan Wu, Lunxi Cao, Shaopan Xiong, Ju Huang, Weixun Wang, Siran Yang, Wenbo Su, Jiamang Wang, Lin Qu, Bo Zheng, and Wei Wang. Rollpacker: Mitigating long-tail rollouts for fast, synchronous rl post-training, 2025.

- [37] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. Multi-resource fair queueing for packet processing. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, page 1–12, New York, NY, USA, 2012. Association for Computing Machinery.
- [38] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, page 323–336, USA, 2011. USENIX Association.
- [39] Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Choosy: Max-min fair sharing for datacenter jobs with constraints. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, page 365–378, New York, NY, USA, 2013. Association for Computing Machinery.
- [40] GitLab. Postmortem of database outage of january 31. <https://about.gitlab.com/blog/postmortem-of-database-outage-of-january-31/>, February 2017.
- [41] Raúl Gracia-Tinedo, Josep Sampé, Edgar Zamora, Marc Sánchez-Artigas, Pedro García-López, Yosef Moatti, and Eran Rom. Crystal: Software-Defined storage for Multi-Tenant object stores. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 243–256, Santa Clara, CA, February 2017. USENIX Association.
- [42] Ajay Gulati, Irfan Ahmad, and Carl A. Waldspurger. PARDA: Proportional allocation of resources for distributed storage access. In *7th USENIX Conference on File and Storage Technologies (FAST 09)*, San Francisco, CA, February 2009. USENIX Association.
- [43] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature*, 645(8081):633–638, 2025.
- [44] Zhenyu Guo, Chuntao Hong, Mao Yang, Dong Zhou, Lidong Zhou, and Li Zhuang. Rex: replication at the speed of multi-core. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.
- [45] Jingkai He, Tianjian Li, Erhu Feng, Dong Du, Qian Liu, Tao Liu, Yubin Xia, and Haibo Chen. History rhymes: Accelerating LLM reinforcement learning with RhymeRL, 2025.

- [46] Jeffrey Helt, Abhinav Sharma, Daniel J. Abadi, Wyatt Lloyd, and Jose M. Faleiro. C5: cloned concurrency control that always keeps up. *Proc. VLDB Endow.*, 16(1):1–14, September 2022.
- [47] Jian Hu, Xibin Wu, Wei Shen, Jason Klein Liu, Zilin Zhu, Weixun Wang, Songlin Jiang, Haoran Wang, Hao Chen, Bin Chen, Weikai Fang, Xianyu, Yu Cao, Haotian Xu, and Yiming Liu. OpenRLHF: An easy-to-use, scalable and high-performance RLHF framework, 2025.
- [48] Qinghao Hu, Shang Yang, Junxian Guo, Xiaozhe Yao, Yujun Lin, Yuxian Gu, Han Cai, Chuang Gan, Ana Klimovic, and Song Han. Taming the long-tail: Efficient reasoning RL training with adaptive drafter, 2025.
- [49] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. LAMA: Optimized locality-aware memory allocation for key-value cache. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 57–69, Santa Clara, CA, July 2015. USENIX Association.
- [50] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. TiDB: a Raft-based HTAP database. *Proc. VLDB Endow.*, 13(12):3072–3084, August 2020.
- [51] IMARC Group. In-memory database market report by data type, application, end user, vertical, and region 2026–2034. <https://www.imarcgroup.com/in-memory-database-market>, 2025.
- [52] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, SOSP '09*, page 261–276, New York, NY, USA, 2009. Association for Computing Machinery.
- [53] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 463–479. USENIX Association, November 2020.
- [54] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. Aether: a scalable approach to logging. *Proc. VLDB Endow.*, 3(1–2):681–692, September 2010.

- [55] Ricardo Koller, Ali José Mashtizadeh, and Raju Rangaswami. Centaur: Host-side SSD caching for storage performance control. In *2015 IEEE International Conference on Autonomic Computing*, pages 51–60, 2015.
- [56] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP '23*, pages 611–626, New York, NY, USA, 2023. Association for Computing Machinery.
- [57] Seung-seob Lee, Jachym Putta, Ziming Mao, and Anurag Khandelwal. Spirit: Fair allocation of interdependent resources in remote memory systems. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles, SOSP '25*, page 120–135, New York, NY, USA, 2025. Association for Computing Machinery.
- [58] Conglong Li and Alan L. Cox. Gd-wheel: a cost-aware replacement policy for key-value stores. In *Proceedings of the Tenth European Conference on Computer Systems, EuroSys '15*, New York, NY, USA, 2015. Association for Computing Machinery.
- [59] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 583–598, Broomfield, CO, October 2014. USENIX Association.
- [60] Jing Liu, Anthony Rebello, Yifan Dai, Chenhao Ye, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Scale and performance in a filesystem semi-microkernel. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, SOSP '21*, page 819–835, New York, NY, USA, 2021. Association for Computing Machinery.
- [61] Shaoteng Liu, Haoqi Yuan, Minda Hu, Yanwei Li, Yukang Chen, Shu Liu, Zongqing Lu, and Jiaya Jia. Rl-gpt: Integrating reinforcement learning and code-as-policy. *Advances in Neural Information Processing Systems*, 37:28430–28459, 2024.
- [62] MariaDB Foundation. Parallel replication. <https://mariadb.com/docs/server/ha-and-performance/standard-replication/parallel-replication>, 2026. Accessed: 2026-04-21.
- [63] R.L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

- [64] Zhiyu Mei, Wei Fu, Kaiwei Li, Guangju Wang, Huanchen Zhang, and Yi Wu. ReaL: Efficient RLHF training of large language models with parameter reallocation, 2025.
- [65] Microsoft. Garnet. <https://github.com/microsoft/garnet>, 2024.
- [66] Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. Looking beyond GPUs for DNN scheduling on Multi-Tenant clusters. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 579–596, Carlsbad, CA, July 2022. USENIX Association.
- [67] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, Carlsbad, CA, October 2018. USENIX Association.
- [68] Michael Noukhovitch, Shengyi Huang, Sophie Xhonneux, Arian Hosseini, Rishabh Agarwal, and Aaron Courville. Asynchronous RLHF: Faster and more efficient off-policy RL for language models, 2025.
- [69] Nvidia. Nvidia/nccl: Optimized primitives for collective multi-GPU communication. <https://github.com/NVIDIA/nccl>.
- [70] Oracle Corporation. Mysql 8.4 reference manual: Replication threads. <https://dev.mysql.com/doc/refman/8.4/en/replication-threads.html>, 2024. Accessed: 2026-04-21.
- [71] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 27730–27744. Curran Associates, Inc., 2022.
- [72] Jinsu Park, Seongbeom Park, and Woongki Baek. CoPart: Coordinated partitioning of last-level cache and memory bandwidth for fairness-aware workload consolidation on commodity servers. In *Proceedings of the Fourteenth EuroSys Conference 2019, EuroSys '19*, New York, NY, USA, 2019. Association for Computing Machinery.

- [73] Dylan Patel, Daniel Nishball, and Jeremie Eliahou Ontiveros. Multi-datacenter training: Openai’s ambitious plan to beat google’s infrastructure, September 2024.
- [74] PostgreSQL Global Development Group. Replication configuration. <https://www.postgresql.org/docs/16/runtime-config-replication.html>, 2026. Accessed: 2026-04-21.
- [75] Guna Prasaad, Badrish Chandramouli, and Donald Kossmann. Concurrent prefix recovery: Performing CPR on a database. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD ’19*, page 687–704, New York, NY, USA, 2019. Association for Computing Machinery.
- [76] Qifan Pu, Haoyuan Li, Matei Zaharia, Ali Ghodsi, and Ion Stoica. FairRide: Near-Optimal, fair cache sharing. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 393–406, Santa Clara, CA, March 2016. USENIX Association.
- [77] Dai Qin, Angela Demke Brown, and Ashvin Goel. Scalable replay-based replication for fast databases. *Proc. VLDB Endow.*, 10(13):2025–2036, September 2017.
- [78] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Trading more storage for less computation — a KVCache-centric architecture for serving LLM chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 155–170, Santa Clara, CA, February 2025. USENIX Association.
- [79] Shi Qiu, Weinan Liu, Yifan Hu, Jianqin Yan, Zhirong Shen, Xin Yao, Renhai Chen, Gong Zhang, and Yiming Zhang. GeminiFS: A companion file system for GPUs. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 221–236, Santa Clara, CA, February 2025. USENIX Association.
- [80] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO’06)*, pages 423–432, 2006.
- [81] Ajeet Raina. Build a leaderboard with redis. <https://redis.io/tutorials/howtos/leaderboard/>, 2026.
- [82] Ajeet Raina. Building an analytics dashboard app using redis. <https://redis.io/tutorials/building-an-analytics-dashboard-app-using-redis/>, 2026.

- [83] Redis. Redis FAQ. <https://redis.io/docs/latest/develop/get-started/faq/>.
- [84] Redis. Redis Module API Docs. <https://redis.io/docs/latest/develop/reference/modules/>.
- [85] Redis. Scale with Redis Cluster. [https://redis.io/docs/latest/operate/oss\\_and\\_stack/management/scaling/](https://redis.io/docs/latest/operate/oss_and_stack/management/scaling/).
- [86] Redis. Session management. <https://redis.io/solutions/session-management/>.
- [87] Redis. Redis: The real-time data platform. <https://redis.io/>, 2009. Accessed: 2026-04-21.
- [88] Redis Ltd. Key eviction. <https://redis.io/docs/latest/develop/reference/eviction/>, 2026.
- [89] Redis Ltd. Memory optimization. [https://redis.io/docs/latest/operate/oss\\_and\\_stack/management/optimization/memory-optimization/](https://redis.io/docs/latest/operate/oss_and_stack/management/optimization/memory-optimization/), 2026.
- [90] Sambhav Satija, Chenhao Ye, Ranjitha Kosgi, Aditya Jain, Romit Kankaria, Yiwei Chen, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Kiran Srinivasan. Cloudscape: A study of storage services in modern cloud architectures. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 103–121, Santa Clara, CA, February 2025. USENIX Association.
- [91] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.
- [92] Amazon Web Services. Amazon DynamoDB. <https://aws.amazon.com/dynamodb/>.
- [93] Amazon Web Services. Amazon DynamoDB Pricing. <https://aws.amazon.com/dynamodb/pricing/>.
- [94] Amazon Web Services. Amazon EC2. <https://aws.amazon.com/ec2/>.
- [95] Amazon Web Services. Amazon Relational Database Service. <https://aws.amazon.com/rds/>.
- [96] Amrith Setlur, Saurabh Garg, Xinyang Geng, Naman Garg, Virginia Smith, and Aviral Kumar. RL on incorrect synthetic data scales the efficiency of LLM math reasoning by eight-fold. *Advances in Neural Information Processing Systems*, 37:43000–43031, 2024.

- [97] Amrith Setlur, Chirag Nagpal, Adam Fisch, Xinyang Geng, Jacob Eisenstein, Rishabh Agarwal, Alekh Agarwal, Jonathan Berant, and Aviral Kumar. Rewarding progress: Scaling automated process verifiers for LLM reasoning. *arXiv preprint arXiv:2410.08146*, 2024.
- [98] Pavel Shamis, Manjunath Gorentla Venkata, M Graham Lopez, Matthew B Baker, Oscar Hernandez, Yossi Itigin, Mike Dubman, Gilad Shainer, Richard L Graham, Liran Liss, et al. UCX: an open source framework for HPC network APIs and beyond. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 40–43. IEEE, 2015.
- [99] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseek-math: Pushing the limits of mathematical reasoning in open language models, 2024.
- [100] Weihai Shen, Ansh Khanna, Sebastian Angel, Siddhartha Sen, and Shuai Mu. Rolis: a software approach to efficiently replicating multi-core transactions. In *Proceedings of the Seventeenth European Conference on Computer Systems, EuroSys '22*, page 69–84, New York, NY, USA, 2022. Association for Computing Machinery.
- [101] Guangming Sheng, Yuxuan Tong, Borui Wan, Wang Zhang, Chaobo Jia, Xibin Wu, Yuqi Wu, Xiang Li, Chi Zhang, Yanghua Peng, Haibin Lin, Xin Liu, and Chuan Wu. Laminar: A scalable asynchronous RL post-training framework. In *Proceedings of the 21st European Conference on Computer Systems, EUROSYS '26*, pages 400–422, New York, NY, USA, 2026. Association for Computing Machinery.
- [102] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. HybridFlow: A flexible and efficient RLHF framework. In *Proceedings of the Twentieth European Conference on Computer Systems, EuroSys '25*, pages 1279–1297, New York, NY, USA, 2025. Association for Computing Machinery.
- [103] David Shue, Michael J. Freedman, and Anees Shaikh. Performance isolation and fairness for Multi-Tenant cloud storage. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 349–362, Hollywood, CA, October 2012. USENIX Association.

- [104] Chijun Sima, Yao Fu, Man-Kit Sit, Liyi Guo, Xuri Gong, Feng Lin, Junyu Wu, Yongsheng Li, Haidong Rong, Pierre-Louis Aublin, and Luo Mai. Ekko: A Large-Scale deep learning recommender system with Low-Latency model update. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 821–839, Carlsbad, CA, July 2022. USENIX Association.
- [105] Snap Inc. KeyDB: The faster redis alternative. <https://docs.keydb.dev/>, 2019. Accessed: 2026-04-21.
- [106] SPDK Open-source Team. The Storage Performance Development Kit. <https://spdk.io/doc>, 2021.
- [107] Ioan Stefanovici, Eno Thereska, Greg O’Shea, Bianca Schroeder, Hitesh Ballani, Thomas Karagiannis, Antony Rowstron, and Tom Talpey. Software-defined caching: Managing caches in multi-tenant data centers. In *Proceedings of the Sixth ACM Symposium on Cloud Computing, SoCC ’15*, page 174–181, New York, NY, USA, 2015. Association for Computing Machinery.
- [108] Xiao Sun, Tan N. Le, Mosharaf Chowdhury, and Zhenhua Liu. Fair allocation of heterogeneous and interchangeable resources. *SIGMETRICS Perform. Eval. Rev.*, 46(2):21–23, jan 2019.
- [109] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. CockroachDB: The resilient geo-distributed SQL database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, SIGMOD ’20*, page 1493–1509, New York, NY, USA, 2020. Association for Computing Machinery.
- [110] Eno Thereska, Hitesh Ballani, Greg O’Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. Ioflow: A software-defined storage architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP ’13*, page 182–196, New York, NY, USA, 2013. Association for Computing Machinery.
- [111] Valkey. Valkey. <https://valkey.io/>, 2024. Accessed: 2026-05-14.

- [112] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, James Corey, Kamal Gupta, Murali Brahmadesam, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvilli, and Xiaofeng Bao. Amazon aurora: On avoiding distributed consensus for i/os, commits, and membership changes. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD '18*, page 789–796, New York, NY, USA, 2018. Association for Computing Machinery.
- [113] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R Ganger. Argon: Performance insulation for shared storage servers. In *5th USENIX Conference on File and Storage Technologies (FAST 07)*, San Jose, CA, February 2007. USENIX Association.
- [114] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache modeling and optimization using miniature simulations. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 487–498, Santa Clara, CA, July 2017. USENIX Association.
- [115] Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient MRC construction with SHARDS. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST'15*, page 95–110, USA, 2015. USENIX Association.
- [116] Jim Allen Wallace and Rini Vasan. It's official: We're the #1 AI agent data storage tool. <https://redis.io/blog/best-ai-agent-data-storage-2025/>.
- [117] Borui Wan, Mingji Han, Yiyao Sheng, Yanghua Peng, Haibin Lin, Mofan Zhang, Zhichao Lai, Menghan Yu, Junda Zhang, Zuquan Song, et al. ByteCheckpoint: A unified checkpointing system for large foundation model development. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 559–578, 2025.
- [118] Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. Query fresh: log shipping on steroids. *Proc. VLDB Endow.*, 11(4):406–419, December 2017.
- [119] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, T. S. Eugene Ng, and Yida Wang. Gemini: Fast failure recovery in distributed training with in-memory checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP '23*, pages 364–381, New York, NY, USA, 2023. Association for Computing Machinery.

- [120] Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I Wang. SWE-RL: Advancing LLM reasoning via reinforcement learning on open software evolution. *arXiv preprint arXiv:2502.18449*, 2025.
- [121] Bo Wu, Sid Wang, Yunhao Tang, Jia Ding, Eryk Helenowski, Liang Tan, Tengyu Xu, Tushar Gowda, Zhengxing Chen, Chen Zhu, Xiaocheng Tang, Yundi Qian, Beibei Zhu, and Rui Hou. LlamaRL: A distributed asynchronous reinforcement learning framework for efficient large-scale LLM training, 2025.
- [122] Yongji Wu, Xueshen Liu, Haizhong Zheng, Juncheng Gu, Beidi Chen, Z. Morley Mao, Arvind Krishnamurthy, and Ion Stoica. RLBoost: Harvesting preemptible cloud resources for Cost-Efficient reinforcement learning on LLMs. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*, pages 1323–1339, Renton, WA, May 2026. USENIX Association.
- [123] Yu Xia, Xiangyao Yu, Andrew Pavlo, and Srinivas Devadas. Taurus: lightweight parallel logging for in-memory database management systems. *Proc. VLDB Endow.*, 14(2):189–201, October 2020.
- [124] Jianhao Yan, Yafu Li, Zican Hu, Zhi Wang, Ganqu Cui, Xiaoye Qu, Yu Cheng, and Yue Zhang. Learning to reason under off-policy guidance. *arXiv preprint arXiv:2504.14945*, 2025.
- [125] Juncheng Yang, Yao Yue, and K. V. Rashmi. Anonymized cache request traces from twitter production. <https://github.com/twitter/cache-trace>, 2020.
- [126] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208. USENIX Association, November 2020.
- [127] Mao Yang, Dong Zhou, Carbo Kuo, Chuntao Hong, Lintao Zhang, and Lidong Zhou. KuaFu: Closing the parallelism gap in database replication. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, page 1186–1195, USA, 2013. IEEE Computer Society.
- [128] Chenhao Ye, Shawn Zhong, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. BunnyFS codebase. <https://github.com/WiscADSL/bunnyfs>, 2025.
- [129] Chenhao Ye, Shawn Zhong, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. HopperKV codebase. <https://github.com/WiscADSL/hopperkv>, 2025.

- [130] Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Mu Qiao, Yonghui Wu, and Mingxuan Wang. DAPO: An open-source LLM reinforcement learning system at scale, 2025.
- [131] Yao Yue. Scaling redis at twitter. <https://www.youtube.com/watch?v=rP9EKvWt0zo>, 2014.
- [132] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. SGLang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583, 2024.
- [133] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. Fast databases with fast durability and recovery through multicore parallelism. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 465–477, Broomfield, CO, October 2014. USENIX Association.
- [134] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210, Santa Clara, CA, July 2024. USENIX Association.
- [135] Yinmin Zhong, Zili Zhang, Xiaoni Song, Hanpeng Hu, Chao Jin, Bingyang Wu, Nuo Chen, Yukun Chen, Yu Zhou, Changyi Wan, Hongyu Zhou, Yimin Jiang, Yibo Zhu, and Daxin Jiang. StreamRL: Scalable, heterogeneous, and elastic RL for LLMs with disaggregated stream generation, 2025.
- [136] Yinmin Zhong, Zili Zhang, Bingyang Wu, Shengyu Liu, Yukun Chen, Changyi Wan, Hanpeng Hu, Lei Xia, Ranchen Ming, Yibo Zhu, and Xin Jin. Optimizing RLHF training for large language models with stage fusion. In Theophilus A. Benson and Radhika Niranjana Mysore, editors, *22nd USENIX Symposium on Networked Systems Design and Implementation, NSDI 2025, Philadelphia, PA, USA, April 28-30, 2025*, pages 489–503. USENIX Association, 2025.

- [137] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. FoundationDB: A distributed unbundled transactional key value store. In *Proceedings of the 2021 International Conference on Management of Data, SIGMOD '21*, pages 2653–2666, New York, NY, USA, 2021. Association for Computing Machinery.
- [138] Yuzhen Zhou, Jiajun Li, Yusheng Su, Gowtham Ramesh, Zilin Zhu, Xiang Long, Chenyang Zhao, Jin Pan, Xiaodong Yu, Ze Wang, et al. April: Active partial rollouts in reinforcement learning to tame long-tail generation. *arXiv preprint arXiv:2509.18521*, 2025.
- [139] Zilin Zhu, Chengxing Xie, Xin Lv, and slime Contributors. slime: An LLM post-training framework for RL scaling. <https://github.com/THUDM/slime>, 2025. GitHub repository. Corresponding author: Xin Lv.
- [140] Siyuan Zhuang, Zhuohan Li, Danyang Zhuo, Stephanie Wang, Eric Liang, Robert Nishihara, Philipp Moritz, and Ion Stoica. Hoplite: efficient and fault-tolerant collective communication for task-based distributed systems. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference, SIGCOMM '21*, pages 641–656, New York, NY, USA, 2021. Association for Computing Machinery.