Yang Zhou

Department of Computer Science
Harvard University
150 Western Ave, SEC 4.429
Allston, MA 02134, USA
+1 617 599 8532
yangzhou@g.harvard.edu

November 28, 2023

Dear Faculty Search Committee:

I am writing to apply for a tenure-track position as an assistant professor in your department. I am currently a PhD candidate in the Department of Computer Science at Harvard University and expect to complete my dissertation work by June 2024.

My research interests lie in the areas of computer systems, especially networking, operating systems, and distributed systems. I am particularly interested in full-stack optimizations for efficient and evolvable datacenter infrastructure, by codesigning networking stacks and datacenter applications. My research has studied kernel eBPF and kernel-bypass techniques, and various applications like fault-tolerant far memory, Paxos consensus, distributed transactions, and microsecond-scale RPCs.

I have enclosed my curriculum vitae with a list of references, research statement, teaching statement, diversity statement, and three representative publications. These materials are also available online at https://yangzhou1997.github.io/application/.

I look forward to hearing from you.

Sincerely,

Yang Zhou

[This page intentionally left blank.]

Yang Zhou

yangzhou1997.github.io
yangzhou@g.harvard.edu ⋄ +1 617 599 8532
150 Western Ave, SEC 4.429, Allston, MA 02134, USA

Research Interests

Networked systems, operating systems, distributed systems, networking stacks, and network telemetry.

Education

Harvard University, Cambridge, MA, USA

Ph.D. in Computer Science *(Expected) June 2024*
M.S. in Computer Science *November 2021*
Thesis title: Codesigning Networking Stacks and Datacenter Applications for High Efficiency and Evolvability
Advisors: Minlan Yu and James Mickens

Peking University, Beijing, China

B.S. in Computer Science *July 2018*
Thesis title: Towards Faster and More Accurate Data Stream Processing
Advisors: Tong Yang

Work Experience

Harvard University, Research Assistant *August 2018–Present*

- *Kernel offloads:* Designed eBPF-based kernel offloads for distributed system protocols including Paxos (Electrode [2]) and serializable transactions (DINT [1]) to reduce kernel networking stack overhead. Implemented and evaluated atop unmodified Linux OSes, and achieved kernel-bypass-like throughput and latency.
- *μs-scale RPCs:* Designed an efficient inter-server load balancing scheme for μs-scale RPCs to achieve low tail latency and high goodput (Mew [12]). Implemented and evaluated for both kernel-bypass and kernel-based networking stacks.
- *SmartNIC architecture:* Designed and prototyped SGX-like trusted execution environments for network functions in SmartNICs under multi-tenant cloud environments (S-NIC [13]).

Google NetInfra Group and System Research Group, Student Researcher *June 2021–May 2023*

- *Far memory:* Designed an efficient far memory system that leverages erasure-coding, remote memory compaction, one-sided RMAs, and offloadable parity calculations to achieve fast, storage-efficient fault tolerance (Carbink [3]). Implemented and evaluated using production networking stack.
- *Distributed runtime:* Designed an efficient fault-tolerant distributed runtime based on tasks and actors by leveraging the Chandy–Lamport consistent checkpointing algorithm and causal logging mechanism.
- *μs-scale RPCs:* Identified and motivated the inter-server scheduling problem for μs-scale RPCs (leading to Mew).

VMware Research, Research Intern *July 2020–September 2020*

- *Geo-distributed data analytics:* Applied traffic redundancy elimination (TRE) technique to accelerate geo-distributed data analytics and save WAN traffic cost. Implemented atop Alluxio, an in-memory data cache system for analytics.

Facebook, Research Collaborator *November 2019–May 2020*

- *Network telemetry:* Conducted extensive measurement and analysis on Facebook's network telemetry system. Identified the importance of being evolvable and handling changes. Proposed a change cube abstraction to systematically track changes, and an intent-based layering design to confine and track changes (PCAT [4]).

SenseTime, Software Engineering Intern *March 2018–May 2018*

- *Distributed storage:* Worked on Ceph storage setup, testing, maintenance, monitoring, and alerting.

Peking University, Research Assistant *April 2016–July 2018*

- *Network telemetry:* Designed and implemented novel probabilistic data structures (e.g., sketches and Bloom filters) to optimize the memory usage, speed, and accuracy of network telemetry tasks (Cold Filter [5], Elastic Sketch [6], Pyramid Sketch [9], and more [7][15][19]).

PUBLICATIONS

Total 780 citations till November 2024 based on Google Scholar.

Conference Publications

[1] **Yang Zhou**, Xingyu Xiang, Matthew Kiley, Sowmya Dharanipragada, and Minlan Yu.
DINT: Fast In-Kernel Distributed Transactions with eBPF. [link]
USENIX NSDI 2024.

[2] **Yang Zhou**, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu.
Electrode: Accelerating Distributed Protocols with eBPF. [link]
USENIX NSDI 2023.

[3] **Yang Zhou**, Hassan Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David Culler, Hank Levy, and Amin Vahdat.
Carbink: Fault-Tolerant Far Memory. [link]
USENIX OSDI 2022.

[4] **Yang Zhou**, Ying Zhang, Minlan Yu, Guangyu Wang, Dexter Cao, Eric Sung, and Starsky Wong.
Evolvable Network Telemetry at Facebook. [link]
USENIX NSDI 2022.

[5] **Yang Zhou**, Tong Yang, Jie Jiang, Bin Cui, Minlan Yu, Xiaoming Li, and Steve Uhlig.
Cold Filter: A Meta-Framework for Faster and More Accurate Stream. Processing [link]
ACM SIGMOD 2018.

[6] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, **Yang Zhou**, Rui Miao, Xiaoming Li, and Steve Uhlig.
Elastic Sketch: Adaptive and Fast Network-Wide Measurements. [link]
ACM SIGCOMM 2018.

[7] Omid Alipourfard, Masoud Moshref, **Yang Zhou**, Tong Yang, and Minlan Yu.
A Comparison of Performance and Accuracy of Measurement Algorithms in Software. [link]
ACM Symposium on SDN Research (SOSR) 2018.

[8] Xiangyang Gou, Chenxingyu Zhao, Tong Yang, Lei Zou, **Yang Zhou**, Yibo Yan, Xiaoming Li, and Bin Cui.
Single Hash: Use One Hash Function to Build Faster Hash Based Data Structures. [link]
IEEE International Conference on Big Data and Smart Computing (BigComp) 2018.

[9] Tong Yang, **Yang Zhou**, Hao Jin, Shigang Chen, and Xiaoming Li.
Pyramid Sketch: A Sketch Framework for Frequency Estimation of Data Streams. [link]
VLDB 2017.

[10] **Yang Zhou**, Peng Liu, Hao Jin, Tong Yang, Shoujiang Dang, and Xiaoming Li.
One Memory Access Sketch: A More Accurate and Faster Sketch for Per-Flow Measurement. [link]
IEEE Global Communications Conference (Globecom) 2017.

[11] Junzhi Gong, Tong Yang, **Yang Zhou**, Dongsheng Yang, Shigang Chen, Bin Cui, and Xiaoming Li.
ABC: A Practicable Sketch Framework for Non-Uniform Multisets. [link]
IEEE International Conference on Big Data (BigData) 2017.

Papers Under Reviews

[12] **Yang Zhou**, Hassan Wassel, James Mickens, Minlan Yu, and Amin Vahdat.
Mew: Efficient Inter-Server Load Balancing for Microsecond-Scale RPCs. [link]
September 2023.

[13] **Yang Zhou**, Mark Wilkening, James Mickens, and Minlan Yu.
SmartNIC Security Isolation in the Cloud with S-NIC. [link]
October 2023.

Workshop and Demo Publications

[14] **Yang Zhou**, Hao Jin, Peng Liu, Haowei Zhang, Tong Yang, and Xiaoming Li.
Accurate Per-Flow Measurement with Bloom Sketch. [link]

IEEE International Conference on Computer Communications Workshops (INFOCOM WKSHPS) 2018.

Journal Publications

[15] Zhuochen Fan, Gang Wen, Zhipeng Huang, **Yang Zhou**, Qiaobin Fu, Tong Yang, Alex X Liu, and Bin Cui.
On the Evolutionary of Bloom Filter False Positives - An Information Theoretical Approach to Optimizing Bloom Filter Parameters. [link]
IEEE Transactions on Knowledge & Data Engineering 2022.

[16] Yuanpeng Li, Xiang Yu, Yilong Yang, **Yang Zhou**, Tong Yang, Zhuo Ma, and Shigang Chen.
Pyramid Family: Generic Frameworks for Accurate and Fast Flow Size Measurement. [link]
IEEE/ACM Transactions on Networking 2021.

[17] Tong Yang, Jie Jiang, **Yang Zhou**, Long He, Jinyang Li, Bin Cui, Steve Uhlig, and Xiaoming Li.
Fast and Accurate Stream Processing by Filtering the Cold. [link]
The VLDB Journal 2019.

[18] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, **Yang Zhou**, Rui Miao, Xiaoming Li, and Steve Uhlig.
Adaptive Measurements Using One Elastic Sketch. [link]
IEEE/ACM Transactions on Networking 2019.

[19] **Yang Zhou**, Omid Alipourfard, Minlan Yu, and Tong Yang.
Accelerating Network Measurement in Software. [link]
ACM SIGCOMM Computer Communication Review 2018.

Talks

- Electrode: Accelerating Distributed Protocols with eBPF
  Duke University, ACE Center for Evolvable Computing, Google, USENIX NSDI — *April 2023*
  Columbia University — *March 2023*
- Carbink: Fault-Tolerant Far Memory
  Cornell University — *November 2023*
  WORDS workshop — *November 2022*
  Microsoft Research Redmond, USENIX OSDI — *July 2022*
  Google — *March & June 2022*
- Evolvable Network Telemetry at Facebook
  USENIX NSDI — *April 2022*
  Boston University, Meta — *March 2022*
- Cold Filter: A Meta-Framework for Faster and More Accurate Stream Processing
  Harvard University — *October 2018*

Mentoring Experience

- Matt Kiley, Harvard College undergraduate — *2023*
  Accelerating distributed transactions using eBPF (NSDI 2024, [1]); AF_XDP-based RPC systems.
- Yunxi Shen, Tsinghua University undergraduate — *2023*
  Resource-efficient job scheduling in data centers.
- Xingyu Xiang, Peking University undergraduate — *2023*
  Accelerating distributed transactions using eBPF (NSDI 2024, [1]).
- Zezhou Wang, Peking University undergraduate $\rightarrow$ University of Washington PhD — *2022*
  Accelerating Paxos using eBPF (NSDI 2023, [2]).

Teaching Experience

- **Guest Lecture** on far memory, CS294-252: Architectures and Systems for Warehouse-Scale Computers, UC Berkeley
  Nov 2023

- **Teaching Assistant** for Prof. Minlan Yu, CS145: Networking at Scale, Harvard University *Spring 2021*
- **Teaching Assistant** for Prof. Tong Yang, Algorithm Design and Analysis, Peking University *Fall 2018*

Patents

- **Yang Zhou**, Hassan Wassel, Minlan Yu, Hank Levy, David Culler, and Amin Vahdat. "Fault Tolerant Disaggregated Memory". Pending (US20230185666A1), filed by Google in December 2022.

Academic Honors

- Google Ph.D. Fellowship in Systems and Networking *2022*
- Finalist, Meta Ph.D. Fellowship in Networking *2022*
- Graduate Fellowship, Harvard University *2018*
- Excellent Bachelor Thesis (10/327), School of EECS, Peking University *2018*
- New Academic Star Award (1/193), School of EECS, Peking University *2018*
- Arawana Scholarship (2/193), Peking University *2017*
- Pinyou Hudong Scholarship, School of EECS, Peking University *2016*
- May Fourth Scholarship, Peking University *2015*

Professional Activities

- PC Member: ACM SIGCOMM Poster/Demo 2023, IEEE INFOCOM Workshop on Networking Algorithms 2020.
- Reviewer (Conferences): ACM SIGKDD 2023.
- Reviewer (Journals): ACM Transactions on Modeling and Performance Evaluation of Computing Systems, IEEE/ACM Transactions on Networking, IEEE Journal on Selected Areas in Communications.
- Panelist: "Getting started with systems research" at Students@Systems 2022.

References

Prof. Minlan Yu
Department of Computer Science
Harvard University
150 Western Ave, SEC 4.415
Allston, MA 02134, USA
+1 617 495 3986
minlanyu@g.harvard.edu

Prof. James Mickens
Department of Computer Science
Harvard University
150 Western Ave, SEC 4.416
Allston, MA 02134, USA
+1 617 384 8132
mickens@seas.harvard.edu

Dr. Amin Vahdat
Google Fellow and Vice President of Engineering
Google LLC
1600 Amphitheatre Parkway
Mountain View, CA 94042, USA
+1 650 390 7073
vahdat@google.com

Prof. Adam Belay
MIT CSAIL
32 Vassar St, 32-G996
Cambridge, MA 02139, USA
+1 617 253 0004
abelay@mit.edu

Dr. Ying Zhang
Senior Engineering Manager
Meta Platforms, Inc.
1 Hacker Way
Menlo Park, CA 94025, USA
+1 408 250 9961
zhangying@meta.com

Research Statement

Yang Zhou

I am a systems researcher spanning the areas of networking, operating systems, and distributed systems, focusing on datacenter environments. A datacenter centralizes hundreds of thousands of machines with high-speed networks, enables computations over huge amounts of data, and hosts popular applications (e.g., Google search, Netflix streaming, ChatGPT) that impact billions of people's lives.

In the era of massive-scale data and computations, networking plays a critical role in supporting scale-out datacenter applications running across multiple machines. Ideally, the underlying datacenter infrastructure should be efficient to maintain steady cloud revenues while meeting high user expectations, and be evolvable to handle the increasingly diverse and performance-hungry applications as well as heterogeneous hardware. However, there is a growing **mismatch** between what networking stacks (involving NICs, kernels, transport layers, and threading) provide and what applications need, causing severe efficiency and evolvability problems. For example, the most widely used kernel networking stack prioritizes security and isolation with separated kernel and user contexts, incurring prohibitive CPU overheads; meanwhile, emerging in-memory applications demand ultra-low latency and high throughput, preferring coalescing different contexts but losing isolation. Even though the networking stacks keep evolving, e.g., the modern kernel-bypass RDMA stacks, applications tend to be network-unaware and take networking resources for granted and unlimited, easily causing resource depletion. Such mismatch gets largely exacerbated in large-scale datacenters where networking stacks and applications are usually developed and maintained by disjoint groups of engineers, i.e., network vs. application engineers (due to their growing complexities and industrial organizational structures). This fundamental mismatch causes less efficient use of datacenter resources and hinders the scaling-out of diverse datacenter applications.

My research has focused on bridging the mismatch by **codesigning** low-level networking stacks and high-level datacenter applications from a systems perspective. My codesign aims to realize high efficiency and agile evolvability for datacenter infrastructure, and it innovates in two directions: (1) application-aware networking by restructuring networking stacks based on application needs, and (2) network-aware applications by redesigning applications to be network-efficient. They have borne fruit for many important datacenter applications, including existing ones (e.g., consensus, distributed transactions) and emerging ones (e.g., far memory over networks, microsecond-scale RPCs). My Electrode [1], Dint [2], and Mew [3] safely inject Paxos, transactions, and RPC load balancing logics into the kernel networking stack respectively via eBPF. This not only achieves remarkable performance improvements (by avoiding kernel overheads) but also allows customizing and evolving the kernel stack based on application needs. My Carbink [4] enables network-aware fault tolerance for far memory with high network and memory efficiency, making it practically usable in datacenters with failures being the norm; it also results in a joint patent with Google. Specific to evolvability, my PCAT [5] helps Facebook design an evolvable telemetry system to handle frequent changes in production networks.

My research methodology has been empiricism-guided *measuring, tailoring, and fitting* to analyze, optimize, and implement real-world systems—just like how tailors made clothes in the old times. First, I thoroughly measure to reason through the performance characteristics of various networking stack primitives and complex applications; I also draw on my two-year experiences in Google's networking and system teams to uncover critical feature requirements in production systems. Second, I aggressively tailor unnecessary or overlapping operations in networking stacks and applications to optimize for high efficiency. Third, I strategically partition and fit applications to the right networking stack primitives to efficiently implement the entire system. This focus on **full-stack optimizations** defines my niche as a systems researcher.

Previous Work

CPU efficient distributed protocols with evolvable kernel networking via eBPF. In-memory distributed protocols such as consensus and distributed transactions are important building blocks for datacenter applications. They require intensive network IOs, while the widely-used kernel networking stack gives low IO performance due to high per-IO CPU overhead. Such mismatch has fostered a popular belief that kernel-bypass is the necessary key to high performance for these protocols. However, kernel-bypass is not a panacea: it essentially trades security, isolation, protection, maintainability, and debuggability for performance; it also burns one or more CPU cores for busy-polling even at low loads, which is usually hard to adopt in public cloud deployments due to per-core pricing [6]. As such, I revisit the above popular belief and ask: is the current kernel networking stack really ill-suited for CPU-efficient distributed protocols, especially given many kernel advancements over decades?

I first measure the source of the high overhead for kernel networking stacks. When running a prior well-designed transaction protocol under a recent Linux kernel networking stack, I find that networking stack traversing dominates the overhead (64% vs. 16% on context switching and 12% on interrupt handling). This motivates me to aggressively tailor unnecessary components of the stack for specific distributed protocols, trading slight genericity loss for performance boosts. For example, the reliable transport along with complex queue disciplines, which incurs costly `sk_buff` maintenance and packet copies, could be cut; this is because (1) distributed protocols themselves can recover from packet loss with application-level timeouts, and (2) packet loss happens rarely within today's well-engineered datacenter networks. To realize such tailoring, I leverage eBPF to *safely* offload protocol-specific request processing logic into the early stages of the kernel stack; this avoids going through the full stack and user space, removing most of the kernel overheads.

However, offloading complex distributed protocols into the kernel is challenging, because eBPF has a constrained programming model for kernel safety and liveness. To address this challenge, I strategically partition the distributed protocols to fit frequent critical paths into the kernel for high performance while complex rare paths into the user space for full functionalities. Take the classic Multi-Paxos protocol as an example. Electrode [1] offloads failure-free Multi-Paxos operations of broadcasting, acknowledging, and waiting-on-quorums into the kernel via eBPF; when failure happens, it runs complex failure-handling operations in the user space. I implement such partitioning for Multi-Paxos and two transaction protocols (version-based and lock-based) atop unmodified Linux kernels, and achieve remarkable performance boosts. For instance, Dint [2] for transaction offloading achieves up to 23× higher throughput than kernel networking stacks, and 2.6× higher than a recent DPDK-based kernel-bypass stack [7] (as the eBPF offloads directly work on raw ethernet packets, bypassing any socket connections). Owing to the kernel-friendliness and high performance, my eBPF offloading work has sparked interest in both industry (e.g., Meta, Intel) and academia (e.g., University of Washington, University of Michigan, NYU).

Looking further out, future kernel networking stacks should be evolvable in order to efficiently tackle increasingly diverse applications and heterogeneous hardware. My Electrode and Dint projects already demonstrate that eBPF can provide significant evolvability to kernel networking stacks for specific applications. I am now working on an evolvable generic RPC framework by implementing a reliable RPC transport in eBPF; it leverages efficient AF_XDP sockets to direct RPC requests to user-space applications for processing. The evolvability of this RPC framework manifests into three aspects: (1) customizing network transport protocols based on application types (e.g., video), (2) customizing the locations of transport layer offloads ranging from host kernels to SmartNICs (many SmartNICs directly support eBPF), and (3) application-informed request load balancing among CPU cores.

Network and memory efficient fault-tolerant far memory. In a datacenter, matching a particular application to just enough memory and CPUs is hard. A commodity server tightly couples memory and compute, hosting a fixed number of CPUs and RAM modules that are unlikely to exactly match the computational requirements of any particular application. Even if a datacenter contains a heterogeneous mix of server configurations, the load on each server (and thus the amount of available resources for a new application) changes dynamically as old applications exit and new applications arrive. Thus, even state-of-the-art cluster schedulers struggle to efficiently bin-pack a datacenter's aggregate collection of CPUs and RAM. For example, Google [8] and Alibaba [9] report that the average server has only 60% memory utilization, with substantial variance across machines.

Disaggregated datacenter memory is a promising solution. It pairs a CPU with an arbitrary set of possibly-remote RAM modules, with a fast network interconnect keeping access latencies to far memory small. Much of the prior work in this space [10, 11] has a common limitation: a lack of fault tolerance. Unfortunately, in a datacenter containing hundreds of thousands of machines, faults are pervasive. Without fault tolerance, the failure rate of an application using far memory will be much higher than the failure rate of an application that only uses local memory; the reason is that the use of far memory increases the set of machines whose failure can impact an application.

Achieving both network and memory efficient fault-tolerant far memory is challenging. Conventional memory-efficient fault tolerance scheme applies erasure coding, and stripes a single memory page across multiple remote nodes with RMA-based swapping. For brevity, I use *span* to denote "memory page". Assuming Reed-Solomon code with 4 data chunks and 2 parity chunks, the conventional scheme requires 6 RMAs per span swap-out and 4 RMAs per swap-in, incurring excessive network IO pressure on the networking stack. In Carbink, I tailor the excessive network IOs by eschewing the span-granularity erasure coding, and instead erasure code at the spanset granularity. A spanset consists of multiple spans with the same size, i.e., 4 data spans and 2 parity spans in our example, and gets swapped out together in a batch. This only requires averagely $(4+2)/4 = 1.5$ RMAs per span swap-out and a single RMA per swap-in, significantly improving network efficiency.

However, spanset-granularity erasure coding inevitably incurs memory fragmentation. This is because each span lives in exactly one place (either local memory or far memory), and swapping a span inside a spanset from far memory to local memory creates dead space (and thus fragmentation) in far memory. To address this problem, I design a pauseless defragmentation mechanism running off the swapping critical path, asynchronously reclaiming dead space for later swap-outs in the background. In contrast to the simple span swapping via RMA, this background defragmentation has complex two-phase commit procedures to guarantee crash consistency; therefore, I choose to implement it using more expressive RPCs. Carbink is implemented and evaluated atop Google's datacenter infrastructure. Compared to a state-of-the-art fault-tolerant design that uses span-granularity erasure coding, Carbink has 29% lower tail latency and 48% higher application performance, with at most 35% higher far memory usage (due to asynchronous memory defragmentation). Carbink also results in a joint patent with Google.

CPU efficient load balancing for microsecond-scale RPCs. Datacenter applications are evolving into microservice architectures, with many small services connected via RPCs to serve user requests. To ensure responsiveness, these services require high throughput and low tail latency, reaching millions of operations/sec per server and microsecond-scale latency respectively. This creates a mismatch between existing RPC frameworks and application demands, in terms of efficiently load balancing microsecond-scale RPCs. Conventional Power-of-Two load balancing probes servers' load too often (i.e., probing before each RPC) and hurts application throughput, as a load probing consumes comparable server CPUs as a microsecond-scale RPC. My measurement shows that it reduces the goodput (i.e., maximum throughput under tail latency SLO) by half compared to naively dispatching RPCs at random. On the other hand, probing too infrequently will result in stale estimates of load, resulting in suboptimal load balancing, the emergence of hot spots, and violated SLOs. To break this dilemma, Mew [3] tailors unnecessary load probings to just fulfill the staleness requirement that does not degrade tail latency. To do so, Mew performs probing statistically following an optimal probing frequency, obtained by running a gradient descent algorithm on the probing frequency vs. tail latency space.

However, there are more challenges in how to efficiently fit RPC load balancing into RPC frameworks. The first is what load signal to use that is general enough to capture different load levels of servers, and is strongly correlated to future RPC's tail latency. Instead of using the conventional signal of CPU utilization, I use the low-level thread and packet queueing delay, because the former cannot differentiate between the ideal case of exactly-saturated CPUs and the bad case of overloaded CPUs. The second challenge is how to efficiently implement load probing, especially for kernel networking stacks with high overhead. My solution is leveraging eBPF to directly return load signal values in the kernel, without going through the full kernel stack or user space. With all the above designs, Mew is able to reduce RPC tail latency by 2×, while achieving 1.7× higher goodput, over a state-of-the-art solution.

Other datacenter infrastructure research:

Evolvable and memory efficient network telemetry. As modern datacenter networks get larger and more complex, operators must rely on network telemetry systems for continuous monitoring, alerting, failure troubleshooting, etc. However, changes happen frequently in production networks (e.g., modifications to monitoring intent, advances of device APIs), impacting the reliability of network telemetry systems. To handle various changes, I helped Facebook develop their evolvable network telemetry system PCAT [5]. PCAT proposes to use a change cube abstraction to systematically track changes, and an intent-based layering design to confine and track changes. The overall result of PCAT is a change-aware network telemetry system that supports fast-evolving datacenter networks at Facebook.

Network telemetry also requires high efficiency for memory. Telemetry data must be stored in memory, at least temporarily, but memory is a precious resource. Network devices (e.g., NICs, switches) often have less than 100MB of memory; server memory is more plentiful, but should be mostly devoted to applications. My Cold Filter [12], Elastic Sketch [13], Pyramid Sketch [14], and more [15, 16, 17] design memory-efficient probabilistic data structures that can be updated at line rate, have low memory footprints, and high accuracy. At the time of this writing, Elastic Sketch is cited over 400 times by follow-up work across many academic research groups (e.g., CMU, Princeton, University of Pennsylvania, Technion, KTH). Some of them try to further optimize its memory usage, speed, or accuracy; some re-purpose its design for more telemetry tasks; and some leverage its implementation for P4 compiler research.

Secure hardware architecture for SmartNICs. Cloud providers are deploying various SmartNICs with wimpy-yet-power-efficient RISC cores to offload simple network functions such as network virtualization and traffic scheduling. Unfortunately, vast cloud tenants are barred from the efficiency benefits of SmartNICs, because they are not allowed to run their own customized functions on SmartNICs. The root cause is that modern SmartNICs provide little isolation between the network functions belonging to different tenants; these NICs also do not protect network functions from the datacenter-provided management OS running on the NIC. My S-NIC [18] project proposes minimal changes to

SmartNIC hardware, so that datacenters can provide offloaded functions with strong isolation, while preserving most of the total-cost-of-ownership benefits with minimal performance degradations. S-NIC's designs target various commodity multi-core SmartNICs, and explicitly isolate their IO subsystems and on-NIC accelerators.

Future Research

Building on my past experiences in networking, memory management, OS kernels, and datacenter applications, I am excited to apply my full-stack optimization approach with cross-layer codesign to the following problems.

Deployment-friendly approaches to memory efficiency via malloc queueing. Previous work on increasing memory efficiency is mostly *not* deployment-friendly, requiring modifying either OS kernels [10] or application code and third-party libraries [11]. In search of deployment-friendly approaches, I have a preliminary insight around separating the provisioning of average memory usage and bursty usage: application's peak memory usage is usually dominated by bursty, large memory allocations (e.g., temporarily loading a large file into memory); if one can time-interleave such allocations from different applications to avoid their memory peaks coinciding with each other, the overall memory provisioning can be reduced, thus improving memory efficiency. One way to implement time-interleaving is overwriting the `Malloc()` function to strategically delay memory allocations, which I believe is far more deployment-friendly than previous work. I call this approach malloc queueing, and it would mostly target batch processing applications whose performance is not sensitive to the incurred memory allocation delays.

eBPF for accelerators and more. eBPF programming language features verified safety and liveness, and has been widely applied to packet processing in kernels and SmartNICs. I intend to extend eBPF to manage heterogeneous hardware accelerators, and build a generic and easy-to-use programming interface between accelerators and application developers. Example accelerators include GPU and FPGA for massively parallel computing, and U2F (Universal 2nd Factor) keys for security. Through verification, this interface would enable strong safety and liveness guarantees for computations running on these accelerators. Besides hardware, I believe eBPF can shed light on more software applications. I intend to explore the following ones: (1) fast task scheduling (e.g., work stealing) for distributed computation framework like Ray [19], and (2) generic shared logs to support various distributed data structures [20]. Both applications would benefit from the efficient network IOs via kernel offloads, and require addressing challenges from the constrained programming model in eBPF.

Resource efficient machine learning. Machine learning (ML) workloads such as the training and inference of Large Language Models (LLMs) are extremely resource-hungry, requiring expensive accelerators like GPUs. I intend to take a full-stack approach to improve the resource efficiency of ML workloads, covering GPU memory efficiency and compute efficiency. One direction is applying far memory techniques to LLM training and inference by swapping to CPU memory. For performance, I plan to codesign far memory swapping with the memory access patterns of LLM weights and key-value cache, e.g., different access frequencies for different weights due to the attention mechanism in LLMs. Another direction is developing a unified GPU memory abstraction that allows easily accessing remote GPU memory over high-speed networks such as NVLink; this kind of GPU memory pooling would help reduce memory stranding and fragmentation caused by dynamic memory allocations in ML workloads. For performance, I plan to codesign such memory pooling with ML workload characteristics, e.g., allowing relaxed consistency. Finally, I am interested in fine-grained GPU kernel scheduling at the microsecond scale possibly with preemption; the goal is to efficiently multiplex GPU compute resources among multiple jobs without losing performance.

Datacenter-scale distributed runtime. A long-term goal of my research is to build a datacenter-scale distributed runtime to not only simplify application development but also increase the whole datacenter efficiency and evolvability. This distributed runtime sits between applications and datacenter resources: (1) for applications, it provides generic and stable interfaces to use compute, memory, storage, and accelerators, and customizable fault tolerance and recovery schemes based on application needs; (2) for resources, it eschews the conventional reservation-based provisioning strategy, and instead provisions resources in a best-effort manner to achieve high resource efficiency.

Today's datacenters have already provisioned network resources in a best-effort manner, and I plan to expand this strategy to cover more resources like compute, memory, storage, and accelerators. For these new best-effort resources, many networking techniques like congestion control can be applied to enable efficient fair sharing. However, unlike the network resources that are delay-tolerable for applications, other resources especially the memory are not (think of out-of-memory errors). To address this challenge, I intend to leverage techniques like far memory and malloc queueing to create a delay-tolerable memory abstraction, at the cost of lower resource utility than normal memory.

References

[1] **Yang Zhou**, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu. Electrode: Accelerating Distributed Protocols with eBPF. In *Proceedings of USENIX NSDI*, 2023.

[2] **Yang Zhou**, Xingyu Xiang, Matthew Kiley, Sowmya Dharanipragada, and Minlan Yu. DINT: Fast In-Kernel Distributed Transactions with eBPF. In *Proceedings of USENIX NSDI*, 2024.

[3] **Yang Zhou**, Hassan Wassel, James Mickens, Minlan Yu, and Amin Vahdat. Mew: Efficient Inter-Server Load Balancing for Microsecond-Scale RPCs. Under submission.

[4] **Yang Zhou**, Hassan Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David Culler, Hank Levy, and Amin Vahdat. Carbink: Fault-Tolerant Far Memory. In *Proceedings of USENIX OSDI*, 2022.

[5] **Yang Zhou**, Ying Zhang, Minlan Yu, Guangyu Wang, Dexter Cao, Eric Sung, and Starsky Wong. Evolvable Network Telemetry at Facebook. In *Proceedings of USENIX NSDI*, 2022.

[6] William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. Revisiting the Open vSwitch Dataplane Ten Years Later. In *Proceedings of ACM SIGCOMM*, 2021.

[7] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating Interference at Microsecond Timescales. In *Proceedings of USENIX OSDI*, 2020.

[8] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the Next Generation. In *Proceedings of ACM EuroSys*, 2020.

[9] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. Imbalance in the Cloud: An Analysis on Alibaba Cluster Trace. In *IEEE International Conference on Big Data (Big Data)*, 2017.

[10] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can Far Memory Improve Job Throughput? In *Proceedings of ACM EuroSys*, 2020.

[11] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. AIFM: High-Performance, Application-Integrated Far Memory. In *Proceedings of USENIX OSDI*, 2020.

[12] **Yang Zhou**, Tong Yang, Jie Jiang, Bin Cui, Minlan Yu, Xiaoming Li, and Steve Uhlig. Cold Filter: A Meta-Framework for Faster and More Accurate Stream Processing. In *Proceedings of ACM SIGMOD*, 2018.

[13] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, **Yang Zhou**, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic Sketch: Adaptive and Fast Network-Wide Measurements. In *Proceedings of ACM SIGCOMM*, 2018.

[14] Tong Yang, **Yang Zhou**, Hao Jin, Shigang Chen, and Xiaoming Li. Pyramid Sketch: A Sketch Framework for Frequency Estimation of Data Streams. *Proceedings of the VLDB Endowment*, 2017.

[15] **Yang Zhou**, Omid Alipourfard, Minlan Yu, and Tong Yang. Accelerating Network Measurement in Software. *ACM SIGCOMM Computer Communication Review*, 2018.

[16] Omid Alipourfard, Masoud Moshref, **Yang Zhou**, Tong Yang, and Minlan Yu. A Comparison of Performance and Accuracy of Measurement Algorithms in Software. In *Proceedings of ACM Symposium on SDN Research (SOSR)*, 2018.

[17] Zhuochen Fan, Gang Wen, Zhipeng Huang, **Yang Zhou**, Qiaobin Fu, Tong Yang, Alex X Liu, and Bin Cui. On the Evolutionary of Bloom Filter False Positives - An Information Theoretical Approach to Optimizing Bloom Filter Parameters. *IEEE Transactions on Knowledge & Data Engineering*, 2022.

[18] **Yang Zhou**, Mark Wilkening, James Mickens, and Minlan Yu. SmartNIC Security Isolation in the Cloud with S-NIC. Under submission.

[19] 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 *Proceedings of USENIX OSDI*, 2018.

[20] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed Data Structures Over a Shared Log. In *Proceedings of ACM SOSP*, 2013.

[This page intentionally left blank.]

Teaching Statement

Yang Zhou

I greatly enjoy the rewards of teaching and mentoring students. For me, the rewards consist of two significant parts: (1) the pride and fulfillment when my teaching helps students carry out their studies smoothly and when my mentored students grow into independent researchers, and (2) the interesting future research directions inspired or confirmed during teaching and mentoring. Driven by these rewards, I have taught as a teaching assistant and as a small-group "supervisor", and mentored four undergraduates and five junior PhD students in their research. Based on my research background, I am qualified to teach undergraduate courses of computer networks, operating systems, distributed systems, and algorithms and data structures, and graduate courses of data center networking and dataplane operating systems (detailed later).

Mentoring Experience and Methodology

I have mentored four undergraduates on system research, and informally mentored five junior PhDs on their research ideas, internship applications, and study experience at Harvard. Among the four undergraduates, one (Zezhou Wang) published an NSDI'23 paper with me and went to University of Washington (UW) as a system PhD; two of them (Xingyu Xiang and Matt Kiley) co-authored an NSDI'24 submission with me, and are about to apply for system PhDs as well as the rest one. Such mentoring brings me enormous pride, e.g., seeing Zezhou gets into the UW PhD program. It inspires my future research—working with Zezhou on eBPF sparks two follow-up projects: one has become the NSDI'24 submission, and another is showing promising results. Below I summarize my mentoring methodology:

- *Building students' confidence.* It is well-known that confidence is crucial for students, but how to build their confidence is challenging. One way I find helpful is respecting students' thoughts by giving them enough freedom to try their thoughts while keeping an eye on the big agendas and goals. Another way is connecting them to experts upon entering a new field, avoiding the steep learning curves overwhelming or destroying their confidence. The experts, who could be the mentors themselves, would point out the proper materials or steps for quick ramp-ups.
- *Encouraging students to form their own opinions and tastes.* I encourage and anticipate students to form their own opinions about systems, develop their own tastes on promising research problems, and stick with them. I do not worry too much about if students' opinions/tastes are wrong, as once they go deep into specific directions they believe, they will learn extensive experiences and insights to refine their previous opinions/tastes.
- *Collaborating widely.* Wide collaboration across industry and academia is especially beneficial for practical system research, and mentors should play the important role in connecting students with proper researchers in the wild. For example, my fault-tolerant far memory project Carbink was collaborated with Google via my co-advisor's connections, and then inspired by Google's desire for high availability. However, collaborating with industry usually requires teasing out real research challenges, while not being misled by massive engineering details; advisors should leverage their experience to help students (especially junior PhDs) navigate efficiently in this space. For another example, my eBPF-for-Paxos project Electrode would not be possible without the collaboration with Sowmya Dharanipragada who is a distributed system PhD at Cornell. Going forward, I would like to expand collaborations to theory, machine learning, architecture, programming languages, etc.

Teaching Experience and Philosophy

System course teaching: I was the teaching assistant (TA) for a computer system course, the Harvard CS145 Networking at Scale, along with an undergraduate TA. This course features eight P4-switch related projects, three of which are designed and developed by me including detailed guides and skeleton code. I held three one-hour sections covering network programming, background knowledge for projects, and handy tools for developing and debugging. Other duties include holding weekly office hours, answering students' questions on forums, and grading projects. In addition to TA, I also had a guest lecture experience at UC Berkeley on far memory techniques in data centers, mainly facing junior graduate students from architecture areas. I started from common and accessible facts like resource utilization and

DRAM prices, then explained why data center operators have an interest in far memory, and finally discussed my work in this space.

Algorithm course teaching: I was the small-group supervisor for the Algorithm Design and Analysis course at Peking University as an undergraduate. This role requires supervising around 14 students in small classes, giving recitations, teaching advanced algorithms and data structures, preparing new problem sets and quizzes, and grading, all on a weekly basis. I extensively introduced non-textbook topics related to my undergraduate research of probabilistic data structures. Although time-consuming, being such a supervisor is truly gratifying, especially when students understand my research and try various optimizations as their final course projects. One student (Yicheng Jin) in my small class is now pursuing a computer science PhD at Duke University.

Introductory teaching: I taught non-CS audiences about the Internet from a computer science perspective during the English Language Program at Harvard. It was a slightly difficult yet fun experience especially when I told the audience that Internet data is transmitted in small packets: they were shocked and immediately asked why, and then I gave them detailed yet understandable explanations until they grasped the design philosophy behind it. This experience gave me a good sense of how to teach introductory courses in the future. Below I summarize my teaching philosophy:

- *Building safe and inclusive environments.* Students in the same class usually have different prior knowledge; thus it is important to create safe and inclusive environments to make students feel they are welcome to ask both the simplest questions and challenging ones. I got such first-hand experiences when I took my co-advisor James Mickens' CS263 System Security course: it has the most open class environment I have ever seen because of James' unique humor, and students ask so many interesting questions during the class. As a result, I personally learned so much security knowledge, though my research is on networked systems.
- *Focusing on hands-on experiences.* I believe the best way to learn computer systems is through reading, running, debugging, and hacking well-written codebases in a hands-on manner. My personal experience in learning dataplane operating systems exactly follows this pattern: after reading relevant papers, I could not understand how specific designs get implemented and contributed to the final performance; then I decided to read the codebase of a dataplane OS called Caladan [1], and run and debug it; finally, I built my own research prototype atop it. After the process, my understanding of dataplane OSes became much clearer, and I gradually began appreciating the merits of various designs in this space. For future system courses I teach, I would like to incorporate well-written teaching systems, such as the WeensyOS [2], into my agenda to help students gain hands-on experiences.
- *Promoting critical thinking on the pros and cons of techniques.* I learned this from the Harvard CS260r Projects and Close Readings in Software Systems—Serverless Computing by Eddie Kohler, where he discussed serverless computing research from a traditional system research perspective. He showed impressive critical thinking on the pros and cons of serverless computing, and helped us grasp the real novel components of this paradigm without deifying any new terms. I plan to apply a similar philosophy to my teaching, encouraging students to critically think about new techniques around us, such as the emerging LLM techniques.

Course plans: In addition to the aforementioned undergraduate courses based on textbook knowledge, I would like to hold two advanced graduate courses and a seminar course based on my research:

- *Data center networking:* I will discuss how modern data centers design and build high-performance network fabrics including topology, routing, congestion control, fault tolerance, load balancing, etc.
- *Dataplane operating systems:* I will discuss how the OS evolves to keep up with the fast hardware in data centers, including user-space networking, efficient threading, light-weight isolation, etc.
- *System seminar course:* I will invite a broad set of system researchers from both academia and industry to give talks on various system research topics, and foster potential collaborations with students.

References

[1] The Caladan authors. Caladan opensource. `https://github.com/shenango/caladan`.

[2] Eddie Kohler. Harvard CS61 Systems Programming and Machine Organization (2023): WeensyOS. `https://cs61.seas.harvard.edu/site/2023/WeensyOS/`.

Diversity Statement

Yang Zhou

I view DEI as the basic soil for growing humanity and excellence in society, including the academic community; it is about the daily respect for people regardless of their self-identifications, and self-introspection on "whether I want to be treated like what I treat others". Everyone has the duty to foster DEI in her/his surroundings, because that eventually determines how the society will treat them in one day. Here, I would like to sample my and my family's experiences of being underrepresented due to educational background, language, political affiliation, and ethnic origin, to motivate how I grow awareness of the challenges faced by underrepresented populations and the importance of DEI, and possible ways to foster DEI—some I have adopted and some I plan to do.

I am a first-generation college student, so my parents could hardly give me advice on how to succeed in college and in my PhD studies. However, I was lucky to receive tremendous emotional support from them. I was also fortunate to receive academic mentorship from a variety of professors and student peers. Thus, I am proud to be a faculty job applicant today, and I look forward to creating a sharing and inclusive environment in the classroom and in my research group.

As a first-generation immigrant to the US, one of the first challenges that I faced was mastering the English language. At Harvard, I greatly benefited from the university's English Language Program (ELP), which offered weekly lectures by experienced English teachers, and recruited native English speakers from the university to serve as language partners. The ELP experience showed me how community building is a critical aspect of helping students integrate into challenging environments. As a professor, I hope to make students aware of programs like the ELP that target specific barriers to students' success (e.g., language issues, or a lack of adequate high school preparation for college-level classes).

Fifty years ago, my uncle was denied admission to his dream civil aviation university, despite his excellent academic performance and physical fitness. He was rejected because his father (my grandfather) was a combat medic for the Chinese Nationalist Party–the party who had fought with the Communist Party of China that founded the People's Republic of China. Such political discrimination prevented a whole generation of my uncles from participating in activities that were even slightly related to military service. I was told this experience at a very young age; thus, I have always known that the political environment of the past can influence personal outcomes in the present.

My mother and her family are Hui Chinese, one of the ethnic minorities that comprises 0.79% of the total Chinese population. Being an ethnic minority in China often results in discrimination by the majority Han population. For example, a popular stereotype is that Hui Chinese are thieves. Fortunately, my parents always taught me to not treat people by their ethnicity, race, or religion. As a result, I am always conscious of potential biases that may impact my interactions with others, and I hope to support DEI principles as a professor.

My Past Contributions to Advancing DEI

I have participated in various activities that supported DEI via mentoring and teaching.

Mentoring: During the summers of 2022 and 2023, I mentored four undergraduate students for research internships at Harvard: three came from non-US schools, with two being in the US for the first time. To help the students get familiar with systems research (and life in the US), I held weekly meetings with each student, talking about not only research but also various cultural acclimation challenges that I had experienced during my own PhD. At the time of this writing, one of them has co-authored a paper with me that was published at a premier system conference. This student was also accepted to the University of Washington as a computer science PhD student. The other three students have also decided to apply to systems PhD programs, including one that was hesitating for a long time before working with me. I also consistently (monthly) shared my research and internship experiences with five junior PhD students over the past two years. All of them are non-native English speakers and are non-white.

Occasionally, I received email inquiries from PhDs who are in other research areas or from underrepresented minorities; I often scheduled one-to-one meetings to learn about their difficulties or puzzles. For example, Jessica Quaye, originally from the Republic of Ghana in West Africa, was interested in system research though she is in an architecture research group. I had long meetings with her both in person and online, and introduced her to my co-advisor Minlan Yu to identify potential opportunities for collaboration and advising.

Besides one-to-one mentoring, I also participate in one-to-many panels to share my research experience with junior system PhDs. For example, I was a panelist for the "Getting started with systems research" panel [1] organized by Students@Systems in 2022. The video recording for the panel is freely accessible online to help systems PhD students regardless of their university or physical location.

Talking: Academic networking (e.g., talking to peer researchers at conferences) is crucial to the success of a PhD student. However, junior graduate students are often afraid of professional networking, e.g., due to fears about having little experience or being from less prodigious schools. However, I vividly remember how, at a conference, James Mickens (one of my co-advisers) stood in front of the door of a breakout room and publicly said "I am James, a Professor at Harvard, and you are welcome to talk to me!" This event inspired me to proactively interact with junior students during conferences, to talk about mutual research interests and identify potential collaboration opportunities. I also like to encourage poster presenters for their research, especially when there are no people who are currently engaging with their posters.

I also talk to undergraduates and high school students regarding computer science research. For example, in October 2022, I gave a research talk at a Harvard AM/CS/EE PhD recruitment event (accessible to all US universities) which targeted students "that hold membership in an underrepresented and/or historically minoritized group in STEM." In 2022, I also gave talks at the Harvard SEAS Undergraduate Research Open House and the SEAS Research Showcase, targeting Harvard freshman and sophomore undergraduates. These talks were well-received, with several undergraduates in the audience later contacting my research lab to learn more about participation opportunities; I still mentor one of these undergraduates. Going back to the time when I was an undergraduate, I had the privilege to talk to juniors in my alma mater high school on why a computer science major is a good college major. Some of these students still contact me for advice.

Teaching: I make an explicit effort to help students with little prior exposure to computer science, and I try to promote inclusiveness during teaching. When I was the small-group "supervisor" for the Algorithm Design and Analysis course at Peking University, I realized that some students lacked high school experience with programming contests; these students often found it hard to catch up with peers who did have this experience. To help them, I wrote step-by-step, thorough explanations for the algorithms discussed in class, and I handed out these explanations after class. When TA'ing a course at Harvard University, I answered all questions that appeared in the Ed forum, no matter whether the questions were anonymous or not, to keep everyone's learning progress on track.

My Future Plans for Fostering DEI

Going forward, as a faculty member, I plan to take the following actions:

- *Advising:* Actively recruiting underrepresented students, being attentive to any anti-DEI atmosphere in my research group, and explicitly adopting counter-measures to foster DEI with affirmative actions.
- *Connecting:* Reducing the barriers of students finding research opportunities by organizing mutual-connecting programs like UCB DARE [2]—matching students with faculty members for research.
- *Teaching:* Being attentive to any students with weaker prior knowledge in my classes, and helping them build confidence with support on a case-by-case basis.
- *Daily life:* Being kind to people I meet, no matter their age, color, disability, gender, ethnicity, politics, religion, education, language, and more. I believe "kindness is the ultimate nobility" [3].

References

[1] Student@Systems. A panel on "Getting started with systems research". `https://students-at-systems.org/pages/events/getting-started-with-systems-research.html`.

[2] UC Berkeley. DARE: Diversifying Access to Research in Engineering. `https://dare.berkeley.edu/`.

[3] Amin Vahdat. SIGCOMM Lifetime Achievement Award 2020 Keynote (48m44s): kindness is the ultimate nobility. `https://youtu.be/Am_itCzkaE0?t=2924`.

This paper is accepted at NSDI'24 and under camera ready

DINT: Fast In-Kernel Distributed Transactions with eBPF

Yang Zhou* Xingyu Xiang†* Matthew Kiley Sowmya Dharanipragada‡ Minlan Yu
Harvard University †*Peking University* ‡*Cornell University*

Abstract

Serializable distributed in-memory transactions are important building blocks for data center applications. To achieve high throughput and low latency, existing distributed transaction systems eschew the kernel networking stack and rely heavily on kernel-bypass networking techniques such as RDMA and DPDK. However, kernel-bypass networking techniques generally suffer from security, isolation, protection, maintainability, and debuggability issues, while the kernel networking stack supports these properties well, but performs poorly.

We present DINT, a kernel networking stack-based distributed transaction system that achieves kernel-bypass-like throughput and latency. To gain the performance back under the kernel stack, DINT offloads frequent-path transaction operations directly into the kernel via eBPF techniques without kernel modifications or customized kernel modules, avoiding most of the kernel stack overheads. DINT does not lose the good properties of the kernel stack, as eBPF is a kernel-native technique on modern OSes. On typical transaction workloads, DINT even achieves up to 2.6× higher throughput than using a DPDK-based kernel-bypass stack, while only adding at most 7%/16% average/99th-tail unloaded latency.

1 Introduction

Serializable distributed transactions are important programming abstractions and building blocks for distributed data center applications, such as object store and online transaction processing (OLTP) systems. With the advance of battery-backed DRAM [14] and fast NVRAM [10], the bottleneck of distributed in-memory transactions shifts from the storage to the networking. This has spurred extensive research on how to implement fast distributed in-memory transactions using kernel-bypass networking techniques, such as RDMA [14, 29, 80] and DPDK [6, 28]. One of the key assumptions for these works is that kernel-bypass is the key to realizing fast distributed in-memory transactions that match the underlying hardware speed.

However, kernel-bypass is not a panacea—it essentially trades security [68], isolation [37,38], protection [3,62], maintainability [52,77], and debuggability [69,77] for performance. In addition to these issues, kernel-bypass techniques such as DPDK usually burn one or more CPU cores for busy-polling even at low loads; this is usually non-acceptable in public cloud deployments due to per-core pricing [78]. These issues collectively have led to the well-known Open vSwitch giving up DPDK-based dataplane designs recently [77].

Instead, we choose to embrace the kernel networking stack with interrupt-driven packet processing. The kernel networking stack provides nice properties of good security, isolation, protection, maintainability, debuggability, and load-aware CPU scaling—but not performance. Its poor performance mainly comes from three sources: heavy-weight networking stack traversing [19, 87], user-kernel context switching [87], and interrupt handling.

This paper therefore asks: *can we remove such kernel stack overheads while keeping all of its nice properties for distributed in-memory transactions?* To this end, we follow a decade-old methodology called *extensible kernels* [4], and realize it in modern OS kernels without any kernel code modifications or customized kernel modules. The key enabler is the eBPF technique that allows users to run customized functions easily, safely, and efficiently inside the kernel networking stack at run time. With eBPF, we can run transaction processing logic at the early stage of the kernel networking datapath without going to the user space, avoiding most of the kernel networking stack functions and user-kernel context switching. For the overhead of interrupt handling, it could be amortized by adaptive batching [3] that the kernel networking stack NAPI [33] already did. Besides the potential performance benefit, eBPF is a kernel-native technique shipped with and well-maintained by each release of modern Linux kernels. Due to its safety and kernel-native nature, it has been rapidly adopted by applications and cloud vendors [2, 16, 54]. For example, Meta runs over 40 eBPF programs on every server with ∼100 loaded on demand [74].

We introduce DINT[1], which accelerates distributed transaction systems using eBPF. DINT handles as many transactions as possible in the kernel to improve their critical path performance. In distributed transaction systems, a transaction usually involves three components in its critical path: it first acquires various locks from a lock manager, then reads relevant key-values from a key-value store and does local updates, next logs key-value updates to a log manager, and finally commits key-value updates to the key-value store. Offloading the three components to eBPF is challenging because *eBPF has a constrained programming model (for kernel safety).*

To address this challenge, our key idea is to redesign transaction-related data structures following the principle of *kernel-offloading* for frequent critical paths to guarantee high performance, and use *user space programs as backups* for rare paths to support full functionalities.

*Equal contribution

[1] DINT as a noun is an archaic word, meaning force and power.

First, the lock manager normally maintains many locks with efficient indexing and complex locking operations. However, it is hard for eBPF to handle hash collision during indexing, because eBPF only allows statically-bounded loops. Further, it is also hard to maintain shared lock states because eBPF does not support common synchronization primitives like Mutex. To address these issues, the DINT lock manager embraces lock sharing to avoid the slow and complex hash collision handling, and directly leverages low-level eBPF atomics to implement transaction locking.

Second, the key-value store normally stores a large number of key-values with different sizes, and requires frequent lookups and updates. However, eBPF does not support dynamic memory allocations, causing low memory efficiency for the key-values. To address these issues, the DINT key-value store directly stores small key-values, which dominate in transaction workloads [12, 46, 75], in kernel memory using a set-associative cache, while leaving large key-values to the user space, avoiding dynamic memory allocations in eBPF. DINT further designs a write-back mechanism with Bloom filters [5] to efficiently handle most key-value lookups and updates in the kernel, while guaranteeing the key-value consistency across the user and kernel.

Third, for the log manager, DINT designs efficient per-CPU log buffers to record logs directly in eBPF, while supporting log replaying from the user space during failure recovery.

We evaluate DINT on two OLTP workloads: a read-intensive TATP workload [46] and a write-intensive SmallBank workload [75]. DINT achieves up to 2.6× higher throughput than using a recent well-engineered kernel-bypass stack based on DPDK (i.e., Caladan [17]), while only adding at most 7% and 16% unloaded latency for the average and 99th-tail respectively. We achieve even higher throughput mainly because the kernel-bypass baseline builds a high-level abstraction for packets and uses a (user-space) threading-based programming model, leaving some performance on the table, while DINT works directly on raw packets in an event-driven way. DINT's designs are also generic to transaction protocols to some extent—it easily supports an OCC (opportunistic concurrency control) protocol for the read-intensive workload and a 2PL (two-phase locking) protocol for the write-intensive workload.

In summary, this paper makes three contributions:

- We design and implement a high-performance distributed transaction system under the widely-deployed kernel networking stack and the widely-available common commodity NICs, with the key idea of kernel offloading via eBPF.
- We are the first to experimentally show that a distributed transaction system under the kernel networking stack can achieve kernel-bypass-like performance and latency.
- We identify a series of open problems for networking stack, transaction protocols, and eBPF research (§6).

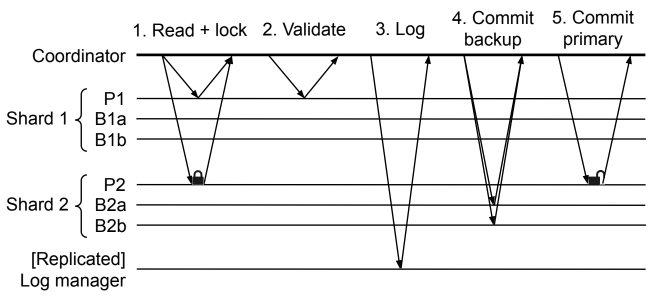


Figure 1: The FaSST [29] transaction protocol with two data shards and three-way replication. P = primary and B = backup. This example transaction reads from the shard 1 and writes to the shard 2.

2 Background

2.1 Distributed Transactions

We focus on serializable distributed transactions over a replicated sharded in-memory key-value store with replicated logging to handle failures. Along with recent works [14, 29, 71, 80] in this space, we assume logging into fast persistent storage like battery-backed DRAM or NVRAM (instead of disks) to match in-memory transaction speed, and having a separate fault-tolerance configuration manager to handle machine failures off the critical path of transaction processing. These works usually employ transaction protocols consisting of optimistic concurrency control (OCC) and two-phase commit for distributed atomic commit, and primary-backup replication to support high availability. Below, we briefly go through the critical path of one of such protocols from FaSST [29].

In the FaSST transaction protocol, each transaction has a set of keys to read (i.e., read-set) and a set of key-values to write (i.e., write-set), and a transaction coordinator issues transaction requests to finish each transaction. As shown in Figure 1, the primary in each shard runs a *lock manager*; both the primary and backups run a replicated *key-value store*; a set of servers run a replicated *log manager* (could just be on the primary and backups). To finish a transaction, the transaction coordinator executes the following phases:

1) **Read+lock:** the coordinator reads all values + locks + versions for the read-set and locks all key-values for the write-set. If any key-value in the two sets is already locked, the transaction aborts. The coordinator buffers key-value writes/updates locally.
2) **Validate:** the coordinator reads again all locks + versions in the read-set, and checks if any read-set value has been changed or locked since the first phase. If so, the transaction aborts.
3) **Log:** the coordinator writes a transaction record containing the write-set's key-values and their versions into the replicated log manager.
4) **Commit backup:** the coordinator updates the write-set values to corresponding backup replicas.
5) **Commit primary:** the coordinator updates the write-set values to corresponding primary replicas, increments key-value versions, and unlocks key-values.

Besides the OCC, there are many more concurrency control protocols for serializable distributed transactions. Another well-known one is two-phase locking (2PL) used in Spanner [9]; it locks before each read and write, and is suitable for write-intensive workloads. More advanced protocols include MDCC [39], Tapir [85], Janus [57], ROCOCO [56], which reduces the number of transaction phases by co-designing concurrency control and replication, and allows more concurrency by tracking fine-grained transaction dependencies.

Distributed transactions inside a data center typically have bottlenecks on the networking stack. For example, when we run the above transaction protocol using a typical OLTP workload under the kernel UDP stack (see §5.2 for a detailed setup), we observe 64% of CPU time is spent on traversing the kernel networking stack, 16% is on the user-kernel context switching, and 12% is on the interrupt handling. This further motivates the huge performance benefits of kernel offloading by avoiding kernel stack overheads.

2.2 eBPF in Kernel Networking Stack

eBPF basics: eBPF (extended Berkeley Packet Filter) is a kernel-native mechanism to let users write *safe, customized* programs that run inside the OS kernel without kernel code modifications or customized kernel modules. Users typically write a high-level C-like eBPF program that gets compiled into low-level eBPF bytecode by Clang/LLVM. Users can then load the eBPF bytecode to predefined attachment points or the so-called *eBPF hooks* in the kernel. Upon loading, the kernel will first verify if the eBPF bytecode meets the safety (e.g., no out-of-bounds memory accesses) and liveness (i.e., it will always terminate in finite steps) requirements. If so, the kernel will compile the eBPF bytecode to native machine code, and run it in a kernel-embedded virtual machine in an event-driven manner; otherwise, the kernel will reject it.

The Linux kernel networking stack has two main eBPF hooks: XDP (eXpress Data Path) [21, 66] and TC (Traffic Control) [49]. The XDP hook only works for ingress packets, and triggers the eBPF program immediately after the NIC driver receives the packet upon NIC interrupts, before `sk_buff` [34] creation. The TC hook works for both ingress and egress packets, and triggers the eBPF program between the NIC driver layer and UDP/TCP layers. For ethernet packet forwarding, TC has lower performance than XDP, as it has run more kernel networking stack functions.

eBPF maps: eBPF programs are event-driven, therefore program states that cross different invocations must be stored in a global heap-like memory region—eBPF maps are exactly for this purpose. eBPF maps are a variety of built-in data structures in the kernel to maintain eBPF program states with various eBPF helper functions. An eBPF map could contain up to $2^{32}-1$ elements each with maximum $2^{32}-1$ bytes, with total size bounded by the server memory; it must be declared and created statically with a fixed size. Typical eBPF maps include arrays, per-CPU arrays, stacks, and queues [48], with

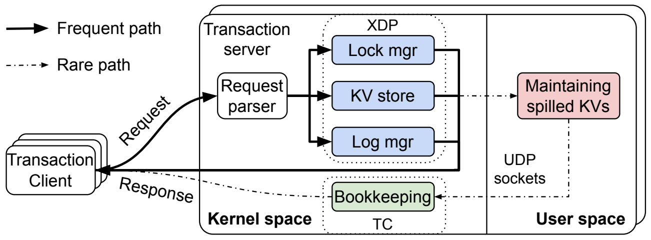


Figure 2: DINT's high-level architecture.

lookup and update functions [32]. The power of eBPF maps is that they can be shared among different eBPF programs and user-space processes. For example, the eBPF program attached to XDP can share an eBPF map with another program on TC and even with a user-space process.

eBPF programming constraints: Due to the safety and liveness verification by the kernel, eBPF programming has some constraints. Perhaps the most important one is not supporting dynamic memory allocations, as correctly handling memory allocation failure and verifying no memory leaks are challenging for eBPF. The second constraint is that eBPF only supports statically-determined bounded loops to ensure liveness. Finally, eBPF lacks high-level thread synchronization primitives such as Mutex. This is because eBPF code runs inside the kernel, and arbitrary/unexpected kernel sleeping by Mutex is dangerous. Instead, eBPF only supports spinlock (i.e., `bpf_spin_lock` [47]) with deep constraints that make it less useful: one cannot call any functions (including built-in eBPF helper functions) while holding the lock, and must release the lock before forwarding/dropping the packet.

3 DINT Design

Figure 2 shows the high-level architecture of DINT. DINT assumes an asymmetric transaction model or the so-called client-side transaction model, similar to [41, 55, 59, 85]. In this model, each transaction client, as the transaction coordinator, sends transaction requests to transaction servers to finish locking, key-value, and logging operations, and then receives responses. As described in Section 6, DINT could also support the symmetric model used in [14, 29, 80]. Like prior works, DINT shards transactions states (i.e., locks, key-values, and logs) among servers, and uses three-way replication and logging for high availability. DINT is generic to a variety of transaction protocols, and currently supports two different ones: a 2PL-based protocol and an OCC-based protocol similar to FaSST [29].

Offloading request frequent path to kernel: To achieve high-performance transaction processing, DINT offloads frequent-path states and operations into the kernel, avoiding kernel stack overheads. Each DINT transaction server maintains *most of* its transaction states in the kernel memory via eBPF maps, and serves *most of* its transaction requests directly in the kernel via an eBPF program attached to the XDP hook. Since eBPF programs cannot generate new packets by themselves, DINT reuses the request packet by modifying its payload to carry the response message, and forwards it back to clients as the response.

Userspace as backups: To support the full functionalities of transaction processing, DINT handles rare-path states and operations in the user space. Each DINT transaction server runs a user-space process listening on UDP sockets to receive and handle *a small portion* of transaction requests that cannot be served directly in the kernel. Transaction responses returned from the user-space process will go through a bookkeeping eBPF program attached to the TC hook, which helps maintain transaction states in eBPF maps, e.g., releasing some internal locks (not transaction locks).

DINT uses UDP protocol between transaction clients and servers to allow easy parsing of transaction requests and responses in eBPF programs. While UDP protocol is lossy, packet losses happen rarely in modern data centers as shown by prior works [28,29,64]. When packet losses happen during severe network hardware failures, DINT would detect such losses using coarse-grained client-side timeouts and handle them by the transaction protocols, similar to FaSST [29]. DINT targets accelerating the handling of transaction requests/responses that can fit into one ethernet packet, i.e., up to 9KB for jumbo frames. This works well for transactions with mostly small key-values, which are quite common in many transaction processing workloads [12,29,46,75,80]. For large key-values, DINT could just pass them to the user-space process to handle, at the cost of lower throughput.

3.1 DINT Lock Manager

The DINT lock manager is responsible for the transaction concurrency control, i.e., controlling how multiple transaction clients concurrently access individual key-values. Such concurrency control mainly involves quickly indexing lock states by lock IDs and maintaining the shared lock states. These two operations are challenging for the constrained programming model in eBPF that lacks dynamic memory allocations, only supports bounded loops, and has nearly no high-level thread synchronization primitives like Mutex (§2.2). For example, lock state indexing usually requires implementing a hash table in eBPF; however, handling hash collisions is nearly impossible or very inefficient in eBPF for either open hashing that requires dynamically allocating a new hash table entry or closed hashing that may require unbounded loops.

To support efficient lock state indexing and shared lock state maintenance in eBPF, DINT leverages two techniques:

- lock sharing to avoid handling hash collisions. Lock sharing means two lock IDs may get mapped to and use the same lock state. DINT further designs a mechanism to avoid possible deadlocks during lock sharing.
- leveraging low-level eBPF atomics [23] to carefully synchronize shared states operations.

Lock sharing: DINT leverages eBPF array maps (i.e., `BPF_MAP_TYPE_ARRAY` [32]) to implement static tables of lock states in the kernel space. Typical lock states include lock status bits, sharer counters (for read-write locks), etc. Each lock ID gets mapped to one shared lock state in the table via a hash function, and later lock acquiring/releasing operations just work on this lock state. Lock sharing avoids handling tricky hash collisions, at the cost of slightly increasing the failure probability when acquiring locks.

However, deadlocks could happen if a transaction client tries to acquire two locks that get mapped to the same lock state (assuming exclusive locking). This is because: the first acquiring operation succeeds, while the second acquiring fails/blocks; however, the first acquiring will not release the lock until the transaction finishes, while the second acquiring always blocks the transaction progress. To resolve such possible deadlocks, DINT lets the lock manager check if any two exclusive lock acquiring operations on the same lock state come from the same transaction client, by maintaining a holder client ID (e.g., IP and port pair) for each exclusive lock; if so, the lock manager directly returns a locking success message.

By leveraging low-level eBPF atomics, DINT supports a variety of locking mechanisms for concurrency control protocols, including the basic read-write locking for 2PL and version-based locking for OCC, in a fail-and-retry manner [8,18,81]. Supporting more advanced concurrency control protocols [39,56,57,85] is also possible in DINT, as they are essentially underpinned by the two basic locking mechanisms; we discuss further in Section 6.

Read-write locking: This locking mechanism includes two types of locks: exclusive locks and shared locks. Transaction clients send lock acquiring/releasing requests with lock IDs to the lock manager, and the manager responds with either success or failure. Lock acquiring requests may receive failure responses, while lock releasing requests always receive success responses. If a client receives a failure response, it will re-send the lock acquiring request again after an optional period of time, until receiving the success response (i.e., fail-and-retry).

To implement the read-write locking, the DINT log manager maintains a per-lock status bit indicating if this lock is held exclusively, and a per-lock counter that counts how many sharers hold the lock. Upon receiving an exclusive lock acquiring request, the lock manager looks up the corresponding lock status bit and executes eBPF atomics to check if it can acquire the lock. It runs the `__sync_val_compare_and_swap()` function inside eBPF to atomically test-and-set the lock status bit. This function gets compiled into corresponding ISA-specific operations and is equally efficient as in the user space. If the lock test-and-set succeeds, which means the lock has not been acquired by other transaction clients, the load manager will return a success response; otherwise, a failure response is returned to let the client retry. Handling the exclusive lock releasing and shared lock operations involves similar atomic operations.

Version-based locking: Version-based locking is widely used in recent high-performance distributed transactions sys-

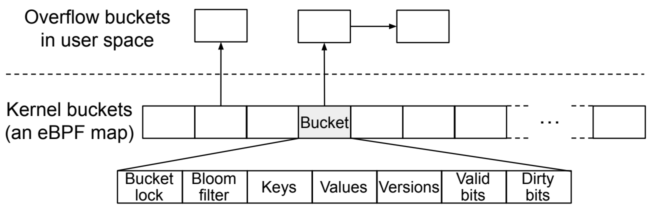


Figure 3: The layout of the key-value store in DINT (assume using the version-based locking).

tems [14, 29, 80], together with the OCC protocol to avoid locking operations for key-value reads. It involves version checking to make sure the read key-values used in a transaction are not stale (see §2.1).

To implement version-based locking, DINT maintains a table for the lock status bit indexed by the lock ID, and maintains a per-key-value version counter in a key-value store that we discuss in the next Section. Every read operation directly reads the key-value and corresponding version from the key-value store. Every write operation tries to test-and-set the lock status bit (i.e., exclusive lock); if test-and-set fails, the transaction aborts. After acquiring all write locks and then finishing all transaction writes locally, the transaction coordinator reads the key-value versions again and compares them with the old versions. If the two version vectors do not change, the coordinator can successfully log and commit the transaction, and increment the versions; otherwise, the transaction aborts.

3.2 DINT Key-Value Store

The DINT key-value store maintains the mapping between keys and values, and supports various operations like GET, INSERT, UPDATE, and DELETE. Conventional user-space key-value store [53, 70] would normally maintain a hash index that maps keys to dynamically-allocated values. Unfortunately, this design does not work for eBPF that lacks dynamic memory allocations.

Figure 3 illustrates how DINT addresses this challenge by storing key-values into an in-kernel set-associative cache backed by a fixed-size eBPF map, while spilling overflowed key-values (includes corresponding versions) into the user space. The eBPF map contains many kernel buckets indexed by the key via a hash function, and each bucket stores multiple key-values and valid bits (denoting whether a key-value field stores object data)[2]. Inside each kernel bucket, DINT stores keys contiguously for better cache locality during lookups; DINT provisions each value field with a fixed size that can cover most of the transaction objects (e.g., dozens of bytes in TPC-C and SmallBank workloads [80, Table 3]). Any kernel bucket that gets too many key-values will spill some key-values into the user space (putting into the overflow buckets); any key-value that cannot fit into the fixed value field in the kernel bucket will also spill into the user space.

A kernel bucket contains a bucket-level lock implemented using eBPF atomics to synchronize concurrent key-value operations on the same bucket. We note that this lock is different from the transaction locks in Section 3.1. Each key-value operation will first try acquiring the bucket lock before touching the bucket data, in a fail-and-retry manner. Most of the time, the key-value operation finishes directly in eBPF and returns the response to clients, before which it releases the bucket lock. In rare cases where its interested key-value is in the user space, the operation needs to pass the operation request/packet to the user-space process via the UDP sockets. Under such cases, the operation still holds the bucket lock when going to the user space, and only releases the lock when it returns to eBPF. By "returns to eBPF", we mean that the response packet sent back by the user-space process will trigger an eBPF program attached to the TC egress hook, which releases the bucket lock.

However, to support high-performance key-value operations in this kernel-user-hybrid key-value store, we must address two additional challenges:

- How to efficiently perform INSERT and UPDATE operations while maintaining *read-all-write* consistency? Prior eBPF-offloaded key-value store BMC [19] adopts a simple write-through cache design and performs well when all operations are GETs. However, in workloads like TATP [46] where 20% of transactions involve INSERTs/UPDATEs, BMC would perform poorly because every such operation will go to the user space.
- How to minimize the chance of going to the user space, especially when clients issue many GET requests for non-existing keys? Non-existing key lookups would require enumerating all keys mapped to the kernel bucket including those spilled into the user space, incurring high kernel stack overheads. Such lookups are common in transaction workloads; e.g., 68.75% of GETs for TATP's largest table target non-existing keys.

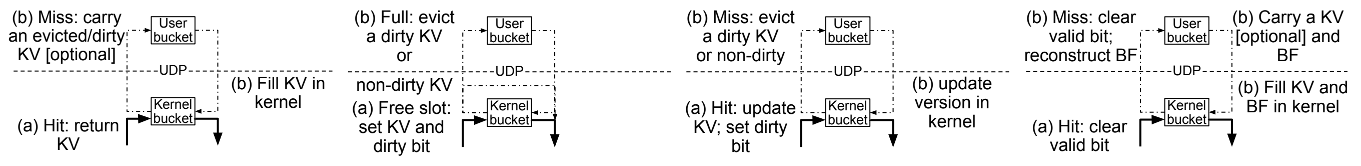
To this end, DINT designs a write-back mechanism that lazily and efficiently maintains the read-after-write consistency, and leverages a per-kernel-bucket Bloom filter to avoid frequently going to the user space for non-existing key lookups.

3.2.1 Write-Back Key-Value Store in eBPF

As shown in Figure 3, a kernel bucket contains a dirty bit for each stored key-value, indicating whether the value is different from the user space; a key-value that only exists in eBPF will always have the dirty bit set. Below, we go through how DINT efficiently realizes each of the key-value operations across eBPF and the user space. A recurring theme in the design of each operation is that: DINT tries to support the majority of key-value operations directly in eBPF by leveraging the dirty bit, while maintaining consistency.

GET (Figure 4a): For simplicity, we assume the looked-up key exists in the key-value store; we describe the non-existing case in the next Section. In the frequent path (a) where the GET operation finds the key in the kernel bucket, DINT di-

[2]Maintaining the valid bit for each key-value should be straightforward; for conciseness, we do not explicitly describe it unless necessary.


(b) Miss: carry an evicted/dirty KV [optional]
User bucket
UDP
(b) Fill KV in kernel
Kernel bucket
(a) Hit: return KV
(b) Full: evict a dirty KV or non-dirty KV
User bucket
UDP
Kernel bucket
(a) Free slot: set KV and dirty bit
(b) Miss: evict a dirty KV or non-dirty
User bucket
UDP
(b) update version in kernel
Kernel bucket
(a) Hit: update KV; set dirty bit
(b) Miss: clear valid bit; reconstruct BF
User bucket
(b) Carry a KV [optional] and BF
UDP
(b) Fill KV and BF in kernel
Kernel bucket
(a) Hit: clear valid bit

(a) GET. **(b)** INSERT. **(c)** UPDATE. **(d)** DELETE.

Figure 4: DINT key-value store operations. Solid thick lines indicate frequent paths, while dotted thin lines mean rare paths. BF = Bloom Filter.

rectly returns the requested value to the client. In the rare path (b) where the GET operation does not find the key: if the kernel bucket is full, DINT chooses one existing key-value to evict following a certain policy (described later) to make a space for the looked-up key-value; otherwise, DINT chooses one dirty key-value (if any) for lazily writing back to the user space. DINT then optionally piggybacks the chosen key-value on the packet and forwards it to the user-space process; DINT uses the `bpf_xdp_adjust_tail()` function [47] to increase the packet size for piggybacking. Once the process receives the request packet, it will look up the key-value in the overflow buckets, and send back the response packet to the client via the UDP sockets. For the piggybacked key-value, the user-space process will update it into the overflow buckets. Finally, the response packet goes through the TC egress eBPF program, which clears the dirty bit of the piggybacked key-value (if any), and fills the requested key-value into an empty or non-dirty key-value field in the kernel bucket.

INSERT (Figure 4b): For simplicity, we assume the to-be-inserted or the incoming key-value does not exist in the key-value store [3]. In the frequent path (a) where the INSERT operation finds an empty slot in the kernel bucket, DINT directly writes the incoming key-value there, sets an initial version and the dirty bit, and returns to the client. In the rare path (b) where there is no empty slot in the kernel bucket, DINT chooses a key-value to evict. Then there will be two cases:

- If the to-be-evicted key-value is not dirty, DINT will directly replace it by the incoming key-value with a *set dirty bit* and an initial version, and return to the client. DINT can directly return as the to-be-evicted key-value has the same copy in the user space, so there is no need to write it back. Since the incoming key-value is marked dirty, a later eviction will lazily write it back to the user space.
- If the to-be-evicted key-value is dirty, DINT will piggyback it on the request packet, replace the bucket's to-be-evicted key-value by the incoming key-value with a *clear dirty bit* and an initial version, then pass the request packet to the user space. The user-space process will then update both the evicted key-value and the incoming key-value into the overflow buckets, and send back a response packet to the client via the UDP sockets.

UPDATE (Figure 4c): For simplicity, we assume the to-be-updated or the incoming key-value exists in the key-value store. In the frequent path (a) where the key-value is found in the kernel bucket, DINT directly updates the key-value there with a set dirty bit, increments the version counter, and returns to the client. In the rare path (b) where the key-value is not found, DINT chooses a key-value to evict. No matter whether the to-be-evicted key-value is dirty or not, DINT always needs to go to the user space, in order to fetch (and increment) the version counter corresponding to the incoming key-value. Therefore, DINT will piggyback the to-be-evicted key-value on the request packet, replace the bucket's to-be-evicted key-value by the incoming key-value with a clear dirty bit and an *undefined version*, then pass the request packet to the user space. The user-space process will then update both the evicted key-value and the incoming key-value into the overflow buckets, increment the version counter of the incoming key-value, and send back a response packet to the client via the UDP sockets. More importantly, this response packet will piggyback the updated version of the incoming key-value, so that the TC egress eBPF program can update the bucket's undefined version to the updated one.

Eviction policy: DINT currently uses a simple eviction policy: it tries to evict the first non-dirty key-value when enumerating the bucket; if all the key-values are dirty, it then evicts a random key-value. Prioritizing evicting non-dirty key-value avoids going to the user space as much as possible (especially under INSERT operations). Randomly choosing a key-value if all is dirty minimizes the compute for selecting a victim key-value. Implementing a more complex eviction policy, e.g., based on key-value accessing frequency, might help further reduce the chance of going to the space. But such policy should be compute-light; otherwise, it may incur performance drops [19]. Recent fast cache eviction algorithms such as QD-LP-FIFO [82] may shed light on this space, and we leave such exploration as future work.

Remark: So far, DINT carefully leverages the dirty bits to run the majority of key-value operations directly in eBPF, while maintaining read-all-write consistency and correct key-value versions. For the DELETE operation, we deliberately leave it for the next Section to describe, as it is highly related to how we efficiently handle non-existing key lookups.

3.2.2 Handling GETs for Non-Existing Keys

So far, in the DINT key-value store, GET requests for non-existing keys would enumerate all keys mapped to the indexed kernel bucket including those spilled into the user space, incurring high kernel stack overheads. Conventional key-value stores implemented in the user space do not have such a prob-

[3] If the to-be-inserted key-value already exists, this INSERT operation is functionally equivalent to an UPDATE operation. Similar equivalence also applies to the UPDATE operation in a reverse way.

lem because the enumeration overhead for them is only a few more memory accesses; however, for the key-value store in eBPF, the overhead escalates into expensive kernel networking stack traversing and user-kernel context switching [19,87].

To handle non-existing key GETs efficiently, DINT maintains a small Bloom filter [5] in each kernel bucket, representing the membership of key-values spilled into the user space (Figure 3). The Bloom filter is updated whenever a key-value gets spilled into the user space. When a GET operation does not find the looked-up key in its kernel bucket, it looks up the Bloom filter to check if the key possibly exists in the user space. If the Bloom filter answers no, then the GET operation can guarantee that the key does not exist in the key-value store, and directly return none to the client; otherwise, the operation must go to the user space to check the overflow buckets (see §3.2.1). Since the Bloom filter never reports an existing key as non-existing (i.e., no false negative errors), the above "early returning" in the GET operation is always correct. DINT currently provisions 64 bits for the Bloom filter in each kernel bucket, sufficient to handle dozens of spilled key-values. To reduce the hash calculation overhead for the Bloom filter, DINT reuses the highest six bits of the raw hash value from the key-value store.

However, the Bloom filter design creates a challenge for the key-value DELETE operation. This is because: when the to-be-deleted key-value is in the user space, the DELETE operation will need to remove the key-value from the Bloom filter; however, the Bloom filter does not support membership removal in order to guarantee no false negative errors. To address this challenge, DINT lets the user-space process reconstruct a new Bloom filter for the remaining key-values whenever it deletes one, and then updates the new Bloom filter to the kernel. Reconstructing the Bloom filter is doable, as the user space records all the spilled key-values in its overflow buckets. Formally, the DELETE operation works as follows.
DELETE (Figure 4d): For simplicity, we assume the to-be-deleted key-value exists in the key-value store. In the frequent path (a) where the INSERT operation finds the key-value in the kernel bucket, it clears the valid bit and directly returns to the client. In the rare path (b) where the key-value is not found in the kernel bucket and the Bloom filter reports its existence in the user space, the DELETE operation must forward the request packet to the user space. The user-space process will look up the key-value in the overflow buckets, clear its valid bit, reconstruct a new Bloom filter based on the remaining spilled key-values, and send back a response packet to the client. The response packet will piggyback the new Bloom filter and an optional spilled key-value (if existing, and this key-value should not be covered in the new Bloom filter), and trigger the TC egress eBPF program, which fills the Bloom filter and key-value into the kernel bucket.

3.3 DINT Log Manager

High-performance distributed transaction systems store transaction logs in memory for failure recovery (assuming battery-backed DRAM or fast NVRAM [14,29]). The transactions logs grow up as the transaction systems run: if they exceed the log space (e.g., memory capacity of the machine), the transaction systems usually truncate the oldest logs [14] or dump them into disks [76]; DINT follows the truncating manner. Since the logging operation is on the transaction critical path, DINT aims to provide a fast logging mechanism entirely inside the eBPF in failure-free cases, while supporting complex offline recovery in failure cases.

To this end, DINT leverages the eBPF maps to implement a circular log buffer abstraction entirely in the kernel. A circular log buffer allows pushing log entries to the tail to support logging operations in transaction systems; it also allows popping log entries from the head (from the user space) to support log replaying during failure recovery. DINT implements such a circular log buffer using a large-size eBPF array map to store log entries, and another eBPF array map to maintain the head and tail, both inside the kernel. These two eBPF maps are also accessible to the user space for log replaying. To avoid thread contentions during logging operations, DINT provisions a circular log buffer on each CPU core. This is achieved by using the eBPF per-CPU array map (`BPF_MAP_TYPE_PERCPU_ARRAY` [32]). When the log manager looks up a per-CPU array map, it will automatically get the map entry corresponding to its local CPU core.

4 DINT Implementation

Our DINT prototype consists of 2.1K lines of eBPF (for kernel code) and 4.3K lines of C++ (for user-space code). DINT uses Clang/LLVM-16 to compile the eBPF program into eBPF bytecode. The eBPF bytecode gets attached to and runs inside the XDP and TC hooks of the standard kernel networking stack, atop unmodified Linux OSes. The user-space process uses the standard POSIX kernel-visible threads (i.e., pthreads) and the Linux UDP socket to receive rare-path request packets and send response packets. Our prototype currently supports two different transaction protocols, i.e., a 2PL-based protocol and an OCC-based protocol, demonstrating the genericity of DINT's designs to some extent. Our DINT prototype currently does not implement failure recovery to handle machine failures; as described in Section 2.1, we assume a separate configuration manager would handle them off the critical path, thus not impacting the critical-path performance we focus on.

To reduce the performance impact of user-kernel context switching when passing request packets to the user space, DINT runs the user-space process (that handles rare-path requests) on CPU cores that do not receive NIC interrupts or run eBPF programs, similar to prior work [87]. This is achieved by configuring the IRQ affinity of the NIC device to exclude the rare-path handling core. Note that the rare-path handling core does not do any busy polling and can be shared with other applications.

To better reason about the performance of DINT, we build two baseline transaction processing systems that run in the user space. One baseline uses the standard kernel UDP socket with `SO_REUSEPORT` enabled to reduce thread contentions [19], and pthreads. Another baseline uses the UDP stack from the kernel-bypass runtime Caladan [17] that supports DPDK-style packet busy-polling and user-space threading for fast context switching. Both baselines leverage DINT's performance optimizations (e.g., lock sharing) if helpful, but without eBPF programming constraints—so that they can handle hash collisions efficiently using state-of-the-art solutions [43]. The two baselines consist of 6.1K lines of C++.

5 Evaluation

This section aims to answer the following questions:

1. What is the throughput and latency of DINT compared to kernel-bypass approaches (§5.1 and §5.2)?
2. Can DINT support different transaction protocols on transaction workloads efficiently (§5.1 and §5.2)?
3. Can DINT provide load-aware CPU scaling (§5.3)?
4. What are the effects of the write-back mechanism, Bloom filter, and rare paths on DINT's performance (§5.4)?

Testbed: We use 13 `r650` physical machines from CloudLab [15]. Each machine has two 36-core (72 logic-core) Intel Xeon Platinum 8360Y CPUs at 2.4GHz, 256GB memory, and a dual-port Mellanox ConnectX-6 100Gb NIC via PCIe 4.0×16. All machines are connected via a Dell Z9432F switch under the same rack. For all experiments, we use a single CPU in the same NUMA domain as the NIC to enforce NUMA locality; we also use a single 100Gb NIC port, as CloudLab currently only wires one such port of `r650` to the switch.

For all experiments, each machine runs an unmodified Ubuntu 20.04 OS. For eBPF and UDP-related experiments, we use kernel v6.1.0 which has full support for eBPF atomics. We use the built-in Mellanox NIC driver on Linux kernel v6.1.0 that has a default NAPI poll budget/batch size of 64 upon each interrupt. We disable Mellanox NIC's interrupt coalescing feature [11], as we find it hurts latency while not increasing throughput, similar to prior work [87]. For Caladan-related experiments, we are not able to run the Caladan runtime on kernel v6.1.0, as it requires a customized kernel module that relies on specific kernels; instead, we manage to run it on kernel v5.8.0. Since Caladan uses the kernel-bypass networking stack and threading, different OS kernels should not have a significant impact on its performance.

Measurement methodology: For transaction benchmarking, we use 3 machines to run transaction servers with three-way replication and sharding; that is, each machine is the primary for one shard and a replica for the other two. For microbenchmarks that benchmark individual lock manager, key-value store, and log manager, we use 1 machine to run the microbenchmark server without replication or sharding to understand their standalone performance. We use the rest machines to run multiple transaction/microbenchmark clients that issue requests in a closed-loop manner. To avoid the client machines becoming the bottleneck, we provision 8 cores on each transaction/microbenchmark server; the client machines further use Caladan's kernel-bypass UDP stack and user-space threading to generate requests. We then vary the number of clients, and measure the achieved throughput and client-perceived median/average and 99th-tail latency, similar to prior transaction works [6, 41, 55, 85].

Comparison baselines: As mentioned in Section 4, we compare DINT to two baseline transaction processing systems: one is based on the Linux kernel UDP socket, another is based on the UDP stack from the kernel-bypass runtime Caladan [17]. For simplicity, we just use kernel UDP and Caladan to refer to these two baselines respectively. The Caladan baseline is a challenging baseline that features DPDK-style packet busy-polling, NIC RSS to evenly spread packets among available cores, and well-implemented and efficient user-space UDP stack and threading.

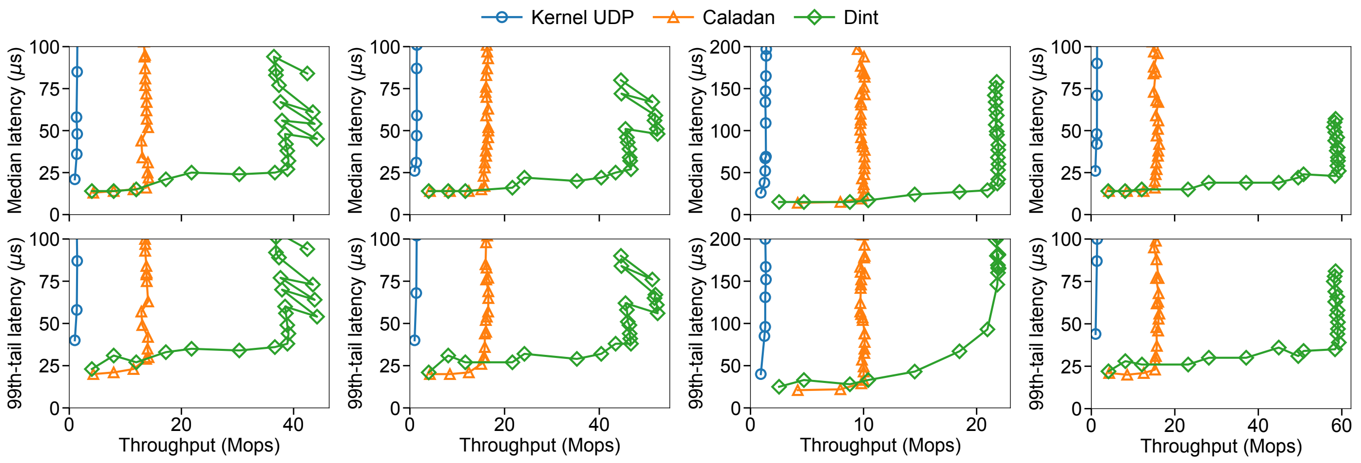
We provision the memory sizes of the user-space key-value store (for the two baselines), eBPF key-value store (for DINT), and lock table (for all three) to be 1.5× of the key-values/locks in corresponding workloads, similar to FaSST [29]. By default, kernel UDP and Caladan use all provisioned cores to handle requests, while Caladan uses one extra core to run its scheduler. DINT devotes one core out of the provisioned cores to handle rare-path requests (§4), while the rest cores handle frequent-path requests.

5.1 Microbenchmarks

To understand how each DINT component compares to baselines, we implement a series of microbenchmarks, including a 2PL-based and an OCC-based lock manager with skewed locking requests (80% shared locking requests or version reads), a key-value store with 40B skewed reads, and a log manager with 56B writes. These microbenchmark parameters (e.g., skewness, value size) are derived from the TATP workload [46]. The two lock managers and the key-value store are provisioned with 36 million lock/key slots, while their requests target 24 million locks/keys.

Lock manager: Figure 5a and 5b show how the latencies (both median and 99th-tail) of the 2PL and OCC lock manager vary with different achieved throughput for different systems, respectively. Each system performs similarly across the two lock managers with the OCC lock manager being slightly faster, as version reads in OCC do not run atomic operations. Overall, DINT achieves 3.1×-3.2× higher throughput than Caladan, with 0%-8%/5%-55% higher unloaded median/99th-tail latency, while kernel UDP performs much worse than others. We notice that DINT has throughput fluctuations at high loads; we think this is because the achieved batch size during interrupt handling gets changed unstably.

It might be supersizing that DINT achieves even higher throughput than the kernel-bypass Caladan system. However, this is achievable, as Caladan wraps raw UDP packets into a high-level connection-oriented abstraction (i.e., `rt::UdpConn`) for applications, which loses some perfor-



(a) 2PL lock manager. **(b)** OCC lock manager. **(c)** Key-value store. **(d)** Log manager.

Figure 5: Microbenchmark load-latency curves (both median and 99th-tail).

mance, while DINT directly works on low-level UDP/ethernet packets. In Caladan, each transaction server creates a `rt::UdpConn` for each transaction client and spawns a user-space thread to handle corresponding transaction requests. Although `rt::UdpConn` only maintains simple connection states and user-space threading is efficient (e.g., 50ns per context switching [60]), they still consume extra CPU time, compared to DINT that directly modifies incoming ethernet packets and forwards back.

In terms of latency, kernel-bypass Caladan achieves lower minimum latency than kernel-stack DINT, e.g., 13μs vs. 14μs of the median and 20μs vs. 23μs of the 99th-tail for the 2PL lock server. The latency gap, especially for the 99th-tail, is mainly caused by the interrupt-driven nature of DINT, which includes the overheads of NIC interrupt delivery and running interrupt handler. We note that such overheads can be effectively amortized under high loads, thus not impacting throughput. We think the small increased latency is acceptable, as the current data center network usually has one or a few tens of microseconds RTT [20, 50].

Key-value store: Figure 5c shows the load-latency curves for the key-value store. Both Caladan and DINT's performance gets dropped compared to the lock managers, due to more compute in key-value operations. DINT achieves 2.17× higher throughput than Caladan, while having 0%-7%/27%-57% higher unloaded median/99th-tail latency. The minimum latency for Caladan and DINT is 14μs vs. 15μs for the median, and 21μs vs. 25μs for the 99th-tail, demonstrating DINT only incurs small interrupt handling overheads.

Log manager: Figure 5d shows the load-latency curves for the log manager. Similarly, DINT outperforms Caladan on throughput (by 3.6×) but sacrifices latency (by 0%-7% for unloaded median and 5%-40% for unloaded 99th-tail). Regarding the absolute performance number, DINT achieves up to 7.4 Mops per core. This translates into as low as 0.14μs per operation/packet, demonstrating the efficiency of offloading frequent-path operations into the kernel. Both DINT and Caladan achieve higher throughput on the log manager than the lock managers, as the serial log appending operations have better cache locality.

5.2 Transaction Benchmarks

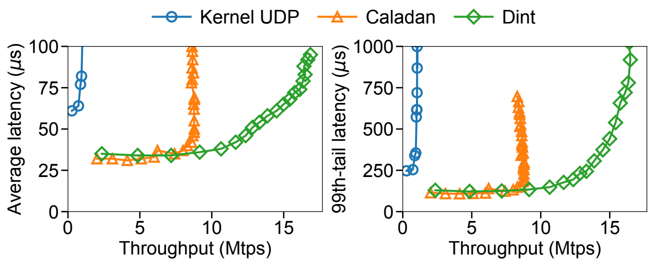
We now evaluate DINT and other baselines on typical OLTP workloads, including TATP [46] and SmallBank [75]. TATP is a read-intensive OLTP benchmark modeling database behaviors of telecommunication providers. It features small key-values (8B keys and 40B values), 80% read-only transactions that read one or more keys, and 20% transactions that modify key-values. We provision 7 million TATP subscribers sharded across the three transaction servers. Similar to prior works [14, 29], we use the OCC-based transaction protocol (see §2.1) for the read-intensive TATP workload.

SmallBank is a write-intensive OLTP benchmark modeling bank account transactions, with 8B keys and values, and 85% write transactions. We provision 24 million bank accounts sharded across the three transaction servers. We use a 2PL-based transaction protocol suitable for write-intensive workloads. Compared to OCC, the 2PL-based protocol uses read-write locks in the read+lock phase without the validate phase; it has similar log and commit phases (§2.1).

DINT can easily support both transaction protocols by leveraging different lock managers and slightly changing client behaviors, demonstrating the genericity of its designs.

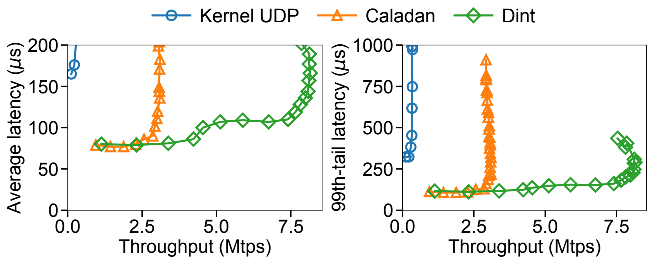
TATP: Figure 6a and 6b show how the average[4] and 99th-tail transaction latencies of different systems change when varying the throughput, respectively. DINT achieves 1.9× higher transaction throughput than Caladan with 7%/12%-16% higher unloaded average/99th-tail latency. As described in Section 5.1, the higher throughput of DINT benefits from directly manipulating and forwarding raw ethernet/UDP packets immediately after the NIC driver receives the packets, in contrast to Caladan that works on a high-level connection-

[4]We show the average rather than the median, as transaction workloads contain many small transactions that dominate the median latency.



(a) Average latency vs. tput. **(b)** 99th-tail latency vs. tput.

Figure 6: OCC on TATP workload. Mtps = Million transactions per second.



(a) Average latency vs. tput. **(b)** 99th-tail latency vs. tput.

Figure 7: 2PL on SmallBank workload.

oriented abstraction. Meanwhile, batching effectively amortizes interrupt handling overheads in DINT, leading to a high sustained load on transaction servers. On the other hand, such batching inevitably causes higher latency for DINT when compared to the kernel-bypass polling-based Caladan, i.e., 1μs/12μs higher minimum average/99th-tail latency.

Although not an apple-to-apple comparison, we cite published performance numbers of RDMA-based transaction systems to demonstrate the throughput achieved by DINT is within the same order of magnitude as RDMA-based ones. For example, FaSST reports 8.7 Mtps/machine with 14 cores and 1 million TATP subscribers per machine [29, §6.2], while DINT achieves 5.62 Mtps/machine with 8 cores and 2.3 million subscribers per machine.

SmallBank: Figure 7a and 7b show the average and 99th-tail transaction latencies of different systems when varying transaction throughput under the SmallBank workload. DINT achieves 2.6× higher throughput than Caladan, while only adding 2%-4%/3%-9% unloaded average/99th-tail latency; the added minimum average/99th-tail latency is 2μs/5μs. Each SmallBank transaction consists of ∼10 transaction requests on average, including locking and key-value operations; therefore, DINT could sustain ∼82 million/sec request rate on 24 cores across three machines. Therefore, DINT's per-core request rate, i.e., ∼3.4 mops, is also within the same order of magnitude as RDMA two-sided operations, i.e., 3.6 mops reported by [79, Figure 3] on a ConnectX-6 NIC.

5.3 CPU Utilization

We now examine whether DINT can scale CPU usage as load changes, avoiding burning CPU cores. We use the same TATP workload as in Section 5.2, but provision enough number of clients, specify different transaction rates (by adjusting the sleeping time interval between two consecutive transaction requests in each client), and measure the CPU core usage of transaction servers. For kernel UDP and DINT, they rely on NIC interrupt to wake up any sleeping kernel-visible thread (i.e., pthread) when packets arrive. For Caladan, it supports a CPU-efficient non-spinning mode where the dedicated scheduler busy polls the NIC, and wakes up sleeping user-space threads when needed via IPIs (Inter-Process Interrupt); the Caladan scheduler also reallocates CPU cores every 5μs for the application process based on various load signals (e.g., packet and thread queueing delay [17]), to provision just-enough CPU cores for the current load.

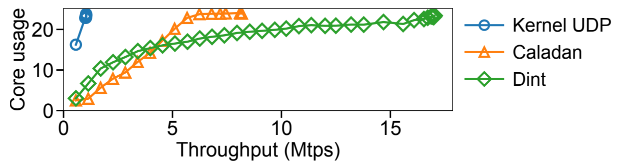


Figure 8: Core usage vs. throughput (on TATP).

KV workload [Throughput (Mops)]	Write-through (BMC [19])	Write-back	Write-back+BF (DINT)
All GETs, all exists	21.6	21.7	21.7
80% GETs, all exists	1.0	21.1	20.9
80% GETs, 31% exists	0.4	0.5	25.0

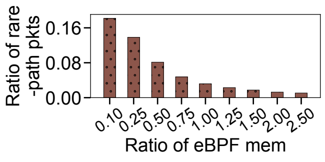
Table 1: Impact of write-back and Bloom filter. "80% GETs" and "31% exists" are based on the TATP workload and its largest table.

Figure 8 shows how the CPU core usage varies with different throughput for different systems. Until 5 Mtps load, Caladan achieves the lowest core usage and can additively allocate more cores as the throughput increases, due to its fast core reallocation. After 5 Mtps load, DINT achieves lower CPU usage than Caladan and can additively scale its CPU usage to 17 Mtps, because of packet batching during interrupt handling. Kernel UDP has the worst CPU scaling curve, caused by the high overheads of frequent kernel networking stack traversing and user-kernel context switching. Nevertheless, to enable more efficient CPU scaling for DINT under low loads, one way could be consolidating multiple NIC interrupts onto fewer cores to leverage batching to reduce per-packet processing overheads. We discuss more in Section 6.

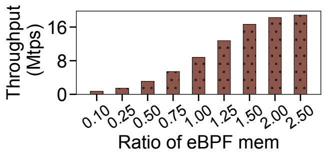
5.4 Design Drill-Down

5.4.1 Impact of Write-Back and Bloom Filter

Table 1 shows how the write-back and Bloom filter designs impact DINT performance on different key-value store workloads. With all GETs and all keys existing, the write-through, write-back, and write-back + Bloom filter achieve similar throughput. Once with 20% PUTs, the write-through throughput drops to 1.0 Mops because of handling PUTs in the user space, while the other two keep similarly high throughput. Furthermore, adding 68.75% key-value operations for non-existing keys, only the write-back + Bloom filter can achieve

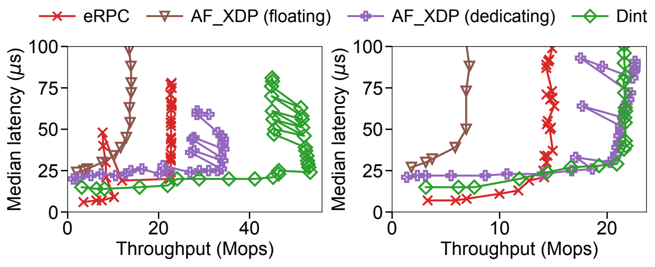


(a) Rare-path ratio.



(b) Throughput.

Figure 9: Impact of varying the eBPF memory size under the TATP workload. "Ratio of eBPF memory" is against the workload dataset size including both locks and key-values.



(a) OCC lock manager. **(b)** Key-value store.

Figure 10: Comparing eRPC and AF_XDP with DINT.

high throughput, as it handles most key-value operations in the kernel, for both existing and non-existing keys.

5.4.2 Impact of Rare-Path Ratio

Figure 9a and 9b shows how different rare-path ratios (by changing the eBPF memory size) impact DINT's transaction throughput. The rare-path ratio significantly impacts DINT performance. For example, with 10% of the workload dataset size in the eBPF memory, which gives 18% of rare-path packet ratio, DINT only achieves 740 Kops. Once we provision the eBPF memory to be 1.5× of the workload dataset size, similar to how FaSST [29] provisions its hash table, there will be only 1.7% of rare-path packet ratio, and DINT reaches 16.7 Mops. This supports the DINT's design principle of offloading frequent-path operations as much as possible into the kernel.

5.5 Comparison to More Baselines

We now compare the performance of DINT with more baselines that leverage other networking stacks. In particular, we compare to eRPC [28] and AF_XDP socket [31]. eRPC is a kernel-bypass event-driven RPC library that builds on top of raw ethernet packets with its own efficient reliable transport protocol. It supports both DPDK and RDMA in busy-polling manners; our testing uses DPDK. AF_XDP is a new kernel socket family that leverages eBPF/XDP to directly DMA packet payload to a pre-registered user-space memory region, so that user-space applications can efficiently receive and send packets in a zero-copy manner. AF_XDP appears to applications as a set of socket APIs, so the application's packet processing logic can be written in a normal programming language (e.g., C/C++, Go) without the strict kernel verification as in eBPF. We run AF_XDP with two modes: floating where all provisioned cores handle NIC interrupts and run transaction servers, and dedicating where half of the cores handle NIC interrupts and another half run transaction servers.

Figure 10a and 10b shows the load-latency curves of eRPC, AF_XDP, and DINT for the OCC lock manager and key-value store respectively. For both applications, eRPC achieves the lowest minimum latency—8μs lower than DINT on both applications. For the lock manager, DINT achieves the highest throughput, and outperforms AF_XDP by 1.6× and eRPC by 2.3×. eRPC suffers from latency spikes at low loads because of insufficient RPC batching. For the key-value store that has more compute per operation, DINT has similar throughput as AF_XDP while achieving 29% lower minimum latency, because of directly handling requests in the kernel without going into the user space; DINT achieves 1.4× higher throughput than eRPC. The throughput results for eRPC must be taken with a grain of salt: eRPC builds a generic loss-tolerant RPC abstraction with session management, while DINT relies on transaction semantics to handle packet losses and works on raw ethernet/UDP packets.

One interesting observation is that AF_XDP in the floating model performs much worse than the dedicating mode; similar results occur for DINT on the CPU placement of rare-path request handling process as described in Section 4. This is caused by the high user-kernel context switching overheads when co-locating interrupt handling and the application process on the same cores. We discuss further in Section 6.

6 Discussion and Future Work

Symmetric vs. asymmetric models: DINT adopts an asymmetric client-side transaction model [41, 55, 59, 85], where each transaction server "passively" handles incoming transaction requests. DINT then leverages eBPF/XDP to offload transaction server operations into the kernel. In contrast, a symmetric model [14, 29, 80] requires the transaction server to also act as a client to issue transaction requests. This creates challenges to DINT, as eBPF/XDP itself cannot generate new packets. Fortunately, by leveraging the AF_XDP technique (see §5.5) that provides fast packet sending functionality, DINT could support symmetric models efficiently. We leave the integration of DINT with AF_XDP as future work.

Implications to networking stack research: DINT shows that the kernel networking stack can achieve kernel-bypass-like throughput and latency, but has worse CPU efficiency under low loads than well-engineered kernel-bypass stacks (§5.3). Therefore, we call for more research on optimizing the CPU efficiency of the kernel networking stack that offloads application operations. One idea may be smartly consolidating NIC interrupts to just-enough CPU cores by manipulating the NIC IRQ affinity, which leverages batching during interrupt handling to reduce per-packet processing overheads. This shares the same goal as Shenango [60] and Caladan [17], but targets the interrupt-driven kernel networking stack.

Another research problem is how to isolate the kernel stack-offloaded operations and user-space operations, as naively co-locating both on the same cores would cause severe performance drop due to frequent user-kernel context switching (see §4 and §5.5). DINT currently uses a simple static partitioning policy, but a more advanced dynamic partitioning policy could possibly provide better performance.

Implications to transaction protocol research: Co-designing transaction protocols with eBPF allows for both high performance and good CPU efficiency. In this work, we co-design an OCC/2PL-based transaction protocol in DINT. DINT should also be able to support more advanced transaction protocols like MDCC [39] and Tapir [85] that essentially rely on read-write and version-based locking. To support advanced protocols like ROCOCO [56] and Janus [57] that maintain transaction dependency DAGs in the lock manager, DINT would need to maintain complex graph data structures in eBPF, which calls for more co-designs to address the challenge of eBPF programming constraints. In an attempt to reduce CPU utilization, many transaction systems have pushed to incorporate network offload devices like RDMA and smartNICs [14, 72]. However, these devices are much more expensive than commodity NICs, and come with customized network stacks that have high maintenance overheads in terms of engineering. DINT provides the opportunity for accomplishing similar goals without the need for expensive customized hardware, and provides a new point in the design space for transaction protocol developers to explore.

Wish list for eBPF: The most helpful feature would be supporting dynamic memory allocations so that offloaded states could be more memory-efficient. Another helpful feature would be the egress XDP hook. When developing DINT, we were thinking of using the AF_XDP socket to process rare-path request packets (instead of the slower UDP socket); however, AF_XDP faces troubles with the egress bookkeeping of in-kernel states (§3.2), as it relies on the ingress-only XDP while bypassing the egress TC hook. Currently, the only way for AF_XDP to work is by calling eBPF functions in the user space, but this suffers from high syscall overheads. If the kernel supports the egress XDP hook, DINT could instead leverage the faster AF_XDP socket to handle rare paths.

7 Related Work

Distributed in-memory transactions: By leveraging battery-backed DRAM or NVRAM, distributed transactions are no longer bottlenecked by disk IOs, but the networking IOs. This has spurred a series of research that leverages RDMA to implement distributed in-memory transactions, e.g., FaRM [14], FaSST [29], `DrTM` [81], `DrTM+R` [8], `DrTM+H` [80], and Prism [6]. Rather than using RDMA that bypasses kernels, DINT sticks to the most common commodity NICs with the kernel networking stack for better security, isolation, protection, maintainability, and debuggability, without losing performance.

High-performance networking stacks: The inefficiency of traditional kernel networking stack has motivated the designs of many kernel-bypass networking stacks, e.g., mTCP [24], eRPC [28], Snap [50], Demikernel [84] and more [17, 30, 36, 60, 62, 73]. These stacks generally require DPDK-style busy polling, and trades security, protection, maintainability, and more for high performance. Instead, DINT provides comparable high performance without busy polling for distributed transaction applications, while guaranteeing kernel-based security, protection, maintainability, etc.

Perhaps the most relevant work to DINT in this space is IX [3] which implements a protected kernel networking stack and achieves kernel-bypass performance. To achieve high networking performance, IX leverages adaptive batching to amortize user-kernel transition overheads, while DINT relies on the built-in batching of the existing kernel networking stack to amortize interrupt handling overheads. One advantage of DINT over IX is that DINT directly works for existing widely-deployed Linux kernels without any kernel modifications or customized kernel modules.

Hardware offloading for applications: Offloading network-intensive operations to specialized hardware such as FPGA [1, 22, 40, 44], SmartNICs [35, 42, 45, 63, 71, 79], and programmable switches [13, 25, 26, 83] significantly improves application performance. However, they are generally hard to deploy in today's cloud environments [28, 87], as these advanced hardware are not widely available in the public cloud. In contrast, DINT aims to be readily-deployable by leveraging the kernel-native eBPF techniques on widely-deployed modern Linux kernels.

eBPF applications: eBPF is mostly used for packet filtering [51], infrastructure monitoring [2, 65], and L4 load balancing [16] in industry. Recent research has proposed more applications including: accelerating key-value stores [19], sidecar proxies [67], Paxos [87], DBMS proxies [7], gathering congestion control signals [58], guiding request scheduling [27], offloading storage functions [86], and optimizing locks [61]. DINT is a new eBPF application targeting distributed transactions.

8 Conclusion

DINT is a distributed in-memory transaction system under the kernel networking stack, yet achieving kernel-bypass-like throughput and latency. DINT achieves this by offloading transaction data structures and operations into the kernel via eBPF techniques, significantly reducing kernel stack overheads. Compared to a transaction system implemented using Caladan, a well-engineered kernel-bypass networking stack, DINT even achieves 2.6× higher throughput and only adds 7%/16% unloaded average/99th-tail latency.

More importantly, DINT challenges the conventional belief that the kernel networking stack is not suitable for distributed in-memory transactions, or generally, μs-scale networked applications; DINT shows that, with proper application-kernel co-design enabled by eBPF, one important class of such applications under the kernel networking stack can achieve kernel-bypass-like performance. We will open source DINT.

References

[1] Mohammadreza Alimadadi, Hieu Mai, Shenghsun Cho, Michael Ferdman, Peter Milder, and Shuai Mu. Waverunner: An Elegant Approach to Hardware Acceleration of State Machine Replication. In *Proceedings of USENIX NSDI*, pages 357–374, 2023.

[2] The Cilium Authors. Cilium: eBPF-Based Networking, Observability, Security. https://cilium.io/.

[3] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of USENIX OSDI*, pages 49–65, 2014.

[4] Brian N Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility Safety and Performance in the SPIN Operating System. In *Proceedings of ACM SOSP*, pages 267–283, 1995.

[5] Burton H Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970.

[6] Matthew Burke, Sowmya Dharanipragada, Shannon Joyner, Adriana Szekeres, Jacob Nelson, Irene Zhang, and Dan RK Ports. PRISM: Rethinking the RDMA Interface for Distributed Systems. In *Proceedings of ACM SOSP*, pages 228–242, 2021.

[7] Matthew Butrovich, Karthik Ramanathan, John Rollinson, Wan Shen Lim, William Zhang, Justine Sherry, and Andrew Pavlo. Tigger: A Database Proxy That Bounces with User-Bypass. *Proceedings of the VLDB Endowment*, 16(11):3335–3348, 2023.

[8] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and General Distributed Transactions Using RDMA and HTM. In *Proceedings of ACM EuroSys*, pages 1–17, 2016.

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

[10] Intel Corporation. Intel Optane Persistent Memory. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/overview.html.

[11] NVIDIA Corporation. Understanding interrupt moderation. https://enterprise-support.nvidia.com/s/article/understanding-interrupt-moderation.

[12] The Transaction Processing Council. TPC-C: On-Line Transaction Processing Benchmark. https://www.tpc.org/tpcc/.

[13] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. Netpaxos: Consensus at Network Speed. In *Proceedings of ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)*, pages 1–7, 2015.

[14] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of ACM SOSP*, pages 54–70, 2015.

[15] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. The Design and Operation of CloudLab. In *Proceedings of USENIX ATC*, pages 1–14, 2019.

[16] Facebook. Katran: A High-Performance Layer 4 Load Balancer. https://github.com/facebookincubator/katran.

[17] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating Interference at Microsecond Timescales. In *Proceedings of USENIX OSDI*, pages 281–297, 2020.

[18] Jian Gao, Youyou Lu, Minhui Xie, Qing Wang, and Jiwu Shu. Citron: Distributed Range Lock Management with One-sided RDMA. In *Proceedings of USENIX FAST*, pages 297–314, 2023.

[19] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing. In *Proceedings of USENIX NSDI*, pages 487–501, 2021.

[20] Dan Gibson, Hema Hariharan, Eric Lance, Moray McLaren, Behnam Montazeri, Arjun Singh, Stephen Wang, Hassan MG Wassel, Zhehua Wu, Sunghwan Yoo, et al. Aquila: A unified, low-latency fabric for datacenter networks. In *Proceedings of USENIX NSDI*, pages 1249–1266, 2022.

[21] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *Proceedings of ACM CoNEXT*, pages 54–66, 2018.

[22] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a Box: Inexpensive Coordination in Hardware. In *Proceedings of USENIX NSDI*, pages 425–438, 2016.

[23] Brendan Jackman. Atomics for eBPF. https://lwn.net/Articles/840224/.

[24] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of USENIX NSDI*, pages 489–502, 2014.

[25] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-Free Sub-RTT Coordination. In *Proceedings of USENIX NSDI*, pages 35–49, 2018.

[26] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of ACM SOSP*, pages 121–136, 2017.

[27] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-Defined Scheduling Across the Stack. In *Proceedings of ACM SOSP*, pages 605–620, 2021.

[28] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be General and Fast. In *Proceedings of USENIX NSDI*, pages 1–16, 2019.

[29] Anuj Kalia, Michael Kaminsky, and David G Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *Proceedings of USENIX OSDI*, pages 185–201, 2016.

[30] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP Acceleration as an OS Service. In *Proceedings of EuroSys*, pages 1–16, 2019.

[31] The Linux kernel development community. AF_XDP. https://docs.kernel.org/networking/af_xdp.html.

[32] The Linux kernel development community. BPF_MAP_TYPE_ARRAY and BPF_MAP_TYPE_PERCPU_ARRAY. https://docs.kernel.org/bpf/map_array.html.

[33] The Linux kernel development community. NAPI. https://docs.kernel.org/networking/napi.html.

[34] The Linux kernel development community. struct sk_buff. https://docs.kernel.org/networking/skbuff.html.

[35] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. LineFS: Efficient SmartNIC Offload of a Distributed File System with Pipeline Parallelism. In *Proceedings of ACM SOSP*, pages 756–771, 2021.

[36] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making RPCs First-Class Datacenter Citizens. In *Proceedings of USENIX ATC*, pages 863–880, 2019.

[37] Xinhao Kong, Jingrong Chen, Wei Bai, Yechen Xu, Mahmoud Elhaddad, Shachar Raindel, Jitendra Padhye, Alvin R Lebeck, and Danyang Zhuo. Understanding RDMA Microarchitecture Resources for Performance Isolation. In *Proceedings of USENIX NSDI*, pages 31–48, 2023.

[38] Xinhao Kong, Yibo Zhu, Huaping Zhou, Zhuo Jiang, Jianxi Ye, Chuanxiong Guo, and Danyang Zhuo. Collie: Finding Performance Anomalies in RDMA Subsystems. In *Proceedings of USENIX NSDI*, pages 287–305, 2022.

[39] Tim Kraska, Gene Pang, Michael J Franklin, Samuel Madden, and Alan Fekete. MDCC: Multi-Data Center Consistency. In *Proceedings of EuroSys*, pages 113–126, 2013.

[40] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of ACM SOSP*, pages 137–152, 2017.

[41] Jialin Li, Ellis Michael, and Dan RK Ports. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *Proceedings of ACM SOSP*, pages 104–120, 2017.

[42] Junru Li, Youyou Lu, Qing Wang, Jiazhen Lin, Zhe Yang, and Jiwu Shu. AlNiCo: SmartNIC-accelerated Contention-aware Request Scheduling for Transaction Processing. In *Proceedings of USENIX ATC*, pages 951–966, 2022.

[43] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of USENIX NSDI*, pages 429–444, 2014.

[44] Jiaxin Lin, Adney Cardoza, Tarannum Khan, Yeonju Ro, Brent E Stephens, Hassan Wassel, and Aditya Akella. RingLeader: Efficiently Offloading Intra-Server Orchestration to NICs. In *Proceedings of USENIX NSDI*, pages 1293–1308, 2023.

[45] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading Distributed Applications onto SmartNICs Using iPipe. In *Proceedings of ACM SIGCOMM*, pages 318–333. 2019.

[46] IBM Software Group Information Management. Telecom Application Transaction Processing Benchmark. https://tatpbenchmark.sourceforge.net/.

[47] Linux Programmer's Manual. bpf-helpers(7). https://man7.org/linux/man-pages/man7/bpf-helpers.7.html.

[48] Linux Programmer's Manual. bpf(2). https://man7.org/linux/man-pages/man2/bpf.2.html.

[49] Linux Programmer's Manual. tc-bpf(8). https://man7.org/linux/man-pages/man8/tc-bpf.8.html.

[50] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, Steve Gribble, et al. Snap: A Microkernel Approach to Host Networking. In *Proceedings of ACM SOSP*, pages 399–413, 2019.

[51] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX winter*, volume 46, 1993.

[52] John McNamara. API/ABI Stability and LTS: Current state and Future. https://www.dpdk.org/wp-content/uploads/sites/35/2017/09/DPDK-Userspace2017-Day2-2-ABI-Stability-and-LTS-Current-state-and-Future.pdf.

[53] The memcached contributors. Memcached - a Distributed Memory Object Caching System. https://memcached.org/.

[54] Microsoft. eBPF implementation that runs on top of Windows. https://github.com/microsoft/ebpf-for-windows.

[55] Sumit Kumar Monga, Sanidhya Kashyap, and Changwoo Min. Birds of a feather flock together: Scaling RDMA RPCs with Flock. In *Proceedings of ACM SOSP*, pages 212–227, 2021.

[56] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. Extracting More Concurrency from Distributed Transactions. In *Proceedings of USENIX OSDI*, pages 479–494, 2014.

[57] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. Consolidating Concurrency Control and Consensus for Commits under Conflicts. In *Proceedings of USENIX OSDI*, pages 517–532, 2016.

[58] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. Restructuring Endpoint Congestion Control. In *Proceedings of ACM SIGCOMM*, pages 30–43, 2018.

[59] The University of Texas at Austin. Natacha Crooks. A client-centric approach to transactional datastores. https://repositories.lib.utexas.edu/handle/2152/81352.

[60] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-Sensitive Datacenter Workloads. In *Proceedings of USENIX NSDI*, pages 361–378, 2019.

[61] Sujin Park, Diyu Zhou, Yuchen Qian, Irina Calciu, Taesoo Kim, and Sanidhya Kashyap. Application-Informed Kernel Synchronization Primitives. In *Proceedings of USENIX OSDI*, pages 667–682, 2022.

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

[63] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: A Programming System for NIC-Accelerated Network Applications. In *Proceedings of USENIX OSDI*, pages 663–679, 2018.

[64] Dan RK Ports, Jialin Li, Vincent Liu, Naveen Kr Sharma, and Arvind Krishnamurthy. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *Proceedings of USENIX NSDI*, pages 43–57, 2015.

[65] The IO Visor Project. BPF Compiler Collection (BCC). https://github.com/iovisor/bcc.

[66] The IO Visor Project. eXpress Data Path (XDP). https://www.iovisor.org/technology/xdp.

[67] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and KK Ramakrishnan. SPRIGHT: Extracting the Server From Serverless Computing! High-Performance eBPF-Based Event-Driven, Shared-Memory Processing. In *Proceedings of ACM SIGCOMM*, pages 780–794, 2022.

[68] Benjamin Rothenberger, Konstantin Taranov, Adrian Perrig, and Torsten Hoefler. ReDMArk: Bypassing RDMA Security Mechanisms. In *Proceedings of USENIX Security*, pages 4277–4292, 2021.

[69] Hugo Sadok, Zhipeng Zhao, Valerie Choung, Nirav Atre, Daniel S Berger, James C Hoe, Aurojit Panda, and Justine Sherry. We Need Kernel Interposition over the Network Dataplane. In *Proceedings of ACM HotOS*, pages 152–158, 2021.

[70] Salvatore Sanfilippo. Redis: An In-Memory Database That Persists on Disk. https://github.com/redis/redis.

[71] Henry N Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. Xenic: SmartNIC-Accelerated Distributed Transactions. In *Proceedings of ACM SOSP*, pages 740–755, 2021.

[72] Henry N. Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. Xenic: Smartnic-accelerated distributed transactions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 740–755, New York, NY, USA, 2021. Association for Computing Machinery.

[73] ScyllaDB. SeaStar High Performance Server-Side Application Framework. https://github.com/scylladb/seastar.

[74] Alexei Starovoitov. BPF at Facebook. https://kernel-recipes.org/en/2019/talks/bpf-at-facebook/.

[75] The H-Store Team. SmallBank Benchmark. https://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank/.

[76] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy Transactions in Multicore In-Memory Databases. In *Proceedings of ACM SOSP*, pages 18–32, 2013.

[77] William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. Revisiting the Open vSwitch Dataplane Ten Years Later. In *Proceedings of ACM SIGCOMM*, pages 245–257, 2021.

[78] VMware. Update to VMware's per-CPU Pricing Model | VMware. http://web.archive.org/web/20211023072913/https://news.vmware.com/company/cpu-pricing-model-update-feb-2020.

[79] Xingda Wei, Rongxin Cheng, Yuhan Yang, Rong Chen, and Haibo Chen. Characterizing Off-Path SmartNIC for Accelerating Distributed Systems. In *Proceedings of USENIX OSDI*, pages 987–1004, 2023.

[80] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing RDMA-Enabled Distributed Transactions: Hybrid is Better! In *Proceedings of USENIX OSDI*, pages 233–251, 2018.

[81] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast In-Memory Transaction Processing Using RDMA and HTM. In *Proceedings of ACM SOSP*, pages 87–104, 2015.

[82] Juncheng Yang, Ziyue Qiu, Yazhuo Zhang, Yao Yue, and KV Rashmi. FIFO can be Better than LRU: the Power of Lazy Promotion and Quick Demotion. In *Proceedings of ACM HotOS*, pages 70–79, 2023.

[83] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. NetLock: Fast, Centralized Lock Management Using Programmable Switches. In *Proceedings of ACM SIGCOMM*, pages 126–138, 2020.

[84] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, et al. The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems. In *Proceedings of ACM SOSP*, pages 195–211, 2021.

[85] Irene Zhang, Naveen Kr Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan RK Ports. Building Consistent Transactions with Inconsistent Replication. *ACM Transactions on Computer Systems (TOCS)*, 35(4):1–37, 2018.

[86] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, et al. XRP: In-Kernel Storage Functions with eBPF. In *Proceedings of USENIX OSDI*, pages 375–393, 2022.

[87] Yang Zhou, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu. Electrode: Accelerating Distributed Protocols with eBPF. In *Proceedings of USENIX NSDI*, pages 1391–1407, 2023.

Electrode: Accelerating Distributed Protocols with eBPF

Yang Zhou* *Harvard University*
Zezhou Wang* *Peking University*
Sowmya Dharanipragada *Cornell University*
Minlan Yu *Harvard University*

Abstract

Implementing distributed protocols under a standard Linux kernel networking stack enjoys the benefits of load-aware CPU scaling, high compatibility, and robust security and isolation. However, it suffers from low performance because of excessive user-kernel crossings and kernel networking stack traversing. We present Electrode with a set of eBPF-based performance optimizations designed for distributed protocols. These optimizations get executed in the kernel before the networking stack but achieve similar functionalities as were implemented in user space (e.g., message broadcasting, collecting quorum of acknowledgments), thus avoiding the overheads incurred by user-kernel crossings and kernel networking stack traversing. We show that when applied to a classic Multi-Paxos state machine replication protocol, Electrode improves its throughput by up to 128.4% and latency by up to 41.7%.

1 Introduction

Distributed protocols such as Paxos [37] for state machine replication are important building blocks for highly-available distributed applications. For example, Google's Chubby [6] uses a variant of classic Paxos [37] and Multi-Paxos [36] to implement a highly-available lock service, powering their business-critical GFS [16] and Bigdata [7] applications. Google's globally-distributed database Spanner [8] and Microsoft's data center management tool Autopilot [22] also run Paxos protocols to maintain their high availability.

Existing high-performance implementation of distributed protocols tends to be radical and not readily-deployable. DPDK-based kernel-bypass approaches [27, 79] allow direct access to the underlying NIC hardware, but require application developers to build their own networking stack and maintain compatibility with the evolving kernel networking stack [75]. DPDK also dedicates CPU cores to busily poll the network interface for I/O competition, sacrificing CPU resources and wasting energy during low I/O loads. This is especially a problem for embedded devices [51, 60, 70] where CPU resources are rare. Other approaches co-design specialized distributed systems with niche network hardware including RDMA [11, 28, 76], FPGA [23], SmartNICs [66], and programmable switches [25]. These advanced hardware devices are not widely available in today's cloud environments, and systems built on top of them tend to be difficult to design, implement, and deploy [27].

*Equal contribution

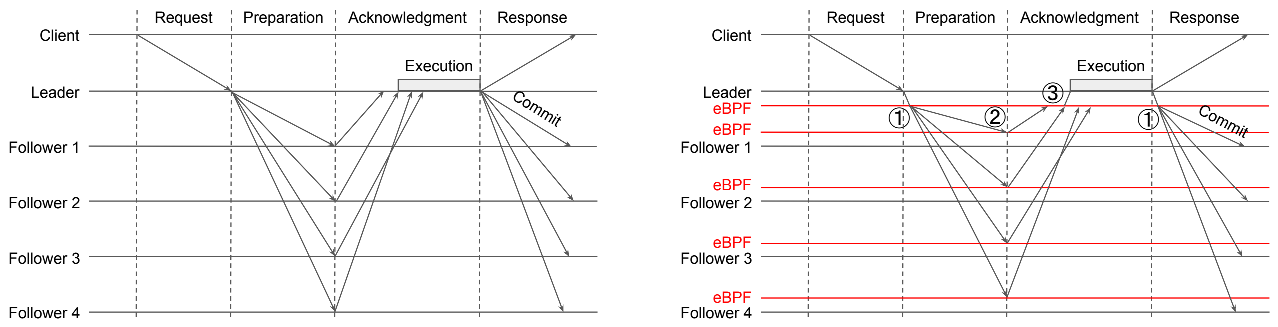
Instead, we would prefer the widely-deployed and well-maintained standard kernel networking stack that also provides load-aware CPU scaling and strong security and isolation among different applications [5, 59]. However, implementing distributed protocols under the standard kernel networking stack often gives poor performance. The root causes are the high packet processing overhead in the kernel networking stack and heavy communications in distributed protocols. Our measurement shows that over half of CPU time is spent on the kernel networking stack in a typical Paxos deployment (§2); such overhead is mainly caused by user-kernel crossings (and associated context switches) and traversing the kernel networking stack. Moreover, when using a classic leader-based Multi-Paxos protocol [43, 54] to implement state machine replication, e.g., with five replicas, processing a single request would require the leader node to send/receive *fourteen* messages in total (see Figure 1a), suffering from the kernel stack overhead fourteen times[1].

In this paper, we focus on accelerating Paxos protocols inside data centers by offloading protocol operations to the kernel via eBPF (i.e., extended Berkeley Packet Filter) [46, 49]. eBPF allows safely executing customized yet constrained functions inside the kernel at various locations. Similar to kernel bypass, the offloaded operations get executed immediately after the NIC driver receives the packet, without user-kernel crossing and kernel networking stack traversing. Unlike kernel bypass, eBPF is an OS-native mechanism such that eBPF-offloaded operations do not sacrifice security and isolation properties while amenable to load-aware CPU scaling without busy-polling.

The key challenge is, given the constrained programming model of eBPF, *which parts of Paxos protocols to offload that can greatly reduce kernel stack overhead while being implementable and efficient in eBPF*. Note that the eBPF subsystem requires every offloaded function to be statically verified to guarantee kernel security, which only allows limited instructions, bounded loops, static memory allocation, etc.

Our insight is that common operations of Paxos protocols, e.g., message broadcasting and waiting on quorums, incur large kernel stack overhead, but are naturally offloadable by existing eBPF programming capacity. For example, Paxos protocols require a leader node to broadcast preparation messages to follower nodes; if implemented using multiple `sendto()` syscalls conventionally, it would incur multiple user-kernel

[1] Linux io_uring [1] can reduce user-kernel crossings, but cannot reduce kernel stack traversing (see §8 for details).



(a) The Multi-Paxos/Viewstamped Replication protocol. **(b)** Electrode-accelerated Multi-Paxos/Viewstamped Replication.

Figure 1: Normal case execution of the leader-based Multi-Paxos/Viewstamped Replication protocol vs. Electrode-accelerated one with 5 replicas. Electrode offloads ①: message broadcasting (§4.1), ②: fast acknowledging (§4.2), and ③: wait-on-quorums (§4.3) to eBPF to reduce the kernel networking stack overhead.

crossings and kernel networking stack traversing. Instead, eBPF has a `bpf_clone_redirect()` [45] function that enables us to clone an in-kernel packet buffer multiple times and send them to different destinations; this eBPF-based message broadcasting only needs one user-kernel crossing and one kernel networking stack traversing. Besides broadcasting, we also utilize eBPF to reduce unnecessary wake-ups of user-space applications when waiting on quorums, and optimize how follower nodes handle preparation messages by early acknowledging before entering the kernel networking stack. The final result of these three eBPF-based optimizations is Electrode[2] (Figure 1b). When applying Electrode to a classic leader-based Multi-Paxos protocol, it achieves up to 128.4% higher throughput and 41.7% lower latency. This translates into up to 112.9% higher throughput and 19.3% lower latency for a Paxos-based transactional replicated key-value store.

Electrode has some limitations: it currently targets protocols implemented in UDP and relies on application-level retransmission to handle packet loss. This works well for Paxos protocols whose requests are usually small enough to fit into a single packet, and data center environments where packet loss is rare [28, 61].

2 Background

2.1 Consensus Protocols

Distributed protocols that coordinate and synchronize among a collection of nodes have become an indispensable part of the modern data center application stack. Storage systems in data centers replicate data for fault tolerance and availability. For instance, Berkeley-DB [55] uses a consensus protocol to replicate its logs over a set of distributed replicas. Transactional storage systems like H-Store [71] and Spanner commit their updates to multiple replicas in order to be more failure resilient. At the heart of most replication-based systems is a consensus protocol [36, 37, 43, 54] that ensures that operations execute in a consistent manner across all replicas.

[2]Electrode is a Pokémon that has a high speed score.

Here, we consider a set of nodes either functioning as clients or replicas. Clients are the users of a particular application-level service hosted by a collection of replicas. It should also be noted here that clients could often just be other servers within the same data center. Clients submit requests to one or more replicas, which triggers a round of agreement to occur. Paxos is a common protocol that is used to obtain an agreement in the presence of node and network failures.

Since applications often need to reach agreements on many client requests, servers use agreement protocols like Paxos to implement a state machine-based abstraction that requires all the replicas to process the exact same set of client requests in the same order. This log-based state machine abstraction is often optimized by the use of a leader. In a leader-based protocol, all the instances of agreement on client requests are mediated through the leader and the leader also dictates the order of the log.

In Figure 1a, we have an example of VR (Viewstamped Replication), a leader-based Multi-Paxos protocol that uses Paxos for running agreements on individual requests. The leader here is responsible for ordering all client requests by assigning sequence numbers to them, and the followers (non-leader nodes) are responsible for responding to the leader and applying all the updates in the order in which they're sequenced by the leader.

The leader is also responsible for initiating agreement by sending out a preparation message to all the other replicas. The leader then waits for a quorum of acknowledgments from all the other replicas before broadcasting a commit message to all the replicas. A successful iteration of this two-round protocol ensures that all non-failed replicas have the client's request. And the sequence number assigned by the leader determines the order in which all the replicas process this client's request. This pattern of broadcasting and waiting on quorums is common in many distributed protocols [38, 39, 80].

To gain more insights into the performance of the Multi-Paxos/VR protocol under the standard Linux kernel networking stack, we measure the CPU time breakdown of the leader node, shown in Table 1. There is $44.7\% +$

Function Name	Description	% CPU
`__libc_sendto()`	User function to send packets.	44.7
⊢ `sock_sendmsg()`	Kernel function to send packets.	32.2
| ⊢ `__alloc_skb()`	Allocate `sk_buff` for packets.	4.5
| ⊢ `dev_queue_xmit()`	Transmit `sk_buff`.	6.8
| ⊢ bookkeeping	For sock, IP, and UDP layers.	**20.9**
⊢ user-kernel crossing	Interrupt, mode switching, etc.	**12.5**
`__libc_recvfrom()`	User function to recv packets.	11.8
⊢ `sock_recvmsg()`	Kernel function to recv packets.	5.7
⊢ user-kernel crossing	Interrupt, mode switching, etc.	**6.1**

Table 1: CPU time breakdown for the leader node when running the Multi-Paxos/Viewstamped Replication protocol with 5 replicas. See §7 for measurement setup.

$11.8\% = 56.5\%$ of time spent on the `__libc_sendto()` and `__libc_recvfrom()` functions, while $20.9\% + 12.5\% + 6.1\% = 39.5\%$ of time spent on user-kernel crossing and kernel networking stack bookkeeping. These numbers concrete our previous motivations that implementing distributed protocols under kernel networking stack incurs significant overhead on user-kernel crossings and kernel stack traversing (while eBPF can potentially save them).

2.2 eBPF and Hooks

BPF (i.e., Berkeley Packet Filter) [49] enables user-space applications to customize packet filtering in the kernel. A BPF program, written in some predicates on packet fields, is triggered by the kernel event that a packet arrives at a NIC driver. Once triggered, the BPF program will run inside a kernel virtual machine with limited registers and scratch memory, and a reduced instruction set [49]. For example, the well-known *tcpdump* [20] command-line packet analyzer is based on BPF.

eBPF extends the BPF by increasing the number of registers and adding stack memory. The increased number of registers and stack memory enable the eBPF program to execute more complex operations—the developers can use a C-like language to express customized operations. This C-like code is compiled into an eBPF bytecode by the Clang/LLVM toolchain and runs inside the kernel virtual machine via just-in-time compilation.

eBPF also introduces various powerful in-kernel data structures called *eBPF maps*, which, paired with various helper functions, are used to store and maintain states across multiple triggering of eBPF programs. Example eBPF maps include array, per-CPU arrays, queues, stacks, and hashMaps [46]. These maps are also used to communicate among different eBPF programs and between eBPF programs and user-space processes. Each eBPF map can be identified by a `map_path` through the file system, e.g., `/sys/fs/bpf/<map_name>`, and user-space processes can access a map based on its path.

The kernel events that can trigger eBPF programs are called *eBPF hooks*. There are many hooks existing in Linux kernels

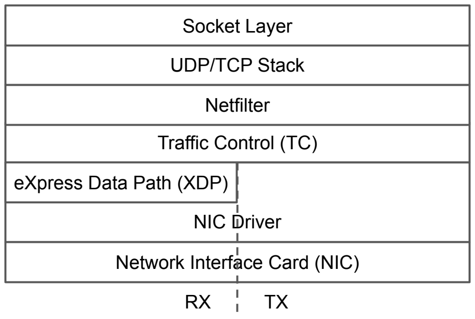


Figure 2: Linux kernel networking stacks and eBPF XDP/TC hooks.

and various device drivers, such as hooks in NIC drivers right after it receives a packet. User-space applications can attach eBPF programs to these eBPF hooks to customize the handling of corresponding kernel events.

Constrained programming model: An eBPF program needs to go through strict verification by an in-kernel eBPF verifier before attaching to an eBPF hook and running inside the kernel. The verification process does a static sanity check to make sure the eBPF program does not have out-of-bounds memory access (i.e., safety) and will always terminate (i.e., liveness). The verifier basically enumerates all possible cases of every conditional branch and loop to make sure every execution path meets the safety and liveness requirements. Because the verification tends to be time-consuming, each eBPF program can only contain up to 1 million instructions. For a larger eBPF program, the developer needs to split it into multiple smaller eBPF programs and uses *tail calls* to let one eBPF program call another one in a continuation manner.

Because of the strict verification process, dynamical memory allocation is not supported in eBPF programs; instead, eBPF programs can only rely on eBPF maps with capacity *specified statically* to maintain in-kernel states.

Due to these limitations, eBPF is commonly used in kernel tracing, profiling, and monitoring [3, 63] and L2-L4 low-level packet processing such as load balancing [14].

XDP (eXpress Data Path) [21, 64] technique implements an in-kernel eBPF hook that enables attached eBPF programs to process RX packets directly out of the NIC driver (Figure 2). Such processing gets triggered before any `sk_buff` [31] allocation or entering software socket queues, thus bypassing any higher-level networking stacks (e.g., UDP, TCP, Socket). XDP-based packet processing normally achieves comparable throughput and latency as DPDK-based kernel-bypass packet processing [21].

TC (Traffic Control) [47] is another important layer/hook which locates right after the XDP (Figure 2). In the TC layer, the `sk_buff` data structure has already been allocated by the kernel networking stack, thus the performance of TC-based packet processing will be slightly worse than XDP. However, the TC hook allows attached eBPF programs to process both RX and TX packets and manipulate the packet `sk_buff`. For example, one can clone the `sk_buff` for a TX packet and thus implements packet broadcasting in the TC layer.

3 Electrode Overview

Electrode is a framework for offloading Paxos protocols under kernel networking stack to in-kernel eBPF programs to reduce user-kernel crossings and kernel networking stack traversing. Electrode has two goals in designing its eBPF offloads: 1) largely reducing kernel stack overhead to improve performance, and 2) carefully partitioning user- and kernel-space functionalities to keep offloads implementable and efficient inside the eBPF subsystem.

To achieve the first goal, Electrode carefully extracts generic and performance-critical fast-path operations from Paxos protocols to offload to the eBPF. As shown in Figure 1b, Electrode offloads message broadcasting (§4.1), fast acknowledging (§4.2), and wait-on-quorums (§4.3). These operations, if purely implemented in the user space, would involve many user-kernel crossings and kernel stack traversing, causing significant kernel stack overhead as shown in §2. Once implemented in the eBPF, message broadcasting allows the leader node to efficiently send preparation and commit messages to multiple follower nodes, by cloning and sending packets in the kernel; fast acknowledging enables follower nodes to buffer preparation messages in the kernel, and quickly respond to the leader node without involving user-space processes; wait-on-quorums lets the leader node eBPF program wait for a quorum number of acknowledgments from follower nodes, and only notify user-space processes once. Moreover, to simplify how user-space applications use these eBPF-based accelerations, Electrode further designs a set of user-space APIs (Table 2). Each API corresponds to one operation that Electrode offloads to the eBPF, and is used to invoke the offloaded function or retrieve eBPF processing results.

To achieve the second goal, Electrode keeps complicated slow-path operations of Paxos protocols in the user space. Specifically, Electrode leaves the procedures of failure recovery and handling message loss/reordering (i.e., gap agreement) to user-space applications, using similar mechanisms as VR [43] and NOPaxos [40]. These procedures involve accessing dynamic ranges of memory, which is hard to implement in eBPF under the static verification (see §8 for details).

Overall, Electrode has the following workflow: first, user-space applications attach eBPF programs to various hook locations corresponding to a network interface; then, user-space applications use Electrode APIs to invoke eBPF-offloaded functions or retrieve eBPF processing results; finally, the eBPF programs intercept and process target packets in the kernel without going through the networking stack or user-space applications (i.e., Paxos protocols in our case). Electrode targets accelerating the handling of messages that can fit into one ethernet packet (i.e., up to 9KB for jumbo frames). This is well-suited for locks, barriers, and configuration parameters [25, 78] that Paxos protocols commonly maintain. Non-target packets still go through the stack and reach user-space applications, without impacting applications' other operations or protocol semantics.

Finally, we note that Electrode does not aim to offload every operation of Paxos protocols to the eBPF, because of eBPF's constrained programming model vs. the diverse set of operations that Paxos protocols and related services could have. For example, currently, Electrode does not offload client-facing request/response handling. There are two reasons: 1) Paxos clients normally serialize/deserialize their requests using widely-used libraries such as protocol buffers [19]; however, parsing or constructing protocol buffers is difficult in eBPF, because it involves complex pointer arithmetics and conditional branches which cannot easily pass the eBPF verifier. 2) client-facing requests/responses are normally embedded into application-level services like the Chubby lock service [6], but it is hard and inefficient to implement them in eBPF because of the strict eBPF verifier and the lack of dynamic memory allocation. We discuss more on Electrode's offloading decisions in §8.

4 Electrode Designs

4.1 Message Broadcasting in TC

In Paxos protocols, one-to-all message broadcasting is widely used. For example, 1) the leader node sends preparation messages to all follower nodes, and 2) (after receiving enough acknowledgments from followers) the leader node sends commit messages to all follower nodes.

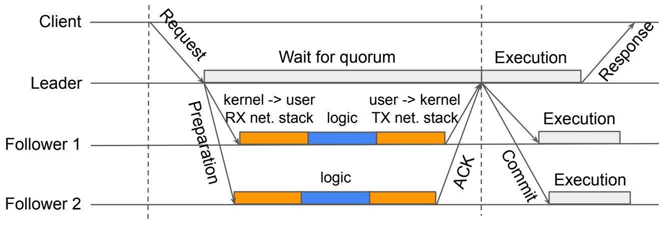
To implement the above message broadcasting, the most common way is sending the same message multiple times in the user space to different destinations. However, the overhead (i.e., user-kernel crossing and kernel networking stack traversing) of this implementation on the leader node increases linearly as the number of followers increases, while the overhead on each follower node remains constant. Thus, the leader node essentially becomes the system bottleneck, e.g., Table 1 has shown that 44.7% of CPU time is spent on sending messages on the leader node.

An alternative implementation is to use IP multicast [42, 68, 77]. However, IP multicast normally requires support from the underlying network switches (e.g., storing a large number of multicast group-table entries for the whole network topology) [68, 77] or considerable modifications of the Linux networking stack [42].

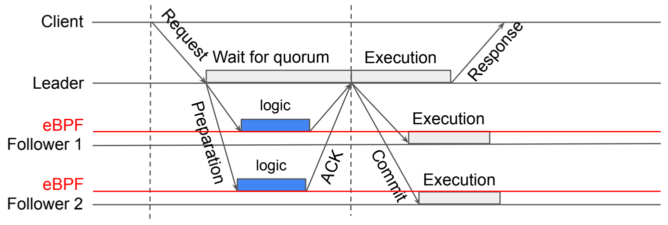
Electrode approach: Electrode provides a flexible host-based broadcasting solution by utilizing eBPF on the TC hook. Here, we require the eBPF program that implements broadcasting operations to attach to the TC hook, because only the TC hook can intercept and process outgoing packets (§2.2). After attaching the eBPF program, user-space applications can call the `elec_broadcast()` function shown in Table 2 with specified `sock_fd`, message, and a list of destination IPs to broadcast the message to these destinations through the socket.

Function Name	Arguments	Output	Description
elec_broadcast	sock_fd, message, {dst_ips}	status	Broadcasts <message> to all destinations through <sock_fd>
elec_poll_message	map_path	messages	Polls buffered messages from an eBPF-maintained in-kernel ring buffer identified by <map_path>
elec_check_quorum	received_message	bool	Checks if <received_message> (acknowledgment) indicates quorum reaching

Table 2: Electrode user-space APIs.



(a) Without fast acknowledging.



(b) With fast acknowledging in eBPF.

Figure 3: Fast acknowledging in eBPF reduces Paxos request latency. This example follows Figure 1, but omits followers 3 and 4 for brevity.

Under the hood, the eBPF program makes clones of the message packet using the `bpf_clone_redirect()` [45] helper function, modifies the destination addresses of cloned packets accordingly, and sends these packets out. The benefit of cloning packets and broadcasting in the kernel compared with sending the same message multiple times in the user space is that we only need to cross the user-kernel boundary and traverse the UDP and socket layer once.

Handling message loss: Electrode relies on application-level timeout and retransmission to handle message loss, similar to modern RPC-based applications [13, 69]. Specifically, if the leader node does not receive a response after a certain time of sending a request, it will resend the request; once a request experiences several timeouts, the leader node will mark the destination node as dead and start Paxos failure recovery. An alternative approach to handling message loss is doing retransmission in the kernel, which could save user-kernel context switching overheads, but such savings become marginal as packet loss happens rarely in data centers [28, 61]; it would also involve complex message buffer management in kernel/eBPF, hurting performance.

4.2 Fast Acknowledging in XDP

As shown in Figure 3a, a significant portion of Paxos request latency comes from the round-trip delay between the leader node and follower nodes. Note that the ACK messages in this figure mean Paxos protocol acknowledgments, not TCP acknowledgments. For Paxos protocols under the kernel networking stack, this round-trip delay includes not only physical propagation and transmission delay, but also the delay caused by the kernel networking stack (i.e., user-kernel crossing and networking stack traversing). As the fabric latency of nowadays data center network reaches a few tens of microseconds [48] or sub-ten microseconds [18, 27], the latency of the kernel networking stack, which is also around sub-ten microseconds [59], becomes non-negligible.

Electrode approach to reducing the Paxos request latency is to optimize the preparation handling in follower nodes by directly buffering the preparation messages into an in-kernel log and early acknowledging to the leader node. At the same time, user-space applications asynchronously poll and consume the buffered messages from the log, using the `elec_poll_message()` function shown in Table 2. Under the hood, the function calls a corresponding eBPF syscall to poll messages in batches, amortizing kernel crossing overhead. This asynchrony does not break the correctness of Paxos protocols because as long as a preparation message gets buffered into the log, it will be eventually processed by the user-space Paxos protocols, and the message processing order has been specified by the sequence number assigned by the leader node. Figure 3b shows that this approach removes *two* user-kernel crossings and networking stack traversing from the critical path of the Paxos request.

Note that not every preparation message can be handled using fast acknowledging; in some non-critical path cases (e.g., message loss/reordering, and node failure) where the eBPF program cannot handle because of its constrained programming model, our eBPF program can detect them and directly forward preparation messages to user-space Paxos protocols (detailed in §6).

In-kernel log implementation: The in-kernel log temporally stores incoming early-acknowledged preparation messages, which are polled and consumed by user-space applications concurrently. To implement this in-kernel log, we use a special eBPF map named `BPF_MAP_TYPE_RINGBUF` [30] (introduced from Linux kernel 5.8). This map implements an efficient multi-producer single-consumer (MPSC) ring buffer using shared memory and a lightweight spinlock, where we can have multiple writers in eBPF and one reader in user space. Based on our measurement, the time of pushing a preparation message into the ring buffer is roughly equal to memcpying this message, in cases without any lock contention. Note that the in-kernel ring buffer also has a fixed size, because eBPF does not support dynamic memory allocation; in case it becomes full, the eBPF program can detect them and directly forward preparation messages to user-space applications.

4.3 Wait-on-Quorums in TC + XDP

Another common operation in Paxos protocols is the leader node waiting for a quorum number of acknowledgments (ACKs) from follower nodes (i.e., wait-on-quorums). Assume there are $2f+1$ replicas including one leader node and $2f$ follower nodes. In most Paxos protocols, once the leader collects f ACKs from different follower nodes, the Paxos request is considered *committed*.

Conventionally, wait-on-quorums is implemented by the user-space applications that receive all ACKs and count towards the quorum number. However, each acknowledgment handling incurs the overhead of the user-kernel crossing and traversing the kernel networking layer. The total overhead of handling all ACKs is linear to the number of follower replicas (i.e., $2f$). Moreover, among these $2f$ ACKs, only the first f ones are required to commit a Paxos request.

Electrode approach: Electrode moves the leader-side wait-on-quorums operations to the eBPF, requiring only one user-kernel crossing and one networking stack traversing. Electrode maintains an array of bitsets (and other metadata) in eBPF, each of which indicates whether a Paxos request has reached the quorum. Electrode only forwards ACK messages that indicate reaching the quorum to the user-space applications, while dropping others. Electrode maps each Paxos request to a specific bitset by using the unique increasing sequence number assigned by the leader node (§2). Note that we use the bitset instead of a counter to check if the quorum gets reached; this is because a timed-out preparation request could cause duplicate ACK messages from follower nodes, and we want to avoid double counting.

Electrode maintains the bitset setting and clearing (i.e., zeroing out) operations through two eBPF programs hooked at TC and XDP layers, respectively. The TC-hooked eBPF program intercepts each outgoing preparation message and clears the indexed bitset, while the XDP-hooked eBPF program intercepts each incoming ACK message from follower nodes and sets the bit corresponding to the follower node's index in replicas.

As shown in Listing 1, the `tc_ebpf` function/program intercepts each outgoing preparation message and clears a specific bitset indexed by the sequence number in each message. Line 6 checks if it is the first time to intercept a preparation message corresponding to this Paxos request, by comparing the `seq` stored along this bitset and the `seq` extracted from the message; if so, it updates the stored `seq` in the array and clears the bitset that may have been used by previous Paxos requests (line 17-18).

```
1  # Processing outgoing preparation message
2  # pkt: the packet of the message
3  # seq: unique increasing sequence number (from pkt)
4  def tc_ebpf(pkt, seq):
5      idx = seq % array_length
6      if array[idx].seq != seq
7          array[idx].seq = seq
8          array[idx].bitset.clear()
9      forward(pkt) # to follower node
10
11 # Processing incoming ACK message
12 # pkt   : the packet of the message
13 # seq   : unique increasing sequence number (from pkt)
14 # node_i: follower node index (from pkt)
15 def xdp_ebpf(pkt, seq, node_i):
16     idx = seq % array_length
17     if array[idx].seq == seq
18         array[idx].bitset.set(node_i)
19         if array[idx].bitset.count() == f
20             pkt.mark_quorum_reach(true)
21             forward(pkt) # to user-space application
22         else: drop(pkt)
23     else: # bitset overwritten by tc_ebpf
24         pkt.mark_quorum_reach(false)
25         forward(pkt)
```

Listing 1: Maintaining the fixed-length bitset array to achieve wait-on-quorums in eBPF. Each bitset operation is also protected by a spinlock; we omit it here for simplicity.

The `xdp_ebpf` program intercepts each incoming ACK message, updates the indexed bitset, drops most of the ACK packets, and only forwards packets to user-space applications that indicate reaching quorum or array overflow (explained in the next paragraph). Lines 17-18 check if this bitset corresponds to the `seq` in the ACK message, and set the proper bitset bit if so. Line 19 further checks if this ACK message reaches the quorum: if so, lines 20-21 will mark the packet as quorum-reaching and forward it to user-space applications; otherwise, line 22 just drops the packet. Once the user-space applications receive a quorum-reaching packet—checked by calling the `elec_check_quorum()` function shown in Table 2, it can directly consider this Paxos request as committed.

Handling array overflow: In some cases, a bitset might be overwritten by the `tc_ebpf` because of the fixed size of the bitset array. `xdp_ebpf` detects such array overflow in lines 17&23; once detected, lines 24-25 will mark the packet as *not*-quorum-reaching and forward it to user-space applications. Once the user-space applications receive a not-quorum-reaching packet, it resends the preparation messages to all follower nodes and waits for ACKs again. In practice, the leader node could limit the number of in-flight preparations while provisioning a large bitset array, such that the array overflow does not normally happen.

RSS: Electrode supports RSS (Receive-Side Scaling) which distributes incoming packets to different NIC queues and CPU cores. Specifically, Electrode has two receive-side optimizations: fast acknowledging and wait-on-quorums. For fast acknowledging, the eBPF programs in the follower node could maintain separate in-kernel ring buffers on different cores to avoid synchronization overhead during log append-

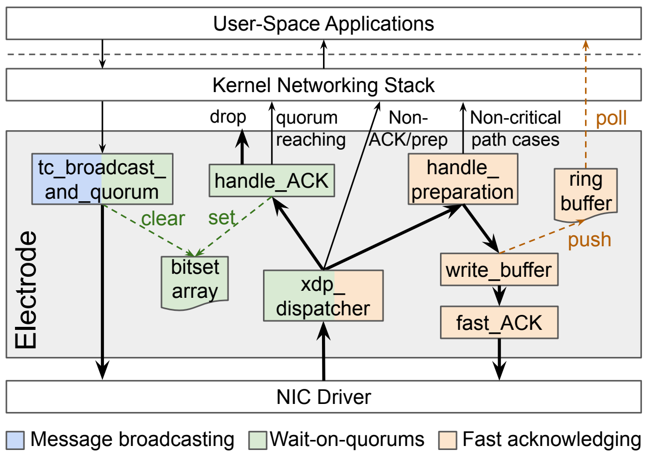


Figure 4: eBPF program structure of Electrode. The thickness of solid lines indicates traffic volume (the thicker, the higher).

ing, and use spinlocks to synchronize accesses to small shared in-kernel states (e.g., `ebpf_seq` in §6); the user-space applications asynchronously pull messages from all ring buffers, and process messages following the order specified by their embedded sequence numbers. For wait-on-quorums, the eBPF programs in the leader node could use atomic instructions to count how many ACKs it has received and check if the quorum is reached.

5 Electrode Implementation

Electrode is prototyped with six eBPF programs written in a restricted C language, and we utilize the Clang/LLVM toolchain for compiling source code to eBPF bytecode. These eBPF programs consist of 500 lines of C code in total. Application developers can also customize their own eBPF programs based on needs, e.g., only processing packets with a specific source port like [25]. Our prototype does not implement the RSS handling yet.

Figure 4 shows the structure of the six eBPF programs. One program can transfer its control flow to the next program via the eBPF tail call. We break the implementation into these six programs because of 1) avoiding breaking the instruction limits in the eBPF verifier (§2.2), and 2) modularity. In the following, we describe each program in detail.

- `tc_broadcast_and_quorum:` This program intercepts outgoing preparation messages. It implements the message broadcasting mechanism (§4.1) and the `tc_ebpf` function in Listing 1 for wait-on-quorums (§4.3). For broadcasting, we generate multiple clones of the preparation packets using the `bpf_clone_redirect()` [45] helper function.
- `xdp_dispatcher:` This program checks the types of incoming messages and calls corresponding message handlers. It only intercepts the ACK (only received on the leader node) and preparation (only received on follower nodes) messages, and calls the corresponding `handle_ACK` and `handle_preparation` programs. It directly forwards other types of messages to user-space applications.
- `handle_ACK:` This program implements the `xdp_ebpf` function in Listing 1 for wait-on-quorums (§4.3). In common cases, it drops most ACK messages, and only forwards the quorum-reaching ACK messages to user-space applications.
- `handle_preparation:` This program implements various checks to detect non-critical path cases where it should forward messages to user-space applications (§4.2). In normal cases (mostly), it will call `write_buffer` to begin `fast_ACK`.
- `write_buffer:` This program stores message/packet data into an in-kernel log for user-space applications to poll and consume. As mentioned earlier, We use the eBPF ring buffer [30] to implement the log data structure. This program then calls the `fast_ACK` program.
- `fast_ACK:` This program reuses and modifies the received packet buffer to create an ACK packet and sent it out. This requires swapping the src-dst IP addresses and filling the corresponding fields of the ACK message.

6 Apply Electrode to Multi-Paxos

Optimizing throughput: We apply the eBPF-based message broadcasting (§4.1) and wait-on-quorums (§4.3) mechanisms to the leader node in the Multi-Paxos protocol. This implies two throughput optimizations: 1) when the leader node sends out preparation messages to follower nodes, it relies on eBPF to broadcast these messages instead of sending them one by one; and 2) when the leader node is waiting for a quorum number of ACK messages from follower nodes, it only needs to process the quorum-reaching ACK message while the other ACK messages are pruned/dropped by the eBPF program. These two optimizations largely reduce the number of user-kernel crossings and kernel networking stack traversing, thus alleviating the CPU bottleneck on the leader node and improving system throughput.

Optimizing latency: We apply the eBPF-based fast acknowledging mechanism (§4.2) to each follower node in the Multi-Paxos protocol. In normal cases (e.g., without packet loss/reordering, and all nodes are alive), the preparation messages from the leader node are quickly buffered and acknowledged by the eBPF program in the follower nodes, bypassing both the kernel networking stack and the user-space Multi-Paxos protocol. This reduces the commit latency of each Multi-Paxos request by twice the time of user-kernel crossing and kernel networking stack traversing.

Detecting non-critical path cases in fast acknowledging: As mentioned in §4.2, there are some non-critical path cases in fast acknowledging where the eBPF program must detect them and forward the incoming packets to the user-space Paxos protocols. To understand why non-critical path cases happen and how to detect them, we first elaborate on the Multi-Paxos/VR protocol shown in §2, following the literature [43]. In the Multi-Paxos protocol, the leader node assigns each Multi-Paxos request a unique and strictly increasing sequence number, `seq`. Each replica including both the leader node and follower nodes maintains locally a `view` number, a `status`, and its last observed `seq`; each message sent by a replica will piggyback these three variables. The `view` number indicates which (leader) election epoch this replica is in; the `status` indicates if this replica is during a leader election (`status_viewchange`), recovering (`status_recovering`), or normal state (`status_normal`). This protocol requires a follower node to only process a preparation message if the node is in the normal state, and the message has a matched `view` and strictly increasing `seq`; otherwise, the follower node needs to drop the message, or execute a complex view-change or state-transfer procedure [43,54]. Therefore, the non-critical path cases for Multi-Paxos are:

1. the follower is during a leader election or recovering,
2. the follower receives a message with an unmatched `view` that is either (a) stale or (b) newer,
3. the follower receives a message with a non-strictly-increasing `seq` caused by message (a) loss/reordering or (b) duplication.

These cases only happen when replicas fail or join, or messages get lost/reordered, which is less common in data centers [27,61].

To detect these non-critical path cases in eBPF, we maintain an `ebpf_status`, an `ebpf_view`, and an `ebpf_seq` variable in the eBPF program using the eBPF map. In particular, these three variables can be updated by the user-space Multi-Paxos protocols to reflect the current protocol state. Listing 2 shows the detection pseudocode. Line 5 detects case 1, and line 6 detects case 2(a); for these two cases, the eBPF program needs to drop the packet. Line 7 detects cases 2(b) and 3(a), and forwards the packet to the user space to execute the view-change or state-transfer procedure. For case 3(b), i.e., $\texttt{msg\_seq} < \texttt{ebpf\_seq} + 1$, the eBPF program function replies an ACK (line 11), because it could be a re-transmitted preparation message due to timeout.

Handling the cases 2(a)&3(a) in fast acknowledging is tricky, because it (i.e., forwarding packets to the user space for processing) involves the concurrency between the user-space protocols and the kernel-space eBPF program, while eBPF only supports map-based communication *but not synchronization* between the user and kernel. Our approach is to let the user-space protocols *detach* the eBPF program from the hook while executing the view-change or state-transfer procedure. Specifically, once a user-space protocol receives a preparation message corresponding to the case 2(a) or 3(a), it detaches the eBPF program, then it finishes the view-change or state-transfer procedure, next it updates the `ebpf_status`, `ebpf_view`, and `ebpf_seq` properly, and finally it reattaches the eBPF program. This guarantees the cases 2(a)&3(a) are exclusively handled by the user-space protocol, avoiding the synchronization between the user and kernel. An alternative approach to achieving the same effect as eBPF detach-reattach

```
 1 # pkt     : the packet of the preparation message
 2 # msg_view: view piggybacked by the pkt
 3 # msg_seq : unique increasing sequence number (from pkt)
 4 def detect_non_crit_path_cases(pkt, msg_view, msg_seq):
 5     if (ebpf_status != status_normal): drop(pkt)
 6     if (msg_view < ebpf_view): drop(pkt)
 7     if (msg_view > ebpf_view or msg_seq > ebpf_seq + 1):
 8         forward(pkt)
 9     if (msg_seq == ebpf_seq + 1):
10         append_log(++ebpf_seq, pkt)
11     reply_ack(pkt)
```

Listing 2: Detecting non-critical path cases during fast acknowledging for Multi-Paxos. Assume the protocol works in a single core, in line with prior Paxos work [40,44,61].

is to use an eBPF map with a branch testing before any Electrode logic. The first packet in the non-critical path can update this map atomically and let all following packets directly go to the user-space application (i.e., closing Electrode optimizations); later, the user-space application can update this map to reopen Electrode optimizations.

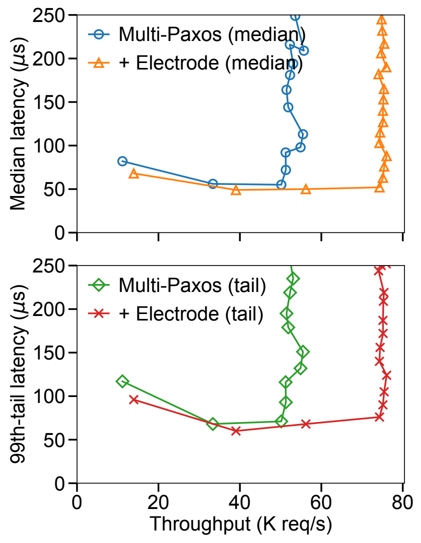
There are a few caveats: 1) After the user-space protocol detaches the eBPF program, it needs to poll the in-kernel ring buffer again, in case the eBPF program still appends a few messages to the ring buffer before detaching. Note that the eBPF map can outlive the eBPF program, as long as the user-space process holds a reference to it, because its lifetime is managed through reference counting [50]. 2) While the user-space protocol is setting the `ebpf_seq` value and is about to reattach the eBPF program, some preparation packets might just pass the eBPF hook location but have not been processed by the user-space protocol, e.g., queued in the socket layer. In this case, the user-space protocol actually has set a smaller `ebpf_seq` value in the map; once the eBPF program gets reattached, it will trigger more case 3(a) (lines 7&8). Our solution to this problem is: after the user-space protocol finishes the view-change or state-transfer procedure, it first sends a `stop_sending_preparation` message to the leader node to stop it from sending preparation messages, then it polls the socket to drain and process any queued packet, next it sets the proper `ebpf_seq` value, finally it sends a `resume_sending_preparation` message to the leader node to resume sending preparation messages, and reattaches the eBPF program. These two messages should be sent using reliable transport like TCP to handle packet loss.

Generalizability: Electrode's eBPF-based optimizations are generic to many more distributed protocols, which normally consist of broadcasting and wait-on-quorums operations. More discussions can be found in Appendix A.

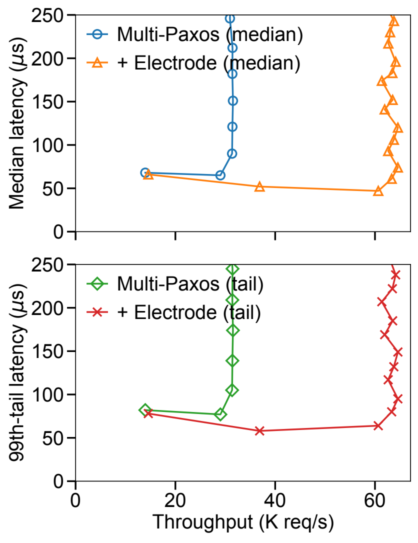
7 Evaluation

This section answers the following questions:

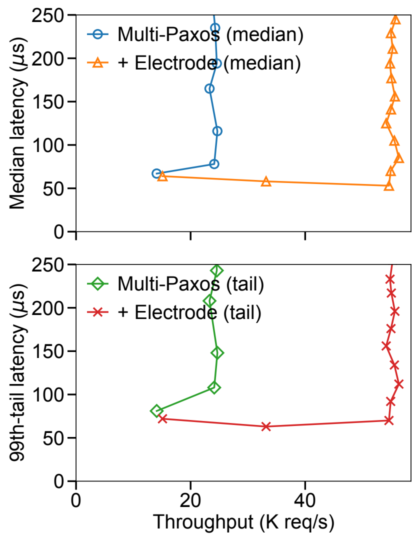
1. How do Electrode and each optimization improve the performance of the Multi-Paxos protocol (§7.1 and §7.2)?



(a) 3 replicas.



(b) 5 replicas.



(c) 7 replicas.

Figure 5: Performance comparison of the Multi-Paxos protocol vs. Electrode-accelerated one with different numbers of replicas.

2. How does Electrode improve the performance of real-world Paxos-based applications (§7.3)?
3. How does Electrode save kernel stack overhead (§7.4)?
4. How does Electrode compare to kernel-bypassing (§7.5)?

Testbed setup: We use eight `xl170` servers from CloudLab [12], each of which has a ten-core Intel E5-2640v4 CPU at 2.4 Ghz, 64GB memory, and a Mellanox ConnectX-4 25 Gbps NIC. Each server runs an unmodified Ubuntu 20.04 OS with kernel v5.8.0. All servers are connected using a two-level topology: five Mellanox 2410 as rack switches (each connecting to forty `xl170` servers) and one Mellanox 2700 as the spine switch. One server is dedicated as the client server that generates Paxos requests, and other servers run the Paxos protocol with 3/5/7-replica configurations. By default, we configure each server to use one core for interrupt processing and another core for Paxos processing, following the performance optimizations in [41]. We disable irqbalance to avoid out-of-order packet deliveries as much as possible (which would hurt Paxos performance), in line with prior Paxos work [40,44,61]. Unlike prior Paxos work [32,40,61], we *do not* use IP multicast which requires specialized support from the network (§4.1).

Measurement methodology: The client server runs multiple Paxos/application clients, and each client sends Paxos/application requests in either a closed-loop or open-loop manner. In closed-loop experiments, each client sends the next request once it receives the response of the last request; we vary the number of clients and measure the corresponding throughput, and median and 99th-percentile tail latency, in line with prior Paxos work [40,44,61]. In open-loop experiments, each client sends requests one by one at a specific time interval, such that the overall request rate reaches a specified value; we use enough clients (i.e., they could saturate the Paxos servers), specify different request rates, and measure the corresponding CPU utilization of each replica node.

Comparisons: We use the Multi-Paxos/VR protocol implementation in the SpecPaxos [61] open-sourced code [35] as the baseline, and optimize it using Electrode. We also run a transactional replicated key-value store similar to the one in SpecPaxos [61] atop the baseline Multi-Paxos protocol and Electrode-accelerated Multi-Paxos protocol. All implementation uses the standard UDP stack and socket layer from the Linux kernel.

7.1 Overall Results

Figure 5a, 5b, and 5c show the performance comparison of the Multi-Paxos protocol and the Electrode-accelerated one when using 3, 5, and 7 replicas, respectively. In each figure, we vary the number of clients sending Multi-Paxos requests in a closed-loop manner, and report throughput and median and 99th-percentile tail latency. All curves eventually hit a "hockey stick" in their median or tail latency growth when the system reaches its maximum throughput.

Throughput: the Electrode-accelerated Multi-Paxos protocol achieves 34.9%, 104.8%, and 128.4% higher maximum throughput than the original Multi-Paxos protocol under 3, 5, and 7 replicas, respectively. The large throughput improvements benefit from the eBPF-based broadcasting and wait-on-quorums which reduce the kernel stack overhead significantly on the leader node. With more replicas, the improvement becomes more significant. This is because, for each Multi-Paxos request, the leader node will send more preparation and commit messages, and handle more ACK messages; thus the eBPF-based broadcasting and wait-on-quorums can save more user-kernel crossings and kernel networking stack traversing.

Latency: the Electrode-accelerated Multi-Paxos protocol achieves 12.5%, 20.0%, and 25.6% lower median latency than the original Multi-Paxos protocol with 2 clients (before

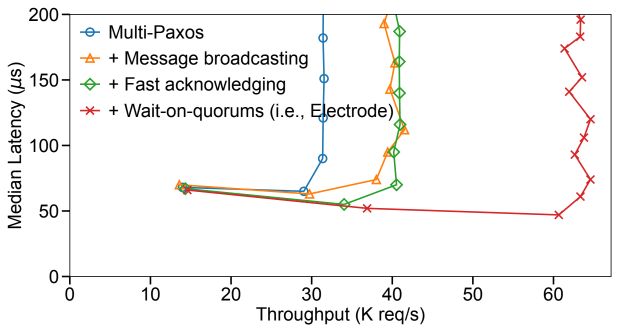
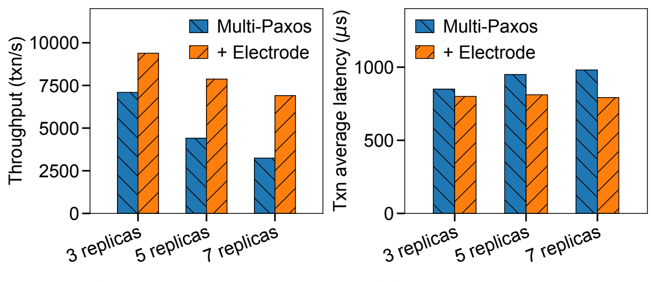


Figure 6: Performance impact of different optimizations for Electrode-accelerated Multi-Paxos protocol (with 5 replicas).



(a) Throughput. **(b)** Latency (one client).

Figure 7: Performance comparison of a transactional key-value store atop the Multi-Paxos protocol vs. Electrode-accelerated one.

the "hockey stick") under 3, 5, and 7 replicas, respectively; the corresponding tail latency is 11.8%, 24.7%, and 41.7% lower. The latency reduction mostly comes from the fast acknowledging in the follower nodes, which, for each Multi-Paxos request, saves the time of two user-kernel crossings, two kernel networking stack traversing, and one wake-up of the user-space process. With more replicas, the latency reduction becomes larger. This is because the fast acknowledging bypasses user-space process scheduling and avoids unpredictable scheduling delays [48] by the OS; for the original Multi-Paxos, with more follower nodes, such unpredictable scheduling delays would raise the chance of follower nodes straggling, thus increasing commit latency. Besides, for Multi-Paxos under 3/5 replicas and Electrode under 7 replicas, their latency curves first decline a bit and arrive at the lowest point, then rise and reach the "hockey stick". This is because, under lower throughput, the Linux scheduler would schedule the Paxos process off the CPU more frequently, while under higher throughput, the Paxos process is mostly scheduled on the CPU.

7.2 Performance Gain Breakdown

Figure 6 shows the performance impact of different optimizations for the Electrode-accelerated Multi-Paxos protocol with 5 replicas. Similar to §7.1, we vary the number of clients sending Multi-Paxos requests in a closed-loop manner, and report the throughput and latency. eBPF-based message broadcasting improves the maximum throughput of the Multi-Paxos protocol by 31.7%; fast acknowledging further reduces the median latency by 4.3%-12.7% (before the "hockey stick"); finally, wait-on-quorums improves the maximum throughput by 57.7%. Overall, we find that the two throughput optimizations (i.e., eBPF-based message broadcasting and wait-on-quorums) have almost no impact on the median latency, while the latency optimization (i.e., fast acknowledging) does not nearly impact maximum throughput. This division of labor demonstrates good modularity of each optimization design in Electrode, and each design can be independently used to accelerate more distributed protocols as shown in Table 4.

7.3 Application Performance

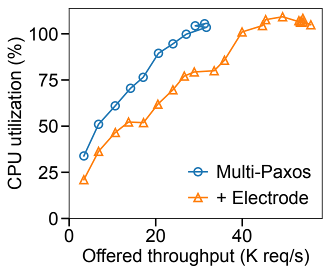
To demonstrate how Electrode can bring benefits to real-world Paxos-based applications, we run a transactional replicated key-value store (similar to the one in SpecPaxos [61]) atop the Multi-Paxos protocol and Electrode-accelerated one. This key-value store supports serializable transactions using two-phase commit and optimistic concurrency control (OCC). Clients use `BEGIN_TXN`, `COMMIT_TXN`, `ABORT_TXN`, `SET`, and `GET` operations to express transactions. We use a synthetic workload derived from the Retwis application [56]—an open-source Twitter clone. This workload consists of four types of transactions with different ratios, and each one issues different numbers of `GET` and `PUT` operations. The workload details can be found in Table 2 of [80]. We vary the number of clients that execute transactions in a closed-loop manner, and measure the maximum throughput these clients can achieve and the average latency under one client.

Figure 7a and 7b shows the maximum throughput and average latency of the key-value store atop the Multi-Paxos protocol vs. Electrode-accelerated one under different numbers of replicas, respectively. Overall, Electrode improves the key-value store throughput by 32.3%-112.9% and latency by 5.9%-19.3%. The improvement becomes larger with more replicas, due to the similar reasons described in §7.1. The latency of the key-value store atop the original Multi-Paxos gradually increases with more replicas, while Electrode-accelerated one's remains relatively stable, because the former is more vulnerable to follower nodes straggling (§7.1).

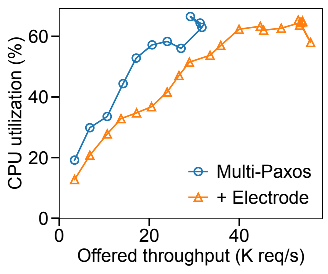
7.4 CPU Utilization

One design goal of Electrode is to reduce the kernel networking stack overhead (§3) when implementing Paxos protocols. Thus, in this subsection, we study the impact of Electrode on CPU utilizations, which indicates how much kernel stack overhead gets reduced.

Figure 8a and 8b show the CPU utilization of the leader node and follower nodes, respectively, for the Multi-Paxos protocol and Electrode-accelerated one with different offered throughput. The experiments are done in an open-loop manner to control the offered throughput when measuring CPU utilization. The CPU utilization covers both the core handling



(a) The leader node.



(b) Follower nodes in average.

Figure 8: CPU utilization comparison of the Multi-Paxos protocol vs. Electrode-accelerated one (with 5 replicas).

interrupts and the core running Paxos. With higher offered throughput, the CPU utilization gradually increases, demonstrating the load-aware CPU scaling provided by the kernel networking stack (§1). We note that for DPDK-based Multi-Paxos protocol implementation, the CPU utilization would be always 100% because DPDK busily polls the network interface. Overall, Electrode reduces the CPU utilization by 22.7%-38.0% on the leader node and 16.0%-35.7% on the follower nodes, benefiting from the reduced user-kernel crossings and kernel stack traversing.

7.5 Comparison with Kernel-Bypassing

Electrode still handles client-facing requests/responses and initiates message broadcasting using the Linux kernel networking stack (§3); thus, it will achieve lower performance than pure kernel-bypassing approaches. This subsection compares the performance of Electrode with a kernel-bypassing baseline, aiming to reveal the performance upper bound of kernel-based approaches and identify the possible improvements for future work.

We choose Caladan [15] and use its high-performance DPDK-based UDP stack to implement our kernel-bypassing baseline. Similar to Caladan, our baseline dedicates one CPU core for packet polling and another core for running the Paxos protocol. We also configure the Caladan runtime to never idle the Paxos core even under low request load.

Table 3 compares the latency and throughput of kernel-based Multi-Paxos and the kernel-bypassing one. To exclude the latency incurred by the client-side kernel stack, we tested all three Paxos implementations with a request generator implemented using Caladan. Electrode achieves 1.4-1.6x lower latency and 2.0x higher throughput than vanilla Linux, but it still has 2.2x higher latency and 2.4x lower throughput compared to pure kernel-bypassing. The performance gap between Electrode and kernel-bypassing exists, because there are still substantial Paxos messages going through the kernel networking stack in Electrode. In particular, our profiling shows that, on the leader node, around 59.5% CPU time is spent on `__libc_sendto()` caused by frequent `dev_queue_xmit()` and `sk_buff` clones. Although eBPF-based broadcasting reduces a significant number of user-kernel crossings and sock-

	Lowest median/99p latency	Maximum throughput
Vanilla Linux	59/69 μs	32 K req/s
Electrode	38/49 μs	65 K req/s
Kernel-bypassing	17/22 μs	154 K req/s

Table 3: Performance comparison of kernel-based Multi-Paxos vs. kernel-bypassing one (with 5 replicas).

/UDP/IP layer traversing, it cannot fundamentally optimize how the Linux kernel manages NICs and packet buffers. Finally, we note that Electrode's goal is to provide generic eBPF-based accelerations for distributed protocol implementations that stick to kernel networking stacks because of compatibility, security, isolation, and elastic CPU scaling.

An additional evaluation regarding how the interrupt coalescing feature of modern NICs impacts Electrode is in Appendix B.

8 Discussion and Future Work

Electrode's offloading decisions: Electrode decides to leave four components of the Multi-Paxos protocol to the user space: 1) failure recovery, 2) handling packet loss and reordering, 3) handling client-facing requests/responses, and 4) executing application-specific operations after reaching the consensus. The first two components involve complex operations on the log, e.g., scanning the log and sending inconsistent entries to other replicas, and inserting missing log entries received from others. These operations require accessing dynamic ranges of log entries, which would fail the eBPF static verification. The last two involve complex serialization/deserialization and application-level operations (see §3). We note that it is possible to offload these four components into eBPF by modifying the kernel eBPF subsystem or verifier—we leave this as future work.

How to improve the eBPF subsystem for offloading? Verifying memory accesses more smartly could make more application operations offloadable. The current eBPF verifier only allows accessing static ranges of memory, which hinders many applications with complex memory accessing behaviors. Another useful construct in eBPF would be dynamic memory allocation, which could ease the maintenance of more advanced data structures in eBPF. To avoid memory leaks, a possible solution could be enforcing Rust-style single-owner memory semantics.

io_uring [1] was recently introduced into the Linux kernel to support efficient batching of asynchronous I/Os via shared memory between the user and kernel space, thus reducing the overhead of frequent user-kernel crossings. Therefore, when implementing Paxos protocols using io_uring, it can help reduce the overhead of message broadcasting, which accounts for 12.5% of CPU time based on Table 1. However, each preparation and ACK message still goes through the full Linux networking stack and wakes up user-space applications, incurring significant overhead; Electrode can be used together with io_uring to reduce such overhead. A recent work XRP [82] shares a similar view regarding io_uring.

Electrode on shared environments: Electrode requires attaching eBPF programs to the network interface, which then processes every packet accordingly. However, multiple Electrode applications might share the same NIC and attach different eBPF programs that might interfere with each other. We can use the SR-IOV (Single Root IO Virtualization) feature that is widely available in modern NICs [2, 9] to avoid such interference. SR-IOV virtualizes a physical network interface into multiple virtualized ones; the Electrode eBPF program can be attached to only one virtualized interface, without impacting others (e.g., used by non-Paxos applications). Besides SR-IOV, Electrode can also check the port numbers of incoming packets in eBPF, and only execute optimizations if the port numbers belong to target Paxos applications.

Accelerating leader-less consensus protocols using eBPF: Electrode targets at leader-based consensus protocols such as Paxos [37] and its variants [36, 43, 54], because they are the most-used ones by modern distributed applications [6, 8, 22]. Electrode's eBPF-based optimizations could also be applied to leader-less consensus protocols, e.g., EPaxos [52], Mencius [4], SD-Paxos [81], etc. For example, replicas in EPaxos could acknowledge preparation messages earlier in an eBPF program before entering the kernel networking stack, thus reducing latency. We leave the exploration of applying Electrode to leader-less consensus protocols as future work.

9 Related Work

Kernel-bypass and hardware offloading: Overheads of the monolithic kernel networking stack have spurred various attempts to design new kernel-bypassed networking stacks like mTCP [24], eRPC [27], Demikernel [79] and more [15, 29, 33, 48, 57, 67], which attempt to eliminate the kernel from the I/O datapath. But all of these solutions are not backward compatible with solutions that already use the standard kernel networking stack, and they incur more costs in terms of CPU cycles and energy during low I/O loads due to busy-polling. Electrode attempts to leverage eBPF to unclog some of the bottlenecks in the kernel networking stack for distributed protocols without completely having to shift to kernel-bypassed stacks.

Similarly, network offload solutions attempt to offload I/O-intensive operations to specialized hardware, e.g., RDMA [11, 28, 76], FPGA [23], SmartNICs [66], and programmable switches [10, 25]. But they come with limited interfaces for programmability and need custom hardware to be installed.

Co-designing distributed systems with networks: There have been attempts to optimize distributed systems by co-designing them with data center networks for improved performance. SpecPaxos [61] attempts to leverage the natural order of packet delivery in data centers to optimize the ordering of messages needed for state machine replication. NoPaxos [40] uses in-network switches to sequence packets for a similar purpose. Eris [39] further applies in-network sequencing to distributed transactions to avoid coordination overhead. These are orthogonal ways to optimize distributed systems and can be used in conjunction with Electrode.

Distributed protocols in data centers: Data centers have a variety of distributed protocols that are deployed for fault tolerance and data consistency. These include replication protocols like Mencius [4], EPaxos [52], chain replication [74], SDPaxos [81], and transaction protocols like TAPIR [80] and Meerkat [72]. Since many distributed protocols share similar patterns of communication like broadcasting and quorum responses, Electrode can be applied to speed up these distributed protocols as well.

eBPF applications: For a long time, eBPF was only used for packet filtering [49], monitoring [3, 63], and load balancing [14] because of its restricted programming model. Now, it is shown to be able to offload small yet critical operations to improve application performance. CCP [53] mentions that it may be possible to leverage the JIT feature of eBPF to gather datapath's congestion measurements for congestion control. BMC [17] uses eBPF to implement an in-kernel cache to accelerate UDP-based Memcached GET requests and achieves significant throughput improvement. Syrup [26] uses eBPF maps to share incoming request information across OS, networking stacks, and application runtimes to enable user-defined scheduling. SPRIGHT [65] employs fast eBPF-based packet forwarding to accelerate sidecar proxies in serverless computing. XRP [82] offloads storage functions (e.g., B-tree lookups) into the kernel using eBPF to reduce kernel storage stack overhead. SynCord [58] leverages eBPF to inject workload-specific and hardware-aware kernel lock policies specified by application developers. Electrode further demonstrates that eBPF can be used to accelerate distributed protocols under the kernel networking stack.

10 Conclusion

Electrode is a system that accelerates distributed protocols using safe in-kernel eBPF-based packet processing before the networking stack. Electrode retains the benefits of using the standard Linux networking stack (e.g., good maintenance, elastic CPU scaling, security, and isolation), while optimizing the performance-critical operations of distributed protocols (e.g., broadcasting, and wait-on-quorums) in a non-intrusive manner. When applying Electrode to a classic Multi-Paxos protocol, we achieve up to 128.4% higher throughput and 41.7% lower latency. We believe that the designs of eBPF-based optimizations in Electrode can motivate more research on improving networked application performance while maintaining the standard Linux networking stack.

Electrode code is available at `https://github.com/Electrode-NSDI23/Electrode`.

Acknowledgments

We thank our shepherd Adam Belay and the anonymous reviewers for their insightful comments. We thank Cloudlab [12] for providing us with the development and evaluation infrastructure. This work was supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. Yang Zhou is also supported by the Google PhD Fellowship.

References

[1] Efficient IO with io_uring. https://kernel.dk/io_uring.pdf.

[2] NVIDIA Corporation affiliates. Single Root IO Virtualization (SR-IOV) for Mellanox NICs. https://docs.nvidia.com/networking/pages/viewpage.action?pageId=43718746.

[3] The Cilium Authors. Cilium: eBPF-Based Networking, Observability, Security. https://cilium.io/.

[4] Catalonia-Spain Barcelona. Mencius: Building Efficient Replicated State Machines for WANs. In *Proceedings of USENIX OSDI*, 2008.

[5] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of USENIX OSDI*, pages 49–65, 2014.

[6] Mike Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Proceedings of USENIX OSDI*, pages 335–350, 2006.

[7] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

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

[9] Intel Corporation. Single Root IO Virtualization (SR-IOV) for Intel NICs. https://www.intel.com/content/www/us/en/support/articles/000005722/ethernet-products.html.

[10] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. Netpaxos: Consensus at Network Speed. In *Proceedings of ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)*, pages 1–7, 2015.

[11] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of ACM SOSP*, pages 54–70, 2015.

[12] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. The Design and Operation of CloudLab. In *Proceedings of USENIX ATC*, pages 1–14, 2019.

[13] Facebook. Facebook's Branch of Apache Thrift, Including a New C++ Server. https://github.com/facebook/fbthrift/blob/main/thrift/doc/cpp/cpp2.md#options.

[14] Facebook. Katran: A High-Performance Layer 4 Load Balancer. https://github.com/facebookincubator/katran.

[15] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating Interference at Microsecond Timescales. In *Proceedings of USENIX OSDI*, pages 281–297, 2020.

[16] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of ACM SOSP*, pages 29–43, 2003.

[17] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing. In *Proceedings of USENIX NSDI*, pages 487–501, 2021.

[18] Dan Gibson, Hema Hariharan, Eric Lance, Moray McLaren, Behnam Montazeri, Arjun Singh, Stephen Wang, Hassan MG Wassel, Zhehua Wu, Sunghwan Yoo, et al. Aquila: A unified, low-latency fabric for datacenter networks. In *Proceedings of USENIX NSDI*, pages 1249–1266, 2022.

[19] Google. Protocol Buffers. https://developers.google.com/protocol-buffers/.

[20] The Tcpdump Group. tcpdump. https://www.tcpdump.org/.

[21] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David

Ahern, and David Miller. The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *Proceedings of ACM CoNEXT*, pages 54–66, 2018.

[22] Michael Isard. Autopilot: Automatic Data Center Management. *ACM SIGOPS Operating Systems Review*, 41(2):60–67, 2007.

[23] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a Box: Inexpensive Coordination in Hardware. In *Proceedings of USENIX NSDI*, pages 425–438, 2016.

[24] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of USENIX NSDI*, pages 489–502, 2014.

[25] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-Free Sub-RTT Coordination. In *Proceedings of USENIX NSDI*, pages 35–49, 2018.

[26] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-Defined Scheduling Across the Stack. In *Proceedings of ACM SOSP*, pages 605–620, 2021.

[27] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be General and Fast. In *Proceedings of USENIX NSDI*, pages 1–16, 2019.

[28] Anuj Kalia, Michael Kaminsky, and David G Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *Proceedings of USENIX OSDI*, pages 185–201, 2016.

[29] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP Acceleration as an OS Service. In *Proceedings of EuroSys*, pages 1–16, 2019.

[30] The Linux kernel development community. BPF Ring Buffer. `https://www.kernel.org/doc/html/latest/bpf/ringbuf.html`.

[31] The Linux kernel development community. struct sk_buff. `https://docs.kernel.org/networking/skbuff.html`.

[32] Marios Kogias and Edouard Bugnion. HovercRaft: Achieving Scalability and Fault-tolerance for microsecond-scale Datacenter Services. In *Proceedings of EuroSys*, pages 1–17, 2020.

[33] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making RPCs First-Class Datacenter Citizens. In *Proceedings of USENIX ATC*, pages 863–880, 2019.

[34] Hsiang-Tsung Kung and John T Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[35] UW Systems Lab. Speculative Paxos Open Source. `https://github.com/UWSysLab/specpaxos`.

[36] Leslie Lamport. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, 2001.

[37] Leslie Lamport. The Part-Time Parliament. In *Concurrency: the Works of Leslie Lamport*, pages 277–317. 2019.

[38] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical Paxos and Primary-Backup Replication. In *Proceedings of ACM PODC*, pages 312–313, 2009.

[39] Jialin Li, Ellis Michael, and Dan RK Ports. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *Proceedings of ACM SOSP*, pages 104–120, 2017.

[40] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan RK Ports. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *Proceedings of USENIX OSDI*, pages 467–483, 2016.

[41] Jialin Li, Naveen Kr Sharma, Dan RK Ports, and Steven D Gribble. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *Proceedings of ACM SoCC*, pages 1–14, 2014.

[42] John C Lin and Sanjoy Paul. RMTP: A Reliable Multicast Transport Protocol. In *Proceedings of IEEE INFOCOM*, volume 96. Citeseer, 1996.

[43] Barbara Liskov and James Cowling. Viewstamped Replication Revisited. 2012.

[44] Xuhao Luo, Weihai Shen, Shuai Mu, and Tianyin Xu. DepFast: Orchestrating Code of Quorum Systems. In *Proceedings of USENIX ATC*, pages 557–574, 2022.

[45] Linux Programmer's Manual. bpf-helpers(7). `https://man7.org/linux/man-pages/man7/bpf-helpers.7.html`.

[46] Linux Programmer's Manual. bpf(2). `https://man7.org/linux/man-pages/man2/bpf.2.html`.

[47] Linux Programmer's Manual. tc-bpf(8). https://man7.org/linux/man-pages/man8/tc-bpf.8.html.

[48] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, Steve Gribble, et al. Snap: A Microkernel Approach to Host Networking. In *Proceedings of ACM SOSP*, pages 399–413, 2019.

[49] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX winter*, volume 46, 1993.

[50] Paul E McKenney. Overview of Linux-Kernel Reference Counting. *N2167*, pages 07–0027, 2007.

[51] Henrique Moniz, Nuno Ferreira Neves, and Miguel Correia. Turquois: Byzantine Consensus in Wireless Ad hoc Networks. In *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, pages 537–546. IEEE, 2010.

[52] Iulian Moraru, David G Andersen, and Michael Kaminsky. There is More Consensus in Egalitarian Parliaments. In *Proceedings of ACM SOSP*, pages 358–372, 2013.

[53] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. Restructuring Endpoint Congestion Control. In *Proceedings of ACM SIGCOMM*, pages 30–43, 2018.

[54] Brian M Oki and Barbara H Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of ACM PODC*, pages 8–17, 1988.

[55] Michael A Olson, Keith Bostic, and Margo I Seltzer. Berkeley DB. In *Proceedings of USENIX ATC, FREENIX Track*, pages 183–191, 1999.

[56] VMware Inc. or its affiliates. Spring Data Redis - Retwis-J. https://docs.spring.io/spring-data/data-keyvalue/examples/retwisj/current/.

[57] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-Sensitive Datacenter Workloads. In *Proceedings of USENIX NSDI*, pages 361–378, 2019.

[58] Sujin Park, Diyu Zhou, Yuchen Qian, Irina Calciu, Taesoo Kim, and Sanidhya Kashyap. Application-Informed Kernel Synchronization Primitives. In *Proceedings of USENIX OSDI*, pages 667–682, 2022.

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

[60] Valentin Poirot, Beshr Al Nahas, and Olaf Landsiedel. Paxos Made Wireless: Consensus in the Air. In *EWSN*, pages 1–12, 2019.

[61] Dan RK Ports, Jialin Li, Vincent Liu, Naveen Kr Sharma, and Arvind Krishnamurthy. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *Proceedings of USENIX NSDI*, pages 43–57, 2015.

[62] Ravi Prasad, Manish Jain, and Constantinos Dovrolis. Effects of Interrupt Coalescence on Network Measurements. In *International Workshop on Passive and Active Network Measurement*, pages 247–256. Springer, 2004.

[63] The IO Visor Project. BPF Compiler Collection (BCC). https://github.com/iovisor/bcc.

[64] The IO Visor Project. eXpress Data Path (XDP). https://www.iovisor.org/technology/xdp.

[65] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and KK Ramakrishnan. SPRIGHT: Extracting the Server From Serverless Computing! High-Performance eBPF-Based Event-Driven, Shared-Memory Processing. In *Proceedings of ACM SIGCOMM*, pages 780–794, 2022.

[66] Henry N Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. Xenic: SmartNIC-Accelerated Distributed Transactions. In *Proceedings of ACM SOSP*, pages 740–755, 2021.

[67] ScyllaDB. SeaStar High Performance Server-Side Application Framework. https://github.com/scylladb/seastar.

[68] Muhammad Shahbaz, Lalith Suresh, Jennifer Rexford, Nick Feamster, Ori Rottenstreich, and Mukesh Hira. Elmo: Source Routed Multicast for Public Clouds. In *Proceedings of ACM SIGCOMM*, pages 458–471. 2019.

[69] Gráinne Sheerin. gRPC and Deadlines. https://grpc.io/blog/deadlines/.

[70] Alberto Spina, Julie McCann, Michael Breza, and Anandha Gopalan. *Reliable Distributed Consensus for Low-Power Multi-Hop Networks*. PhD thesis, Master's thesis, Imperial College London, 2019.

[71] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland.

The End of an Architectural Era: (It's Time for a Complete Rewrite). In *Proceedings of VLDB*, page 1150–1160. VLDB Endowment, 2007.

[72] Adriana Szekeres, Michael Whittaker, Jialin Li, Naveen Kr Sharma, Arvind Krishnamurthy, Dan RK Ports, and Irene Zhang. Meerkat: Multicore-Scalable Replicated Transactions Following the Zero-Coordination Principle. In *Proceedings of EuroSys*, pages 1–14, 2020.

[73] Amy Tai, Igor Smolyar, Michael Wei, and Dan Tsafrir. Optimizing Storage Performance with Calibrated Interrupts. *ACM Transactions on Storage (TOS)*, 18(1):1–32, 2022.

[74] Robbert Van Renesse and Fred B Schneider. Chain Replication for Supporting High Throughput and Availability. In *Proceedings of USENIX OSDI*, volume 4, 2004.

[75] Ed. W. Eddy. RFC 9293: Transmission Control Protocol (TCP). https://datatracker.ietf.org/doc/html/rfc9293.

[76] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing RDMA-Enabled Distributed Transactions: Hybrid is Better! In *Proceedings of USENIX OSDI*, pages 233–251, 2018.

[77] IJsbrand Wijnands, E Rosen, Andrew Dolganow, Tony Przygienda, and Sam Aldrin. RFC 8279: Multicast Using Bit Index Explicit Replication (BIER). https://www.rfc-editor.org/rfc/rfc8279.

[78] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. NetLock: Fast, Centralized Lock Management Using Programmable Switches. In *Proceedings of ACM SIGCOMM*, pages 126–138, 2020.

[79] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, et al. The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems. In *Proceedings of ACM SOSP*, pages 195–211, 2021.

[80] Irene Zhang, Naveen Kr Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan RK Ports. Building Consistent Transactions with Inconsistent Replication. *ACM Transactions on Computer Systems (TOCS)*, 35(4):1–37, 2018.

[81] Hanyu Zhao, Quanlu Zhang, Zhi Yang, Ming Wu, and Yafei Dai. SDPaxos: Building Efficient Semi-Decentralized Geo-Replicated State Machines. In *Proceedings of ACM SoCC*, pages 68–81, 2018.

[82] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, et al. XRP: In-Kernel Storage Functions with eBPF. In *Proceedings of USENIX OSDI*, pages 375–393, 2022.

Types	Protocols	Applying message broadcasting	Applying fast acknowledging	Applying wait-on-quorums
Replication	Primary-backup	The primary broadcasts requests to backups.	Each backup buffers messages in the kernel and quickly responds to the primary.	The primary waits for responses from all backups.
	Chain	None	Each replica (except for the last one) buffers write requests in the kernel and forwards them to the next replica.	None
Concurrency control	Two-phase locking	A transaction coordinator broadcasts `LOCK` and `UNLOCK` requests to all shards.	Each shard maintains a lock table in the kernel and directly handles lock acquiring and releasing.	A transaction coordinator waits for responses from all shards.
	OCC	None	Each shard checks in the kernel if the committing transaction's timestamp conflicts with all other running ones.	None
Atomic commitment	Two-phase commit	A transaction coordinator broadcasts `PREPARE` and `COMMIT` requests to all shards.	Each shard buffers `PREPARE` messages in the kernel and responds to the coordinator, and handles `COMMIT` requests by polling the buffered messages.	A transaction coordinator waits for responses from all shards

Table 4: Applying Electrode to more distributed protocols.

APPENDIX

A Electrode Generalizability

Table 4 summarizes how the classic replication, concurrency control, and atomic commitment protocols can leverage Electrode optimizations. For example, the primary-back replication, two-phase locking, and two-phase commit protocols follow the pattern of sending requests to multiple nodes and waiting for a quorum number of responses; thus they naturally fit well with the eBPF-based message broadcasting and wait-on-quorums. Together with the above protocols, the chain replication [74] and opportunistic concurrency control (OCC) [34] protocols include some critical-yet-simple operations like storing messages in memory, maintaining a lock table, and checking timestamp conflicts; these operations are also suitable for offloading to the eBPF following the fast acknowledging mechanism.

B Impact of Interrupt Coalescing

During benchmarking, we noticed that the interrupt coalescing [62] (IC) feature of modern NICs has a big impact on the measured performance. In IC, after an incoming packet triggers an interrupt, the kernel networking stack waits until a threshold of packets arrives or a timeout gets triggered, aiming to amortize the interrupt cost. In our scenarios, we find it significantly hurts latency and performance predictability in our settings; similar results are also reported in [73]. Thus, in all our experiments, we disable IC by default.

Figure 9 shows the performance impact of IC on the Multi-Paxos protocol and Electrode-accelerated one, by varying the number of open-loop clients. With IC, load-latency curves become unpredictable with two "hockey stick"s. The second "hockey stick" is because the extremely high load triggers coalescing/batching much more packets in one interrupt. Overall, IC does not nearly impact the maximum throughput for the Multi-Paxos protocol and Electrode-accelerated one, but it increases the latency by 57.4%-129.2% and 9.1%-246.8% with 1-3 clients (before the first "hockey stick"). Moreover,

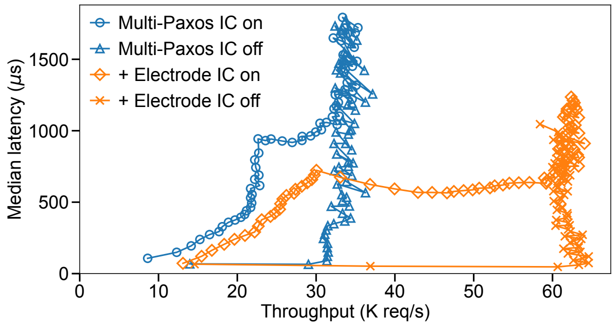


Figure 9: Performance impact of interrupt coalescing (IC) on the Multi-Paxos protocol vs. Electrode-accelerated one (with 5 replicas).

enabling IC decreases the one-client throughput by 38.3% and 10.1% for the original Multi-Paxos and Electrode-accelerated one, respectively.

Electrode performance with IC: Electrode accelerates the maximum throughput of the Multi-Paxos protocol by 81.4% and latency by 32.7% with 1 client (before the first "hockey stick") when IC is on.

[This page intentionally left blank.]

Carbink: Fault-Tolerant Far Memory

Yang Zhou[†][*] Hassan M.G. Wassel[‡] Sihang Liu[§][*] Jiaqi Gao[†] James Mickens[†] Minlan Yu[†][‡]
Chris Kennelly[‡] Paul Turner[‡] David E. Culler[‡] Henry M. Levy[||][‡] Amin Vahdat[‡]

[†]*Harvard University* [‡]*Google* [§]*University of Virginia* [||]*University of Washington*

Abstract

Far memory systems allow an application to transparently access local memory as well as memory belonging to remote machines. Fault tolerance is a critical property of any practical approach for far memory, since machine failures (both planned and unplanned) are endemic in datacenters. However, designing a fault tolerance scheme that is efficient with respect to both computation and storage is difficult. In this paper, we introduce Carbink, a far memory system that uses erasure-coding, remote memory compaction, one-sided RMAs, and offloadable parity calculations to achieve fast, storage-efficient fault tolerance. Compared to Hydra, a state-of-the-art fault-tolerant system for far memory, Carbink has 29% lower tail latency and 48% higher application performance, with at most 35% higher memory usage.

1 Introduction

In a datacenter, matching a particular application to just enough memory and CPUs is hard. A commodity server tightly couples memory and compute, hosting a fixed number of CPUs and RAM modules that are unlikely to exactly match the computational requirements of any particular application. Even if a datacenter contains a heterogeneous mix of server configurations, the load on each server (and thus the amount of available resources for a new application) changes dynamically as old applications exit and new applications arrive. Thus, even state-of-the-art cluster schedulers [51, 52] struggle to efficiently bin-pack a datacenter's aggregate collection of CPUs and RAM. For example, Google [52] and Alibaba [34] report that the average server has only ~60% memory utilization, with substantial variance across machines.

Memory is a particularly vexing resource for two reasons. First, for several important types of applications [19, 20, 33, 54], the data set is too big to fit into the RAM of a single machine, even if the entire machine is assigned to a single application instance. Second, for these kinds of applications, alleviating memory pressure by swapping data between RAM and storage [14] would lead to significant application slowdowns, because even SSD accesses are orders of magnitude slower than RAM accesses. For example, Google runs a graph analysis engine [28] whose data set is dozens of GBs in size. This workload runs 46% faster when it shuffles data purely through RAM instead of between RAM and SSDs.

Disaggregated datacenter memory [2, 5, 15, 16, 22, 44, 46] is a promising solution. In this approach, a CPU can be paired with an arbitrary set of possibly-remote RAM modules, with a fast network interconnect keeping access latencies to far memory small. From a developer's perspective, far memory can be exposed to applications in several ways. For example, an OS can treat far RAM as a swap device, transparently exchanging pages between local RAM and far RAM [5, 22, 46]. Alternatively, an application-level runtime like AIFM [44] can expose remotable pointer abstractions to developers, such that pointer dereferences (or the runtime's detection of high memory pressure) trigger swaps into and out of far memory.

Much of the prior work on disaggregated memory [2, 44, 55] has a common limitation: a lack of fault tolerance. Unfortunately, in a datacenter containing hundreds of thousands of machines, faults are pervasive. Many of these faults are planned, like the distribution of kernel upgrades that require server reboots, or the intentional termination of a low-priority task when a higher-priority task arrives. However, many server faults are unpredictable, like those caused by hardware failures or kernel panics. Thus, any *practical* system for far memory has to provide a scalable, fast mechanism to recover from unexpected server failures. Otherwise, the failure rate of an application using far memory will be much higher than the failure rate of an application that only uses local memory; the reason is that the use of far memory increases the set of machines whose failure can impact an application [8].

Some prior far-memory systems do provide fault tolerance via replication [5, 22, 46]. However, replication-based approaches suffer from high storage overheads. Hydra [29] uses erasure coding, which has smaller storage penalties than replication. However, Hydra's coding scheme stripes a single memory page across multiple remote nodes. This means that a compute node requires multiple network fetches to reconstruct a page; furthermore, computation over that page cannot be outsourced to remote memory nodes, since each node contains only a subset of the page's bytes.

[*]Contributed to this work during internships at Google.

In this paper, we present Carbink,[1] a new framework for far memory that provides efficient, high-performance fault recovery. Like (non-fault-tolerant) AIFM, Carbink exposes far memory to developers via application-level remoteable pointers. When Carbink's runtime must evict data from local RAM, Carbink writes erasure-coded versions of that data to remote memory nodes. The advantage of erasure coding is that it provides equivalent redundancy to pure replication, while avoiding the double or triple storage overheads that replication incurs. However, straightforward erasure coding is a poor fit for the memory data created by applications written in standard programming languages like C++ and Go; those applications allocate variable-sized memory objects, but erasure coding requires equal-sized blocks. To solve this problem, Carbink eschews the object-granularity swapping strategy of AIFM, and instead swaps at the granularity of *spans*. A single span consists of multiple memory pages that contain objects with similar sizes. Carbink's runtime asynchronously and transparently moves local objects within the spans in local memory, grouping cold objects together and hot objects together. When necessary, Carbink batch-evicts cold spans, calculating parity bits for those spans at eviction time, and writing the associated fragments to remote memory nodes. Carbink utilizes one-sided remote memory accesses (RMAs) to efficiently perform swapping activity, minimizing network utilization. Unlike Hydra, Carbink's erasure coding scheme allows a compute node to fetch a far memory region using a single network request.

In Carbink, each span lives in exactly one place: the local RAM of a compute node, or the far RAM of a memory node. Thus, swapping a span from far RAM to local RAM creates dead space (and thus fragmentation) in far RAM. Carbink runs pauseless defragmentation threads in the background, asynchronously reclaiming space to use for later swap-outs.

We have implemented Carbink atop our datacenter infrastructure. Compared to Hydra, Carbink has up to 29% lower tail latency and 48% higher application performance, with at most 35% more remote memory usage. Unlike Hydra, Carbink also allows computation to be offloaded to remote memory nodes.

In summary, this paper has four contributions:

- a span-based approach for solving the size mismatch between the granularity of erasure coding and the size of the objects allocated by compute nodes;
- new algorithms for defragmenting the RAM belonging to remote memory nodes that store erasure-encoded spans;
- an application runtime that hides spans, object migration within spans, and erasure coding from application-level developers; and
- a thorough evaluation of the performance trade-offs made by different approaches for adding fault tolerance to far memory systems.

[1]Carbink is a Pokémon that has a high defense score.

2 Background

Recent work on far memory has used one of two approaches. The first approach modifies the OS that runs applications, exploiting the fact that preexisting OS abstractions already decouple application-visible in-memory data from the backing storage hierarchy. For example, INFINISWAP [22], Fastswap [5], and LegoOS [46] leverage virtual memory support to swap application memory to far RAM instead of a local SSD or hard disk. Applications use standard language-level pointers to interact with memory objects; behind the scenes, the OS swaps pages between local RAM and far RAM, e.g., in response to page faults for non-locally-resident pages. In contrast, the remote region approach [2] exposes far memory via file system abstractions. Applications name remote memory regions using standard filenames, and interact with regions using standard file operations like `open()` and `read()`.

Exposing far memory via OS abstractions is attractive because it requires minimal changes to application-level code. However, invasive kernel changes are needed; such changes require substantial implementation effort, and are difficult to maintain as other parts of the kernel evolve.

The second far-memory approach requires more help from application-level code. For example, AIFM [44] uses a modified C++ runtime to hide the details of managing far memory. The runtime provides special pointer types whose dereferencing may trigger the swapping of a remote C++-level object into local RAM. AIFM's runtime tracks object hotness using GC-style read/write barriers, and uses background threads to swap out cold local objects when local memory pressure is high. To synchronize the local memory accesses generated by application threads and runtime threads, AIFM embeds a variety of metadata bits (e.g., `present`, `isBeingEvicted`) in each smart pointer, leveraging an RCU-like scheme [36] to protect concurrent accesses to a pointer's referenced object.

Listing 1 provides an example of how applications use AIFM's smart pointers. Like AIFM, Carbink exposes far memory via smart pointers, but unlike AIFM, Carbink provides fault tolerance.

3 Carbink Design

Figure 1 depicts the high-level architecture of Carbink. **Compute nodes** execute single-process (but potentially multithreaded) applications that want to use far memory. **Memory nodes** provide far memory that compute nodes use to store application data that cannot fit in local RAM. A logically-centralized **memory manager** tracks the liveness of compute nodes and memory nodes. The manager also coordinates the assignment of far memory **regions** to compute nodes. When a memory node wants to make a local memory region available to compute nodes, the memory node *registers* the region with the memory manager. Later, when a compute node requires far memory, the compute node sends an *allocation* request to the memory manager, who then assigns a registered, unallo-

```
RemUniquePtr<Node> rem_ptr = AIFM::MakeUnique<Node>();
{
  DerefScope scope;
  Node* normal_ptr = rem_ptr.Deref(scope);
  computeOverNodeObject(normal_ptr);
} // Scope is destroyed; Node object can be evicted.
```

Listing 1: Example of how AIFM applications interact with far memory. In the code above, the application first allocates a `Node` object that is managed by a particular `RemUniquePtr`. Such a remote unique pointer represents a pointer to an object that (1) can be swapped between local and far memory, and (2) can only be pointed to by a single application-level pointer. The code then creates a new scope via an open brace, declares a `DerefScope` variable, and invokes the `RemUniquePtr`'s `Deref()` method, passing the `DerefScope` variable as an argument. `Deref()` essentially grabs an RCU lock on the remotable memory object, and returns a normal C++ pointer to the application. After the application has finished using the normal pointer, the scope terminates and the destructor of the `DerefScope` runs, releasing the RCU lock and allowing the object to be evicted from local memory.

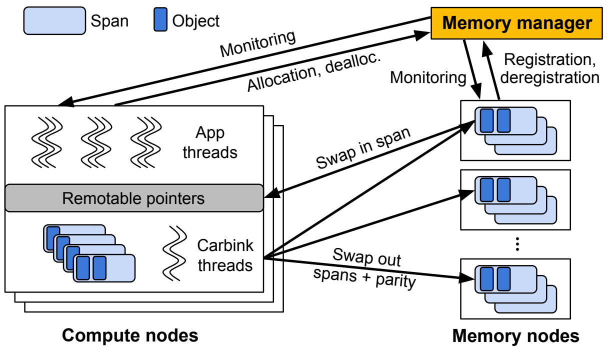


Figure 1: Carbink's high-level architecture.

cated region. Upon receiving a *deallocation* message from a compute node, the memory manager marks the associated region as available for use by other compute nodes. A memory node can ask the memory manager to *deregister* a previously registered (but currently unallocated) region, withdrawing the region from the global pool of far memory.

Carbink does not require participating machines to use custom hardware. For example, any machine in a datacenter can be a memory node if that machine runs the Carbink memory host daemon. Similarly, any machine can be a compute node if that node's applications use the Carbink runtime.

From the perspective of an application developer, the Carbink runtime allows a program to dynamically allocate and deallocate memory objects of arbitrary size. As described in Section 3.2, programs access those objects through AIFM-like remotable pointers [44]. When applications dereference pointers that refer to non-local (i.e., swapped-out) objects, Carbink pulls the desired objects from far memory. Under the hood, Carbink's runtime manages objects using **spans** (§3.3) and **spansets** (§3.4). A span is a contiguous run of memory pages; a single region allocated by a compute node contains one or more spans. Similar to slab allocators like Facebook's jemalloc [17] and Google's TCMalloc [21, 24], Carbink rounds up each object allocation to the bin size of the relevant span, and aligns each span to the page size used by compute nodes and memory nodes. Carbink swaps far memory into local memory at the granularity of a span; however, when local memory pressure is high, Carbink swaps local memory out to far memory at the granularity of a spanset (i.e., a collection of spans of the same size). In preparation for swap-outs, background threads on compute nodes group cold objects into cold spans, and bundle a group of cold spans into a spanset; at eviction time, the threads generate erasure-coding parity data for the spanset, and then evict the spanset and the parity data to remote nodes. As we discuss in Sections 3.4 and 3.5, this approach simplifies memory management and fault tolerance.

Carbink disallows cross-application memory sharing. This approach is a natural fit for our target applications, and has the advantage of simplifying failure recovery and avoiding the need for expensive coherence traffic [46].

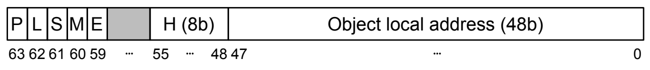
3.1 Failure Model

Carbink implements the logically-centralized memory manager as a replicated state machine [1, 45]. Thus, Carbink assumes that the memory manager will not fail. Carbink assumes that memory nodes and compute nodes may experience fail-stop faults. Carbink does not handle Byzantine failures or partial network failures.

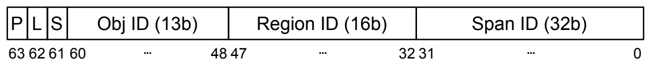
The memory manager tracks the liveness of compute nodes and memory nodes via heartbeats. When a compute node fails, the memory manager instructs the memory nodes to deallocate the relevant spans; if applications desire, they can use an application-level fault tolerance scheme like checkpointing to ensure that application-level data is recoverable. When a memory node fails, the memory manager deregisters the node's regions from the global pool of far memory. However, erasure-coding recovery of the node's regions is initiated by a compute node when the compute node unsuccessfully tries to read or write a span belonging to the failed memory node. If an application thread on a compute node tries to read a span that is currently being recovered, the read will use Carbink's degraded read protocol (§3.5), reconstructing the span using data from other spans and parity blocks.

3.2 Remotable Pointers

Like AIFM, Carbink exposes far memory through C++-level smart pointers. However, as shown in Figure 2, Carbink uses a different pointer encoding to represent span information.



(a) Local object.



(b) Far object.

Field	Meaning
Present	Is the object in local RAM or far RAM?
Lock	Is the object (spin)locked by a thread?
Shared	Is the pointer a unique pointer or a shared pointer?
Moving	Is the object being moved by a background thread?
Evicting	Is the object being evicted by a background thread?
Hotness	Is the object frequently accessed?

(c) Field semantics.

Figure 2: Carbink's `RemUniquePtr` representation. In contrast to AIFM [44], Carbink does not embed information about a data structure ID or an object size. Instead, Carbink embeds span metadata (namely, a Region ID and a Span ID) to associate a pointed-to object with its backing span.

A Carbink `RemUniquePtr` has the same size as a traditional `std::unique_ptr` (i.e., 8 bytes). The **P**resent bit indicates whether the pointed-to object resides in local RAM. The **S**hared bit indicates whether a pointer implements unique-pointer semantics or shared-pointer semantics; the former only allows a single reference to the pointed-to object. The **L**ock, **M**oving, and **E**victing bits are used to synchronize object accesses between application threads and Carbink's background threads (§3.6). The **H**otness byte is consulted by the background threads when deciding whether an object is cold (and thus a priority for eviction).

If an object is local, the local virtual address of the object is directly embedded in the pointer. If an object has been evicted, the pointer describes how to locate the object. In particular, the Obj ID indicates the location of an object within a particular span; the Span ID identifies that span; and the Region ID denotes the far memory region that contains the span.

Carbink supports two smart pointer types: `RemUniquePtr`, which only allows one reference to the underlying object, and `RemSharedPtr`, which allows multiple references. When moving or evicting an object, Carbink's background threads need a way to locate and update the smart pointer(s) which reference the object. To do so, Carbink uses AIFM's approach of embedding a "reverse pointer" in each object; the reverse pointer points to the object's single `RemUniquePtr`, or to the first `RemSharedPtr` that references the object. An individual `RemSharedPtr` is 16 bytes large, with the last 8 bytes storing a pointer that references the next `RemSharedPtr` in the list. Thus, Carbink's runtime can find all of an object's `RemSharedPtrs` by discovering the first one via the object's reverse pointer, and then iterating across the linked list.

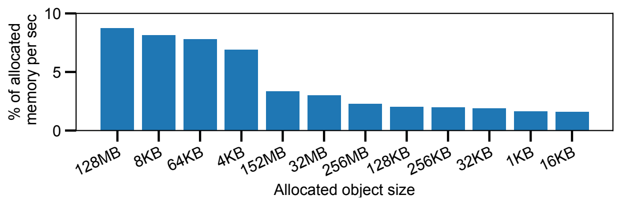


Figure 3: Allocation sizes in our production workloads.

3.3 Span-Based Memory Management

Local memory management: A span is a contiguous set of pages that contain objects of the same size class. Carbink supports 86 different size classes, and aligns each span on an 8KB boundary; Carbink borrows these configuration parameters from TCMalloc [21, 24], which observed these parameters to reduce internal fragmentation. When an application allocates a new object, Carbink tries to round the object size up to the nearest size class and assign a free object slot from an appropriate span. If the object is bigger than the largest size class, Carbink rounds the object size up to the nearest 8KB-aligned size, and allocates a dedicated span to hold it.

To allocate spans locally, Carbink uses a *local page heap*. The page heap is an array of free lists, with each list tracking 8KB-aligned free spans of a particular size (e.g., 2MB, 4MB, etc.). If Carbink cannot find a free span big enough to satisfy an allocation request, Carbink allocates a new span, using `mmap()` to request 2MB huge pages from the OS.

Allocating and deallocating via the page heap is mutex-protected because application threads may issue concurrent allocations or deallocations. To reduce contention on the page heap, each thread reserves a private (i.e., thread-local) cache of free spans for each size class. Carbink also maintains a global cache of free lists, with each list having its own spinlock. When a thread wants to allocate a span whose size can be handled by one of Carbink's predefined size classes, the thread first tries to allocate from the thread-local cache, then the global cache, and finally the page heap. For larger allocation requests, threads allocate spans directly from the page heap.

Carbink associates each span with several pieces of metadata, including an integer that describes the span's size class, and a bitvector that indicates which object slots are free. To map a locally-resident object to its associated span metadata, Carbink uses a two-level radix tree called the *local page map*. The lookup procedure is similar to a page table walk: the first 20 bits of an object's virtual address index into the first-level radix tree table, and the next 15 bits index into a second-level table. The same mapping approach allows Carbink to map the virtual address of a locally-resident span to its metadata.

Far memory management: On a compute node, local spans contain a subset of an application's memory state. The rest of that state is stored in far spans that live in far memory regions. Recall from Figure 2b that a Carbink pointer to a non-local object embeds the object's Region ID and Span ID.

To allocate or deallocate a region, a compute node sends a request to the memory manager. A single Carbink region is 1GB or larger, since Carbink targets applications whose total memory requirements are hundreds or thousands of GBs. Upon successfully allocating a region, the compute node updates a *region table* which maps the Region ID of the allocated region to the associated far memory node.

A compute node manages far spans and far regions using additional data structures that are analogous to the ones that manage local spans. A *far page heap* handles the allocation and deallocation of far spans belonging to allocated regions. A *far page map* associates a far Span ID with metadata that (1) names the enclosing region (as a Region ID) and (2) describes the offset of the far span within that region.

Each application thread has a private far cache; Carbink also maintains a global far cache that is visible to all application threads. To swap out a local span of size *s*, a compute node must first use the far page heap (or a far cache if possible) to allocate a free far span of size *s*. Similarly, after a compute node swaps in a far span, the node deallocates the far span, returning the far span to its source (either the far page heap or a far cache).

Span filtering and swapping: The Carbink runtime executes *filtering threads* that iterate through the objects in locally-resident spans and move those objects to different local spans. Carbink's object shuffling has two goals.

- First, Carbink wants to create *hot spans* (containing only hot objects) and *cold spans* (containing only cold ones); when local memory pressure is high, Carbink's *eviction threads* prefer to swap out spansets containing cold spans. Carbink tracks object hotness using GC-style read/write barriers [4, 23]. Thus, by the time that a filtering thread examines an object, the Hotness byte in the object's pointer (see Figure 2) has already been set. Upon examining the Hotness byte, a filtering thread updates the byte using the CLOCK algorithm [12].
- Second, object shuffling allows Carbink to garbage-collect dead objects by moving live objects to new spans and then deallocating the old spans. During eviction, Carbink utilizes efficient one-sided RMA writes to swap spansets out to far memory nodes; this approach allows Carbink to avoid software-level overheads (e.g., associated with thread scheduling) on the far node.

From the application's perspective, object movement and spanset eviction are transparent. This transparency is possible because each object embeds a reverse pointer (§3.2) that allows filtering threads and evicting threads to determine which smart pointers require updating.

Carbink swaps far memory into local memory at the granularity of a span. As with swap-outs, Carbink uses one-sided RMAs for swap-ins. Swapping at the granularity of a span simplifies far memory management, since compute nodes only have to remember how spans map to memory nodes (as opposed to how the much larger number of *objects* map to memory nodes). However, swapping in at span granularity instead of object granularity has a potential disadvantage: if a compute node swaps in a span containing multiple objects, but only uses a small number of those objects, then the compute node will have wasted network bandwidth (to fetch the unneeded objects) and CPU time (to update the remotable pointers for those unneeded objects). We collectively refer to these penalties as *swap-in amplification*.

To reduce the likelihood of swap-in amplification, Carbink's filtering and eviction threads prioritize the scanning and eviction of spansets containing large objects. The associated spans contain fewer objects per span; thus, swapping in these spans will reduce the expected number of unneeded objects. Figure 3 shows that, for our production workloads, large objects occupy the majority of memory. Moreover, most hot objects are small; for example, in our company's geo-distributed database [13], roughly 95% of accesses involve objects smaller than 1.8KB. As a result, an eviction scheme which prioritizes large-object spansets is well-suited for our target applications.

In Carbink, a local span has a three-state lifecycle. A span is first *created* due to a swap-in or local allocation. The span transitions to the *filtering* state upon being examined by filtering threads. Once filtering completes, those spans transition to the *evicting* state when evicting threads begin to swap out spansets. The transition from created to filtering to evicting is fixed, and determines which Carbink runtime threads race with application threads at any given moment (§3.6).

3.4 Fault Tolerance via Erasure Coding

Erasure coding provides data redundancy with lower storage overhead than traditional replication. However, the design space for erasure coding schemes is more complex. Carbink seeks to minimize both average and long-tail access penalties for far objects; per our fault model (§3.1), Carbink also wants to efficiently recover from the failure of memory nodes. Achieving these goals forced us to make careful decisions involving coding granularity, parity recalculation, and cross-node transport protocols.

Coding granularity: To motivate Carbink's decision to erasure-code at the spanset granularity, first consider an approach that erasure-codes individual spans. In this approach, to swap out a span, a compute node breaks the span into data fragments, generates the associated parity fragments, and then writes the entire set of fragments (data+parity) to remote nodes. During the swap-in of a span, a compute node must fetch multiple fragments to reconstruct the target span.

This scheme, which we call EC-Split, is used by Hydra [29]. With EC-Split, handling the failure of memory nodes during swap-out or swap-in is straightforward: the compute node who is orchestrating the swap-out or swap-in will detect the memory node failure, select a replacement memory node, trigger span reconstruction, and then restart the swap-in or

Schemes	EC data fragment size	Network transport	Parity computation	Defragmentation
EC-Split (Hydra [29])	Span chunk	RMA in & out	Local	N/A
EC-2PC	Full span	RMA in, RPC out (+updating parity via 2PC)	Remote	N/A
EC-Batch Local (**Carbink**)	Full span	RMA in & out	Local	Remote compaction
EC-Batch Remote (**Carbink**)	Full span	RMA in & out (+parallel 2PC for compaction)	Local (swap-out)+ Remote (compaction)	Remote compaction

Table 1: The erasure-coding approaches that we study.

swap-out. The disadvantage of EC-Split is that, to reconstruct a single span, a compute node must contact multiple memory nodes to pull in all of the needed fragments. This requirement to contact multiple memory nodes makes the swap-in operation vulnerable to stragglers (and thus high tail latency[2]). This requirement also frequently prevents a compute node from offloading computation to memory nodes; unless a particular object is small, the object will span multiple fragments, meaning that no single memory node will have a complete local copy of the object.

An alternate approach is to erasure-code across a group of equal-sized spans. We call such a group a *spanset*. In this approach, each span in the spanset is treated as a fragment, with parity data computed across all of the spans in the set. To reconstruct a span, a compute node merely has to contact the single memory node which stores the span. Carbink uses this approach to minimize tail latencies.

Parity updating: Erasure-coding at the spanset granularity but swapping in at the span granularity does introduce complications involving parity updates. The reason is that swapping in a span s leaves an invalid, span-sized hole in the backing spanset; the hole must be marked as invalid because, when s is later swapped out, s will be swapped out as part of a new spanset. The hole created by swapping in s causes fragmentation in the backing spanset. Determining how to garbage-collect the hole and update the relevant parity information is non-trivial. Ideally, a scheme for garbage collection and parity updating would not incur overhead on the critical path of swap-ins or swap-outs. An ideal scheme would also allow parity recalculations to occur at either compute nodes or memory nodes, to enable opportunistic exploitation of free CPU resources on both types of nodes.

Cross-node transport protocols: In systems like RAMCloud [39], machines use RPCs to communicate. RPCs involve software-level overheads on both sides of a communication. Carbink avoids these overheads by using one-sided RMA, avoiding unnecessary thread wakeups on the receiver. However, in and of itself, RMA does not automatically solve the consistency issues that arise when offloading parity calculations to remote nodes (§3.4.2).

Throughout the paper, we compare Carbink's erasure-coding approach to various alternatives.

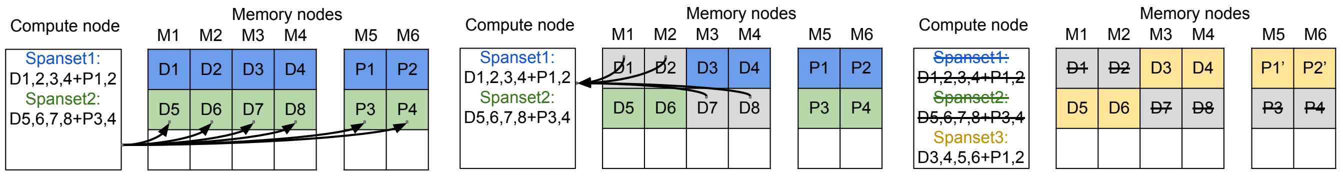
[2]Hydra [29] and EC-Cache [42] try to minimize straggler-induced latencies by contacting $k+\Delta$ memory nodes instead of the minimum k, using the first k responses to reconstruct an object. This approach increases network traffic and compute-node CPU overheads.

- **EC-Split** is Hydra's approach, which erasure-codes at the span granularity, swaps data using RMA, and synchronously recalculates parity at compute nodes when swap-outs occur. Fragmentation within an erasure-coding group never occurs, as a span is swapped in and out as a full unit.
- **EC-2PC** erasure-codes using spansets, and uses RMA to swap in at the span granularity. During a swap-out (which happens at the granularity of a span), EC-2PC writes the updated span to the backing memory node; the memory node then calculates the updates to the parity fragments, and sends the updates to the relevant memory nodes which store the parity fragments. To provide crash consistency for the update to the span and the parity fragments, EC-2PC implements a two-phase commit protocol using RPCs. There is no fragmentation within an erasure-coding group because swap-ins and swap-outs both occur at the span granularity.
- **EC-Batch Local** and **EC-Batch Remote** are the approaches used by Carbink. Both schemes erasure-code at spanset granularity, using RMA for swap-in as well as swap-out. Swap-ins occur at the granularity of a span, but swap-outs occur at the granularity of spansets (§3.4.1); thus, both EC-Batch approaches deallocate a span's backing area in far memory upon swapping that span into a compute node's local RAM. The result is that swap-ins create dead space on a remote memory node. Both EC-Batch schemes reclaim dead space and recalculate parity data using asynchronous garbage collection. EC-Batch Local always recalculates parity on compute nodes, whereas EC-Batch Remote can recalculate parity on compute nodes or memory nodes. When EC-Batch Remote offloads parity computations to remote nodes, it employs a parallel commit scheme that avoids the latencies of traditional two-phase commit (§3.4.2).

Table 1 summarizes the various schemes. We now discuss EC-Batch Local and Remote in more detail.

3.4.1 EC-Batch: Swapping

Swapping out: In both varieties of EC-Batch, a spanset contains multiple spans of the same size. At swap-out time, a compute node writes a *batch* (i.e., a spanset and its parity fragments) to a memory node. Figure 4a shows an example. In that example, the compute node has two spansets: spanset1 (consisting of data spans $<D1,D2,D3,D4>$ and parity fragments $<P1,P2>$), and spanset2 (containing data spans $<D5,D6,D7,D8>$ and parity fragments $<P3,P4>$). Carbink uses Reed-Solomon codes [43] to create parity data, and prioritizes the eviction of spansets that contain cold spans



(a) Swapping out spans and parity in a batch. **(b)** Swapping in individual spans. **(c)** Compacting spansets to reclaim space.

Figure 4: EC-Batch swapping-out, swapping-in, and far compaction.

(§3.3). Neither variant of EC-Batch updates spansets in place, so eviction may require a compute node to request additional far memory regions from the memory manager.

Swapping in: When an application tries to access an object that is currently far, the Carbink runtime inspects the application pointer and extracts the Span ID (see Figure 2b). The runtime consults the far page map (§3.3) to discover which remote node holds the span. Finally, the runtime initiates the appropriate RMA operation to swap in the span.

However, swapping in at the span granularity creates *remote fragmentation*. In Figure 4b, the compute node in the running example has pulled four spans ($D1$, $D2$, $D7$, and $D8$) into local memory. Any particular span lives exclusively in local memory or far memory; thus, the swap-ins of the four spans creates dead space on the associated remote memory nodes. If Carbink wants to fill (say) $D1$’s dead space with a new span $D9$, Carbink must update parity fragments $P1$ and $P2$. For a Reed-Solomon code, those parity fragments will depend on both $D1$ and $D9$.

There are two strawman approaches to update $P1$ and $P2$:

- The compute node can read $D1$ into local memory, generate the parity information, and then issue writes to $P1$ and $P2$.
- Alternatively, the compute node can send $D9$ to memory node $M1$, and request that $M1$ compute the new parity data and update $P1$ and $P2$.

The second approach requires a protocol like 2PC to guarantee the consistency of data fragments and parity fragments; without such a protocol, if $M1$ fails after updating $P1$, but before updating $P2$, the parity information will be out-of-sync with the data fragments.

The first approach, in which the compute node orchestrates the parity update, avoids the inconsistency challenges of the second approach. If a memory node dies in the midst of a parity update, the compute node will detect the failure, pick a new memory node to back the parity fragment, and retry the parity update. If the compute node dies in the midst of the parity update, then the memory manager will simply deallocate all regions belonging to the compute node (§3.1).

Unfortunately, both approaches require a lot of network bandwidth to fill holes in far memory. To reclaim one vacant span, the first approach requires three span-sized transfers—the compute node must read $D1$ and then write $P1$ and $P2$. The second approach requires two span-sized transfers to update $P1$ and $P2$. To reduce these network overheads, Carbink performs *remote compaction*, as described in the next section.

3.4.2 EC-Batch: Remote Compaction

Carbink employs *remote compaction* to defragment far memory using fewer network resources than the two strawmen above. On a compute node, Carbink executes several *compaction threads*. These threads look for “matched” spanset pairs; in each pair, the span positions containing dead space in one set are occupied in the other set, and vice versa. For example, the two spansets in Figure 4b are a matched pair. Once the compaction threads find a matched pair, they create a new spanset whose data consists of the live spans in the matched pair (e.g., $< D3, D4, D5, D6 >$ in Figure 4b). The compaction threads recompute and update the parity fragments $P1'$ and $P2'$ using techniques that we discuss in the next paragraph. Finally, the compaction threads deallocate the dead spaces in the matched pair (e.g., $< D1, D2, D7, D9, P3, P4 >$ in Figure 4b), resulting in a situation like the one shown in Figure 4c. Carbink’s compaction can occur in the background, unlike the synchronous parity updates of EC-2PC which place consensus activity on the critical path of swap-outs.

So, how should compaction threads update parity information? Carbink uses Reed-Solomon codes over the Galois field $GF(2^8)$. The new parity data to compute in Figure 4c is therefore represented by the following equations on $GF(2^8)$:

$$P1' - P1 = A_{1,1}(D5 - D1) + A_{2,1}(D6 - D2)$$
$$P2' - P2 = A_{1,2}(D5 - D1) + A_{2,2}(D6 - D2)$$

where $A_{i,j}$ ($i \in \{0,1,2,3\}, j \in \{0,1\}$) are fixed coefficient vectors in the Reed-Solomon code. Carbink provides two approaches for updating the parity information.

- In EC-Batch Local, the compute node that triggered the swap-out orchestrates the updating of parity data. In the running example, the compute node asks $M1$ to calculate the span delta $D5 - D1$, and asks $M2$ to calculate the span delta $D6 - D2$. After retrieving those updates, the compute node determines the parity deltas (i.e., $P1' - P1$ and $P2' - P2$) and pushes those deltas to the parity nodes $M5$ and $M6$.
- In EC-Batch Remote, the compute node offloads parity recalculation and updating to memory nodes. In the running example, the compute node asks $M1$ to calculate the span delta $D5 - D1$, and $M2$ to calculate the span delta $D6 - D2$. The compute node also asks $M1$ and $M2$ to calculate partial parity updates (e.g., $A_{1,1}(D5 - D1)$ and $A_{1,2}(D5 - D1)$ on $M1$). $M1$ and $M2$ are then responsible for sending the partial parity updates to the parity nodes. For example, $M1$ sends $A_{1,1}(D5 - D1)$ to $M5$, and $A_{1,2}(D5 - D1)$ to $M6$.

In EC-Batch Local, recovery from memory node failure is orchestrated by the compute node in a straightforward way, as in EC-Split (§3.4). In EC-Batch Remote, a compute node performs remote compaction by offloading parity updates to memory nodes. The compute node ensures fault tolerance for an individual compaction via 2PC. However, the compute node aggressively issues compaction requests in parallel. Two compactions (i.e., two instance of the 2PC protocol) are safe to concurrently execute if the compactions involve different spansets; the prepare and commit phases of the two compactions can partially or fully overlap.

On a compute node, Carbink's runtime can monitor the CPU load and network utilization of remote memory nodes. The runtime can default to remote compaction via EC-Batch Local, but opportunistically switch to EC-Batch Remote if spare resources emerge on memory nodes. During a switch to a different compaction mode, Carbink allows all in-flight compactions to complete before issuing new compactions that use the new compaction mode.

The strawmen defragmentation schemes in Section 3.4.1 require two or three span-sized network transfers to recover one dead span. In the context of Figure 4, EC-Batch Local recovers four dead spans using four span-sized network transfers. EC-Batch Remote requires four span-sized network transfers (plus some small messages generated by the consistency protocol) to recover four dead spans.

3.5 Failure Recovery

Carbink handles two kinds of memory node failures: planned and unplanned. Planned failures are scheduled by the cluster manager [51, 52] to allow for software updates, disk reformatting, and so on. Unplanned failures happen unexpectedly, and are caused by phenomena like kernel panics, defective hardware, and power disruptions.

Planned failures: When the cluster manager decides to schedule a planned failure, the manager sends a warning notification to the affected memory nodes. When a memory node receives such a warning, the memory node informs the memory manager. In turn, the memory manager notifies any compute nodes that have allocated regions belonging to the soon-to-be-offline memory node. Those compute nodes stop swapping-out to the memory node, but may continue to swap-in from the node as long as the node is still alive. Meanwhile, the memory manager orchestrates the migration of regions from the soon-to-be-offline memory node to other memory nodes. When a particular region's migration has completed, the memory manager informs the relevant compute node, who then updates the local mapping from Region ID to backing memory node. At some point during this process, the memory manager may also request non-failing memory nodes to contribute additional regions to the global pool of far memory.

Unplanned Failures: On a compute node, the Carbink runtime is responsible for detecting the unplanned failure of a memory node. The runtime does so via connection timeouts or more sophisticated leasing protocols [15, 16]. Upon detecting an unplanned failure, the runtime spawns background threads to reconstruct the affected spans using erasure coding. The runtime is also responsible for allowing application threads to read spans whose recovery is in-flight.

Span reconstruction: To reconstruct the spans belonging to a failed memory node M_{fail}, a compute node first requests a new region from the memory manager. Suppose that the new region is provided by memory node M_{new}. The compute node iterates through each lost spanset associated with M_{fail}; for each spanset, the compute node tells M_{new} which external spans and parity fragments to read in order to erasure-code-restore M_{fail}'s data. As the relevant spans are restored, a compute node can still swap in and remotely compact those spans. However, the swap-in and remote compaction activity will have to synchronize with recovery activity (§3.6).

In EC-Batch Local, when a compute node detects a memory node failure, the compute node cancels all in-flight compactions involving that node. A compute node using EC-Batch Remote does the same; however, for each canceled compaction, the compute node must also instruct the surviving memory nodes in the 2PC group to cancel the transaction.

The data and parity for a swapped-out spanset reside on multiple memory nodes. As a compute node recovers from the failure of one of the nodes in that group, another node in the group may fail. As long as the number of failed nodes does not exceed the number of parity nodes, Carbink can recover the spanset. The reason is that all of the information needed to recover is stored on a compute node, e.g., in the far page heap (§3.3). Due to space limitations, we omit a detailed explanation of how Carbink deals with concurrent failures.

Degraded reads: During the reconstruction of an affected span, application threads may try to swap in the span. The runtime handles such a fetch using a *degraded read* protocol. For example, consider Figure 4a. Suppose that $M1$ fails unexpectedly, and while the Carbink runtime is recovering $M1$'s spans ($D1$ and $D5$), an application thread tries to read an object residing in $D1$. The runtime will swap in data spans $D2$, $D3$, and $D4$, as well as parity fragment $P1$, and then reconstruct $D1$ via erasure coding. Degraded reads ensure that the failure of a memory node merely slows down an application instead of blocking it. In Section 5.3, we show that application performance only drops for 0.6 seconds, and only suffers a throughput degradation of 36% during that time.

Network bandwidth consumption: During failure recovery, Carbink consumes the same amount of network bandwidth as Hydra. For example, suppose that both Hydra and Carbink use RS4.2 encoding and have 4 spans, with a span stored on each of 4 memory nodes. In Hydra, a single node failure will lose four 1/4th spans. Reconstructing each 1/4th span will require the reading of four 1/4th span/parity regions from the surviving nodes, resulting in an aggregate network bandwidth requirement of 1 full span. So, reconstructing four 1/4th spans will require an aggregate network bandwidth of 4 full spans. In Carbink, the failure of a single memory node results in the loss of 1 full span. To recover that span, Carbink (like Hydra) must read 4 span/parity regions.

3.6 Thread Synchronization

On a compute node, the main kinds of Carbink threads are applications threads (which read objects, write objects, and swap in spans), filtering threads (which move objects within local spans), and eviction threads (which reclaim space by swapping local spansets to far memory). At any given time, a span may be in one of two concurrency regimes (§3.3): the span is either accessible to application threads and filtering threads, or to application threads and eviction threads. In both regimes, Carbink has to synchronize how the relevant threads update Carbink's smart pointers (§3.2).

At a high level, Carbink uses an RCU locking scheme that is somewhat reminiscent of AIFM's approach [44]. Due to space restrictions, we merely sketch the design. Carbink optimizes for the common case in which a span is only being accessed by an application thread. In this common case, an application thread grabs an RCU read lock on the pointer via the pointer's `Deref()` method, as shown in Listing 1. The thread sees that either (1) the **P**resent bit is not set, in which case the Carbink runtime issues an RMA read to swap in the appropriate span; (2) alternatively, the thread sees that the **P**resent bit is set, but the **M** and **E** bits are unset. In the second case, `Deref()` can just return a normal pointer back to the application. The application can be confident that concurrent filtering or evicting threads will not move or evict the object, because those threads cannot touch the object until application-level threads have released their RCU read locks via the `DerefScope` destructor (Listing 1).

The more complicated scenarios arise when the **P**resent bit is set and either the **M** or **E** bit are set as well. In this case, the (say) **M** bit has been set because the filtering thread set the bit and then called `SyncRCU()` (i.e., the RCU write waiting lock). The concurrent application thread and filtering thread essentially race to acquire the pointer's spinlock; if the application thread (i.e., `Deref()`) wins, it makes a copy of the object, clears **M**, releases the spinlock, and returns the address of the object copy to the application. Otherwise, if the filtering thread wins, it moves the object, clears **M**, and releases the spinlock. The losing thread has to retry the desired action. An analogous situation occurs if the **E** bit is set.

Carbink's eviction and remote compaction threads directly poll the network stack to learn about RMA completions and RPC completions. An application thread which has issued an RMA swap-in operation will yield, but a dedicated RMA poller thread detects when application RMAs have completed and awakens the relevant application threads. Polling avoids the overheads of context switching to new threads and notifying old threads that network events have occurred.

During recovery (§3.5), Carbink spawns additional threads to orchestrate the reconstruction of spans. Those threads acquire per-spanset mutexes which are also acquired by threads performing swap-ins, swap-outs, and remote compactions.

4 Implementation

Our Carbink prototype contains 14.3K lines of C++. It runs atop unmodified OSes, using standard POSIX abstractions for kernel-visible threads and synchronization. The runtime leverages the PonyExpress user-space network stack [35]. On a compute node, all threads in a particular application (both application-defined threads and Carbink-defined threads) execute in the same process. On a memory node, a Carbink daemon exposes far memory via RMAs or RPCs. We use Intel ISA-L v2.30.0 [25] for Reed-Solomon erasure coding.

Our current prototype has a simplified memory manager that is unreplicated, does not handle planned failures, and statically assigns memory nodes to compute nodes. Implementing the full version of the memory manager will be conceptually straightforward, since we can use off-the-shelf libraries for replicated state machines [1, 45] and cluster management [51, 52]. We also note that the experiments in §5 are insensitive to the performance of the memory manager, regardless of whether the manager is replicated or not. The reason is that memory allocations and deallocations (which must be routed through the memory manager) are rare and are not on the critical path of steady-state compute node operations like swap-in and swap-out.

To better understand the performance overheads of Carbink's erasure-coding approach, we built an AIFM-like [44] far memory system. That system uses remotable pointers like Carbink, but swaps in and out at the granularity of objects, and provides no fault tolerance. Like Carbink, it leverages the PonyExpress [35] user-space network stack. Our AIFM clone is 5.8K lines of C++.

5 Evaluation

In this section, we answer the following questions:

1. What is the latency, throughput, and remote memory usage of EC-Batch compared with the other fault tolerance schemes (§5.1 and §5.2)?
2. How does an unplanned memory node failure impact the performance of Carbink applications (§5.3)?
3. How does the performance of Carbink's span-based memory organization compare to the performance of an AIFM-like object-level approach (§5.4)?

Testbed setup: We deployed eight machines in the same rack, including one compute node and seven memory nodes; one of the memory nodes was used for failover. Each machine was equipped with dual-socket 2.2 GHz Intel Broadwell processors and a 50 Gbps NIC.

Fault tolerance schemes: Using the Carbink runtime, we compared our proposed EC-Batch schemes to four ap-

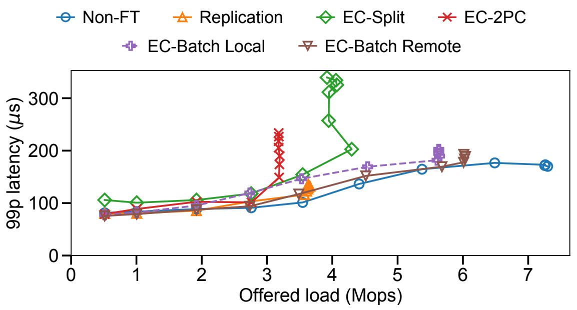


Figure 5: Microbenchmark load-latency curves.

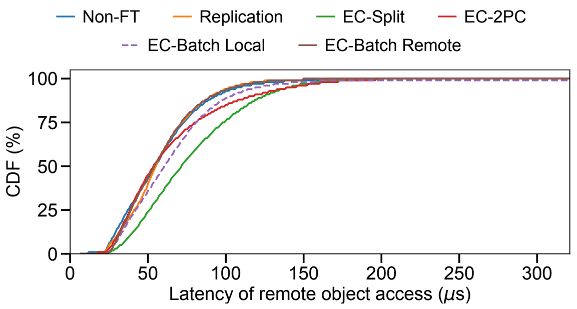


Figure 6: Latency distribution of remote object accesses in the microbenchmark under an offered load of 2 Mops.

proaches: Non-FT (a non-fault-tolerant scheme that used RMA to swap spans), Replication (which replicated spans on multiple nodes), EC-Split (the approach used by Hydra [29]), and EC-2PC (Table 1). We configured all fault tolerance schemes to tolerate up to two memory node failures. So, the Replication scheme replicated each swapped-out span on three memory nodes, whereas the EC schemes used six memory nodes—four held data, and two held RS4.2 parity bits [43]. EC-Batch spawned two compaction threads by default.

As mentioned in Section 4, we also built an AIFM-like far memory system. This system did not provide fault tolerance, but it provided a useful comparison with our Non-FT Carbink version.

Carbink borrows the span sizes that are used by TCMalloc (§3.3). These parameters have been empirically observed to reduce internal fragmentation. In our evaluation, EC-Batch (both Local and Remote) grouped four equal-size spans into a spanset, swapping out at the granularity of a spanset. Increasing spanset sizes would allow Carbink to issue larger batched RMAs, improving network efficiency. However, spansets whose evictions are in progress must be locked in local memory while RMAs complete; thus, larger spanset sizes would delay the reclamation of larger portions of local memory.

5.1 Microbenchmarks

To get a preliminary idea of Carbink's performance, we created a synthetic benchmark that wrote 15 million 1 KB objects (totalling 15 GB) to a remotable array. The compute node's local memory had space to store 7.5 GB of objects (i.e., half of the total set). By default, the compute node spawned 128 threads on 32 logical cores to access objects; the access pattern had a Zipfian-distributed [41] skew of 0.99. Such skews are common in real workloads for key/value stores [7].

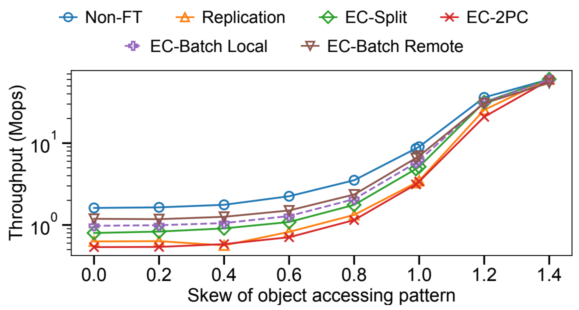


Figure 7: Impact of skew on throughput.

Object access throughput and tail latency: Figure 5 shows the 99th-percentile latency with various object access loads. All of the fault-tolerant schemes eventually hit a "hockey stick" in tail latency growth when the schemes could no longer catch up with the offered load. EC-Batch Remote had the highest sustained throughput (6.0 Mops), which was 40% higher than the throughput of the state-of-the-art EC-Split (4.3 Mops). EC-Batch Local achieved 5.6 Mops, which was 30% higher than EC-Split. EC-Split had worse performance because it had to issue four RMA requests to swap in one span; thus, EC-Split quickly became bottlenecked by network IO. In contrast, EC-Batch only issued one RMA request per swap-in.

EC-Batch Remote had 18%-29% lower tail latency than EC-Split under the same load (before reaching the "hockey-stick"). The reason was that EC-Split's larger number of RMAs per swap-in left EC-Split more vulnerable to stragglers [29]. Also recall that EC-Batch can support computation offloading [3, 27, 44, 57], which is hard with EC-Split (§3.4).

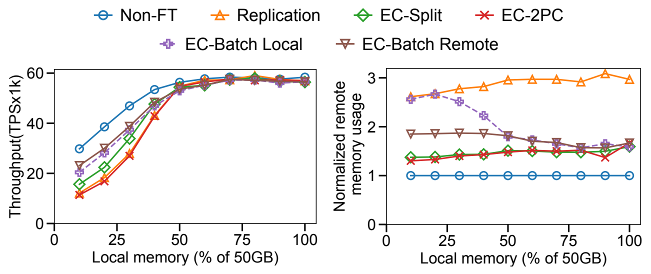
EC-2PC had the worst throughput because it relied on costly RPCs and 2PC protocols to swap out spans. Thus, EC-2PC could not reclaim local memory as fast as other schemes. The Replication scheme was bottlenecked by network bandwidth, since every swap-out incurred a 3× network write penalty; in contrast, EC-based schemes used RS4.2 erasure coding to reduce the write penalty to 1.5×.

Latency distribution of remote object accesses: Figure 6 shows the latency of accessing remote objects under 2 Mops of offered load. With this low offered load, Replication and EC-Batch Remote achieved similar access latencies as Non-FT because none of the schemes were bottlenecked by network bandwidth. EC-Batch Local had slightly higher remote access latencies. However, EC-Split had significantly higher access latencies (e.g., at the median and tail) than EC-Batch Local and Remote; the reason was that EC-Split issued four times as many network IOs and thus was more sensitive to stragglers. EC-2PC's tail latency was slightly higher than that of EC-Batch Local and Remote due to the overhead of costly RPCs and 2PC traffic.

Impact of skewness: Figure 7 shows how the skewness of object accesses impacted throughput. EC-Batch Remote and

	# Compaction threads	Norm. remote mem usage	Avg. # remote logical cores	Avg. BW (Gbps)
EC-Batch Local	1	2.54	0.23	1.27
	2	2.35	0.53	1.64
	3	2.28	0.56	1.76
EC-Batch Remote	1	1.89	1.97	2.98
	2	1.83	2.10	3.15
	3	1.74	2.27	3.40
W/o compaction	0	3.03	–	–

Table 2: Remote resource usage in the microbenchmark. The remote memory usage is normalized with respect to the usage of Non-FT. The number of remote logical cores and the network bandwidth are averaged across all six memory nodes.



(a) Transaction throughput. **(b)** Remote memory usage.

Figure 8: Transactional KV-store evaluation.

Local performed best due to their more efficient swapping approaches. However, the throughput of all schemes increased with higher skewness. The reason is that high skewness led to a smaller working set and thus a higher likelihood that hot objects were locally resident. In these scenarios, schemes with faster swapping were not rewarded as much.

Remote resource usage with compaction: Table 2 shows the impact of compaction on the average memory, CPU, and bandwidth usage per memory node. Without compaction, EC-Batch used 3.03× remote memory (normalized with respect to Non-FT memory consumption). With two local compaction threads, EC-Batch Remote's memory overhead reduced to 1.83×. The memory reduction was at the expense of 2.1 cores and 3.15 Gbps bandwidth on each memory node. With more compaction threads, Carbink could further reduce memory usage at the cost of higher CPU and bandwidth utilization. That being said, we note that the synthetic microbenchmark application represented an extreme case of remote CPU and network usage, since the workload accessed objects without actually computing on them.

EC-Batch Remote vs. Local: EC-Batch Remote had higher throughput and lower tail latency than EC-Batch Local (Figure 5). This was because EC-Batch Local's compaction required (1) local CPUs for parity computation and (2) network bandwidth for transferring span deltas and parity updates, leaving fewer local resources for application threads and RMA reads. Because of EC-Batch Remote's faster compaction, EC-Batch Remote also used 28%-34% less remote memory than EC-Batch Local (Table 2). However, EC-Batch Remote consumed more remote CPUs (2.10 vs. 0.53 cores) and more network bandwidth (3.15 vs. 1.64 Gbps) than Local. In practice, the Carbink runtime could transparently switch between EC-Batch Remote and Local based on an application developer's policy about resource/performance trade-offs.

5.2 Macrobenchmarks

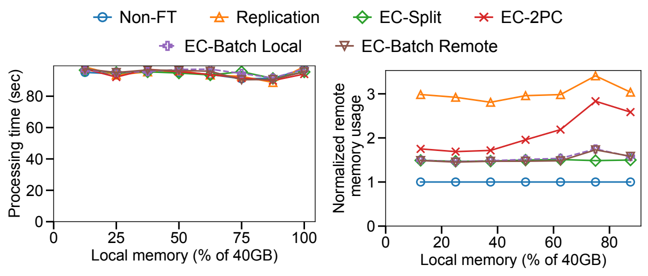
We evaluated Carbink using two memory-intensive applications that would benefit from remote memory: an in-memory transactional key-value store, and a graph processing algorithm. The two applications exhibited different patterns of object accesses, and had different working set behaviors.

Transactional KV-store: This application implemented a transactional in-memory B-tree, exposing it via a key/value interface similar to that of MongoDB [37]. Each remotable object was a 4 KB value stored in a B-tree leaf. The application spawned 128 threads, and each thread processed 20 K transactions. The compute node provisioned 32 logical cores, with the application overlapping execution of the threads for higher throughput [26, 38, 44, 56]. Each transaction contained three reads and three writes, similar to the TPC-A benchmark [53]. Each update created a new version of a particular key's value; asynchronously, the application trimmed old versions. The maximum working set size during the experiment was roughly 50 GB.

Throughput: Figure 8a shows the KV-store throughput when varying the size of local memory (normalized as a fraction of the maximum working set size). In scenarios with less than 50% local memory, EC-Batch Remote achieved higher transactions per second (TPS) than all other fault tolerance schemes. For example, TPS for EC-Batch Remote was 1.5%-48% higher than that of EC-Split; this was because EC-Batch only needed one RMA request to swap in a span. EC-Batch Remote was at most 29% slower than Non-FT, mainly due to the additional parity update required for fault tolerance. EC-Batch Local was at most 13% slower than EC-Batch Remote. EC-2PC performed the worst among EC schemes.

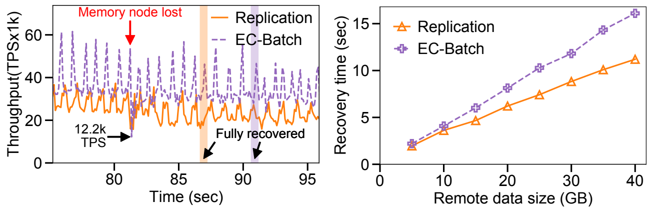
All schemes achieved similar throughput when the local memory size was above 50%. The reason was that the *average* working set size of the workload was only half the size of the *maximum* memory usage. The maximum memory usage only occurred when the B-Tree had fallen very behind in culling old versions of objects.

Remote memory usage: Figure 8b plots remote memory usage as a function of local memory sizes; remote memory usage is normalized with respect to that of Non-FT. Compared to EC-Split, EC-Batch Remote and Local used up to 35% and 93% more remote memory, respectively. EC-Batch schemes defragmented remote memory using compaction, but when local memory space was less than 50%, remote compaction could not immediately defragment the spanset holes created by frequent span swap-ins. As local memory grew larger, span fetching became less frequent, making it



(a) Processing time. **(b)** Remote memory usage.

Figure 9: Graph processing evaluation.



(a) KV-store TPS over time. **(b)** Microbenchmark recovery.

Figure 10: Failure recovery evaluation.

easier for remote compaction to reclaim space. In this less hectic environment, EC-Batch's remote memory usage was similar to that of the other erasure-coding schemes.[3]

Graph processing: We implemented a connected-components algorithm [50] that found all sets of linked vertices in a graph. This kind of algorithm is critical to various Google services. We evaluated the algorithm using the Friendster graph [30] which contained 65 million vertexes and 1.8 billion edges. In the graph analysis code, each vertex's adjacency list was referenced via remotable pointers. The total size of the objects stored in Carbink was roughly 40 GB. The application used 80 application threads that ran atop 80 logical cores. In our experimental results, the reported processing times exclude graph loading, since graph loading is dominated by disk latencies.

Figure 9a shows that all schemes had similar processing times as Non-FT, regardless of the local memory size. The reason was that the graph application had a high compute-to-network ratio—the application fetched all neighbors associated with each vertex and then spent non-trivial time enumerating each neighbor and computing on them. As a result of this good spatial locality and high "think time," the graph application did not incur frequent data swapping, and thus avoided fault tolerance overhead that the KV-store could not.

Figure 9b shows that EC-Batch Local and Remote had similar remote memory usage as EC-Split: 15%-39% lower than EC-2PC and roughly 50% lower than Replication. All EC-based schemes had lower remote memory overheads than Replication because the erasure coding only incurred a 1.5× space overhead for the extra parity data.

EC-2PC used more memory than EC-Batch because the graph workload randomly fetched diverse-sized spans. The random fetch sizes reflected the fact that different vertices had different sizes for their adjacency lists. This lack of span size locality hindered dead space reclamation, since EC-2PC had to wait longer for all of the spans in an erasure-coding group to be swapped in. EC-Batch avoided this problem by bundling equal-sized spans into the same spanset and using remote compaction.

[3]The remote memory usage of triple-replication was slightly less than 3× the usage of Non-FT because Non-FT could swap out memory faster during periods of high local memory pressure.

5.3 Failure Recovery

We measured the recovery time for an unplanned memory node failure in the KV-store, the graph processor, and the microbenchmark application. For the graph application, all schemes achieved similar processing time during unplanned failures; thus, in the text below, we focus on the KV-store and the microbenchmark.

Transactional KV-store: Figure 10a shows the KV-store throughput of Replication and EC-Batch Local, with a data point collected every 100 ms before and after an unplanned memory node failure. Upon detecting the failure, EC-Batch Local immediately reconstructed the lost data on a pre-configured failover memory node. We gave the KV-store 15 GB of local memory, equivalent to 30% of the 50 GB maximum working set size.

The throughput of both schemes fluctuated sinusoidally because the KV-store frequently tried to swap in remote objects, but the swap-ins sometimes had to synchronously block until eviction threads could reclaim enough local memory. After a memory node failed, EC-Batch needed 0.6 seconds to restore normal throughput, while replication needed 0.3 seconds. This is because, during failure recovery, an EC-Batch read that targeted an affected span used the degraded read protocol which uses more bandwidth than a normal read (§3.5); in contrast, a Replication read that targeted an affected span consumed the same amount of bandwidth as a read during non-failure-recovery. During recovery, the throughput of Replication and EC-Batch dropped an average of 35% and 36% respectively.

EC-Batch required 9.7 seconds to fully regenerate the lost data on the failover node, taking 1.7× longer than Replication. This difference arose because, in EC-Batch, the new memory node read 4× span/parity information involving the lost data and computed erasure codes to reconstruct the lost data. In contrast, Replication lost more data per memory node, but only read one copy of the lost data. Note that with EC-Batch, degraded reads mostly happened during the first second of failure recovery; the skewed workload meant that a small number of objects were the targets of most reads, and once a hot object was pulled into local memory (perhaps by a degraded read), the object would not generate additional degraded reads.

Microbenchmark: Figure 10b shows recovery times as a function of the remote data size. The recovery time of EC-Batch increased almost linearly with the remote data size,

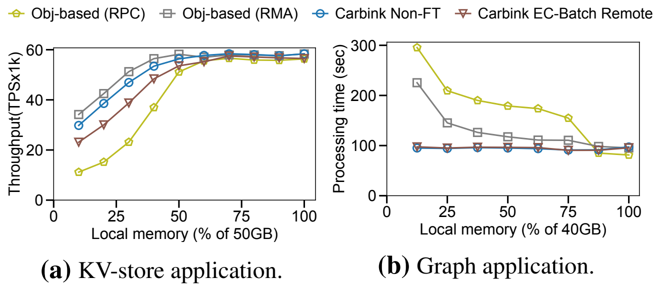


Figure 11: Application performance: AIFM-like object-based systems and Carbink.

with 0.6 GB/s recovery speed. This speed was 12%-44% slower than Replication due to the larger amount of recovery information that EC-Batch had to transfer around the network, and the computational overhead of generating erasure codes.

Prior work [10, 29, 58] also found that, during recovery, erasure-coding schemes had longer recovery times and worse performance degradation than replication schemes. However, this drawback only happens for unplanned failures which, in our production environment, are rare compared to planned failures; in an erasure-coding scheme, handling a *planned* failure just requires simple copying of the information on a departing memory node, and does not incur additional work to find parity information or recompute erasure coding. Thus, in our deployment setting where unplanned failures are rare, erasure-coding schemes (which have lower memory utilization than replication schemes) are very attractive.

5.4 Comparison with AIFM-like Systems

We compared span-based swapping in Carbink with the object-based approach used in AIFM [44]. We implemented two AIFM-like systems using our threading and network stack (§4). The first system used RPCs to swap individual objects, with the remote memory nodes tracking the object-to-remote-location mapping (as done in AIFM). Our second object-granularity swapping system used more-efficient RMAs to swap objects, and had compute nodes track the mapping between objects and their remote locations; recall that RMA is one-sided, so compute nodes could not rely on memory nodes to synchronously update mappings during swaps. Like the original AIFM, neither system provided fault tolerance.

Transactional KV-store: Figure 11a shows that, if local memory was too small to hold the average working set, Non-FT Carbink had 45%-167% higher throughput than the AIFM-like system with RPC. The reason is that, when local memory pressure was high, more swapping occurred, and the better efficiency of RMAs over RPCs became important. However, Non-FT Carbink achieved 5.6%-15% lower throughput than the object-based system with RMA. This was due to swap-in amplification. For example, Non-FT Carbink might swap in an 8KB span but only use one 4KB object in the span; this never happens in a system that swaps at an object granularity.

Graph processing: Figure 11b shows the graph application's processing time. When the local memory size was below 87.5%, Carbink performed 18%-58% faster than the object-based system with RMA. This is because, in the graph workload, 4% of large objects occupied 50% of the overall data set. Carbink prioritized swapping out large cold objects (§3.3), keeping most small objects in local memory and reducing the miss rate for those objects. In contrast, the object-based systems did not consider object sizes when swapping, leading to an increased miss rate for small objects. Note that, with larger local memories, all schemes had similar performance; indeed, when all objects fit into local memory, the object-based system with RPC slightly outperformed the rest because it did not require a dedicated core to poll for RMA completions.

6 Discussion

EC-Batch for paging-based systems: Carbink uses EC-Batch to transparently expose far memory via remotable pointers. However, EC-Batch can also be used to expose far memory via OS paging mechanisms [5, 22, 46]. In a traditional paging-based approach for far memory, a compute node swaps in and out at the granularity of a page. However, a compute node can use EC-Batch to treat each page as a span, such that pages are swapped out at the "pageset" granularity, and pages are swapped in at the page granularity.

Custom one-sided operations: EC-Batch requires memory nodes to calculate span deltas and parity updates (§3.4.2). In our Carbink prototype, memory nodes use separate threads to execute these calculations. However, memory nodes could instead implement them as custom one-sided operations in the network stack, such that the network stack itself performs the calculations, avoiding the need to context-switch to external threads. This approach has been used in prior work [6, 9, 35, 47, 48] to avoid thread scheduling overheads.

Designing the memory manager: We used a centralized manager because such a manager (1) simplified our overall design, and (2) made it easier to drive memory utilization high (because a centralized manager will have a global, accurate view of memory allocation metadata). A similarly-centralized memory manager is used by the distributed transaction system FaRM [16]. If the centralized manager became unavailable, Carbink could fall back to a decentralized memory allocation scheme like the one used by Hydra [29] or INFINISWAP [22].

The state maintained by the memory manager is not large. With 1 GB regions, we expect up to 500 regions in a typical memory node (similar to FaRM [16]). With thousands of memory nodes, the memory manager just needs to store a few MBs of state for region assignments.

Fault tolerance for compute nodes: In Carbink, a compute node does not share memory with other compute nodes. Thus, a Carbink application can checkpoint its own state without fear of racing with other compute nodes that modify the state being checkpointed. Checkpoint data could be placed in a

	Fast s/o	Low mem	Fast s/i	Interface	Coding granularity
On-disk rpl.	✗	✓	✓	Various	–
In-memory rpl.	✓	✗	✓	Various	–
Hydra [29]	✓	✓	✗	Paging	Split 4KB pages
Cocytus [10]	✓	✓	✗	KV-store	Across 4KB pages
BCStore [31]	✓	✓	✗	KV-store	Across objs
Hybrid [32]	✗	✗	✓	KV-store	Split 4KB pages
Carbink	✓	✓	✓	Remotable pointers	Across spans

Table 3: Comparison of existing fault-tolerant approaches for far memory. "Fast s/o" indicates whether a system can swap out at network/memory speeds. "Low mem" means that a system has relatively low memory pressure. "Fast s/i" refers to whether a system can swap in at network/memory speeds.

non-Carbink store, obviating the need to track how checkpointed spans move across Carbink memory nodes during compaction and invalidation. Alternatively, Carbink itself could store checkpoints, e.g., in the fault-tolerant address space of a well-known Carbink application whose sole purpose is to store checkpoints.

7 Related Work

Fault tolerance for far memory: Many far memory systems do not provide fault tolerance [2, 44, 55]. Of the systems that do, most replicate swapped-out data to local disks or remote ones [5, 22, 46]. Unfortunately, this approach forces application performance to bottleneck on disk bandwidth or disk IOPs during bursty workloads or failure recovery [29]. This behavior is unattractive, since a primary goal of a far memory system is to have applications run at *memory* speeds as much as possible.

Like Carbink, Hydra [29] is a far memory system that provides fault tolerance by writing erasure-coded local memory data to far RAM. Hydra uses the EC-Split coding approach that we describe in Section 3.4. As we demonstrate in Section 5, Carbink's erasure-coding scheme provides better application performance in exchange for somewhat higher memory consumption. Carbink's coding scheme also enables the offloading of computations to far memory nodes. Such offloading can significantly improve the performance of various applications [3, 27, 44, 57].

Fault tolerance for in-memory transactions and KV-stores: In-memory transaction systems typically provide fault tolerance by replicating data across the memory of multiple nodes [15, 16, 26]. These approaches suffer from the classic disadvantages of replication: double or triple storage overhead, and the associated increase in network traffic.

Recent in-memory KV-stores use erasure coding to provide fault tolerance. For example, Cocytus [10] and BCStore [31] only rely on in-memory replication to store small instances of metadata; object data is erasure-coded using a default page size of 4KB. Cocytus erasure-codes using a scheme that resembles EC-2PC (§3.4). To reduce the network utilization of a Cocytus-style approach, a BCStore compute node buffers outgoing writes; this approach allows the node to batch the computation of parity fragments (and thus issue fewer updates to remote data and parity regions). Batching reduces network overhead at the cost of increasing write latency.

Both Cocytus and BCStore rely on two-sided RPCs to manipulate far memory. RPCs incur software-level overheads involving thread scheduling and context switching on remote nodes. To avoid these costs, Carbink eschews RPCs for one-side RMA operations. Carbink also issues fewer parity updates than Cocytus; whereas Cocytus uses expensive 2PC to update parity information during every write, Carbink defers parity updates until compaction occurs on remote nodes (§3.4.2). Carbink's compaction approach is also more efficient than that of BCStore. BCStore's compaction algorithm performs actual copying of data objects on memory nodes, whereas Carbink compaction just manipulates span pointers inside of spanset metadata.

A far memory system could use both replication and erasure coding [32]. For example, during a Hydra-style swap-out, a span would be erasure-coded and the fragments written to memory nodes; however, a full replica of the span would also be written out. Relative to Carbink, this hybrid approach would have lower reconstruction costs (assuming that the full replica did not live on the failed node). However, Carbink would have lower memory overheads because no full replica of a span would be stored. Carbink would also have faster swap-outs, because swap-outs in the hybrid scheme would require an EC-2PC-like mechanism to ensure consistency.

Table 3 summarizes the strengths and weaknesses of the various systems discussed above.

Memory compaction: In Carbink, the far memory regions used by a program become fragmented as spans are swapped in. Memory compaction is a well-studied topic in the literature about "moving" garbage collectors for managed languages (e.g., [11, 18, 49]). Moving garbage collection is also possible for C/C++ programs; Mesh [40] represents the state-of-the-art. With respect to this prior work, Carbink's unique challenge is that the compaction algorithm (§3.4.2) must compose well with an erasure coding scheme that governs how objects move between local memory and far memory.

8 Conclusion

Carbink is a far memory system that provides low-latency, low-overhead fault tolerance. Carbink erasure-codes data using a span-centric approach that does not expose swap-in operations to stragglers. Whenever possible, Carbink uses efficient one-sided RMAs to exchange data between compute nodes and memory nodes. Carbink also uses novel compaction techniques to asynchronously defragment far memory. Compared to Hydra, a state-of-the-art fault-tolerant system for far memory, Carbink has 29% lower tail latency and 48% higher application performance, with at most 35% higher memory usage.

Acknowledgments

We thank our shepherd Luís Rodrigues and the anonymous reviewers for their insightful comments. We also thank Kim Keeton and Jeff Mogul for their comments on early drafts of the paper, and Maria Mickens for her comments on a later draft. Yang Zhou and Minlan Yu were supported in part by NSF CNS-1955422 and CNS-1955487.

References

[1] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. Sync HotStuff: Simple and Practical Synchronous State Machine Replication. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 106–118, 2020.

[2] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, and et al. Remote Regions: A Simple Abstraction for Remote Memory. In *Proceedings of USENIX ATC*, pages 775–787, 2018.

[3] Marcos K Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. Designing Far Memory Data Structures: Think Outside the Box. In *Proceedings of ACM HotOS*, pages 120–126, 2019.

[4] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. Write-Rationing Garbage Collection for Hybrid Memories. *ACM SIGPLAN Notices*, 53(4):62–77, 2018.

[5] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can Far Memory Improve Job Throughput? In *Proceedings of ACM EuroSys*, pages 1–16, 2020.

[6] Emmanuel Amaro, Zhihong Luo, Amy Ousterhout, Arvind Krishnamurthy, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Remote Memory Calls. In *Proceedings of ACM HotNets*, pages 38–44, 2020.

[7] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. *ACM SIGMETRICS Performance Evaluation Review*, 40(1):53–64, 2012.

[8] Cristina Băsescu and Bryan Ford. Immunizing Systems from Distant Failures by Limiting Lamport Exposure. In *Proceedings of ACM HotNets*, pages 199–205, 2021.

[9] Matthew Burke, Shannon Joyner, Adriana Szekeres, Jacob Nelson, Irene Zhang, and Dan R.K. Ports. PRISM: Rethinking the RDMA Interface for Distributed Systems. In *Proceedings of USENIX SOSP*, pages 228–242, 2021.

[10] Haibo Chen, Heng Zhang, Mingkai Dong, Zhaoguo Wang, Yubin Xia, Haibing Guan, and Binyu Zang. Efficient and Available In-Memory KV-Store with Hybrid Erasure Coding and Replication. *ACM Transactions on Storage (TOS)*, 13(3):1–30, 2017.

[11] Jon Coppeard. Compacting Garbage Collection in SpiderMonkey. https://hacks.mozilla.org/2015/07/compacting-garbage-collection-in-spidermonkey/, 2015.

[12] Fernando J. Corbato. A Paging Experiment with the Multics System. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC, 1968.

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

[14] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[15] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *Proceedings of USENIX NSDI*, pages 401–414, 2014.

[16] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of ACM SOSP*, pages 54–70, 2015.

[17] Jason Evans. A Scalable Concurrent malloc (3) Implementation for FreeBSD. In *Proceedings of BSDCan Conference*, 2006.

[18] Robert R. Fenichel and Jerome C. Yochelson. A LISP Garbage-Collector for Virtual-Memory Computer Systems. *Communications of the ACM*, 12(11):611–612, 1969.

[19] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of USENIX OSDI*, pages 17–30, 2012.

[20] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. Graphx: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of USENIX OSDI*, pages 599–613, 2014.

[21] Google. TCMalloc Open Source. https://github.com/google/tcmalloc.

[22] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient Memory Disaggregation with INFINISWAP. In *Proceedings of USENIX NSDI*, pages 649–667, 2017.

[23] Xianglong Huang, Stephen M Blackburn, Kathryn S. McKinley, J. Eliot B. Moss, Zhenlin Wang, and Perry Cheng. The Garbage Collection Advantage: Improving Program Locality. *ACM SIGPLAN Notices*, 39(10):69–80, 2004.

[24] Andrew Hamilton Hunter, Chris Kennelly, Paul Turner, Darryl Gove, Tipp Moseley, and Parthasarathy Ranganathan. Beyond Malloc Efficiency to Fleet Efficiency: A Hugepage-Aware Memory Allocator. In *Proceedings of USENIX OSDI*, pages 257–273, 2021.

[25] Intel. Intel Intelligent Storage Acceleration Library. https://github.com/intel/isa-l.

[26] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided RDMA Datagram RPCs. In *Proceedings of USENIX OSDI*, pages 185–201, 2016.

[27] Dario Korolija, Dimitrios Koutsoukos, Kimberly Keeton, Konstantin Taranov, Dejan Milojičić, and Gustavo Alonso. Farview: Disaggregated Memory with Operator Off-loading for Database Engines. In *Proceedings of Conference on Innovative Data Systems Research*, 2022.

[28] Jakub Łącki, Vahab Mirrokni, and Michał Włodarczyk. Connected Components at Scale via Local Contractions. *arXiv preprint arXiv:1807.10727*, 2018.

[29] Youngmoon Lee, Hasan Al Maruf, Mosharaf Chowdhury, Asaf Cidon, and Kang G. Shin. Mitigating the Performance-Efficiency Tradeoff in Resilient Memory Disaggregation. *arXiv preprint arXiv:1910.09727*, 2019.

[30] Jure Leskovec. Friendster Social Network Dataset. https://snap.stanford.edu/data/com-Friendster.html.

[31] Shenglong Li, Quanlu Zhang, Zhi Yang, and Yafei Dai. BCStore: Bandwidth-Efficient In-Memory KV-store with Batch Coding. *Proceedings of IEEE International Conference on Massive Storage Systems and Technology*, 2017.

[32] Yuzhe Li, Jiang Zhou, Weiping Wang, and Yong Chen. RE-Store: Reliable and Efficient KV-Store with Erasure Coding and Replication. In *Proceedings of IEEE International Conference on Cluster Computing*, pages 1–12, 2019.

[33] Yucheng Low, Joseph E. Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E. Guestrin, and Joseph Hellerstein. Graphlab: A New Framework for Parallel Machine Learning. *arXiv preprint arXiv:1408.2041*, 2014.

[34] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. Imbalance in the Cloud: An Analysis on Alibaba Cluster Trace. In *Proceedings of IEEE International Conference on Big Data*, pages 2884–2892, 2017.

[35] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, and et al. Snap: A Microkernel Approach to Host Networking. In *Proceedings of ACM SOSP*, pages 399–413, 2019.

[36] Paul E. McKenney and John D. Slingwine. Read-Copy Update: Using Execution History to Solve Concurrency Problems. In *Proceedings of Parallel and Distributed Computing and Systems*, pages 509–518, 1998.

[37] MongoDB Inc. MongoDB Open Source. https://github.com/mongodb/mongo.

[38] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-Sensitive Datacenter Workloads. In *Proceedings of USENIX NSDI*, pages 361–378, 2019.

[39] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, and et al. The RAMCloud Storage System. *ACM Transactions on Computer Systems (TOCS)*, 33(3):1–55, 2015.

[40] Bobby Powers, David Tench, Emery D. Berger, and Andrew McGregor. Mesh: Compacting Memory Management for C/C++ Applications. In *Proceedings of ACM PLDI*, pages 333–346, 2019.

[41] David M.W. Powers. Applications and Explanations of Zipf's Law. In *Proceedings of New Methods in Language Processing and Computational Natural Language Learning*, 1998.

[42] KV Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. EC-Cache: Load-Balanced, Low-Latency Cluster Caching with Online Erasure Coding. In *Proceedings of USENIX OSDI*, pages 401–417, 2016.

[43] Irving S. Reed and Gustave Solomon. Polynomial Codes over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[44] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-Performance, Application-Integrated Far Memory. In *Proceedings of USENIX OSDI*, pages 315–332, 2020.

[45] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[46] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *Proceedings of USENIX OSDI*, pages 69–87, 2018.

[47] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. StRoM: Smart Remote Memory. In *Proceedings of ACM EuroSys*, pages 1–16, 2020.

[48] Arjun Singhvi, Aditya Akella, Maggie Anderson, Rob Cauble, Harshad Deshmukh, Dan Gibson, Milo M.K. Martin, Amanda Strominger, Thomas F. Wenisch, and Amin Vahdat. CliqueMap: Productionizing an RMA-Based Distributed Caching System. In *Proceedings of ACM SIGCOMM*, pages 93–105, 2021.

[49] SUN Microystems. Memory Management in the Java HotSpot Virtual Machine, 2006.

[50] Michael Sutton, Tal Ben-Nun, and Amnon Barak. Optimizing Parallel Graph Connectivity Computation via Subgraph Sampling. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium*, pages 12–21, 2018.

[51] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, and et al. Twine: A Unified Cluster Management System for Shared Infrastructure. In *Proceedings of USENIX OSDI*, pages 787–803, 2020.

[52] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the Next Generation. In *Proceedings of ACM EuroSys*, pages 1–14, 2020.

[53] Transaction Processing Performance Council (TPC). TPC-A. `http://tpc.org/tpca/default5.asp`.

[54] Volt Active Data. VoltDB. `https://www.voltdb.com/`.

[55] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A Memory-Disaggregated Managed Runtime. In *Proceedings of USENIX OSDI*, pages 261–280, 2020.

[56] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing RDMA-Enabled Distributed Transactions: Hybrid Is Better! In *Proceedings of USENIX OSDI*, pages 233–251, 2018.

[57] Jie You, Jingfeng Wu, Xin Jin, and Mosharaf Chowdhury. Ship Compute or Ship Data? Why Not Both? In *Proceedings of USENIX NSDI*, pages 633–651, 2021.

[58] Zhe Zhang, Amey Deshpande, Xiaosong Ma, Eno Thereska, and Dushyanth Narayanan. Does Erasure Coding Have a Role to Play in My Data Center? *Microsoft Research Technical Report*, 2010.