UNIVERSITY OF CALIFORNIA SAN DIEGO

Building End-to-end Disaggregation Stack via Cross Layer Co-Design

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy

in

Computer Science

by

Zhiyuan Guo

Committee in charge:

Professor Yiying Zhang, Chair
Professor Alex C. Snoeren
Professor Geoffrey M. Voelker
Professor Hao Zhang

2025

The Dissertation of Zhiyuan Guo is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2025

EPIGRAPH

The problems are solved, not by giving new information,

but by arranging what we have known since long.

– Ludwig Wittgenstein, *"Philosophical Investigations"*

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ALGORITHMS

ACKNOWLEDGEMENTS

I feel very fortunate for having the support from many people in my life. This dissertation would not be possible without them.

I begin by expressing my deepest gratitude to my advisor, Professor Yiying Zhang. Yiying granted me the rare freedom to pursue research problems I genuinely believe in and enjoy, and she never hesitated to back ambitious ideas, especially our works many consider too uncertain or daunting. She taught me to attack hard problems the *right* way: with rigor, integrity, and a willingness to dig into the details until they yield. Most importantly, Yiying instilled a spirit of fearless exploration: because of her example I now view uncharted territory not as a warning sign but as an invitation. That mindset is the single greatest gift of my Ph.D., and it will guide my career long after this thesis rests, hardbound as she wished, on her shelf.

I thank my committee members, Professor Alex C. Snoeren, Professor Geoffrey M. Voelker, and Professor Hao Zhang, for their thoughtful questions and constructive guidance. During my internship at Google I was fortunate to learn from Kimberly Keeton, Kan Wu, Suli Yang, and Stanko Novakovic. Their mentorship and technical advice greatly enriched chapters of this dissertation.

A special thank goes to my partner, Jiaxin Lin. I am endlessly grateful for her love, her intellect, and her steady companionship. She is simultaneously my close collaborator, sounding board, and constant and unconditional source of encouragement. Her belief in me never faltered, even in moments when I doubted myself. Having her with me, during this journey and beyond, has been my greatest stroke of luck in life. This work, in many ways, carries her fingerprints too.

I am grateful to Yizhou Shan for introducing me to systems research; to Yutong Huang and Xuhao Luo for their invaluable contributions to the *Clio* project; to Zjian He for his pivotal work on *Mira* and many illuminating discussions; to Zachary Blanco, Mohammad Shahrad, Junda Chen, Zerui Wei, Bili Dong, Jinmou Li, Ishaan Pota, and Harry Xu for pushing the *Scad* project forward; and to Arvind Krishnamurthy for his guidance on the *SuperNIC* project. I also thank William Lin, Ryan Kosta, Vikranth Srivatsa and Reyna Abhyankar for sparkling

discussions and shouldering day-to-day responsibilities and fostering a lively, candid atmosphere that I begin to love to stay in. I thank my UC San Diego peers, Zachary Blanco, Stewart Grant, and Anil Yelam, for many spirited hallway conversations that sharpened my thinking.

Finally, I owe everything to my parents, Jianzhu Guo and Yan He. Though an ocean has separated us for the entire of my doctoral studies, their unconditional love and steadfast support gave me the confidence to follow this path.

To everyone who offered advice, criticism, or a friendly cup of coffee. Thank you.

Chapter 3, in part, is a reprint of the material as it appears in the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, Yiying Zhang. The dissertation author was the primary investigator and author of this paper.

Chapter 2, in part, is a reprint of the materials as it apears in 29th Symposium on Operating Systems Principles. Zhiyuan Guo, Zijian He, Yiying Zhang. The dissertation author was the primary investigator and author of this paper.

Chapter 4, in part, is currently being prepared for submission for publication of material. Zhiyuan Guo, Yiying Zhang, Arvind Krishnamurthy. The dissertation author was the primary investigator and author of this paper.

VITA

2019	Bachelor of Engineering, Beihang University
2022	Master of Science, University of California, San Diego
2025	Doctor of Philosophy, University of California, San Diego

PUBLICATIONS

"PageFlex: Flexible and Efficient User-space Delegation of Linux Paging Policies with eBPF" USENIX Annual Technical Conference (ATC '25), 2025

"Portable and High-Performance SmartNIC Programs with Alkali" Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI '25), 2025

"Zenix: Efficient Execution of Bulky Serverless Applications" arXiv preprint, 2024

"Mira: A Program-Behavior-Guided Far Memory System" Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP '23), October 2023

"Towards a Fully Disaggregated and Programmable Data Center" Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '22), August 2022

"Clio: A Hardware–Software Co-Designed Disaggregated Memory System" Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), March 2022

"Accelerating End-to-End Deep Learning Workflows with Codesign of Data Preprocessing and Scheduling" *IEEE Transactions on Parallel and Distributed Systems*, Special Section on Parallel and Distributed Computing Techniques for AI, ML, and DL, December 2021

"Direct Universal Access: Making Data Center Resources Available to FPGA" Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19), February 2019

"DLBooster: Boosting End-to-End Deep Learning Workflow with Offloading Data Preprocessing Pipelines" Proceedings of the 48th International Conference on Parallel Processing (ICPP '19), August 2019

ABSTRACT OF THE DISSERTATION

Building End-to-end Disaggregation Stack via Cross Layer Co-Design

by

Zhiyuan Guo

Doctor of Philosophy in Computer Science

University of California San Diego, 2025

Professor Yiying Zhang, Chair

Modern datacenter applications' increasing resource usage amount and diverse resource utilization patterns challenge the datacenter systems' design on scalability and resource efficiency: workloads demand over 10 $\times$ more and 110 $\times$ more diverse of mixes of CPU and GPU computation power, DRAM and SSD capacity, and network bandwidth. As a result, server-centric scaling strands up to 55% of memory and 43% of computation capacity. Hardware resource disaggregation emerges as a solution. High-speed fabrics such as 400 GbE and CXL 3.0 now blur the boundary between local and remote devices, allowing them to be accessed by application transparently and effectively decouple hardware into shared pools. Unfortunately, naïve "plug-and-play" disaggregation shows immitigable performance overhead and struggles to

make it way into real deployment.

This dissertation defends the thesis that resource disaggregation becomes practical and performant only when *resource semantics*and *application intent* cross traditional layer boundaries, enabling joint optimization across the application, runtime, operating system, networking, and hardware. We show that the dominant overheads stem from semantic mismatches across the layers, and that aligning those semantics through co-design unlocks efficiency with reducing 94% performance overhead.

The claim is validated through three end-to-end systems:

Clio co-designs a stateless, connection-less transport with a hash-based hardware page table, sustaining 100 Gbps and 2.5 µs median load-to-use latency while cutting energy by up to 3.4× versus CPU-centric far-memory solutions.

Mira uses static analysis and runtime profiling to classify objects, tailor a software-defined DRAM cache, and emit remote-aware code; across data-intensive workloads it accelerates execution by up to 18× and halves 99th-percentile latency relative to swap-based or API-driven approaches.

NetPool pools SmartNICs at rack scale, design fairness-oriented itered scheduler with µs-level local deflection, and drives first-hop processing over peer links; it network-traffic driven design reduces network-device capital cost by 7.4× and boosts application throughput by 44% under burst load.

Together, these results demonstrate that a semantic-guided, cross-layer co-design of disaggregation stack can deliver the fine-grain flexibility without sacrificing the performance their applications require.

Chapter 1

Introduction

Over just the past few years the shape of datacenter workloads has shifted dramatically. Application portfolios have exploded, and the resource mix they demand now varies by orders of magnitude across *different applications*, *different inputs*. The compute–memory-network ratio gap is further deepen, the change of Large-language-model workloads illustrates the point clearly: In the past five years, one training job requires $4\times$ more computational power, but requires over $14\times$ more memory capacity; Context length has grown by more than $110\times$, so a single request can consume anything from a few kilobytes to multiple gigabytes of RAM, yet the DRAM per server has risen less than $2\times$ in the same period. Meanwhile, ever more resource types, including tensor cores, DPUs, compress–offload engines, cohabit the same racks, each with its own performance envelope.

With such heterogeneity and volatility, two pressures dominate datacenter design. First, there's a stronger need for resource efficiency. Industry studies still observe average server-side utilization below 40–45% for memory and 20–30% for accelerator cycles. Every idle gigabyte or tensor core wastes CAPEX and inflates OPEX through cooling and power budgets that now rival CPU costs. Second, application execution shows a increasing requirement for elasticity and scalability. Across applications or execution from the same application, the resource demand swings could change for over $20\times$ within minutes or across inputs, operators must add head-room instantly yet reclaim it just as fast to avoid stranded capital.

Traditional server-centric scaling forces operators to scale at the level of *entire* machines even when only one resource is scarce. A video-transcoding job drags along an unneeded terabyte of DRAM; an LLM session that is memory-heavy but compute-light hauls an idle motherboard into the rack. Datacenter builders and tenants alike now seek to share and scale resources at far finer granularity.

These advances invite a fresh architectural paradigm: hardware resource disaggregation. By breaking the implicit CPU + memory + storage + accelerator bundle and leveraging RDMA, CXL, and similar links, datacenter hardware can be reorganized into independently scalable, failure-isolated pools. To an application, the facility appears as multiple infinite reservoirs, each growing or shrinking on demand, unconstrained by the physical chassis that once defined a "server."

Three converging hardware trends make that wish plausible. High-speed, low-latency fabrics export remote access semantics identical to local ones, erasing the programmer's awareness of device placement. Logically, the fabric becomes a new system bus: any compute node may map remote pages, enqueue work to a GPU pool, or attach block devices with the same instruction stream it uses for on-board peripherals.

1.1 Resource Disaggregation and Its Intrinsic Performance Overhead

Physical-location transparency is the defining promise of resource disaggregation. Regardless of the underlying hardware, interconnect, or software stack, every disaggregated platform offers the same abstraction: *a single resource pool masking physically separate devices*. From an application's perspective, a job can acquire CPU cycles, memory capacity, or accelerators on demand, scaling only the resource it lacks, dynamically regardless of the physical location of resources at the runtime.

This vision contrasts sharply with the well-studied domain of *distributed systems*. Classic

distribution partitions the *application*: developers shard datasets, replicate state machines, and reason explicitly about partial failure. Disaggregation instead partitions the *hardware substrate* beneath an unmodified application. A single run might bind to local CPU and DRAM at the beginning, then draw extra DRAM from a remote blade at the runtime, all transparent without altering program logic.

This abstraction enables higher resource flexibility. Because each pool scales independently, an in-memory analytic can double its footprint by annexing remote DRAM nodes without consuming an extra CPU cycle, while a batch job can burst onto idle CPUs cores at night without touching the machines hot DRAM cache. Pooling also converts stranded capacity into a reservoir large enough for contiguous allocations, pushing DRAM utilization above 87% in prototype CXL fabrics, compared with roughly 70% in conventional clusters [105]. Finally, heterogeneity becomes tractable: GPUs, FPGAs, compression engines, and NVRAM all surface through a uniform interface.

However, the intrinsic performance issue prevent resource disaggregation from widely deployed. Because the unified, homogeneous and on-demand pool is an illusion created by the disaggregation system, even when the API is uniform, several performance penalties remain. the most severe ones including longer latency, different granularity and limited bandwidth. Remote DRAM, for instance, incurs hundreds of nanoseconds of wire delay and protocol overhead. At the runtime, an application that is unaware of the fact could still use local access patterns on remote memory, with frequent and small accesses that go against the chunked and cached remote accesses, and magnify that cost until it dominates execution time.

The overheads surface and become an unavoidable performance penalty at three layers. **At the application level, flexibility obscures optimization opportunities**: code could not be optimized based on the resource features and performance characteristics, as the actual underlaying hardware could change at the runtime. **Within the system stack, server-centric policies misfire and go against each other**: kernels, runtimes, and caches manage duplicated state, apply conflicting heuristics, and interpret stale metrics, all because the true hardware

topology is hidden. **At the data-center level, theoretical savings are hard to realize**: scheduling must trade off resource utilization against the performance penalty, and mismatches lead to unused capacity or inflated and unacceptable cluster level performance drop.

Our survey of modern disaggregated memory and network systems shows that every lineage wrestles with the same open question: *how can we preserve the flexibility of disaggregation while driving its overhead toward zero?*

1.2 Pierce The Veil: End-to-End Disaggregation Stack

In this dissertation, we advance resource disaggregation beyond building resource compatible layers and embrace an end-to-end system design that spans hardware, network, operating system, runtime, and application boundaries.

We identify the major cause behind the efficiency–performance dilemma of resource disaggregation. The fundamental issue is a differently layers manages and utilizes the resources in a different, locally optimized way, this *behavior mismatch* causes performance lost due to friction across disaggregation layers. Using RDMA-backed memory disaggregation as an illustrative case: the operating-system page cache migrates hot pages without regard to fabric topology, which triggers ping-pong thrashing and moves untouched read or write data across the fabric; NIC drivers treat every remote-memory fetch as a bulky RDMA and ignore the spatial locality that application and CPU could exploit; user-level runtimes schedule threads under the assumption of uniform memory model costs, and even small deviations in latency violate shared data-structure performance optimizations that depend on low variance. No single layer behaves incorrectly. Each of them is locally optimized. However, when combined, the stack performance can degrade sharply.

I defend the following thesis statement: *Disaggregation becomes practical and performant only when resource semantics (what is a common behaves from a specific hardware type) and application-semantic intent (why and how the application uses the resource) are*

exposed across datacenter system layer boundaries, thereby enabling joint optimizations that span multiple layers of resource disaggregation stack.

The most important contribution of this dissertation is the insight that optimization hinges on reducing misalignment among stack layers. To realize disaggregation effectively, the illusion of a monolithic server cannot be created in any single layer, and layers must no longer be optimized in isolation. For best performance, each layer should consider the behavior of the others, which demands a full-stack design that supports cross-layer co-optimization.

We pursue this end-to-end solution by answering three questions in sequence. First, *what behaviors must cross layer boundaries and prove critical to performance optimization?* To enable co-optimization, we focus on two previously under-leveraged semantics: application semantics and resource semantics. We analyze how each is currently handled across multiple layers. The semantics reveal, for example, that sharing one resource can differ dramatically from sharing another, which leads to distinct requirements for management, performance modeling, and interconnection.

Second, *how can these behaviors be extracted?* This dissertation develops a general methodology for capturing and exploiting the identified semantics. Through cross-layer profiling, we introduce lightweight probes that correlate micro-architectural events such as LLC misses with fabric telemetry such as queue depth and with application-level metrics such as key-value hot-set size. Through semantic programming interfaces, we design narrow, vendor-neutral APIs that convey resource characteristics, including memory latency sensitivity and prefetching applicability, as well as workload hints such as access pattern and QoS class, all while hiding proprietary hardware details. Finally, we present comprehensive analysis and comparison techniques that quantify the impact of semantic exposure.

Last, *how can the extracted behavior be used to optimize systems?* We propose an end-to-end redesign of the entire stack that aligns the behavior across stack layers, specializes for disaggregation purposes, and deduplicates components. Duplication is removed where possible, latency is excised from critical paths, and functionality is rescheduled across resource boundaries

to match the semantics uncovered earlier.

Taken together, these techniques achieve *cross-layer semantic alignment*: information that the compiler extracts is understood by the runtime and enforced cooperatively by the cache hierarchy, network transport, and resource serving device. We showcase the benefits of cross-layer co-design with three end-to-end systems prototypes. Each system, guided by the same principle, resolves friction between specific layers for a particular resource type and set of system layers. Collectively, they form an end-to-end disaggregation-stack blueprint that reaches from the application layer down to the operating system, networking substrate, and resource hardware.

Chapter 2 reviews the performance gap that arises when existing far-memory runtimes treat applications as black boxes. Swap–based systems evict and prefetch at 4 KB page granularity, causing severe read/write amplification, while API-level libraries off-load whole objects, demanding manual refactoring and still missing future-access context. The root problem is a semantic disconnect: the runtime sees addresses, but the compiler alone understands an application's data phases, lifetimes, and locality. Mira bridges this gap through a co-design of the *application layer (static analysis + profiling)* and the *memory-disaggregation runtime*. The compiler classifies objects and phases, emits behavior hints, and auto-generates remote-aware code; the runtime exposes a configurable DRAM cache whose sections—size, associativity, line width, prefetch window, and RDMA mode—are tuned to those hints. Together they slash data amplification, hide network latency with just-in-time prefetching, and adapt on the fly as phases shift, delivering near-local performance without developer intervention.

Chapter 3 reviews how the flexibility promised by memory disaggregation clashes with the RDMA-centric stack beneath it. OS managed page allocation hinder dynamic allocation, host-side page walks inject long tails, and metadata round trips stall critical paths—symptoms of a stack tuned for local NUMA, not remote memory pools. We fix these pathologies through a cross-layer co-design of the *network transport, hardware page translation, and virtual memory management semantics*. The transport becomes a stateless, connection-less protocol; translation is pushed into a hash-based hardware page table; every remote access is packaged as a self-

contained request that flows through the memory node without per-client state. This re-aligned pipeline trims tail latency, and preserves line-rate bandwidth even with thousands of concurrent clients, demonstrating that deterministic performance emerges only when network, virtual memory, and hardware datapath are designed together.

Chapter 4 further go beyond single application's performance and fulfils disaggregation's system level performance promises. It reviews how per-host SmartNICs optimize a *single application's* packet path yet undermine *rack-wide* cost and tail-latency goals: each server must size its NIC for the sum-of-peaks, so hardware sits idle in the common case, while bursty traffic can still overflow an individual card and stall flows. NetPool restores both economic and performance efficiency through a *system-level co-design* that couples a rack-scale pooling pattern with a disaggregation control stack. The fabric connects every host to a shared bank of SmartNICs and peers those NICs with lightweight inter-NIC links; a global controller allocates the bulk of encryption, compression, and bandwidth units by fairness and locality, while a per-NIC micro-controller elastically deflects spikes across the pool in microseconds. This rack-aware data–control synergy cuts over-provisioning, smooths burst latency, and lets thousands of tenant flows share accelerators at line rate—demonstrating that only a joint design of topology, resource scheduler, and NIC datapath achieves both single-flow speed and fleet-wide efficiency.

Chapter 5 reviews the process of building the end-to-end resource disaggregation stack and discuss the future directions.

Chapter 3, in part, is a reprint of the material as it appears in the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, Yiying Zhang. The dissertation author was the primary investigator and author of this paper.

Chapter 2, in part, is a reprint of the materials as it apears in 29th Symposium on Operating Systems Principles. Zhiyuan Guo, Zijian He, Yiying Zhang. The dissertation author was the primary investigator and author of this paper.

Chapter 4, in part, is currently being prepared for submission for publication of material.

Zhiyuan Guo, Yiying Zhang, Arvind Krishnamurthy. The dissertation author was the primary investigator and author of this paper.

Chapter 2

Mira: A Progam-Behavior-Guided Far Memory System

2.1 Introduction

As memory becomes one of the most contended hardware resources in data centers and as more applications require huge memory to execute, a promising and popular approach is to allow applications to use memory beyond traditional main memory, such as unused memory on a remote server [50], disaggregated memory blades in a server or a rack [105, 72], and other forms of slower but cheaper memory [96, 145]. Some of these non-local memory has attached computation power (*e.g.*, an ARM processor) [182, 143]. In this paper, we call all of them *far memory*.

Because far memory is slower than local memory, existing systems have all utilized local memory as a cache for far memory, with two approaches. The first is to transparently swap memory pages between local and far memory [66, 18, 147, 172, 16]. These systems all suffer from the coarse granularity of a 4 KB page, which is often larger (2.3× to 31× [36]) than what is actually read/written by an application. Data amplification not only consumes extra network bandwidth but could also slow down overall application performance. The second far-memory approach is to use a new programming model or extend an existing one with new APIs for far-memory accesses [143, 51, 164, 182]. Through explicit and precise control of what to access in far memory, this approach reduces amplification but requires non-trivial application-programmer

or library-writer effort.

These two approaches respectively perform optimizations dynamically by a run-time system and statically by programmers. The former is a completely transparent system-level approach that treats user programs as a black box, while the latter is a white-box approach that puts the responsibility of optimization on programmers. *Is it possible to overcome the drawbacks of these approaches, harness their benefits, and even surpass their best-case performance?*

Our answer lies in a *program-behavior-guided far-memory approach*, by exploring an unexplored layer in far-memory research: program-analysis tools and compilers. With compilers, we can automatically convert programs written for local memory to accessing far memory, optimize the transferred code for better performance, and do so without any programming burden. Program analysis can reveal information unknown to run-time systems or even programmers. For example, it can detect indirect memory accesses like `for (i=0; i< size; i++) B[A[i]]++;`. With this knowledge, a compiler can insert prefetching operations like `%1=(fetch A[i+distance])` and `fetch B[%1]` at `distance` elements ahead. In contrast, without program knowledge, a runtime-based far-memory system often exhibits amplification or prefetching of incorrect data based on history. While static program analysis and code optimization offer many benefits, a key limitation is their inability to incorporate run-time information, which may result in suboptimal decisions. As with prior solutions to static approaches' limitations [129, 81], we can leverage run-time profiling of applications and utilize profiling outcomes to steer program analysis and compilation for far-memory systems.

We leverage program analysis, compiling, and profiling together to automate and optimize far-memory accesses. These technologies have been extensively studied in a traditional server setting for optimizing the performance of CPU cache and main memory [38, 95, 107, 129]. However, cache for far memory is fundamentally different in one key aspect: *cache for far memory is DRAM-based and can be controlled by software*. This feature gives us a great opportunity to customize the cache for program behavior (which we acquire from program analysis and run-time profiling) and to generate and optimize code for far memory based on

the customized cache configurations using a compiler. Together, they call for the *co-design of program analysis, compiler, profiling, and run-time cache systems for far memory*. This co-design opportunity also brings significant challenges: while traditional compiler optimizations target a fixed CPU cache architecture, we need to configure our cache based on our program analysis and profiling, and our compiler should generate and optimize code for far memory via this non-fixed cache.

To leverage opportunities and confront challenges, our core idea is to separate the local cache into spaces dedicated to and configured for different program behaviors. We observe that a program often exhibits several different memory access patterns with different objects or at different phases, and they benefit from different cache configurations. For example, sequential accesses fit a small directly mapped cache with a cache line size of multiple consecutive data elements, while accesses with good locality but large working sets fit a relatively large set-associative cache. With this observation, we propose to divide the local cache into different *cache section*s, each tailored to a distinct access pattern. Based on the analyzed and profiled behavior of one program scope for one object or multiple objects with the same behavior, we configure a cache section's size, cache structure (*e.g.*, set-/full- associative), cache line size, prefetching and eviction patterns, and communication method (*e.g.*, one-/two-sided RDMA). Our compiler then optimizes code in that scope to best fit the configuration. Section separation allows us to customize cache configurations for one access pattern at a time and to in turn optimize code for one cache configuration at a time. Additionally, we decompose a whole-program-whole-cache co-design problem into manageable per-access-pattern subproblems that we can more precisely solve.

With this core idea, we build *Mira*, a far-memory system that co-designs program analysis, compilation, a configurable cache layer, and run-time profiling. It follows an iterative approach shown in Figure 2.1. Initially, Miraprofiles the application running on our generic swap layer to identify scopes for analysis. For these scopes and based on analysis and profiling results, Mira identifies objects to place in far memory, generates far-memory accessing code, and optimizes

the code and cache configuration. Additionally, Mira identifies and compiles functions to offload to far memory with computation power, also based on program behavior. The next iteration uses the new configuration and code. If high overhead is detected, Miraperforms another optimization iteration, until user-specified stopping criteria is met.

Mainly two goals: reduce the high far-memory accesss overhead; Mitigating the effect of large scale Apart from the co-design challenge with a configurable cache, Mira confronts two unique challenges in a far-memory environment: 1) inefficient implementation of far-memory pointers and their dereferences will largely hurt application performance; 2) larger program scopes and more objects need to be potentially analyzed, as far-memory accesses are slower and local cache is larger than CPU cache. For 1), we design a novel far-memory pointer dereferencing mechanism that is performance efficient and metadata-space efficient, by leveraging program behavior to turn as many dereferences into native memory loads as possible. For 2), we perform coarse-grained, cache-section-specific profiling to narrow down program scopes and objects to those with the highest potential gain from further optimization, and we analyze and optimize each of them while globally optimizing the partition of local cache space across them.

We implement Mira's static parts on top of MLIR [99], a Multi-Layer Intermediate Representation ecosystem that allows us to choose the proper abstraction levels to build our program analysis and compiler and to support a variety of front-end programs and back-end execution architectures. We build all run-time parts as user-level libraries. We evaluate Mira using micro-benchmarks and three real programs: MCF [21], DataFrame [75], and GPT-2 [126] inference [13]. We compare Mira with FastSwap [18], a kernel-level swap-based far-memory system, Leap [16], a run-time pre-fetching solution for swap-based far-memory system, and AIFM [143], a far-memory system with a new programming model. Our results show that Mira outperforms these prior swap-based and programming-model-based systems by up to 18 times.

Mira is available at https://github.com/WukLab/Mira.

2.2 Related Works

2.2.1 Existing Far-Memory Systems

Page-based far-memory swapping. A common way to build far-memory systems is via page-based memory swapping. InfiniSwap [66] is the first RDMA-based remote memory swap system. FastSwap [18] improves InfiniSwap's performance with better scheduling and polling mechanisms. Leap [16] prefetches memory pages to avoid remote-memory accesses in the critical path based on a process' majority access pattern. Canvas [172] and Hermit [138] are two recent works that improve Linux's swap system by enforcing better isolation mechanisms in a multi-application environment and by executing non-urgent but time-consuming tasks asynchronously. LegoOS [147] is a non-Linux based system that swaps 4 KB pages between a compute node's "extended cache" and disaggregated memory.

These swap-based systems all suffer from two common problems: 1) they are all 4 KB page based. Such coarse granularity could result in huge network bandwidth wastage and reduced application performance [36]; 2) they are all agnostic to program semantics. As we will show, program semantics are crucial in enabling a variety of optimizations.

3PO [33] is a recent system that uses an offline process to analyze memory accesses of *oblivious* applications, whose memory accesses are independent of program inputs. 3PO then uses the analysis results to perform prefetching. 3PO still performs prefetching in 4KB-page granularity. Moreover, it only works for completely oblivious applications.

Cache-line-based and other far-memory systems. New hardware like CXL [43] and research-based prototypes [37, 63] enable access to far memory in cache-line size and with much faster speed than today's network communication. Moreover, CXL allows CPU cache misses being directly served by a memory device connected to the CPU. Software systems on top of these hardware technologies can utilize the high speed and/or fine access granularity to improve far-memory performance [36, 105]. Unlike Mira, none of these existing software systems consider program semantics or configure local cache based on program behavior. Note that even though

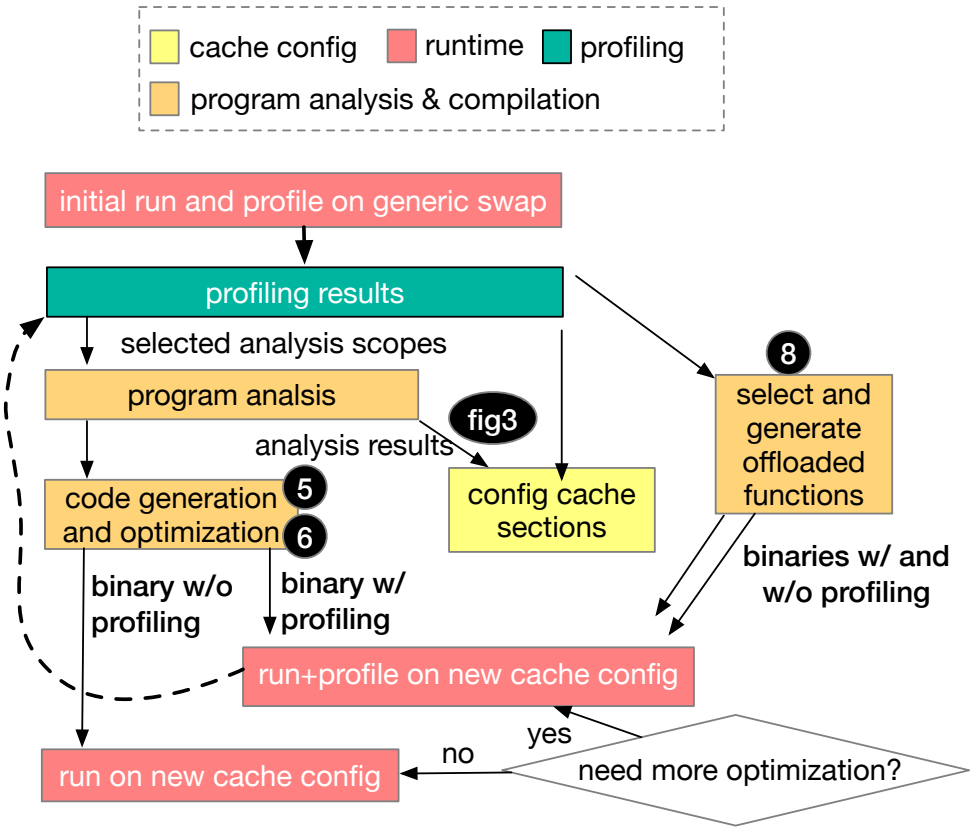

Figure 2.1. Mira Overall Flow

our implementation of Mira focuses on RDMA-based remote memory, our general designs apply to a broad definition of far memory, including CXL-based memory pools, local- or remote-node persistent memory, and slower storage layers, because Mira's optimizations can adapt to different far-memory accessing speeds and computation power.

New programming models. In addition to transparent approaches, another type of far-memory solution is introducing new far-memory-specific programming interfaces. FaRM [50, 51] and many other RDMA-based systems [182, 164, 148] use simplified or richer APIs for programmers to perform remote memory allocation, read, write, etc. AIFM [143] proposes a new programming model for far memory, including remotable pointers, dereferencing scope, eviction handler, etc. To avoid application programmers' burden, AIFM tries to confine far-memory-specific programming within libraries. A common limitation of these works is their burden on application or library developers, who can also make unoptimized decisions. Moreover, these works only

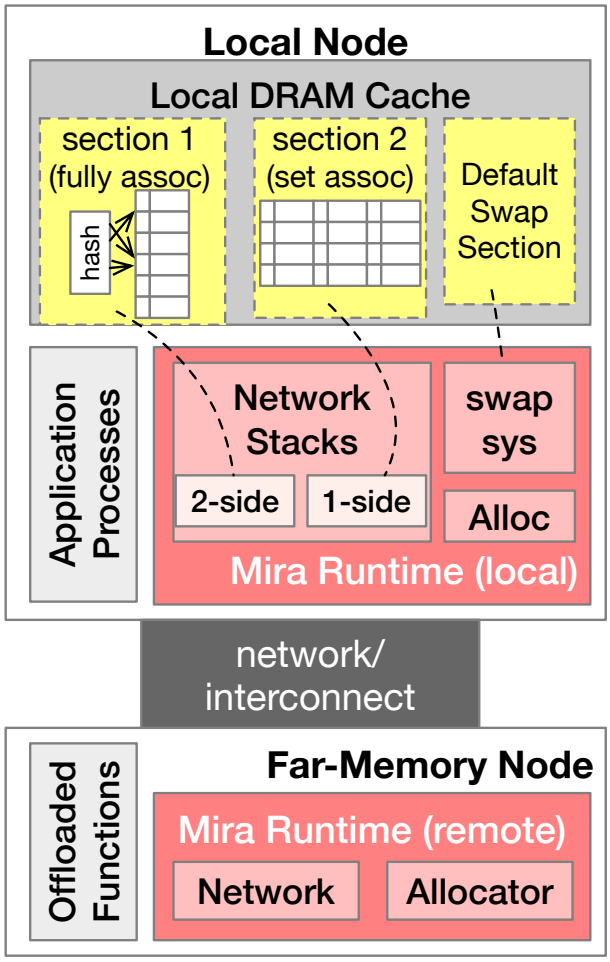

Figure 2.2. Mira Architecture

optimize their added APIs or library calls and do not analyze other program behavior for further performance optimization opportunities. Finally, systems like AIFM incur high runtime overhead, as each far-memory pointer dereferencing requires the manipulation of fair amounts of metadata.

2.2.2 Non-Far-Memory Optimizations

Memory accesses in a traditional, non-far-memory environment have been highly optimized at various layers. However, as far as we know, there is no work that co-designs program analysis, compiler, and a configurable cache.

Compiler and system optimizations for CPU cache. A host of compiler-level and system-level solutions have been proposed to optimize applications' performance on CPU caches.

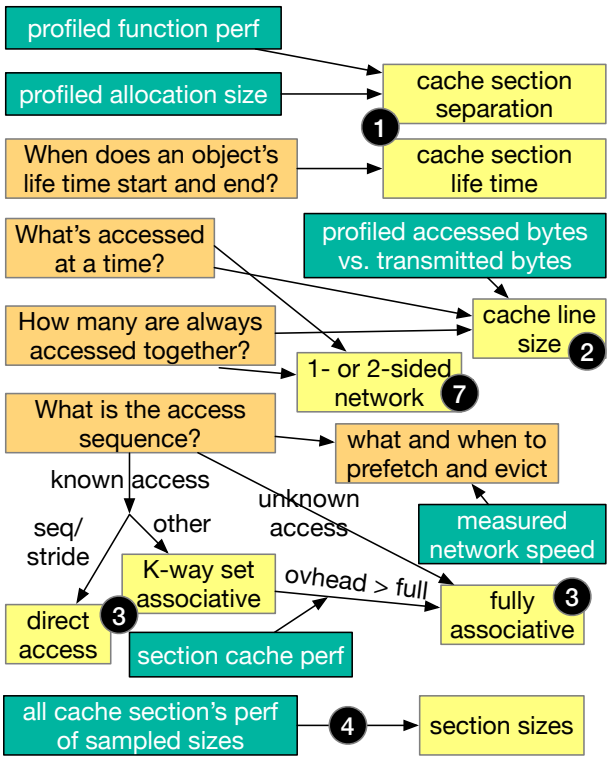

Figure 2.3. Mira Decision Making Process

They can be roughly categorized into three types. The first transfers programs and/or data layout to make memory accesses more cache friendly, *e.g.*, via data structure padding, peeling, field reordering, hot-cold code region separation, etc. [38, 95, 107]. The second allocates different CPU cache spaces to different parts of applications. For example, CPU cache coloring assigns different memory regions to different cache regions to avoid cache conflicts across memory regions [181, 47]. The third guides memory-access optimizations using run-time profiling results (*i.e.*, PGO) [129]. For example, APT-GET [81] improves prefetching of memory accesses to the CPU cache using profiling information collected from CPU counters.

These techniques cannot directly be used in a far-memory setting. Unlike Mira, they do not target a software configurable cache environment, do not co-design program analysis, compiler, and cache systems, do not work for far memory or perform any of our far-memory-oriented optimizations, and do not support function offloading.

Software-defined and configurable cache. There have been several works proposing config-

urable CPU cache architectures and software mechanisms to utilize such reconfigurable cache architectures [176, 162, 102]. For example, Jenga [162] proposes to assign different parts of CPU cache to different hierarchies (levels) based on measured cache miss curves for each application. Lee et al.[102] build a customized cache for streaming applications based on offline analysis of memory access traces. These solutions focus on the architecture and systems level, without the understanding or usage of program behavior and lack compiler optimization. Moreover, they all require non-traditional CPU cache hardware. Mira configures DRAM cache for far memory based on program behavior, with a run-time cache system, program-analysis, compiler, and profiling co-designed approach.

Another software-manageable cache hardware is scratchpad memory. Several works have focused on finding good ways to schedule what data to place in the space-limited scratchpad memory [87, 167, 156]. For example, Susu et al. [156] use static analysis and code transformation to perform space planning on scratchpad memory for an accelerator. Unlike Mira, these works do not configure scratchpad memory based on program behavior and instead seek good data placement and scheduling to fit the scratchpad.

Finally, TriCache [54] proposes to customize DRAM cache using a user-space block cache with a virtual memory interface to access fast storage devices. Unlike Mira, it does not utilize program behavior when configuring its block cache, and its usage scenario and cache designs are both different.

2.3 Mira Overview

folks, the figures are too busy to read... Basically, my feeling of reading the figure and S3 is that I don't have an overaching mental model on how everything fits in a principled approach. I feel there are a lot of rules—each rule makes sense in its own context, but are they complete and principled? Also, how do you decide the "cuts"? as performance numbers are not binary and dynamic – how large is a large object?

Mira consists of program analysis tools, a compiler, a run-time system for local nodes, a run-time system for far-memory nodes, and a profiling system. They work together to iteratively adapt system configurations and user programs for far-memory accesses, as shown in Figure 2.1. Figure 3.2 shows the run-time architecture of Mira. Mira takes an unmodified program as input and generates 1) a cache configuration based on the program's behavior and 2) a compiled code that runs on far memory via the Mira run-time system.

Overall flow. Initially, without run-time information or program analysis, Miraconfigures the local cache as a universal swap section and places all heap objects and static data in it (we never use far memory for stack or code, as they are small and frequently accessed). The initial execution works almost the same as traditional page swap-based systems, except for the profiling code our compiler inserts. At this and each of the later profiling runs, we collect per-function miss rate, miss latency, hit overhead (*i.e.*, the additional latency to access data in cache over a regular memory load), and function execution time. Additionally, we collect allocation sizes of all data objects.

We then decide how to split cache sections (initially, only the swap section) based on profiled per-function performance results and object sizes (§2.4.1). As each non-swap section needs program analysis and code generation/optimization, having many of them increases the static tools' complexity and is often unnecessary. Thus, we identify the functions that "suffer the most" from executing on the current cache configuration and compiled code, and we find larger objects in them to place in their own sections for further optimization.

Figure 2.3 illustrates the type of program analysis we perform and how we use the analysis results together with profiling results to determine various cache section configurations (details in §2.4.2). Overall, we use lifetime analysis to determine when to start and end a section, the amount of (batched) data accessed with profiled network performance to determine cache line size, and memory access sequences together with profiled cache section performance to determine cache structure. We determine the sizes of cache sections by globally optimizing the overall performance based on each section's profiled performance characteristics (§2.4.3). How

is the profiling done? Is it online or offline? Will it be super expensive if you want to track every load and store?

For code ranges in each non-swap cache section, Mira compiles code to access the cache section or in the case of a cache miss, the far memory (§2.4.4). Mira converts memory operations like allocation, read, and write to *remotable* operations at the IR level, which then is lowered to either cache or network accesses. Afterward, we perform various code optimizations based on program analysis and profiling results, *e.g.*, prefetching data, batching far-memory accesses, flushing and marking data evictable, etc. (§2.4.5). We also generate code to access different network stacks of Mira based on program behavior (§3.4.4). Finally, we instrument the compiled code with coarse-grained profiling operations for the next round of profiling execution.

In addition to the above, we consider per-function computation load and network traffic to determine which functions to offload to far memory for optimal performance, and Mira generates binaries for them (§2.4.8).

Input adaptation. To adapt our compilation and cache configurations to inputs, we invoke profiling on sampled inputs. When the current compilation and cache configurations' performance degrades, we trigger a round of iterative code optimization in the background while the user invocation of a program keeps using the current compilation. Each iteration uses the previous iteration's profiling results to potentially set a new cache configuration and generates a new compilation. System administrators of Mira set an optimization target for each round (*e.g.*, at most 10 profiling-optimization iterations, or keep optimizing until no further gain is observed). The final compilation of a round is used for subsequent invocation of the program until another round of iterative optimization is needed. Each round of optimization converges fast, usually in two to three iterations, and our profiling adds negligible performance overhead (§3.7). Our iterative approach reduces analysis and optimization scopes and complexity at each iteration while allowing for inaccuracy in one iteration to be fixed in the next one.

Overall, this sample-based input-adaptation approach has been taken by most prior profiling-guided-optimization (PGO) works [81, 93, 158, 154, 129, 27, 94, 92] and has been

```
1 edges, nodes = malloc()
2 void traverse_graph(struct edge *edges) {
3   for (int i = 0; i < num_edges; i++)
4     update_node(edges[i], edges[i].from, edges[i].to);
5     // edges[i].from and edges[i].to point to nodes
6 }
```

Figure 2.4. (Simplified) Code Example of Graph Traversal.

adopted in production [140, 129, 39]. As we (§3.7) and prior works show, this approach has only little mis-profiling overhead. This is because production workloads' inputs change slowly [39], and a fair amount of datacenter applications like machine learning benefit from the same sets of optimizations regardless of their inputs [33]. Additionally, as we will show in §4.4, many of Mira's designs are resistant to input changes.

Targeted applications. Mira optimizes code with memory access patterns that can be inferred from static analysis and dynamic profiling. Many datacenter applications fit this feature. Our evaluation results show the benefits of Mira for data analytics, machine learning, and graph processing applications (§3.7). Apart from our evaluated applications, Mira is potentially beneficial to other types of applications such as key-value stores and event-triggered applications. For applications or parts of an application that Mira does not optimize, we guarantee performance that is on par with existing swap-based far-memory systems.

Note that we assume each application to have its own cache space, far memory space, and cache runtime that are isolated from other applications. A datacenter/cloud manager can decide the amount of local/far memory space and CPU cores assigned to each tenant (application) and then run Mira in each tenant's container/VM.

2.4 Mira Design

This section presents Mira' design, including how we perform and utilize profiling, how we configure cache sections and their sizes, how we generate and optimize remote-access code, how we support multi-threading, different communication methods, and automated function

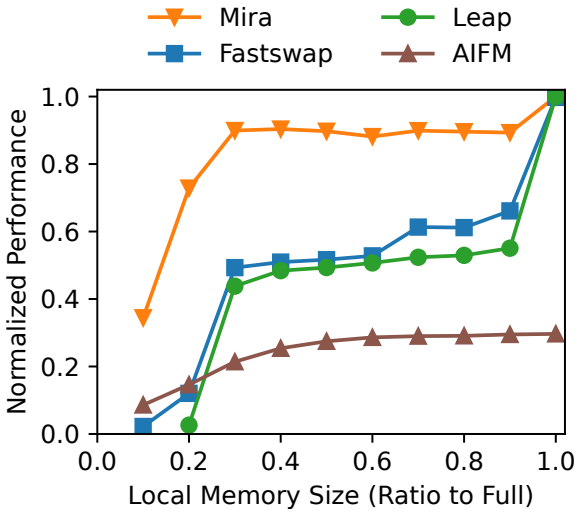

Figure 2.5. Overall Performance of Edge Traverse Application

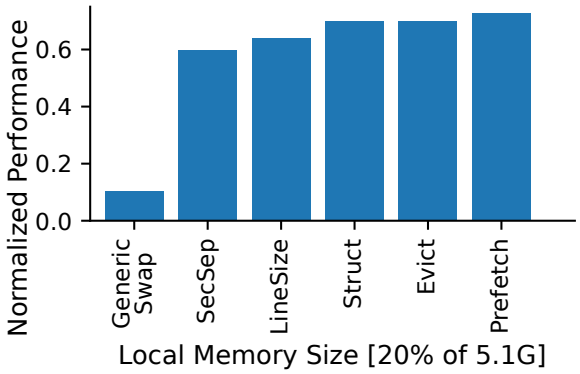

Figure 2.6. Overall Breakdown of Edge Traverse Application

offloading. We use a simple graph traversal program shown in Figure 2.4 as the rundown example of Mira's major designs. It traverses an edge array sequentially and updates the edge's source and destination nodes in a node array. Figure 2.5 shows the overall superior performance of this example when running on Mira as compared to FastSwap [18], Leap [16], and AIFM [143] for all local memory sizes. Figure 2.6 summarizes the effect of Mira techniques on this example. Here, and throughout the paper, we show relative performance that is normalized over native execution on full local memory (*i.e.*, no far memory).

2.4.1 Profiling for Cache Configurations

As discussed in §4.3, to make program analysis more manageable, we leverage profiling results to pinpoint specific segments of a program that require analysis. Moreover, profiling results aid in identifying configurations that are challenging to determine through static analysis alone.

Profiling mechanism. Traditional profiling that happens at the run-time system can add fairly high-performance overhead that is not necessary for our profiling purpose. We instrument profiling code during compilation and only profile coarse-grained cache section performance at the function level or at allocation sites. Most of our profiling is related to a cache section's behavior (*e.g.*, miss rate, miss latency, hit overhead). These metrics are collected only when a non-native cache event happens, leaving native memory access intact and achieving lightweight profiling.

Determining cache sections and analysis scopes 1. We leverage Mira's overall iterative flow to adaptively decide what data and code regions to place in a cache section, improving section selection with each iteration. After a profiling run, Mira collects the cache overhead and execution time of all functions. We compare the cache performance overhead across all functions and pick the highest 10% functions to analyze. Here and throughout the paper, we define cache performance overhead as the ratio of time spent in Mira runtime over the remaining program execution time, where the former includes handling cache hits (*e.g.*, cache lookup), misses (going across the network to fetch cache lines from far memory), and evictions. When selecting a function, we also implicitly select all its callee functions recursively for analysis. In the next iteration, if more optimization is needed, Mira uses new profiling results to pick the highest 20% functions to analyze, and so on (*i.e.*, 30%, 40% in the subsequent iterations until iteration stops).

After picking functions, we further nail down the analysis scope to large objects, as they need more space and will likely cause more cache misses. Similarly, we pick the largest 10% objects in the first iteration. If this function still needs to be analyzed in later iterations, we pick

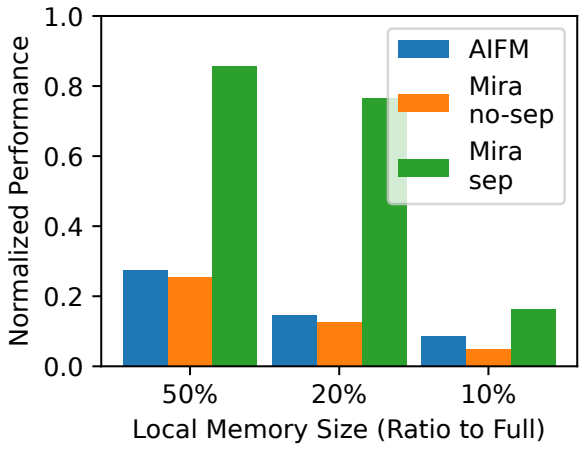

Figure 2.7. Overall Performance of Edge Traverse Application

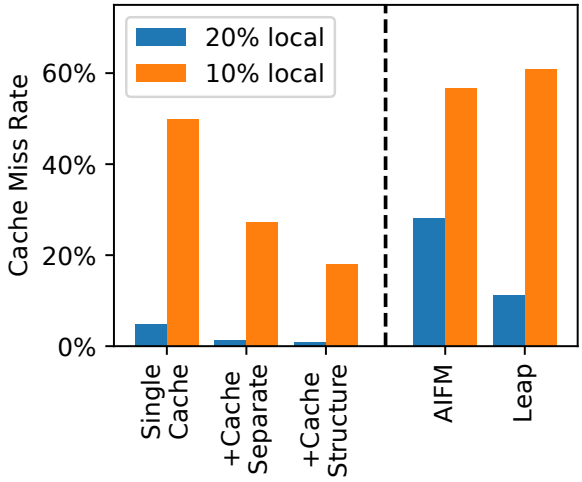

Figure 2.8. Overall Breakdown of Edge Traverse Application

the largest 20% objects. Users can set their own thresholds to replace these values we use for functions and objects. Even if we pick non-ideal objects and functions to optimize (*e.g.*, because of program input changes), our optimizations still improve application performance over generic swap-based systems.

After performing an analysis of the selected functions and objects and knowing their access patterns (§2.4.2), we group similar patterns into one section and leave different patterns in different sections. That means multiple objects can be in one section if their access patterns are similar, while one object can be in different sections at different times if its access pattern

changes. Note that with the complexity and uncertainty of cache/code optimizations, separating a cache section may worsen its functions' performance. In this case, we roll back to the previous iteration's configuration.

Figure 2.7 shows the performance of Mira when not separating and separating cache sections with the graph traversal example. We also show AIFM [143]'s performance as a reference. Cache separation significantly improves Mira's performance. After initial iteration, Mira separates out two sections, one for the node array, and one for the edge array. To further understand where the benefit of cache separation comes from, we measure the miss rate of accessing the node array in Figure 2.8. In a joint cache, the sequentially accessed edge array could evict the randomly accessed node array and end up taking more space than what it needs (a few lines because of the sequentiality). After cache separation and assigning appropriate sizes (§2.4.3) to each section, the node array's miss rate drops by 44%-78%, while the edge array's miss rate stays the same. Cache separation also allows us to apply different cache structures to each section (more in §2.4.2), which further reduces the node array's miss rate.

2.4.2 Program Analysis for Cache Configurations

For the code regions selected using the profiling results (§2.4.1), we perform static program analysis to infer their access patterns, including their lifetime, access sequence, access granularity, access being read or write, and what data are often accessed together. We then use these analysis results together with profiling results to determine various cache section configurations to be used for the application's execution.

Determining cache line size 2. A cache line in our system can contain one or multiple data items. We determine the cache line size of a section based on several factors. On the one hand, we want a cache line to be no larger than the data access granularity to avoid read/write amplification. On the other hand, if data items are accessed contiguously, we want to enlarge the cache line to cover as many of them as possible, as long as the line size is not bigger than what the network can transmit efficiently at a time. This is because accesses to each cache line need to

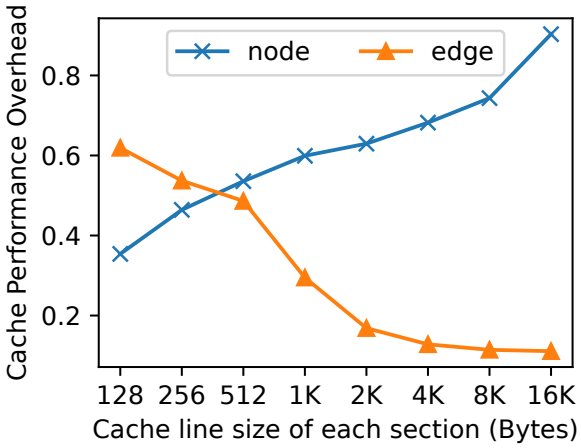

Figure 2.9. Effect of Different Line Size.

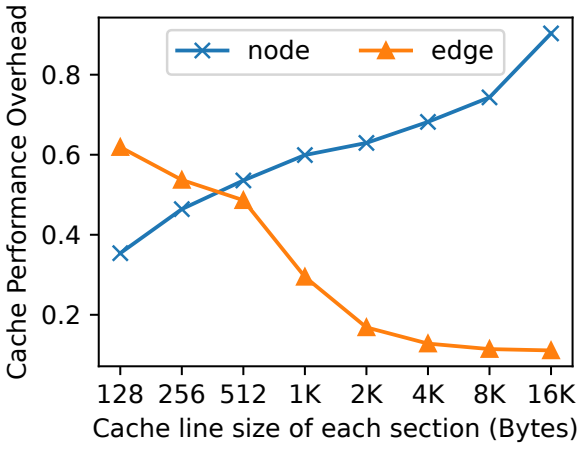

Figure 2.10. Effect of Cache Structures on Node Objects.

go through a relatively costly pointer dereferencing process but accesses to an offset within a dereferenced cache line do not incur this overhead (§2.4.4). We take into consideration all these factors when setting cache line sizes.

Figure 2.9 shows the cache performance overhead (§2.4.1) when using different cache line sizes for the node and the edge sections. For the node array, a smaller size is better, as it is accessed randomly. 128 bytes is the smallest size that can hold the accessed data unit. The edge array is accessed sequentially and thus benefits from larger line sizes. The cache overhead decreases dramatically when the line size is smaller than 2 KB because of our measured network characteristics.

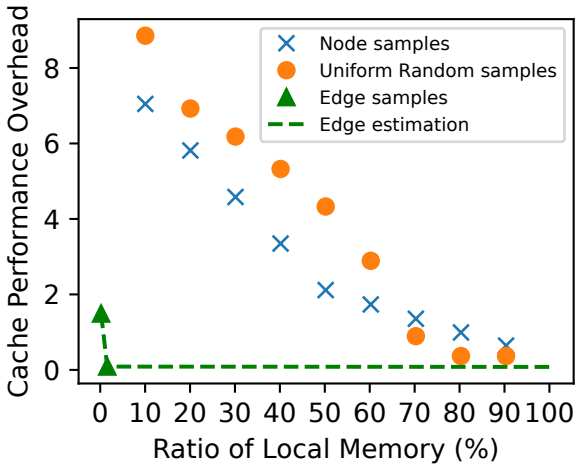

Figure 2.11. Cache Performance Overhead with Sampled Sizes Different datastructure have different performance under memory pressure. Through estimation and sampling Miracould know performance.

Determining cache section structure 3. Mira currently supports three cache section structures: directly mapped, set associative, and fully associative, following classical CPU cache architectures. Future works can add other structures. As with CPU cache, full associativity has the best utilization of cache space (*i.e.*, no conflict miss) but has a higher run-time overhead for cache lookup. This tradeoff shifts the other way with set associativity and then direct mapping.

To determine the structure of a cache section, we first analyze the access sequences of the program scope for a section to estimate the potential amount of conflicts that contend for a cache set or a direct location. If the access pattern is sequential or stride, then we use a directly mapped cache, as there will be no conflict. Otherwise, we analyze the locality set (*i.e.*, the entries of data that need to live in the local cache at the same time) and addresses in the locality set. If we cannot identify a locality set, we set the section to be fully associative. If we can find locality sets, we infer the potential amount of conflicts when using a K-way set-associative cache and set K accordingly.

Figure 2.10 shows the effect of using different cache structures on the node section. When local memory is large, full associativity has a constant overhead over set associativity and direct mapping. As local memory gets smaller, full associativity turns better than set associativity.

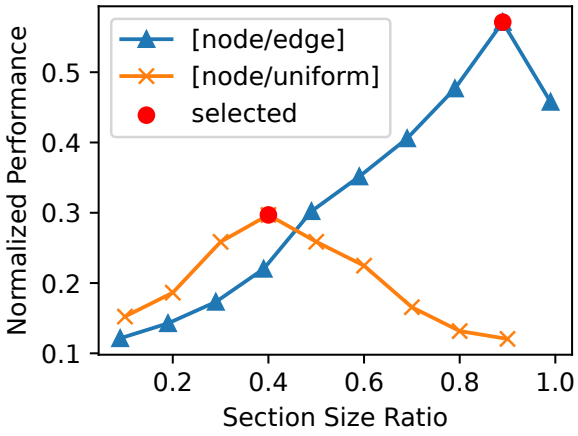

Figure 2.12. Section Size Selection

Note that even though different cache structures for the node section have small differences, choosing the right cache structure for different sections has a larger impact on performance.

2.4.3 Determining Cache Section Size

As discovered by previous far-memory systems [143, 18, 147], the amount of local cache can largely impact far-memory system performance. Different from previous systems that only consider the effect of total cache size for an application's performance, we consider the effect of each cache section's size, as different objects and their access patterns can be affected differently by the amount of local cache. We use sampling and profiling to determine section sizes. 4

We first sample a few sizes for each section. In each sampled run, we profile the cache performance overhead of the section. Sequential and strided cache sections only need a small size that can fit enough prefetched data to hide network delay. Beyond this size, the performance of these sections would stay the same. Thus, we only need to sample very few sizes to find a sequential/strided section's optimal size. For other cache sections, we sample a few section sizes as ratios of total local memory size (*e.g.*, 20%, 40%, 60%, 80%). After acquiring the relationship between section size and section performance for them and with our program analysis results of section lifetime, we construct an integer linear programming (ILP) problem with the target of minimizing the total cache overhead and the constraint that during any time, the total size of live

sections should be no larger than the total application's local memory space. The solution to this ILP problem is the sizes we use for these sections.

Figure 2.11 shows different cache sections' performance overhead when sampling different section sizes. As the edge array is accessed sequentially, a small size can already achieve the same performance as the full size. The node array's accesses are indirect, and its section cache overhead is non-linear from our sampled results. To make the section size selection problem more interesting, we add a third array that is accessed uniformly randomly to be in another section, which also exhibits non-linear behavior with different section sizes. Figure 2.12 shows normalized application performance when partitioning the local memory differently across multiple sections and the partition ratios Mira's ILP solutions give (which are the optimal ratios). As expected, the optimal selection between node and edge arrays is to give most memory to the non-sequentially accessed node array. The ratio between the node array and the third array follows their sampled performance results.

2.4.4 Conversion to Remote Code

Depends on the cache configuration results, one remotable access or region could be lower to different form.I don't get what this actually means

Converting to remote pointers and operations. Our compiler generates explicit remote operations for objects in non-swap cache sections; swap sections run the original code. As explained in §2.4.2 and discovered by previous API-based far-memory solutions [143], explicit remote operations can more precisely control far-memory accesses and thereby improve application performance. Specifically, Mira turns all pointers that point to selected objects (as in §2.4.1) in non-swap sections to remote pointers (defined in Mira's IR §2.5.1). It then turns allocation, load, and store operations of these remote pointers to their corresponding remote APIs (*e.g.*, remote load/store, see §2.5.1). Figure 2.13 shows a simplified code converted to remote operations for the graph-traversal example, using notations in §2.5.1.

Lowering remote operations 5. When a remote pointer is dereferenced, resolving it could

```
1  @_redges, @_rnodes = remotable.alloc(..)
2
3  // parameter uses internal edge struct representation
4  remotable.func @trvs_graph_rmt(%arg0: !remotable<struct<edge>>){
5    scf.for %i = %0 to %num_edges { // scf is an MLIR dialect
6      // dereference remote pointer to local pointer
7      %1 = rmem.deref %arg0[%0]
8      %2 = rmem.deref %1->from
9      %3 = rmem.deref %1->to
10     func.call @update_node (%1, %2, %3)
11   }
12 }
```

Figure 2.13. Convert to Remotable for Graph Example.

involve three steps: 1) looking up the pointer in the local cache; 2) if not found in the cache, fetching the data from far memory to the local cache; and 3) the actual data access. The third step is unavoidable. We perform prefetching to hide the overhead of far-memory accesses (step 2), to be discussed in §2.4.5.

We now describe how we optimize the first step of cache lookup. Normally, each cache lookup would require a set of instructions to locate whether or not and where the pointed-to data sits in the local cache. However, if we have already accessed a cache line and know that it is still in the local cache, we would know its local memory address. For future accesses of any data item in the same cache line, we can directly resolve the dereferencing by using the already obtained local address and an offset in the cache line. In these cases, the Mira compiler converts a remote pointer dereferencing to a native memory load instruction.

Note that the above optimization is only possible if the cache line is in the cache when the dereferencing happens. In a single-threaded program, Mira knows from program analysis whether or not there are any potential accesses to data that may fall into the same cache set (set-associative) or cache slot (direct mapped) before the dereferencing site. If no such "conflicting" accesses exist, we can safely know that the cache line will not be evicted and can perform the above optimization for that dereferencing site. When our analysis finds conflicting accesses or is unsure about the occurrence of conflict accesses, Mira can mark cache lines as "*dont-evict*" to

```
%SEdge = rmem.cache_section {#type = "direct", #line = 2M, ...}
%SNode = rmem.cache_section {#type = "full", #line = 128B, ...}

func.func @trvs_graph_opt(%arg0: !remotable<struct<edge>>){
  scf.for %i <- %0 to %num_edges step %elements_per_line {
    // prefetch %n_ahead elements ahead from far memory
    rmem.fetch %SEdge, %arg0 + %i + %n_ahead
    // wait for current requested data (at %i) to be in cache
    rmem.wait %SEdge, %arg0 + %i
    // get corresponding phyiscal address (paddr) of cache line
    %wide_cache_line = rmem.paddr %SEdge, %arg0 + %i

    scf.for %j = %0 to %elements_per_line {
      // directly load element in (already resolved) cache line
      %1 = memref.load %wide_cache_line[%j]

      // use later element in the line to prefetch node elements
      %2 = memref.load %wide_cache_line[%j + %n_ahead_node]
      // node elements may be in cache already, fetch if not
      rmem.fetch_if_not_in_cache %SNode, %2 -> from
      rmem.fetch_if_not_in_cache %SNode, %2 -> to

      // wait for node elements to be in cache and access
      rmem.wait %SNode, %1 -> from
      %3 = rmem.paddr %SNode, %1 -> from
      rmem.wait %SNode, %1 -> to
      %4 = rmem.paddr %SNode, %1 -> to
      func.call @update_node (%1, %3, %4)
    }
    // flush used %i element for eviciton hint
    rmem.flush %SEdge, %i
  }
}
```

Figure 2.14. Mira Optimizations for Graph Example. We show optimizations of prefetching and eviction flush, not showing others for simplicity.

indicate that evicting them would cause a huge overhead. Our runtime would choose to evict them the last. When our intended dereferencing sites all finish for a dont-evict cache line, we will remove the mark.

With these optimizations, we reduce not only the runtime overhead but also the metadata needed for far memory. Compared to Mira, AIFM's [143] library-based remote operation implementation has a much higher run-time overhead. AIFM needs to perform pointer dereferencing for each remote data item (*e.g.*, an element in a remote array), as AIFM does not perform program analysis and cannot apply native-instruction optimizations like ours. Moreover, AIFM maintains a significant amount of metadata for each remote pointer, *e.g.*, a "dereferencing scope" to manage pointer lifetime. It encounters high run-time overhead using and bookkeeping the metadata. Mira's analysis directly infers object lifetime and other information and uses them to compile code as native memory instructions if possible. Mira does not need any metadata for cache lines whose lifetime it can fully control. For example, in a loop whose accessed far-memory data can all be prefetched and do not have cache line access conflict, we do not need to maintain or access any metadata like cache line tags and pointers referencing to the line, all accesses are compiled as native memory instructions.

2.4.5 Program Optimization

Apart from generating remote code, our compiler performs code optimizations in various ways as discussed below. 6 These program-based optimizations provide benefits across different inputs without the need for recompilation.

Adaptive prefetching. Prefetching is a common technique used to reduce the overhead of far-memory data accesses. Previous systems [16, 34] use generic policies to determine what data to prefetch based on run-time access history. Instead of predicting future accesses based on run-time history, we use program analysis to determine what will be accessed in the future. For example, for a multi-level loop over a set of memory accesses, we prefetch them based on the loop pattern. Different from traditional CPU cache prefetching, we determine when to prefetch

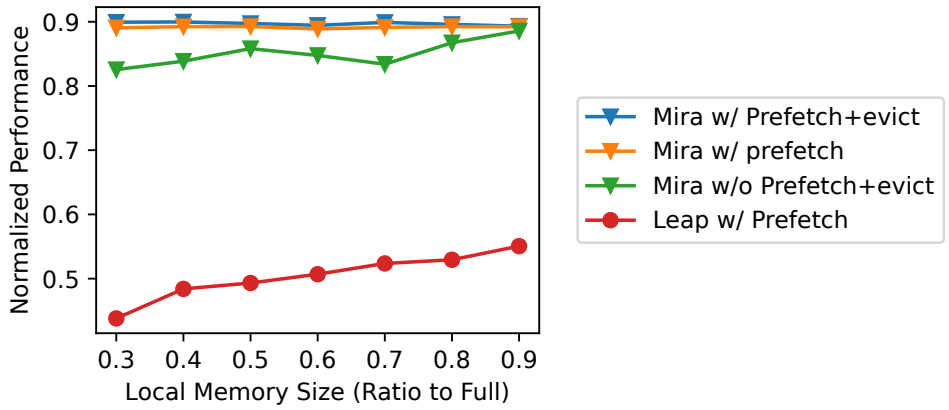

Figure 2.15. Effect of Prefetch and Eviction Hints.

based on system environments (*e.g.*, measured network delay). Our compiler inserts prefetch operations at the program location that is estimated to be one network round trip earlier than actual access.

Eviction hints. From our program analysis (§2.4.2), in many cases, we can find the last access of a data element in a program scope (*e.g.*, a function). In these cases, our compiler inserts an asynchronous cache-line flushing operation after the last access and marks the line as *evictable*. When inserting a new cache line, we check which existing lines are marked evictable and evict those first. As the least useful lines are marked with our program-guided hints, Mira improves the local cache utilization. If there is no line marked as evictable, Mira uses a default LRU-like eviction policy.

Figure 2.15 shows the benefit of adding prefetching and eviction hints in Mira when running the graph-traversal example. Prefetching hides the latency for sequential edge accesses, and early eviction hides the write-back overhead behind the performance critical path. For this application, the former has a larger impact. We also evaluate Leap [16], which performs majority-history-based prefetching. Leap only aims at capturing global access patterns and cannot properly prefetch for an interleaved access pattern like this example. Moreover, Leap uses the default Linux global eviction policy, not getting any benefits from program hints.

Selective transmission. A major problem with swap-based systems is their coarse far-memory

access granularity. New programming models like AIFM [143] allow programmers to define the exact data structures to move between local and far memory. However, programmers could make unoptimized decisions and fetch more data from far memory than needed. For example, if a programmer defines a large data structure as a object, AIFM fetches the entire data structure from far memory even when only a few fields are accessed.

To solve this problem and minimize traffic between local and far memory, our approach is to use program analysis to determine the parts in a data structure that are accessed in each program scope (*e.g.*, a function). We then generate code to only fetch or prefetch these parts.

Data access batching. For most networks and interconnects, one large communication event (*e.g.*, a message with multiple scatter-gathered data pieces) is more efficient than multiple smaller communication events. We seek program transformation opportunities leveraging this feature. If our program analysis identifies multiple addresses to be accessed at different locations, we batch them into a single network message by transforming the code. For example, when we identify two arrays to be accessed by two adjacent loops, we fuse the loops and batch access the two arrays.

Read/write optimization. In many cases, a read-only or write-only access pattern can be leveraged to achieve better performance. If a loop only contains read operations, we can safely discard the local cached objects after the loop. If it only contains writes that cover whole cache lines, we can avoid fetching the objects from far memory.

2.4.6 Multi-Threading Support

Multi-threaded programs have non-deterministic shared memory access behavior, bringing new challenges. Our solution for supporting multi-threading differs for programs that have no shared-memory writes and those that have them. For the former, *i.e.*, multi-threaded programs that are shared-nothing, read-only, or have unique ownership [28], we create separated cache sections for each thread. If multiple threads read the same data, each thread's cache section will have a copy of it. Thus, we could treat each cache section in isolation and apply all our

optimizations discussed above.

We use shared cache sections for writable shared-memory multi-threading. We configure shared sections in a conservative way: full associative with cache line size being the largest access granularity among all accessing threads. We apply all optimizations presented in §2.4.5 except for eviction hints. Shared cache sections complicate the no-conflict analysis discussed in §2.4.4, as static analysis alone cannot determine whether a cache line could be evicted by another thread before it is accessed by the current thread. Instead, we mark a cache line as ”dont-evict” from a thread's dereferencing time until the end of the line's lifetime in all threads. We perform lifetime analysis for “dont-evict” cache lines by keeping the reference count for each shared object and decreasing the count when all accesses from a thread finish.

Finally, traditional thread synchronization methods such as locks still work as is on Mira since we never make synchronization primitives remotable, and real data accesses only occur at local caches that are protected by traditional synchronization primitives.

2.4.7 Data Communication Methods

An important part of far-memory systems is the data communication between local and far-memory nodes, either over the network or over a local bus/interconnect. Many prior works have studied the benefits and use cases for one-sided communication where data is directly read/written from/to far memory vs. two-sided communication where data is sent as messages and far-memory nodes copy the messages to their final locations [164, 174]. These works manually design the communication methods for specific application domains.

We decide what communication method to use for each cache section based on its access pattern 7. If our program analysis finds that a section's access pattern is reading/writing the entire data structure, then we use one-sided communication for this section to directly read/write the data structure with zero memory copy. If a section only accesses partial data structure (*e.g.*, one or two fields of it), then we use two-sided communication to only transfer the partial structure, avoiding read/write amplification. To achieve this, our compiler inserts code to prepare/process a

message by copying from/to the partially accessed data fields.

2.4.8 Function Offloading

Certain types of far memory nodes have computation power that can execute application code [182, 143], allowing the offloaded computation to access data in far memory locally, reducing the network transfer overhead. To exploit this benefit, existing works require programmers to decide what computation to offload to far memory nodes and sometimes even rewrite offloaded computation. Mira automatically determines and offloads computation to far memory in the following program- and profiling-guided manner 8.

To reduce the program-analysis complexity, we only consider program functions as the unit of offloading and functions that do not have shared writable data. Future work could include functions with shared writeable data with the support of new coherence hardware like CXL [43]. Among the candidate functions, we determine which ones to offload to far memory based on their amount of computation and required network communication. As far-memory nodes usually have less computation power (*e.g.*, with a low-power ARM processor), it is more beneficial to offload computation-light functions to far memory. Additionally, it reduces network communication to offload functions whose accessed data are already in far memory. Thus, we consider both factors when choosing functions to offload.

To implement function offloading, we insert code at the compute node to flush the local cache that contains data the function accesses before invoking the function. The compute node then calls the offloaded function with an RPC call and sends the function inputs to far memory. After the far memory node finishes executing the offloaded function, it sends the return data to the local side.

Currently, Mira only supports offloading to CPU-based far-memory nodes. It could be extended to support other types of computing units by leveraging MLIR's capability of generating code for accelerators like GPU [90] and co-processors [7]. Similar to CPU-based nodes, offloading decisions for accelerators could be made based on computation needs and

data-movement overhead.

2.5 Implementation

We implement Mira's program analysis and compiler on top of MLIR with 7.7K LOC in C++. We implement Mira's runtime libraries that run on the local node and far-memory node with 12.1K LOC in C++. This section discusses some of the implementation details.

Mira currently runs on one compute and one memory node. Supporting multiple memory nodes, or *memory pooling*, can be done via the integration of Mira and a distributed memory management layer such as the one used in LegoOS [147], where Mira decides what objects and functions to offload and the distributed memory manager decides which memory node to offload them to.

2.5.1 Far-Memory MLIR Abstractions

MLIR. MLIR (Multi-Level Intermediate Representation) [99] is a compiler ecosystem that allows multiple abstractions at different levels. Each abstraction is called a *dialect*. Currently, MLIR supports tens of dialects for common operations, such as memory accesses, control flow, arithmetic, machine learning, and LLVM [10]. We choose to build our compiler in the MLIR ecosystem because it supports multiple frontend languages and backend architectures. Moreover, it allows us to easily add various far-memory abstractions and code optimizations as dialects at different layers while reusing existing MLIR dialects and their optimizations. Note that Mira analyzes and optimizes all libraries whose source code is available (*e.g.*, C++ STL), in the same way as application programs. We run pre-compiled library calls on our generic swap cache.

We add two new MLIR dialects for far memory:

. The dialect defines a new abstraction for data objects in non-swap cache sections and for functions that can be offloaded. Lines 1 and 4 in Figure 2.13 shows the allocation of a object and the definition of a function.

. The dialect defines operations to access and manipulate objects and functions, including two

main types. The first is basic object accesses such as load and store, by extending traditional pointer operations and `memref` [11] operations in MLIR to work with objects. For example, lines 7 and 8 in Figure 2.13 perform memory loading from objects `%arg0[%0]` and `%1`. The second type is code optimizations such as prefetch. For example, lines 7 and 9 in Figure 2.14 perform an asynchronous `fetch` of an object to be accessed in a future loop iteration and blocking `wait` the data needed for the current iteration.

2.5.2 Static Analysis and Code Generation

We now discuss how we analyze programs and generate code with the and dialects. Our analysis is sound, as we trade completeness for correctness and fast analysis time. There could be rare cases where our analysis cannot infer (*i.e.*, "undecidable"), and we avoid their optimizations.

Implementing and

We now discuss how we implement and abstractions.

Converting to and . Mira identifies data objects to place in far memory based on analysis explained in §2.4.1 and turn them into objects. If a field in a structure is identified, we turn the whole structure into . Afterward, Mira finds all pointers pointing to objects via forward dataflow analysis (lattice static-single-assignment, or SSA-based, analysis [8]) and type-based alias analysis [48]. These pointers all become pointers, and we convert the original memory accesses to the corresponding operations.

Afterward, we perform an SSA-based backward analysis to find all the functions where an pointer is passed as a parameter. If a function only accesses objects, stack variables, and heap variables allocated and released within the function scope, then we mark the function as . Note that the same function may be called with a non-remote pointer (*i.e.*, pointing to a local object). In this case, we create another version of the function definition that is not .

As the above backward and forward analysis involves the whole program, we avoid

invoking them as much as possible by storing analysis results, including the relationships between functions and objects and each function's references to objects. Later compiler optimizations could reuse these results without going through the costly whole-program analysis again.

Implementing `.alloc`. We use the combination of a local allocator and a remote allocator to implement the allocation of memory space on the far memory node. The remote allocator works like a low-level systems allocator (*e.g.*, `mmap` in Linux) and performs the actual memory allocation at far memory. The local allocator acquires allocated far-memory addresses from the remote allocator and buffers the addresses locally; so it works like an allocator in a language library (*e.g.*, `malloc` in clib). When a `.alloc` is called, the local allocator first checks if there is a buffered memory address range that is no smaller than the allocated size. If so, it directly assigns one to the allocation site. Otherwise, it asks the remote allocator for more addresses. As the allocated addresses are the virtual memory addresses at a far-memory node, our RDMA-based network stack can use them to perform one-sided accesses directly (§3.4.4).

Loading an pointer from far memory. We now explain how Mira dereferences an pointer. Initially, an pointer has the value of an allocated far-memory address for a remotable memory space. When an `.load` happens, Mira first checks if the data the pointer points to has been fetched to the local cache already by searching for the far-memory address in the designated local cache section. If not, Mira fetches the data object from far memory and places it in the section. For the next step of this case or for the cache-hit case, we set the section ID and the offset of the object within the section as the value of the pointer, with the former occupying the highest 16 bits and the latter occupying the lower 48 bits.After fetching or a cache hit, we generate a cache token that contains the section ID and the offset of this object within the corresponding section. The token can be reused to avoid repetitive mapping from a far-mem remote address to a cache line slot within a local section. Then to access the actual data, we map the section ID and offset to the virtual memory address of this cache line plus an offset within the line. This is the virtual memory address seen by the local node MMU, which performs the actual memory access. The

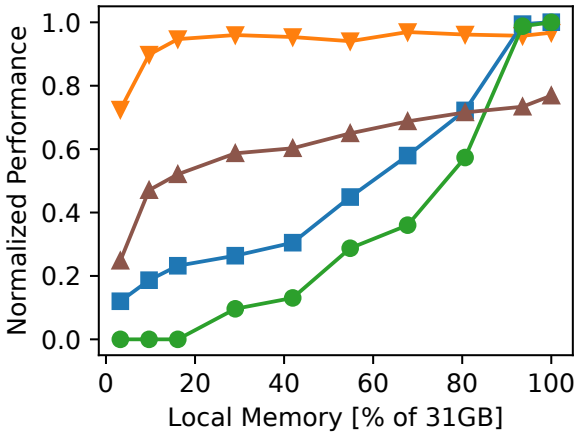

Figure 2.16. DataFrame Performance.

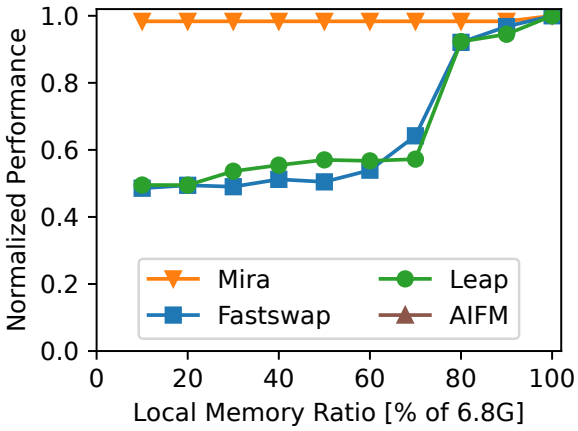

Figure 2.17. GPT2 Performance.

Mira compiler generates corresponding code for all the above steps during compilation.

Pointers to both local and objects. An pointer could be set to point to a object or a local object at runtime in different executions (*e.g.*, based on an `if` condition). A potential problem of such cases is that the pointer will have a normal memory address when pointing to a local object but address constructed as section ID and offset when pointing to a object. If we use the same process to dereference an pointer by locating the cache section and offset, accesses to local objects would be wrong. To solve this problem, we use a simple method: reserving a dummy, non-existent cache section (section 0, as the highest 16 bits for normal addresses are 0) to represent all pointers that point to local objects. When Mira finds a cache section ID zero

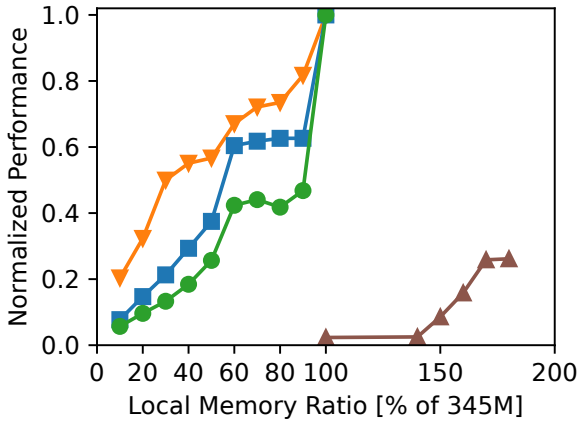

Figure 2.18. MCF Performance.

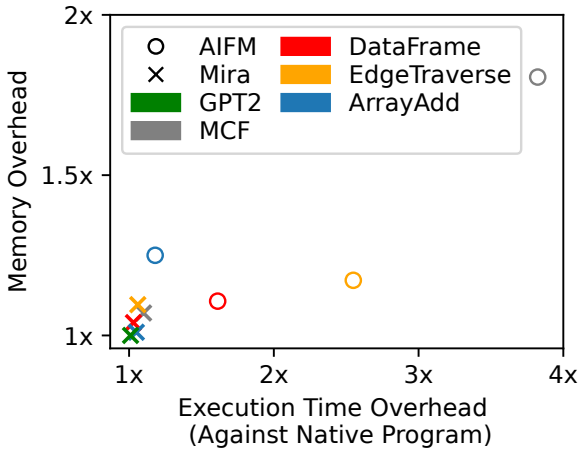

Figure 2.19. Runtime Overhead.

during dereferencing, it treats it as a local object and maps it to the proper local address.

Generating offloaded function binaries. At compile time, Mira turns accesses within the offloaded function into raw memory accesses and pointers into raw pointers, as the function will run on the node that contains the objects. The pointer addresses will be assigned by the remote allocator in the remote virtual address space. On the local node side, we implement the call of a function as an RPC call. To ensure that a function sees the up-to-date objects during its execution, we flush all cached objects that the function accesses to far-memory before calling the function.

Behavior Analysis

For the performance-critical sections we identified, Miraperforms a detailed analysis of memory operations, concerning the range of addresses that will be accessed in each section. We use memory dependency analysis [125] together with scalar evolution [32] to reason about memory accesses and their patterns within a code block (access address sequence, granularity, read/write, possible batching). We further analyze memory accesses across code blocks and function boundaries. For example, if addresses touched within a basic block suggest certain locality, we can batch multiple pointer dereferences within that block to reduce the runtime overhead. If this block happens to be the body of a loop, the address representation at the loop level will guide our prefetch optimization and reveal batching opportunities across iterations.

2.5.3 Cache Section Implementation

Fully-associative cache. We maintain remote-address-to-physical-address maps and a list of available free physical cache lines for fully associative caches. The former is used for cache lookup, while the latter is used for cache insertion. For our compiler-inserted prefetch and eviction hints, we implement the actual operations in our runtime. Additionally, we implement an approximation of LRU eviction using active and inactive lists for when an on-demand eviction is needed.

Swap-based cache section. Different from other sections that use compiler-generated code for cache accesses, the swap cache transparently executes the original code via our implemented user-space swap system (on top of Linux `userfaultfd` [24]). The line size in the swap cache is 4 KB, consistent with OS default page size. Miramanages a physical page pool in RDMA region for the swap section. Unlike other sections, the mapping between virtual addresses to physical pages in the swap section is dynamic. Mira sets up, tears down, or changes mappings when there are `userfaultfd` events, prefetching operations, or eviction hints. Mira evicts a page based on an approximate global LRU policy.

2.6 Evaluation

We evaluate Mira on a Cloudlab [42] cluster of eight c6220 servers, each equipped with two 8-core Intel Xeon E5-2560 CPUs (2.6 GHz), 64 GB RAM, and a 50 Mellanox FDR-CX3 NIC with 50 Infiniband network.

Applications. We select three applications to evaluate Mira: *DataFrame*, *MCF*, and *GPT-2 inference*, representing common code patterns (*e.g.*, data access pattern, threading model, etc.) and common datacenter application types (data analytic, ML inference, graph processing), being open sourced, and having fairly large memory consumption.

DataFrame [75] is a data analytic system written in 24.3K LOC C++. The Dataframe system provides a set of data analytic operations, such as filtering, grouping, etc., on a collection of named columns called a DataFrame. When operating on large data sets, DataFrame can be both compute and memory intensive, making it a good candidate for far memory.

GPT-2 [126, 153] is a transformer-based [168] large language machine-learning model with 100M to 1.5B parameters. We perform GPT-2 inference on ONNX [13], an open AI ecosystem that is compatible with MLIR [12]. The MLIR representation of GPT-2 inference on ONNX has more than 36K lines of code. We run this inference on sequences of 256-token length with a batch size of 64 in a CPU-based far memory environment. Both industry and academia have adopted the use of CPU to perform large machine learning model inference [106, 9, 130], as GPU is not always available (*e.g.*, in serverless computing services). A common technique used by transformer inference is to cache computed values called keys and values to avoid recomputation for better inference latency. Key-value caches consume device memory that can be several times bigger than the model itself [136]. Instead of manually figuring out what data to place in far memory, Mira automatically identifies key-value data for far memory.

MCF [21] is a benchmark from the SPEC 2006 benchmark suites [71]. MCF is derived from a program used for single-depot vehicle scheduling in public transportation and performs graph-based computation. It is written in C and contains 1.8K LOC. Even though MCF is a

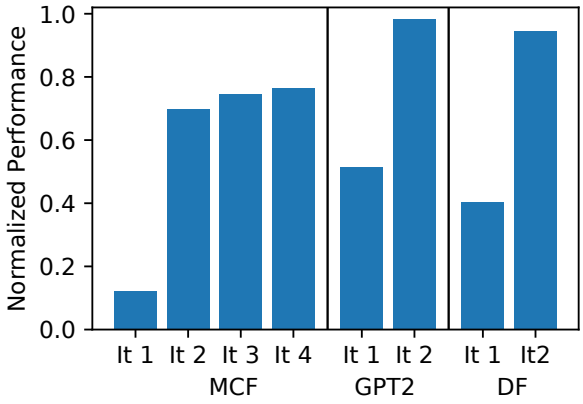

Figure 2.20. Iterative Optimization with Applications.

smaller application than DataFrame and GPT-2 inference, it is representative of graph-processing applications that are common in data centers and can benefit from far memory.

Systems in comparison. We compare Mira to three systems: AIFM [143], FastSwap [18], and Leap [16]. AIFM is a far-memory system that introduces a new programming model. We use AIFM's DataFrame implementation for DataFrame and its array library for MCF. FastSwap is a Linux-based optimized swap system for far memory. Leap is a Linux-based swap system that performs majority-based prefetching.

2.7 Discussion

Tiered memory hierarchy. Even though our implementation of Mira focuses on RDMA-based remote memory, our general designs apply to a broad definition of far memory, including CXL-based memory pools, local- or remote-node persistent memory, and slower storage layers. In general, these technologies could form a *tiered memory hierarchy*, where accesses to each tier could have a different interface and performance characteristics. Mira's optimizations are guided by far-memory accessing speed and computation power, besides static and dynamic program behaviors. Thus, it could be extended to support and benefit other layers in tiered memory.

Distributed support. Mira currently runs on one compute and one memory node. Supporting multiple memory nodes, or *memory pooling*, can be done via the integration of Mira and a

distributed memory management layer such as the one used in LegoOS [147], where Mira decides what objects and functions to offload and the distributed memory manager decides which memory node to offload them to. Enabling multiple compute nodes is more complex, especially when they share memory. On the one hand, we need to perform profiling and code optimization in a cross-client-aware manner. On the other hand, the compiler and runtime need to handle distributed synchronization and cache coherence.

2.8 Conclusion

We presented Mira, a far-memory platform that co-designed program analysis, compiler, run-time profiling, and run-time systems. By leveraging the unique opportunities of far-memory environments and by overcoming challenges, we show that Mira significantly outperforms existing far-memory systems.

2.9 Acknowledgement

Chapter 2, in full, is a reprint of Zhiyuan Guo, Zijian He, Yiying Zhang, “Mira: A Progam-Behavior-Guided Far Memory System”, SOSP, 2023. The dissertation author was the primary investigator and author author of this paper.

Chapter 3

Clio: A Hardware-Software Co-Designed Disaggregated Memory System

3.1 Introduction

Modern datacenter applications like graph computing, data analytics, and deep learning have an increasing demand for access to large amounts of memory [18]. Unfortunately, servers are facing *memory capacity walls* because of pin, space, and power limitations [73, 80, 177]. Going forward, it is imperative for datacenters to seek solutions that can go beyond what a (local) machine can offer, *i.e.*, using remote memory. At the same time, datacenters are seeing the needs from management and resource utilization perspectives to *disaggregate* resources [161, 170, 40]—separating hardware resources into different network-attached pools that can be scaled and managed independently. These real needs have pushed the idea of memory disaggregation (*Memory Disaggregation* for short): organizing computation and memory resources as two separate network-attached pools, one with compute nodes (*CN*s) and one with memory nodes (*MN*s).

So far, Memory Disaggregation researches have all taken one of two approaches: building/emulating MNs using regular servers [143, 66, 18, 147, 121] or using raw memory devices with no processing power [164, 109, 110, 72]. The fundamental issues of server-based approaches such as RDMA-based systems are the monetary and energy cost of a host server and the inherent performance and scalability limitations caused by the way NICs interact with the

host server's virtual memory system. Raw-device-based solutions have low costs. However, they introduce performance, security, and management problems because when MNs have no processing power, all the data and control planes have to be handled at CNs [164].

Server-based MNs and MNs with no processing power are two extreme approaches of building MNs. We seek a sweet spot in the middle by proposing a hardware-based Memory Disaggregation solution that has the right amount of processing power at MNs. Furthermore, we take a clean-slate approach by starting from the requirements of Memory Disaggregation and designing a Memory Disaggregation-native system.

We built *Clio*, a hardware-based disaggregated memory system. Clio includes a CN-side user-space library called *Clio Libarary* and a new hardware-based MN device called *Clio Memory Board*. Multiple application processes running on different CNs can allocate memory from the same Clio Memory Board, with each process having its own *remote virtual memory address space*. Furthermore, one remote virtual memory address space can span multiple Clio Memory Boards. Applications can perform byte-granularity remote memory read/write and use Clio's synchronization primitives for synchronizing concurrent accesses to shared remote memory .

A key research question in designing Clio is ***how to use limited hardware resources to achieve 100 Gbps, microsecond-level average and tail latency for TBs of memory and thousands of concurrent clients?*** These goals are important and unique for Memory Disaggregation. A good Memory Disaggregation solution should reduce the total CapEx and OpEx costs compared to traditional non-disaggregated systems and thus cannot afford to use large amounts of hardware resources at MNs. Meanwhile, remote memory accesses should have high throughput and low average and tail latency, because even after caching data at CN-local memory, there can still be fairly frequent accesses to MNs and the overall application performance can be impacted if they are slow [57]. Finally, unlike traditional single-server memory, a disaggregated MN should allow many CNs to store large amounts of data so that we only need a few of them to reduce costs and connection points in a cluster. How to achieve each of the above cost, performance, and scalability goals *individually* is relatively well understood. However, achieving

all these seemingly conflicting goals *simultaneously* is hard and previously unexplored.

Our main idea is to ***eliminate state from the MN hardware***. Here, we overload the term "state elimination" with two meanings: 1) the MN can treat each of its incoming requests in isolation even if requests that the client issues can sometimes be inter-dependent, and 2) the MN hardware does not store metadata or deals with it. Without remembering previous requests or storing metadata, an MN would only need a tiny amount of on-chip memory that does not grow with more clients, thereby *saving monetary and energy cost* and achieving *great scalability*. Moreover, without state, the hardware pipeline can be made *smooth* and *performance deterministic*. A smooth pipeline means that the pipeline does not stall, which is only possible if requests do not need to wait for each other. It can then take one incoming data unit from the network every fixed number of cycles (1 cycle in our implementation), achieving constantly *high throughput*. A performance-deterministic pipeline means that the hardware processing does not need to wait for any slower metadata operations and thus has *bounded tail latency*.

Effective as it is, can we really eliminate state from MN hardware? First, as with any memory systems, users of a disaggregate memory system expect it to deliver certain reliability and consistency guarantees (*e.g.*, a successful write should have all its data written to remote memory, a read should not see the intermediate state of a write, etc.). Implementing these guarantees requires proper ordering among requests and involves state even on a single server. The network separation of disaggregated memory would only make matters more complicated. Second, quite a few memory operations involve metadata, and they too need to be supported by disaggregated memory. Finally, many memory and network functionalities are traditionally associated with a client process and involve per-process/client metadata (*e.g.*, one page table per process, one connection per client, etc.). Overcoming these challenges require the re-design of traditional memory and network systems.

Our first approach is to separate the metadata/control plane and the data plane, with the former running as software on a low-power ARM-based SoC at MN and the latter in hardware at MN. Metadata operations like memory allocation usually need more memory but are rarer (thus

not as performance critical) compared to data operations. A low-power SoC's computation speed and its local DRAM are sufficient for metadata operations. On the other hand, data operations (*i.e.*, all memory accesses) should be fast and are best handled purely in hardware. Even though the separation of data and control plane is a common technique that has been applied in many areas [65, 97, 133], a separation of memory system control and data planes has not been explored before and is not easy, as we will show in this paper.

Our second approach is to re-design the memory and networking data plane so that most state can be managed only at the CN side. Our observation here is that the MN only *responds* to memory requests but never *initiates* any. This CN-request-MN-respond model allows us to use a custom, connection-less reliable transport protocol that implements almost all transport-layer services and state at CNs, allowing MNs to be free from traditional transport-layer processing. Specifically, our transport protocol manages request IDs, transport logic, retransmission buffer, congestion, and incast control all at CNs. It provides reliability by ordering and retrying an entire memory request at the CN side. As a result, the MN does not need to worry about per-request state or inter-request ordering and only needs a tiny amount of hardware resources which do not grow with the number of clients.

With the above two approaches, the hardware can be largely simplified and thus cheaper, faster, and more scalable. However, we found that ***complete state elimination at MNs is neither feasible nor ideal***. To ensure correctness, the MN has to maintain some state (*e.g.*, to deal with non-idempotent operations). To ensure good data-plane performance, not every operation that involves state should be moved to the low-power SoC or to CNs. Thus, our approach is to eliminate as much state as we can without affecting performance or correctness and to carefully design the remaining state so that it causes small and bounded space and performance overhead.

For example, we perform paging-based virtual-to-physical memory address mapping and access permission checking at the MN hardware pipeline, as these operations are needed for every data access. Page table is a kind of state that could potentially cause performance and scalability issues but has to be accessed in the data path. We propose a new overflow-free, hash-based page

table design where 1) all page table lookups have bounded and low latency (at most one DRAM access time in our implementation), and 2) the total size of all page table entries does not grow with the number of client processes. As a result, even though we cannot eliminate page table from the MN hardware, we can still meet our cost, performance, or scalability requirements.

Another data-plane operation that involves metadata is page fault handling, which is a relatively common operation because we allocate physical memory on demand. Today's page fault handling process is slow and involves metadata for physical memory allocation. We propose a new mechanism to handle page faults in hardware and finish all the handling within bounded hardware cycles. We make page fault handling performance deterministic by moving physical memory allocation operations to software running at the SoC. We further move these allocation operations off the performance-critical path by pre-generating free physical pages to a fix-sized buffer that the hardware pipeline can pull when handling page faults.

We prototyped Clio Memory Board with a small set of Xilinx ZCU106 MPSoC FPGA boards [178] and built three applications using Clio: a FaaS-style image compression utility, a radix-tree index, and a key-value store. We compared Clio with native RDMA, two RDMA-based disaggregated/remote memory systems [164, 85], a software emulation of hardware-based disaggregated memory [147], and a software-based SmartNIC [116]. Clio scales much better and has orders of magnitude lower tail latency than RDMA, while achieving similar throughput and median latency as RDMA (even with the slower FPGA frequency in our prototype). Clio has 1.1× to 3.4× energy saving compared to CPU-based and SmartNIC-based disaggregated memory systems and is 2.7× faster than SmartNIC solutions. Clio is publicly available at https://github.com/WukLab/Clio.

3.2 Goals and Related Works

Resource disaggregation separates different types of resources into different pools, each of which can be independently managed and scaled. Applications can allocate resources from

any node in a resource pool, resulting in tight resource packing. Because of these benefits, many datacenters have adopted the idea of disaggregation, often at the storage layer [52, 40, 170, 20, 19, 17, 160]. With the success of disaggregated storage, researchers in academia and industry have also sought ways to disaggregate memory (and persistent memory) [109, 26, 79, 110, 147, 148, 128, 164, 143, 18, 66, 171, 122]. Different from storage disaggregation, Memory Disaggregation needs to achieve at least an order of magnitude higher performance and it should offer a byte-addressable interface. Thus, Memory Disaggregation poses new challenges and requires new designs. This section discusses the requirements of Memory Disaggregation and why existing solutions cannot fully meet them.

3.2.1 Memory Disaggregation Design Goals

In general, Memory Disaggregation has the following features, some of which are hard requirements while others are desired goals.

R1: Hosting large amounts of memory with high utilization. To keep the number of memory devices and total cost of a cluster low, each MN should host hundreds GBs to a few TBs of memory that is expected to be close to fully utilized. To most efficiently use the disaggregated memory, we should allow applications to create and access *disjoint* memory regions of arbitrary sizes at MN.

R2: Supporting a huge number of concurrent clients. To ensure tight and efficient resource packing, we should allow many (*e.g.*, thousands of) client processes running on tens of CNs to access and share an MN. This scenario is especially important for new data-center trends like serverless computing and microservices where applications run as large amounts of small units.

R3: Low-latency and high-throughput. We envision future systems to have a new memory hierarchy, where disaggregated memory is larger and slower than local memory but still faster than storage. Since Memory Disaggregation is network-based, a reasonable performance target of it is to match the state-of-the-art network speed, *i.e.*, 100 Gbps throughput (for bigger requests) and sub-2 μs median end-to-end latency (for smaller requests).

R4: Low tail latency. Maintaining a low tail latency is important in meeting service-level objectives (SLOs) in data centers. Long tails like RDMA's 16.8 *ms* remote memory access can be detrimental to applications that are short running (*e.g.*, serverless computing workloads) or have large fan-outs or big DAGs (because they need to wait for the slowest step to finish) [45].

R5: Protected memory accesses. As an MN can be shared by multi-tenant applications running at CNs, we should properly isolate memory spaces used by them. Moreover, to prevent buggy or malicious clients from reading/writing arbitrary memory at MNs, we should not allow the direct access of MNs' physical memory from the network and MNs should check the access permission.

R6: Low cost. A major goal and benefit of resource disaggregation is cost reduction. A good Memory Disaggregation system should have low *overall* CapEx and OpEx costs. Such a system thus should not 1) use expensive hardware to build MNs, 2) consume huge energy at MNs, and 3) add more costs at CNs than the costs saved at MNs.

R7: Flexible. With the fast development of datacenter applications, hardware, and network, a sustainable Memory Disaggregation solution should be flexible and extendable, for example, to support high-level APIs like pointer chasing [143, 15], to offload some application logic to memory devices [143, 149], or to incorporate different network transports [119, 70, 23] and congestion control algorithms [98, 151, 108].

3.2.2 Server-Based Disaggregated Memory

Memory Disaggregation research so far has mainly taken a server-based approach by using regular servers as MNs [66, 18, 171, 147, 143, 121, 50], usually on top of RDMA. The common limitation of these systems is their reliance on a host server and the resulting CPU energy costs, both of which violate **R6**.

RDMA is what most server-based Memory Disaggregation solutions are based on, with some using RDMA for swapping memory between CNs and MNs [66, 18, 171] and some using RDMA for explicitly accessing MNs [143, 121, 50]. Although RDMA has low average latency and high

throughput, it has a set of scalability and tail-latency problems.

A process (P_M) running at an MN needs to allocate memory in its virtual memory address space and *register* the allocated memory (called a memory region, or MR) with the RDMA NIC (RNIC). The host OS and MMU set up and manage the page table that maps P_M's virtual addresses (*VA*s) to physical memory addresses (*PA*s). To avoid always accessing host memory for address mapping, RNICs cache page table entries (PTEs), but when more PTEs are accessed than what this cache can hold, RDMA performance degrades significantly (Figure 3.5 and [50, 165]). Similarly, RNICs cache MR metadata and incur degraded performance when the cache is full. Thus, RDMA has serious performance issues with either large memory (PTEs) or many disjoint memory regions (MRs), violating **R1**. Moreover, RDMA uses a slow way to support on-demand allocation: the RNIC interrupts the host OS for handling page faults. From our experiments, a faulting RDMA access is 14100× slower than a no-fault access (violating **R4**).

To mitigate the above performance and scalability issues, most RDMA-based systems today [50, 165] preallocate a big MR with huge pages and pin it in physical memory. This results in inefficient memory space utilization and violates **R1**. Even with this approach, there can still be a scalability issue (**R2**), as RDMA needs to create at least one MR for each protection domain (*i.e.*, each client).

In addition to problems caused by RDMA's memory system design, reliable RDMA, the mode used by most Memory Disaggregation solutions, suffers from a connection queue pair (QP) scalability issue, also violating **R2**. Finally, today's RNICs violate **R7** because of their rigid one-sided RDMA interface and the close-sourced, hardware-based transport implementation. Solutions like 1RMA [151] and IRN [118] mitigate the above issues by either onloading part of the transport back to software or proposing a new hardware design.

LegoOS [147], our own previous work, is a distributed operating system designed for resource disaggregation. Its MN includes a virtual memory system that maps VAs of application processes running at CNs to MN PAs. Clio's MN performs the same type of address translation. However, LegoOS emulates MN devices using regular servers and we built its virtual memory system

in software, which has a stark difference from a hardware-based virtual memory system. For example, LegoOS uses a thread pool that handles incoming memory requests by looking up a hash table for address translation and permission checking. This software approach is the major performance bottleneck in LegoOS (§3.7), violating **R3**. Moreover, LegoOS uses RDMA for its network communication hence inheriting its limitations.

3.2.3 Physical Disaggregated Memory

One way to build Memory Disaggregation without a host server is to treat it as raw, physical memory, a model we call *PDM*. The PDM model has been adopted by a set of coherent interconnect proposals [59, 44], HPE's Memory-Driven Computing project [72, 53, 169, 74]. A recent disaggregated hashing system [184] and our own recent work on disaggregated key-value systems [164] also adopt the PDM model and emulate remote memory with regular servers. To prevent applications from accessing raw physical memory, these solutions add an indirection layer at CNs in hardware [59, 44] or software [164, 184] to map client process VAs or keys to MN PAs.

There are several common problems with all the PDM solutions. First, because MNs in PDM are raw memory, CNs need multiple network round trips to access an MN for complex operations like pointer chasing and concurrent operations that need synchronization [164], violating **R3** and **R7**. Second, PDM requires the client side to manage disaggregated memory. For example, CNs need to coordinate with each other or use a global server [164] to perform tasks like memory allocation. Non-MN-side processing is much harder, performs worse compared to memory-side management (violating **R3**), and could even result in higher overall costs because of the high computation added at CNs (violating **R6**). Third, exposing physical memory makes it hard to provide security guarantees (**R5**), as MNs have to authenticate that every access is to a legit physical memory address belonging to the application. Finally, all existing PDM solutions require physical memory pinning at MNs, causing memory wastes and violating **R1**.

In addition to the above problems, none of the coherent interconnects or HPE's Memory-

Driven Computing have been fully built. When they do, they will require new hardware at all endpoints and new switches. Moreover, the interconnects automatically make caches at different endpoints coherent, which could cause performance overhead that is not always necessary (violating **R3**).

Besides the above PDM works, there are also proposals to include some processing power in between the disaggregated memory layer and the computation layer. soNUMA [123] is a hardware-based solution that scales out NUMA nodes by extending each NUMA node with a hardware unit that services remote memory accesses. Unlike Clio which physically separates MNs from CNs across generic data-center networks, soNUMA still bundles memory and CPU cores, and it is a single-server solution. Thus, soNUMA works only on a limited scale (violating **R2**) and is not flexible (violating **R7**). MIND [103], a concurrent work with Clio, proposes to use a programmable switch for managing coherence directories and memory address mappings between compute nodes and memory nodes. Unlike Clio which adds processing power to every MN, MIND's single programmable switch has limited hardware resources and could be the bottleneck for both performance and scalability.

3.3 Clio Overview

Clio co-designs software with hardware, CNs with MNs, and network stack with virtual memory system, so that at the MN, the entire data path is handled in hardware with high throughput, low (tail) latency, and minimal hardware resources. This section gives an overview of Clio's interface and architecture (Figure 3.2).

3.3.1 Clio Interface

Similar to recent Memory Disaggregation proposals [143, 22], our current implementation adopts a non-transparent interface where applications (running at CNs) allocate and access disaggregated memory via explicit API calls. Doing so gives users opportunities to perform application-specific performance optimizations. By design, Clio's APIs can also be called by a

```
1  /* Alloc one remote page. Define a remote lock */
2  #define PAGE_SIZE (1<<22)
3  void *remote_addr = ralloc(PAGE_SIZE);
4  ras_lock lock;
5
6  /* Acquire lock to enter critical section.
7     Do two AYSNC writes then poll completion. */
8  void thread1(void *) {
9   rlock(lock);
10  e[0]=rwrite(remote_addr, local_wbuf1,len, ASYNC);
11  e[1]=rwrite(remote_addr+len, local_wbuf2,len, ASYNC);
12  runlock(lock);
13  rpoll(e, 2);
14 }
15
16 /* Synchronously read from remote */
17 void thread2(void *) {
18  rlock(lock);
19  rread(remote_addr, local_rbuf, len, SYNC);
20  runlock(lock);
21 }
```

Figure 3.1. Example of Using Clio.

runtime like the AIFM runtime [143] or by the kernel/hardware at CN like LegoOS' pComponent [147] to support a transparent interface and allow the use of unmodified user applications. We leave such extension to future work.

Apart from the regular (local) virtual memory address space, each process has a separate ***R**emote virtual memory **A**ddress **S**pace* (*RAS* for short). Each application process has a unique global *PID* across all CNs which is assigned by Clio when the application starts. Overall, programming in RAS is similar to traditional multi-threaded programming except that memory read and write are explicit and that processes running on different CNs can share memory in the same RAS. Figure 3.1 illustrates the usage of Clio with a simple example.

An application process can perform a set of virtual memory operations in its RAS, including `ralloc`, `rfree`, `rread`, `rwrite`, and a set of atomic and synchronization primitives (*e.g.*, `rlock`, `runlock`, `rfence`). `ralloc` works like `malloc` and returns a VA in RAS. `rread` and `rwrite` can then be issued to any allocated VAs. As with the traditional virtual memory

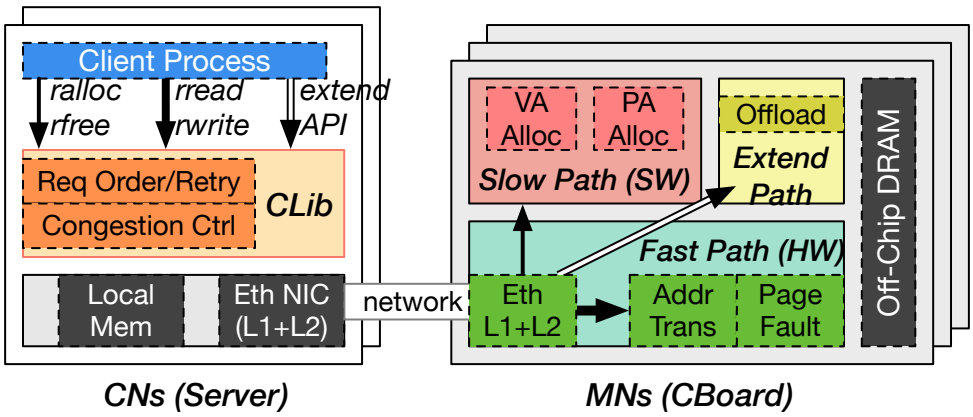

Figure 3.2. Clio Architecture.

interface, allocation and access in RAS are in byte granularity. We offer *synchronous* and *asynchronous* options for `ralloc`, `rfree`, `rread`, and `rwrite`.

Intra-thread request ordering. Within a thread, synchronous APIs follow strict ordering. An application thread that calls a synchronous API blocks until it gets the result. Asynchronous APIs are non-blocking. A calling thread proceeds after calling an asynchronous API and later calls `rpoll` to get the result. Asynchronous APIs follow a release order. Specifically, asynchronous APIs may be executed out of order as long as 1) all asynchronous operations before a `rrelease` complete before the `rrelease` returns, and 2) `rrelease` operations are strictly ordered. On top of this release order, we guarantee that there is no concurrent asynchronous operations with dependencies (Write-After-Read, Read-After-Write, Write-After-Write) and target the same page. The resulting memory consistency level is the same as architecture like ARMv8 [25]. In addition, we also ensure consistency between metadata and data operations, by ensuring that potentially conflicting operations execute synchronously in the program order. For example, if there is an ongoing `rfree` request to a VA, no read or write to it can start until the `rfree` finishes. Finally, failed or unresponsive requests are transparently retried, and they follow the same ordering guarantees.

Thread synchronization and data coherence. Threads and processes can share data even when they are not on the same CN. Similar to traditional concurrent programming, Clio threads can

use synchronization primitives to build critical sections (*e.g.*, with `rlock`) and other semantics (*e.g.*, flushing all requests with `rfence`).

An application can choose to cache data read from `rread` at the CN (*e.g.*, by maintaining `local_rbuf` in the code example). Different processes sharing data in a RAS can have their own cached copies at different CNs. Similar to [147], Clio does not make these cached copies coherent automatically and lets applications choose their own coherence protocols. We made this deliberate decision because automatic cache coherence on every read/write would incur high performance overhead with commodity Ethernet infrastructure and application semantics could reduce this overhead.

3.3.2 Clio Architecture

In Clio (Figure 3.2), CNs are regular servers each equipped with a regular Ethernet NIC and connected to a top-of-rack (ToR) switch. MNs are our customized devices directly connected to a ToR switch. Applications run at CNs on top of our user-space library called *Clio Libarary*. It is in charge of request ordering, request retry, congestion, and incast control.

By design, an MN in Clio is a Clio Memory Board consisting of an ASIC which runs the hardware logic for all data accesses (we call it the *fast path* and prototyped it with FPGA), an ARM processor which runs software for handling metadata and control operations (*i.e.*, the *slow path*), and an FPGA which hosts application computation offloading (*i.e.*, the *extend path*). An incoming request arrives at the ASIC and travels through standard Ethernet physical and MAC layers and a Match-and-Action-Table (MAT) that decides which of the three paths the request should go to based on the request type. If the request is a data access (fast path), it stays in the ASIC and goes through a hardware-based virtual memory system that performs three tasks in the same pipeline: address translation, permission checking, and page fault handling (if any). Afterward, the actual memory access is performed through the memory controller, and the response is formed and sent out through the network stack. Metadata operations such as memory allocation are sent to the slow path. Finally, customized requests with offloaded computation are

handled in the extend path.

3.4 Clio Design

This section presents the design challenges of building a hardware-based Memory Disaggregation system and our solutions.

3.4.1 Design Challenges and Principles

Building a hardware-based Memory Disaggregation platform is a previously unexplored area and introduces new challenges mainly because of restrictions of hardware and the unique requirements of Memory Disaggregation.

Challenge 1: The hardware should avoid maintaining or processing complex data structures, because unlike software, hardware has limited resources such as on-chip memory and logic cells. For example, Linux and many other software systems use trees (*e.g.*, the vma tree) for allocation. Maintaining and searching a big tree data structure in hardware, however, would require huge on-chip memory and many logic cells to perform the look up operation (or alternatively use fewer resources but suffer from performance loss).

Challenge 2: Data buffers and metadata that the hardware uses should be minimal and have bounded sizes, so that they can be statically planned and fit into the on-chip memory. Unfortunately, traditional software approaches involve various data buffers and metadata that are large and grow with increasing scale. For example, today's reliable network transports maintain per-connection sequence numbers and buffer unacknowledged packets for packet ordering and retransmission, and they grow with the number of connections. Although swapping between on-chip and off-chip memory is possible, doing so would increase both tail latency and hardware logic complexity, especially under large scale.

Challenge 3: The hardware pipeline should be deterministic and smooth, *i.e.*, it uses a bounded, known number of cycles to process a data unit, and for each cycle, the pipeline can take in one new data unit (from the network). The former would ensure low tail latency, while the

latter would guarantee a throughput that could match network line rate. Another subtle benefit of a deterministic pipeline is that we can know the maximum time a data unit stays at MN, which could help bound the size of certain buffers (*e.g.*, §3.4.5). However, many traditional hardware solutions are not designed to be deterministic or smooth, and we cannot directly adapt their approaches. For example, traditional CPU pipelines could have stalls because of data hazards and have non-deterministic latency to handle memory instructions.

To confront these challenges, we took a clean-slate approach by designing Clio's virtual memory system and network system with the following principles that all aim to eliminate state in hardware or bound their performance and space overhead.

Principle 1: Avoid state whenever possible. Not all state in server-based solutions is necessary if we could redesign the hardware. For example, we get rid of RDMA's MR indirection and its metadata altogether by directly mapping application process' RAS VAs to PAs (instead of to MRs then to PAs).

Principle 2: Moving non-critical operations and state to software and making the hardware fast path deterministic. If an operation is non-critical and it involves complex processing logic and/or metadata, our idea is to move it to the software slow path running in an ARM processor. For example, VA allocation (`ralloc`) is expected to be a rare operation because applications know the disaggregated nature and would typically have only a few large allocations during the execution. Handling `ralloc`, however, would involve dealing with complex allocation trees. We thus handle `ralloc` and `rfree` in the software slow path. Furthermore, in order to make the fast path performance deterministic, we *decouple* all slow-path tasks from the performance-critical path by *asynchronously* performing them in the background.

Principle 3: Shifting functionalities and state to CNs. While hardware resources are scarce at MNs, CNs have sufficient memory and processing power, and it is faster to develop functionalities in CN software. A viable solution is to shift state and functionalities from MNs to CNs. The key question here is how much and what to shift. Our strategy is to shift functionalities to CNs only if doing so 1) could largely reduce hardware resource consumption at MNs, 2) does not slow

down common-case foreground data operations, 3) does not sacrifice security guarantees, and 4) adds bounded memory space and CPU cycle overheads to CNs. As a tradeoff, the shift may result in certain uncommon operations (*e.g.*, handling a failed request) being slower.

Principle 4: Making off-chip data structures efficient and scalable. Principles 1 to 3 allow us to reduce MN hardware to only the most essential functionalities and state. We store the remaining state in off-chip memory and cache a fixed amount of them in on-chip memory. Different from most caching solutions, our focus is to make the access to off-chip data structure fast and scalable, *i.e.*, all cache misses have bounded latency regardless of the number of client processes accessing an MN or the amount of physical memory the MN hosts.

Principle 5: Making the hardware fast path smooth by treating each data unit independently at MN. If data units have dependencies (*e.g.*, must be executed in a certain order), then the fast path cannot always execute a data unit when receiving it. To handle one data unit per cycle and reach network line rate, we make each data unit independent by including all the information needed to process a unit in it and by allowing MNs to execute data units in any order that they arrive. To deliver our consistency guarantees, we opt for enforcing request ordering at CNs before sending them out.

The rest of this section presents how we follow these principles to design Clio's three main functionalities: memory address translation and protection, page fault handling, and networking. We also briefly discuss our offloading support.

3.4.2 Scalable, Fast Address Translation

Similar to traditional virtual memory systems, we use fix-size pages as address allocation and translation unit, while data accesses are in the granularity of byte. Despite the similarity in the goal of address translation, the radix-tree-style, per-address space page table design used by all current architectures [152] does not fit Memory Disaggregation for two reasons. First, each request from the network could be from a different client process. If each process has its own page table, MN would need to cache and look up many page table roots, causing additional

overhead. Second, a multi-level page table design requires multiple DRAM accesses when there is a translation lookaside buffer (TLB) miss [179]. TLB misses will be much more common in a Memory Disaggregation environment, since with more applications sharing an MN, the total working set size is much bigger than that in a single-server setting, while the TLB size in an MN will be similar or even smaller than a single server's TLB (for cost concerns). To make matters worse, each DRAM access is more costly for systems like RDMA NIC which has to cross the PCIe bus to access the page table in main memory [163, 120].

Flat, single page table design (Principle 4). We propose a new *overflow-free* hash-based page table design that sets the total page table size according to the physical memory size and bounds *address translation to at most one DRAM access*. Specifically, we store *all* page table entries (PTEs) from *all* processes in a single hash table whose size is proportional to the physical memory size of an MN. The location of this page table is fixed in the off-chip DRAM and is known by the fast path address translation unit, thus avoiding any lookups. As we anticipate applications to allocate big chunks of VAs in their RAS, we use huge pages and support a configurable set of page sizes. With the default 4 MB page size, the hash table consumes only 0.4% of the physical memory.

The hash value of a VA and its PID is used as the index to determine which hash bucket the corresponding PTE goes to. Each hash bucket has a fixed number of (K) slots. To access the page table, we always fetch the entire bucket including all K slots in a single DRAM access.

A well-known problem with hash-based page table design is hash collisions that could overflow a bucket. Existing hash-based page table designs rely on collision chaining [30] or open addressing [179] to handle overflows, both require multiple DRAM accesses or even costly software intervention. In order to bound address translation to at most one DRAM access, we use a novel technique to avoid hash overflows at *VA allocation time*.

VA allocation (Principle 2). The slow path software handles `ralloc` requests and allocates VA. The software allocator maintains a per-process VA allocation tree that records allocated VA ranges and permissions, similar to the Linux vma tree [91]. To allocate size k of VAs, it first

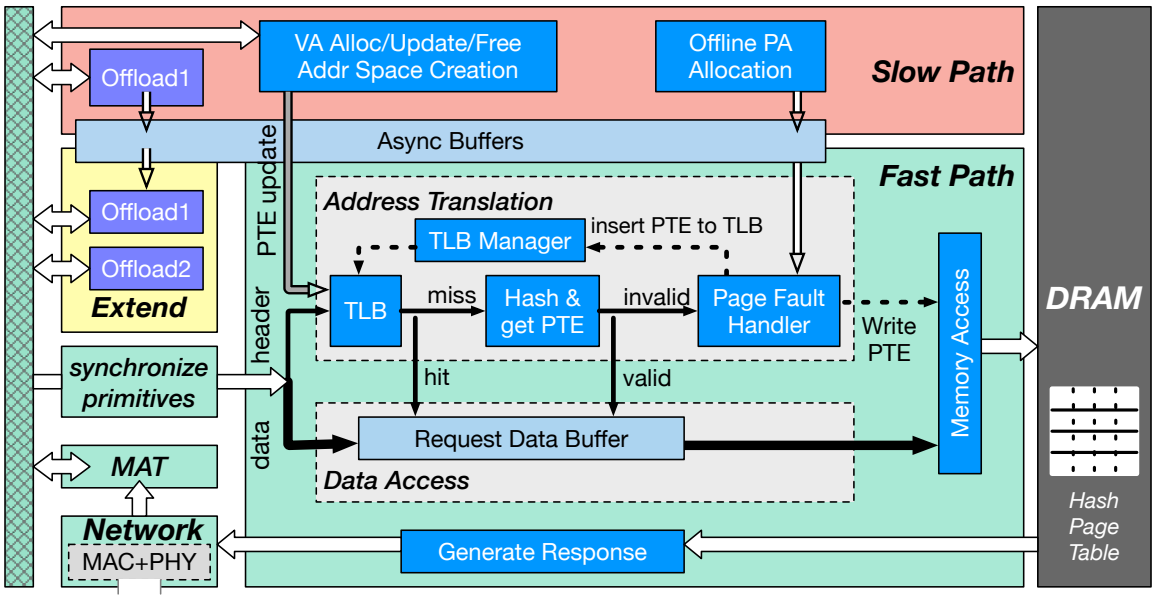

Figure 3.3. Clio Memory Board Design.

finds an available address range of size k in the tree. It then calculates the hash values of the virtual pages in this address range and checks if inserting them to the page table would cause any hash overflow. If so, it does another search for available VAs. These steps repeat until it finds a valid VA range that does not cause hash overflow.

Our design trades potential retry overhead at allocation time (at the slow path) for better run-time performance and simpler hardware design (at the fast path). This overhead is manageable because 1) each retry takes only a few microseconds with our implementation (§4.5), 2) we employ huge pages, which means fewer pages need to be allocated, 3) we choose a hash function that has very low collision rate [175], and 4) we set the page table to have extra slots ($2\times$ by default) which absorbs most overflows. We find no conflicts when memory is below half utilized and has only up to 60 retries when memory is close to full (Figure 3.13).

TLB. Clio implements a TLB in a fix-sized on-chip memory area and looks it up using content-addressable-memory in the fast path. On a TLB miss, the fast path fetches the PTE from off-chip memory and inserts it to the TLB by replacing an existing TLB entry with the LRU policy. When updating a PTE, the fast path also updates the TLB, in a way that ensures the consistency of

inflight operations.

Limitation. A downside of our overflow-free VA allocation design is that it cannot guarantee that a specific VA can be inserted into the page table. This is not a problem for regular VA allocation but could be problematic for allocations that require a fixed VA (*e.g.*, `mmap(MAP_FIXED)`). Currently, Clio finds a new VA range if the user-specified range cannot be inserted into the page table. Applications that must map at fixed VAs (*e.g.*, libraries) will need to use CN-local memory.

3.4.3 Low-Tail-Latency Page Fault Handling

A key reason to disaggregate memory is to consolidate memory usages on less DRAM so that memory utilization is higher and the total monetary cost is lower (**R1**). Thus, remote memory space is desired to run close to full capacity, and we allow memory over-commitment at an MN, necessitating page fault handling. Meanwhile, applications like JVM-based ones allocate a large heap memory space at the startup time and then slowly use it to allocate smaller objects [67]. Similarly, many existing far-memory systems [164, 143, 50] allocate a big chunk of remote memory and then use different parts of it for smaller objects to avoid frequently triggering the slow remote allocation operation. In these cases, it is desirable for a Memory Disaggregation system to delay the allocation of physical memory to when the memory is actually used (*i.e.*, *on-demand* allocation) or to "reshape" memory [150] during runtime, necessitating page fault handling.

Page faults are traditionally signaled by the hardware and handled by the OS. This is a slow process because of the costly interrupt and kernel-trapping flow. For example, a remote page fault via RDMA costs 16.8 *ms* from our experiments using Mellanox ConnectX-4. To avoid page faults, most RDMA-based systems pre-allocate big chunks of physical memory and pin them physically. However, doing so results in memory wastes and makes it hard for an MN to pack more applications, violating **R1** and **R2**.

We propose to *handle page faults in hardware and with bounded latency*—a *constant three cycles* to be more specific with our implementation of Clio Memory Board. Handling

initial-access faults in hardware is challenging, as initial accesses require PA allocation, which is a slow operation that involves manipulating complex data structures. Thus, we handle PA allocation in the slow path (**Challenge 1**). However, if the fast-path page fault handler has to wait for the slow path to generate a PA for each page fault, it will slow down the data plane.

To solve this problem, we propose an asynchronous design to shift PA allocation off the performance-critical path (**Principle 2**). Specifically, we maintain a set of *free physical page numbers* in an *async buffer*, which the ARM continuously fulfills by finding free physical page addresses and reserving them without actually using the pages. During a page fault, the page fault handler simply fetches a pre-allocated physical page address. Note that even though a single PA allocation operation has a non-trivial delay, the throughput of generating PAs and filling the async buffer is higher than network line rate. Thus, the fast path can always find free PAs in the async buffer in time. After getting a PA from the async buffer and establishing a valid PTE, the page fault handler performs three tasks in parallel: writing the PTE to the off-chip page table, inserting the PTE to the TLB, and continuing the original faulting request. This parallel design hides the performance overhead of the first two tasks, allowing foreground requests to proceed immediately.

A recent work [101] also handles page faults in hardware. Its focus is on the complex interaction with kernel and storage devices, and it is a simulation-only work. Clio uses a different design for handling page faults in hardware with the goal of low tail latency, and we built it in FPGA.

Putting the virtual memory system together. We illustrate how Clio Memory Board's virtual memory system works using a simple example of allocating some memory and writing to it. The first step (`ralloc`) is handled by the slow path, which allocates a VA range by finding an available set of slots in the hash page table. The slow path forwards the new PTEs to the fast path, which inserts them to the page table. At this point, the PTEs are invalid. This VA range is returned to the client. When the client performs the first write, the request goes to the fast path. There will be a TLB miss, followed by a fetch of the PTE. Since the PTE is invalid, the page

fault handler will be triggered, which fetches a free PA from the async buffer and establishes the valid PTE. It will then execute the write, update the page table, and insert the PTE to TLB.

3.4.4 Asymmetric Network Tailored for Memory Disaggregation

With large amounts of research and development efforts, today's data-center network systems are highly optimized in their performance. Our goal of Clio's network system is unique and fits Memory Disaggregation's requirements—minimizing the network stack's hardware resource consumption at MNs and achieving great scalability while maintaining similar performance as today's fast network. Traditional software-based reliable transports like Linux TCP incurs high performance overhead. Today's hardware-based reliable transports like RDMA are fast, but they require a fair amount of on-chip memory to maintain state, *e.g.*, per-connection sequence numbers, congestion state [23], and bitmaps [118, 115], not meeting our low-cost goal.

Our insight is that different from general-purpose network communication where each endpoint can be both the sender (requester) and the receiver (responder) that exchange general-purpose messages, MNs only respond to requests sent by CNs (except for memory migration from one MN to another MN (§3.4.7), in which case we use another simple protocol to achieve the similar goal). Moreover, these requests are all memory-related operations that have their specific properties. With these insights, we design a new network system with two main ideas. Our first idea is to maintain transport logic, state, and data buffers only at CNs, essentially making MNs "transportless" (**Principle 3**). Our second idea is to relax the reliability of the transport and instead enforce ordering and loss recovery at the memory request level, so that MNs' hardware pipeline can process data units as soon as they arrive (**Principle 5**).

With these ideas, we implemented a transport in Clio Libarary at CNs. Clio Libarary bypasses the kernel to directly issue raw Ethernet requests to an Ethernet NIC. CNs use regular, commodity Ethernet NICs and regular Ethernet switches to connect to MNs. MNs include only standard Ethernet physical, link, and network layers and a slim layer for handling corner-case requests (§3.4.5). We now describe our detailed design.

Removing connections with request-response semantics. Connections (*i.e.*, QPs) are a major scalability issue with RDMA. Similar to recent works [119, 151], we make our network system connection-less using request-response pairs. Applications running at CNs directly initiate Clio APIs to an MN without any connections. Clio Libarary assigns a unique request ID to each request. The MN attaches the same request ID when sending the response back. Clio Libarary uses responses as ACKs and matches a response with an outstanding request using the request ID. Neither CNs nor MNs send ACKs.

Lifting reliability to the memory request level. Instead of triggering a retransmission protocol for every lost/corrupted packet at the transport layer, Clio Libarary retries the entire memory request if any packet is lost or corrupted in the sending or the receiving direction. On the receiving path, MN's network stack only checks a packet's integrity at the link layer. If a packet is corrupted, the MN immediately sends a NACK to the sender CN. Clio Libarary retries a memory request if one of three situations happens: a NACK is received, the response from MN is corrupted, or no response is received within a `TIMEOUT` period. In addition to lifting retransmission from transport to the request level, we also lift ordering to the memory request level and allow out-of-order packet delivery (see details in §3.4.5).

CN-managed congestion and incast control. Our goal of controlling congestion in the network and handling incast that can happen both at a CN and an MN is to minimize state at MN. To this end, we build the entire congestion and incast control at the CN in the Clio Libarary. To control congestion, Clio Libarary adopts a simple delay-based, reactive policy that uses end-to-end RTT delay as the congestion signal, similar to recent sender-managed, delay-based mechanisms [117, 98, 151]. Each CN maintains one congestion window, *cwnd*, per MN that controls the maximum number of outstanding requests that can be made to the MN from this CN. We adjust *cwnd* based on measured delay using a standard Additive Increase Multiplicative Decrease (AIMD) algorithm.

To handle incast to a CN, we exploit the fact that the CN knows the sizes of expected responses for the requests that it sends out and that responses are the major incoming traffic to it.

Each Clio Libarary maintains one incast window, *iwnd*, which controls the maximum bytes of expected responses. Clio Libarary sends a request only when both *cwnd* and *iwnd* have room.

Handling incast to an MN is more challenging, as we cannot throttle incoming traffic at the MN side or would otherwise maintain state at MNs. To have CNs handle incast to MNs, we draw inspiration from Swift [98] by allowing *cwnd* to fall below one packet when long delay is observed at a CN. For example, a *cwnd* of 0.1 means that the CN can only send a packet within 10 RTTs. Essentially, this situation happens when the network between a CN and an MN is really congested, and the only way is to slow the sending speed.

3.4.5 Request Ordering and Data Consistency

As explained in §3.3.1, Clio supports both synchronous and asynchronous remote memory APIs, with the former following a sequential, one-at-a-time order in a thread and the latter following a release order in a thread. Furthermore, Clio provides synchronization primitives for inter-thread consistency. We now discuss how Clio achieves these correctness guarantees by presenting our mechanisms for handling intra-request intra-thread ordering, inter-request intra-thread ordering, inter-thread consistency, and retries. At the end, we will provide the rationales behind our design.

One difficulty in designing the request ordering and consistency mechanisms is our relaxed network ordering guarantees, which we adopt to minimize the hardware resource consumption for the network layer at MNs (§3.4.4). On an asynchronous network, it is generally hard to guarantee any type of request ordering when there can be multiple outstanding requests (either multiple threads accessing shared memory or a single thread issuing multiple asynchronous APIs). It is even harder for Clio because we aim to make MN stateless as much as possible. Our general approaches are 1) using CNs to ensure that no two concurrently outstanding requests are dependent on each other, and 2) using MNs to ensure that every user request is only executed once even in the event of retries.

Allowing intra-request packet re-ordering (T1). A request or a response in Clio can contain

multiple link-layer packets. Enforcing packet ordering above the link layer normally requires maintaining state (*e.g.*, packet sequence ID) at both the sender and the receiver. To avoid maintaining such state at MNs, our approach is to deal with packet reordering only at CNs in Clio Libarary (**Principle 3**). Specifically, Clio Libarary splits a request that is bigger than link-layer maximum transmission unit (MTU) into several link-layer packets and attaches a Clio header to each packet, which includes sender-receiver addresses, a request ID, and request type. This enables the MN to treat each packet independently (**Principle 5**). It executes packets as soon as they arrive, even if they are not in the sending order. This out-of-order data placement semantic is in line with RDMA specification [118]. Note that only write requests will be bigger than MTU, and the order of data writing within a write request does not affect correctness as long as proper *inter-request* ordering is followed. When a CN receives multiple link-layer packets belonging to the same request response, Clio Libarary reassembles them before delivering them to the application.

Enforcing intra-thread inter-request ordering at CN (T2). Since only one synchronous request can be outstanding in a thread, there cannot be any inter-request reordering problem. On the other hand, there can be multiple outstanding asynchronous requests. Our provided consistency level disallows concurrent asynchronous requests that are dependent on each other (WAW, RAW, or WAR). In addition, all requests must complete before `rrelease`.

We enforce these ordering requirements at CNs in Clio Libarary instead of at MNs (**Principle 3**) for two reasons. First, enforcing ordering at MNs requires more on-chip memory and complex logic in hardware. Second, even if we enforce ordering at MNs, network reordering would still break end-to-end ordering guarantees.

Specifically, Clio Libarary keeps track of all inflight requests and matches every new request's virtual page number (VPN) to the inflight ones'. If a WAR, RAW, or WAW dependency is detected, Clio Libarary blocks the new request until the conflicting request finishes. When Clio Libarary sees a `rrelease` operation, it waits until all inflight requests return or time out. We currently track dependencies at the page granularity mainly to reduce tracking complexity

and metadata overhead. The downside is that false dependencies could happen (*e.g.*, two accesses to the same page but different addresses). False dependencies could be reduced by dynamically adapting the tracking granularity if application access patterns are tracked—we leave this improvement for future work.

Inter-thread/process consistency (T3). Multi-threaded or multi-process concurrent programming on Clio could use the synchronization primitives Clio provides to ensure data consistency (§3.3.1). We implemented all synchronization primitives like `rlock` and `rfence` at MN, because they need to work across threads and processes that possibly reside on different CNs. Before a request enters either the fast or the slow paths, MN checks if it is a synchronization primitive. For primitives like `rlock` that internally is implemented using atomic operations like `TAS`, MN blocks future atomic operations until the current one completes. For `rfence`, MN blocks all future requests until all inflight ones complete. Synchronization primitives are one of the only two cases where MN needs to maintain state. As these operations are infrequent and each of these operations executes in bounded time, the hardware resources for maintaining their state are minimal and bounded.

Handling retries (T4). Clio Libarary retries a request after a `TIMEOUT` period without receiving any response. Potential consistency problems could happen as Clio Memory Board could execute a retried write after the data is written by another write request thus undoing this other request's write. Such situations could happen when the original request's response is lost or delayed and/or when the network reorders packets. We use two techniques to solve this problem.

First, Clio Libarary attaches a new request ID to each retry, essentially making it a new request with its own matching response. Together with Clio Libarary's ordering enforcement, it ensures that there is only one outstanding request (or a retry) at any time. Second, we maintain a small buffer at MN to record the request IDs of recently executed writes and atomic APIs and the results of the atomic APIs. A retry attaches its own request ID and the ID of the failed request. If MN finds a match of the latter in the buffer, it will not execute the request. For atomic APIs, it sends the cached result as the response. We set this buffer's size to be $3\times$`TIMEOUT`$\times$*bandwidth*,

which is 30 KB in our setting. It is one of the only two types of state MN maintains and does not affect the scalability of MN, since its size is statically associated with the link bandwidth and the `TIMEOUT` value. With this size, the MN can "remember" an operation long enough for two retries from the CN. Only when both retries and the original request all fail, the MN will fail to properly handle a future retry. This case is extremely rare [119], and we report the error to the application, similar to [85, 151].

Why T1 to T4? We now briefly discuss the rationale behind why we need all T1 to T4 to properly deliver our consistency guarantees. First, assume that there is no packet loss or corruption (*i.e.*, no retry) but the network can reorder packets. In this case, using T1 and T2 alone is enough to guarantee the proper ordering of Clio memory operations, since they guarantee that network reordering will only affect either packets within the same request or requests that are not dependent on each other. T3 guarantees the correctness of synchronization primitives since the MN is the serialization point and is where these primitives are executed. Now, consider the case where there are retries. Because of the asynchronous network, a timed-out request could just be slow and still reach the MN, either before or after the execution of the retried request. If another request is executed in between the original and the retried requests, inconsistency could happen (*e.g.*, losing the data of this other request if it is a write). The root cause of this problem is that one request can be executed twice when it is retried. T4 solves this problem by ensuring that the MN only executes a request once even if it is retried.

3.4.6 Extension and Offloading Support

To avoid network round trips when working with complex data structures and/or performing data-intensive operations, we extend the core MN to support application computation offloading in the extend path. Users can write and deploy application offloads both in FPGA and in software (run in the ARM). To ease the development of offloads, Clio offers the same virtual memory interface as the one to applications running at CNs. Each offload has its own PID and virtual memory address space, and they use the same virtual memory APIs (§3.3.1) to access

on-board memory. It could also share data with processes running at CNs in the same way that two CN processes share memory. Finally, an offload's data and control paths could be split to FPGA and ARM and use the same async-buffer mechanism for communication between them. These unique designs made developing computation offloads easier and closer to traditional multi-threaded software programming.

3.4.7 Distributed MNs

Our discussion so far focused on a single MN (Clio Memory Board). To more efficiently use remote memory space and to allow one application to use more memory than what one Clio Memory Board can offer, we extend the single-MN design to a distributed one with multiple MNs. Specifically, an application process' RAS can span multiple MNs, and one MN can host multiple RASs. We adopt LegoOS' two-level distributed virtual memory management approach to manage distributed MNs in Clio. A global controller manages RASs in coarse granularity (assigning 1 GB virtual memory regions to different MNs). Each MN then manages the assigned regions at fine granularity.

The main difference between LegoOS and Clio's distributed memory system is that in Clio, each MN can be over-committed (*i.e.*, allocating more virtual memory than its physical memory size), and when an MN is under memory pressure, it migrates data to another MN that is less pressured (coordinated by the global controller). The traditional way of providing memory over-commitment is through memory swapping, which could be potentially implemented by swapping memory between MNs. However, swapping would cause performance impact on the data path and add complexity to the hardware implementation. Instead of swapping, we *proactively* migrate a rarely accessed memory region to another MN when an MN is under memory pressure (its free physical memory space is below a threshold). During migration, we pause all client requests to the region being migrated. With our 10 Gbps experimental board, migrating a 1 GB region takes 1.3 second. Migration happens rarely and, unlike swapping, happens in the background. Thus, it has little disturbance to foreground application performance.

3.5 Clio Implementation

Apart from challenges discussed in §4.4, our implementation of Clio also needs to overcome several practical challenges, for example, how can different hardware components most efficiently work together in Clio Memory Board, how to minimize software overhead in Clio Libarary. This section describes how we implemented Clio Memory Board and Clio Libarary, focusing on the new techniques we designed to overcome these challenges. Currently, Clio consists of 24.6K SLOC (excluding computation offloads and third-party IPs). They include 5.6K SLOC in SpinalHDL [155] and 2K in C HLS for FPGA hardware, and 17K in C for Clio Libarary and ARM software. We use vendor-supplied interconnect and DDR IPs, and an open-source MAC and PHY network stack [55].

Clio Memory Board Prototyping. We prototyped Clio Memory Board with a low-cost ($2495 retail price) Xilinx MPSoC board [178] and build the hardware fast path (which is anticipated to be built in ASIC) with FPGA. All Clio's FPGA modules run at 250 MHz clock frequency and 512-bit data width. They all achieve an *Initiation Interval* (*II*) of one (II is the number of clock cycles between the start time of consecutive loop iterations, and it decides the maximum achievable bandwidth). Achieving II of one is not easy and requires careful pipeline design in all the modules. With II one, our data path can achieve a maximum of 128 Gbps throughput even with just the slower FPGA clock frequency and would be higher with real ASIC implementation.

Our prototyping board consists of a small FPGA with 504K logic cells (LUTs) and 4.75 MB FPGA memory (BRAM), a quad-core ARM Cortex-A53 processor, two 10 Gbps SFP+ ports connected to the FPGA, and 2 GB of off-chip on-board memory. This board has several differences from our anticipated real Clio Memory Board: its network port bandwidth and on-board memory size are both much lower than our target, and like all FPGA prototypes, its clock frequency is much lower than real ASIC. Unfortunately, no board on the market offers the combination of small FPGA/ARM (required for low cost) and large memory and high-speed network ports.

Nonetheless, certain features of this board are likely to exist in a real Clio Memory Board, and these features guide our implementation. Its ARM processor and the FPGA connect through an interconnect that has high bandwidth (90 GB/s) but high delay (40 μs). Although better interconnects could be built, crossing ARM and FPGA would inevitably incur non-trivial latency. With this board, the ARM's access to on-board DRAM is much slower than the FPGA's access because the ARM has to first physically cross the FPGA then to the DRAM. A better design would connect the ARM directly to the DRAM, but it will still be slower for the ARM to access on-board DRAM than its local on-chip memory.

To mitigate the problem of slow accesses to on-board DRAM from ARM, we maintain shadow copies of metadata at ARM's local DRAM. For example, we store a *shadow* version of the page table in ARM's local memory, so that the control path can read page table content faster. When the control path needs to perform a virtual memory space allocation, it reads the shadow page table to test if an address would cause an overflow (§3.4.2). We keep the shadow page table in sync with the real page table by updating both tables when adding, removing, or updating the page table entries.

In addition to maintaining shadow metadata, we employ an efficient polling mechanism for ARM/FPGA communication. We dedicate one ARM core to busy poll an RX ring buffer between ARM and FPGA, where the FPGA posts tasks for ARM. This polling thread hands over tasks to other worker threads for task handling and post responses to a TX ring buffer.

Clio Memory Board's network stack builds on top of standard, vendor-supplied Ethernet physical and link-layer IPs, with just an additional thin checksum-verify and ack-generation layer on top. This layer uses much fewer resources compared to a normal RDMA-like stack (§3.7.3). We use lossless Ethernet with Priority Flow Control (PFC) for less packet loss and retransmission. Since PFC has issues like head-of-line blocking [183, 108, 58, 118], we rely on our congestion and incast control to avoid triggering PFC as much as possible.

Finally, to assist Clio users in building their applications, we implemented a simple software simulator of Clio Memory Board which works with Clio Libarary for developers to test

their code without the need to run an actual Clio Memory Board.

Clio Libarary Implementation. Even though we optimize the performance of Clio Memory Board, the end-to-end application performance can still be hugely impacted if the host software component (Clio Libarary) is not as fast. Thus, our Clio Libarary implementation aims to provide low-latency performance by adopting several ideas (e.g., data inlining, doorbell batching) from recent low-latency I/O solutions [84, 85, 86, 165, 83, 127, 180]. We implemented Clio Libarary in the user space. It has three parts: a user-facing request ordering layer that performs dependency check and ordering of address-conflicting requests, a transport layer that performs congestion/incast control and request-level retransmission, and a low-level device driver layer that interacts with the NIC (similar to DPDK [49] but simpler). Clio Libarary bypasses kernel and directly issues raw Ethernet requests to the NIC with zero memory copy. For synchronous APIs, we let the requesting thread poll the NIC for receiving the response right after each request. For asynchronous APIs, the application thread proceeds with other computations after issuing the request and only busy polls when the program calls `rpoll`.

3.6 Building Applications on Clio

We built five applications on top of Clio, one that uses the basic Clio APIs, one that implements and uses a high-level, extended API, and two that offload data processing tasks to MNs, and one that splits computation across CNs and MNs.

Image compression. We build a simple image compression/decompression utility that runs purely at CN. Each client of the utility (*e.g.*, a Facebook user) has its own collection of photos, stored in two arrays at MNs, one for compressed and one for original, both allocated with `ralloc`. Because clients' photos need to be protected from each other, we use one process per client to run the utility. The utility simply reads a photo from MN using `rread`, compresses/decompresses it, and writes it back to the other array using `rwrite`. Note that we use compression and decompression as an example of image processing. These operations could potentially be

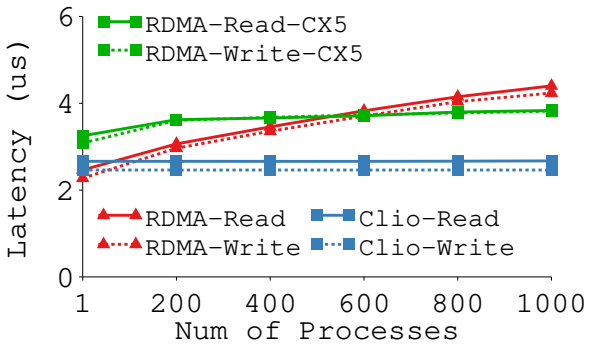

Figure 3.4. Process (Connection) Scalability.

offloaded to MNs. However, in reality, there can be many other types of image processing that are more complex and are hard and costly to implement in hardware, necessitating software processing at CNs. We implemented this utility with 1K C code in 3 developer days.

Radix tree. To demonstrate how to build a data structure on Clio using Clio's extended API, we built a radix tree with linked lists and pointers. Data-structure-level systems like AIFM [143] could follow this example to make simple changes in their libraries to run on Clio. We first built an extended pointer-chasing functionality in FPGA at the MN which follows pointers in a linked list and performs a value comparison at each traversed list node. It returns either the node value when there is a match or null when the next pointer becomes null. We then expose this functionality to CNs as an extended API. The software running at CN allocates a big contiguous remote memory space using `ralloc` and uses this space to store radix tree nodes. Nodes in each layer are linked to a list. To search a radix tree, the CN software goes through each layer of the tree and calls the pointer chasing API until a match is found. We implemented the radix tree with 300 C code at CN and 150 SpinalHDL code at Clio Memory Board in less than one developer day.

Key-value store. We built *Clio-KV*, a key-value store that supports concurrent create/update/read-/delete key-value entries with atomic write and read committed consistency. Clio-KV runs at an MN as a computation offloading module. Users can access it through a key-value interface from multiple CNs. The Clio-KV module has its own virtual memory address space and uses Clio

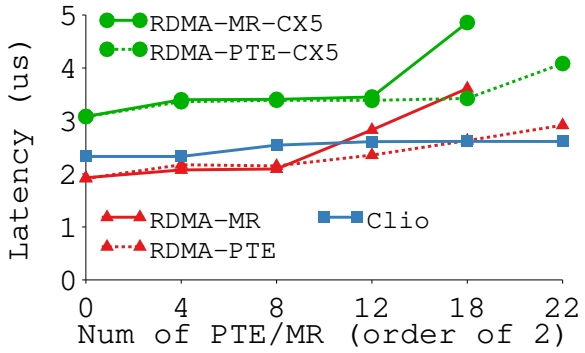

Figure 3.5. PTE and MR Scalability.

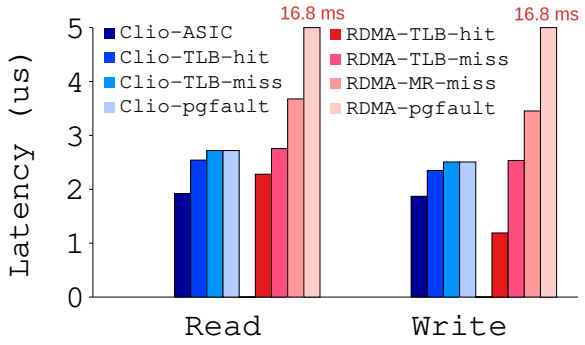

Figure 3.6. Comparison of TLB Miss and page fault.

virtual memory APIs to access it. Clio-KV uses a chained hash table in its virtual memory space for managing the metadata of key-value pairs, and it stores the actual key values at separate locations in the space. Each hash bucket has a chain of slots. Each slot contains the virtual addresses of seven key-value pairs. It also stores a fingerprint for each key-value pair.

To create a new key-value pair, Clio-KV allocates space for the key-value data with an `ralloc` call and writes the data with an `rwrite`. It then calculates the hash and the fingerprint of the key. Afterward, it fetches the last hash slot in the corresponding hash bucket using the hash value. If that slot is full, Clio-KV allocates another slot using `ralloc`; otherwise, it just uses the fetched last slot. It then inserts the virtual address and fingerprint of the data into the last/new slot. Finally, it links the current last slot to the new slot if a new one is created.

To perform a read, Clio-KV locates the hash bucket (with the key's hash value) and fetches one slot in the bucket chain at a time using `rread`. It then compares the fingerprint of

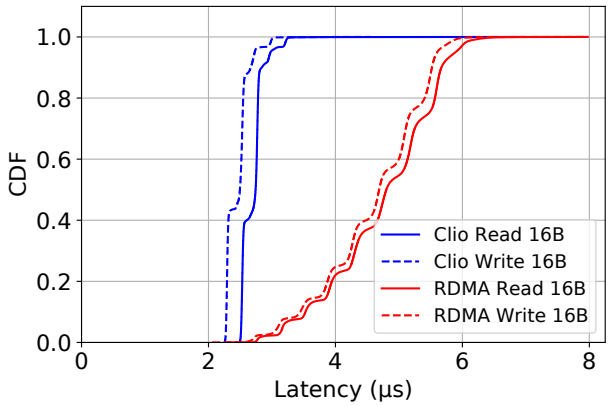

Figure 3.7. Latency CDF.

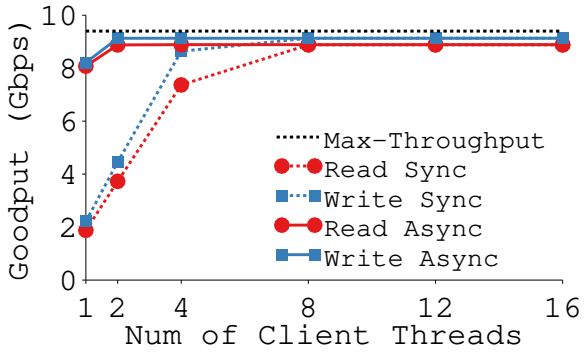

Figure 3.8. End-to-End Goodput.

the key to the seven entries in the slot. If there is no match, it fetches the next slot in the bucket. Otherwise, with a matched entry, it reads the key-value pair using the address stored in that entry with an `rread`. It then compares the full key and returns the value if it is a match. Otherwise, it keeps searching the bucket.

The above describes a single-MN Clio-KV system. Another CN-side load balancer is used to partition key-value pairs into different MNs. Since all CNs requests of the same partition go to the same MN and Clio APIs within an MN are properly ordered, it is fairly easy for Clio-KV to guarantee the atomic-write, read-committed consistency level.

We implemented Clio-KV with 772 SpinalHDL code in 6 developer days. To evaluate Clio's virtual memory API overhead at Clio Memory Board, we also implemented a key-value store with the same design as Clio-KV but with raw physical memory interface. This physical-

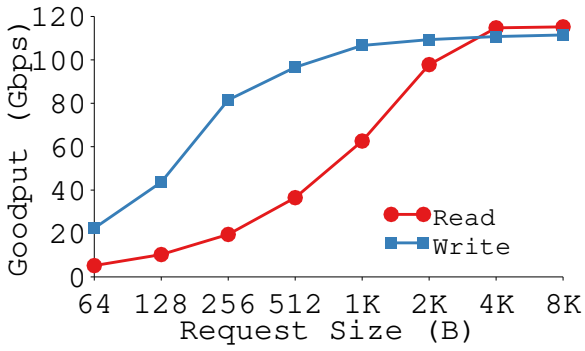

Figure 3.9. On-board Goodput.

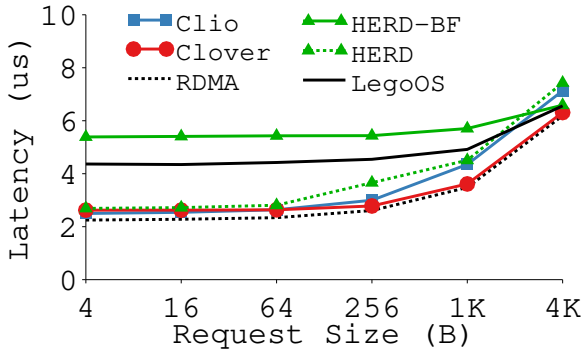

Figure 3.10. Read Latency.

memory-based implementation takes more time to develop and only yields 4%–12% latency improvement and 1%–5% throughput improvement over Clio-KV.

Multi-version object store. We built a multi-version object store (*Clio-MV*) which lets users on CNs create an object, append a new version to an object, read a specific version or the latest version of an object, and delete an object. Similar to Clio-KV, Clio-MV has its own address space. In the address space, it uses an array to store versions of data for each object, a map to store the mapping from object IDs to the per-object array addresses, and a list to store free object IDs. When a new object is created, Clio-MV allocates a new array (with `ralloc`) and writes the virtual memory address of the array into the object ID map. Appending a new version to an object simply increases the latest version number and uses that as an index to the object array for writing the value. Reading a version simply reads the corresponding element of the array.

Clio-MV allows concurrent accesses from CNs to an object and guarantees sequential

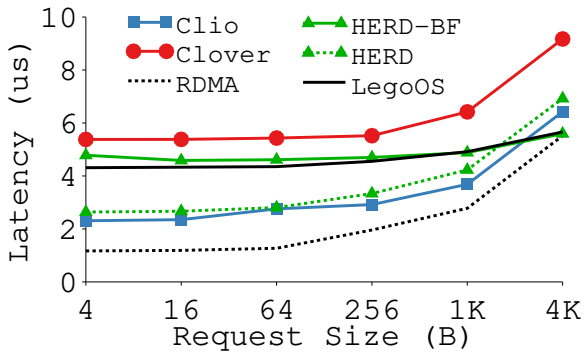

Figure 3.11. Write Latency.

consistency for each object. Each Clio-MV user request involves at least two internal Clio operations, some of which include both metadata and data operations. This compound request pattern makes it tricky to deal with synchronization problems, as Clio-MV needs to ensure that no internal Clio operation of a later Clio-MV request could affect the correctness of an earlier Clio-MV request. We implemented Clio-MV with 1680 lines of C HLS code in 15 developer days.

Simple data analytics. Our final example is a simple DataFrame-like data processing application (*Clio-DF*), which splits its computation between CN and MN. We implement `select` and `aggregate` at MN as two offloads, as offloading them can reduce the amount of data sent over the network. We keep other operations like `shuffle` and `histogram` at CN. For the same user, all these modules share the same address space regardless of whether they are at CN or MN. Thanks to Clio's support of computation offloading sharing the same address space as computations running at host, Clio-DF's implementation is largely simplified and its performance is improved by avoiding data serialization/deserialization. We implemented Clio-DF with 202 lines of SpinalHDL code and 170 lines of C interface code in 7 developer days.

3.7 Evaluation

Our evaluation reveals the scalability, throughput, median and tail latency, energy and resource consumption of Clio. We compare Clio's end-to-end performance with industry-grade

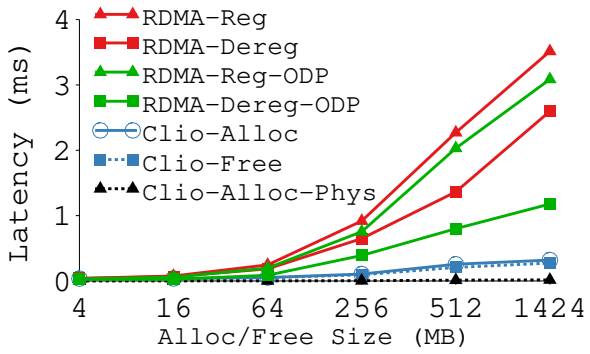

Figure 3.12. Alloc/Free Latency.

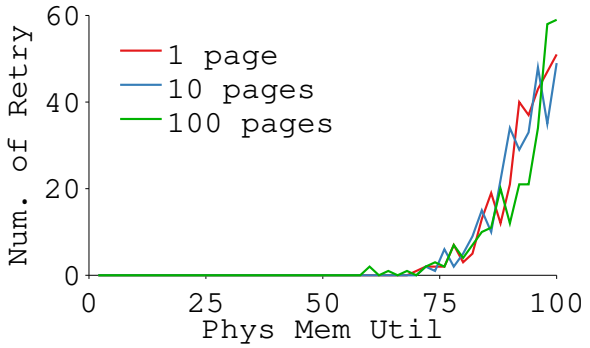

Figure 3.13. Alloc Retry Rate.

NICs (ASIC) and well-tuned RDMA-based software systems. All Clio's results are FPGA-based, which would be improved with ASIC implementation.

Environment. We evaluated Clio in our local cluster of four CNs and four MNs (Xilinx ZCU106 boards), all connected to an Nvidia 40 Gbps VPI switch. Each CN is a Dell PowerEdge R740 server equipped with a Xeon Gold 5128 CPU and a 40 Gbps Nvidia ConnectX-3 NIC, with two of them also having an Nvidia BlueField SmartNIC [116]. We also include results from CloudLab [42] with the Nvidia ConnectX-5 NIC.

3.7.1 Basic Microbenchmark Performance

Scalability. We first compare the scalability of Clio and RDMA. Figure 3.4 measures the latency of Clio and RDMA as the number of client processes increases. For RDMA, each process uses its own QP. Since Clio is connectionless, it scales perfectly with the number of processes. RDMA

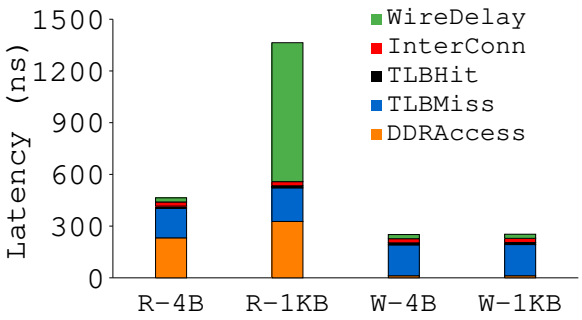

Figure 3.14. Latency Breakdown.

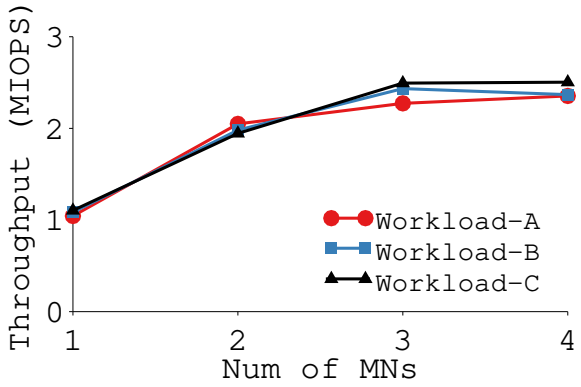

Figure 3.15. Clio-KV Scalability against MNs.

scales poorly with its QP, and the problem persists with newer generations of RNIC, which is also confirmed by our previous works [163, 124].

Figure 3.5 evaluates the scalability with respect to PTEs and memory regions. For the memory region test, we register multiple MRs using the same physical memory for RDMA. For Clio, we map a large range of VAs (up to 4 TB) to a small physical memory space, as our testbed only has 2 GB physical memory. However, the number of PTEs and the amount of processing needed are the same for Clio Memory Board as if it had a real 4 TB physical memory. Thus, this workload stress tests Clio Memory Board's scalability. RDMA's performance starts to degrade when there are more than 2^8 (local cluster) or 2^{12} (CloudLab), and the scalability wrt MR is worse than wrt PTE. In fact, RDMA fails to run beyond 2^{18} MRs. In contrast, Clio scales well and never fails (at least up to 4 TB memory). It has two levels of latency that are both stable: a lower latency below 2^4 for TLB hit and a higher latency above 2^4 for TLB miss (which

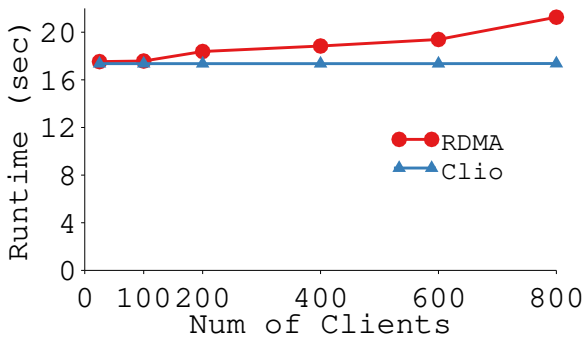

Figure 3.16. Image Compression.

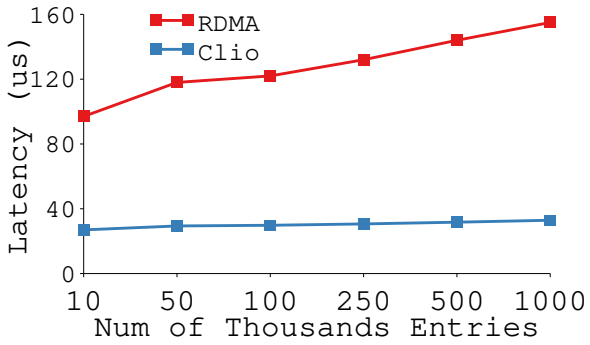

Figure 3.17. Radix Tree Search Latency.

always involves one DRAM access). A Clio Memory Board could use a larger TLB if optimal performance is desired.

These experiments confirm that **Clio can handle thousands of concurrent clients and TBs of memory**.

Latency variation. Figure 3.6 plots the latency of reading/writing 16 B data when the operation results in a TLB hit, a TLB miss, a first-access page fault, and MR miss (for RDMA only, when the MR metadata is not in RNIC). RDMA's performance degrades significantly with misses. Its page fault handling is extremely slow (16.8 *ms*). We confirm the same effect on CloudLab with the newer ConnectX-5 NICs. Clio only incurs a small TLB miss cost and **no additional cost of page fault handling**.

We also include a projection of Clio's latency if it was to be implemented using a real ASIC-based Clio Memory Board. Specifically, we collect the latency breakdown of time spent

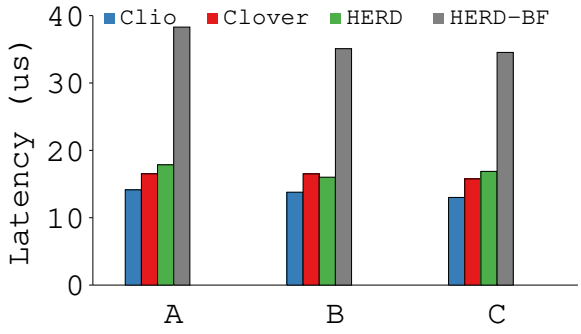

Figure 3.18. Key-Value Store YCSB Latency.

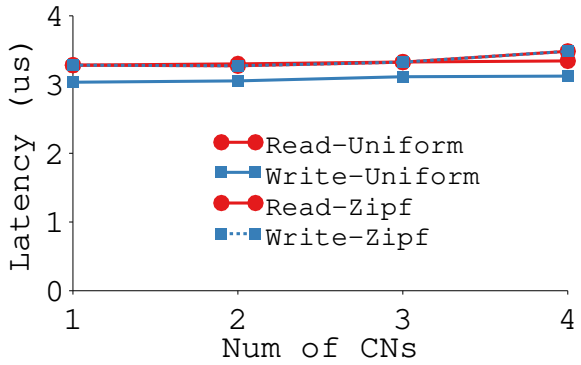

Figure 3.19. Clio-MV Object Read/Write Latency.

on the network wire and at CN, time spent on third-party FPGA IPs, number of cycles on FPGA, and time on accessing on-board DRAM. We maintain the first two parts, scale the FPGA part to ASIC's frequency (2 GHz), use DDR access time collected on our server to replace the access time to on-board DRAM (which goes through a slow board memory controller). This estimation is conservative, as a real ASIC implementation of the third-party IPs would make the total latency lower. Our estimated read latency is better than RDMA, while write latency is worse. We suspect the reason being Nvidia RNIC's optimization of replying a write before it is fully written to DRAM, which Clio could also potentially adopt.

Figure 3.7 plots the request latency CDF of continuously running read/write 16 B data while not triggering page faults. Even without page faults, Clio has much less latency variation and a much shorter tail than RDMA.

Read/write throughput. We measure Clio's throughput by varying the number of concurrent

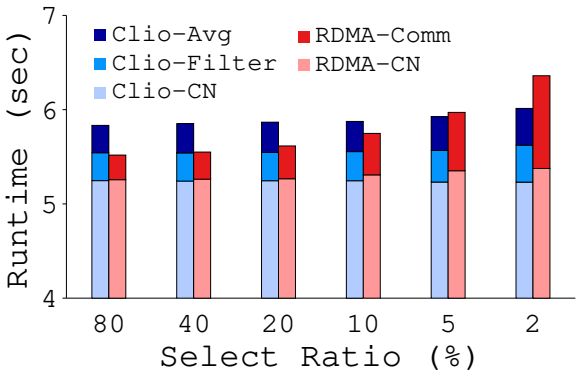

Figure 3.20. Select-Aggregate-Shuffle.

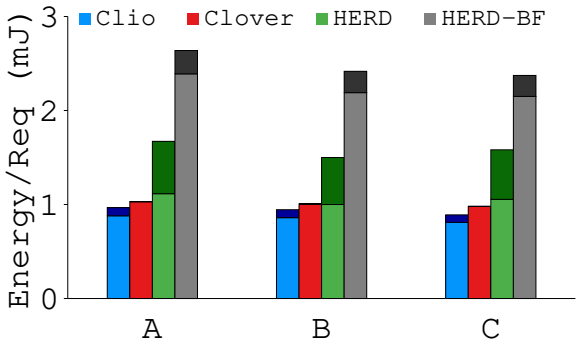

Figure 3.21. Energy Comparison.

client threads (Figure 3.8). Clio's default asynchronous APIs quickly reach the line rate of our testbed (9.4 Gbps maximum throughput). Its synchronous APIs could also reach line rate fairly quickly.

Figure 3.9 measures the maximum throughput of Clio's FPGA implementation without the bottleneck of the board's 10 Gbps port, by generating traffic on board. Both read and write can reach more than 110 Gbps when request size is large. Read throughput is lower than write when request size is smaller. We found the throughput bottleneck to be the third-party non-pipelined DMA IP (which could potentially be improved).

Comparison with other systems. We compare Clio with native one-sided RDMA, Clover [164], HERD [85], and LegoOS [147]. We ran HERD on both CPU and BlueField (HERD-BF). Clover is a passive disaggregated persistent memory system which we adapted as a passive disaggregated

memory (PDM) system. HERD is an RDMA-based system that supports a key-value interface with an RPC-like architecture. LegoOS builds its virtual memory system in software at MN.

Clio's performance is similar to HERD and close to native RDMA. Clover's write is the worst because it uses at least 2 RTTs for writes to deliver its consistency guarantees without any processing power at MNs. HERD-BF's latency is much higher than when HERD runs on CPU due to the slow communication between BlueField's ConnectX-5 chip and ARM processor chip. LegoOS's latency is almost two times higher than Clio's when request size is small. In addition, from our experiment, LegoOS can only reach a peak throughput of 77 Gbps, while Clio can reach 110 Gbps. LegoOS' performance overhead comes from its software approach, demonstrating the necessity of a hardware-based solution like Clio.

Allocation performance. Figure 3.12 shows Clio's VA and PA allocation and RDMA's MR registration performance. Clio's PA allocation takes less than 20 μs, and the VA allocation is much faster than RDMA MR registration, although both get slower with larger allocation/registration size. Figure 3.13 shows the number of retries at allocation time with three allocation sizes as the physical memory fills up. There is no retry when memory is below half utilized. Even when memory is close to full, there are at most 60 retries per allocation request, with roughly 0.5 ms per retry. This confirms that our design of avoiding hash overflows at allocation time is practical.

Close look at Clio Memory Board components. To further understand Clio's performance, we profile different parts of Clio's processing for read and write of 4 B to 1 KB. Clio Libarary adds a very small overhead (250 ns in total), thanks to our efficient threading model and network stack implementation. Figure 3.14 shows the latency breakdown at Clio Memory Board. Time to fetch data from DRAM (DDRAccess) and to transfer it over the wire (WireDelay) are the main contributor to read latency, especially with large read size. Both could be largely improved in a real Clio Memory Board with better memory controller and higher frequency. TLB miss (which takes one DRAM read) is the other main part of the latencies.

3.7.2 Application Performance

Image Compression. We run a workload where each client compresses and decompresses 1000 256*256-pixel images with increasing number of concurrently running clients. Figure 3.16 shows the total runtime per client. We compare Clio with RDMA, with both performing computation at the CN side and the RDMA using one-sided operations instead of Clio APIs to read/write images in remote memory. Clio's performance stays the same as the number of clients increase. RDMA's performance does not scale because it requires each client to register a different MR to have protected memory accesses. With more MRs, RDMA runs into the case where the RNIC cannot hold all the MR metadata and many accesses would involve a slow read to host main memory.

Radix Tree. Figure 3.17 shows the latency of searching a key in pre-populated radix trees when varying the tree size. We again compare with RDMA which uses one-sided read operations to perform the tree traversal task. RDMA's performance is worse than Clio, because it requires multiple RTTs to traverse the tree, while Clio only needs one RTT for each pointer chasing (each tree level). In addition, RDMA also scales worse than Clio.

Key-value store. Figure 3.18 evaluates Clio-KV using the YCSB benchmark [6] and compares it to Clover, HERD, and HERD-BF. We run two CNs and 8 threads per CN. We use 100K key-value entries and run 100K operations per test, with YCSB's default key-value size of 1 KB. The accesses to keys follow the Zipf distribution ($\theta = 0.99$). We use three YCSB workloads with different *get-set* ratios: 100% *get* (workload C), 5% *set* (B), and 50% *set* (A). Clio-KV performs the best. HERD running on BlueField performs the worst, mainly because BlueField's slower crossing between its NIC chip and ARM chip.

Figures 3.15 shows the throughput of Clio-KV when varying the number of MNs. Similar to our Clio scalability results, Clio-KV can reach a CN's maximum throughput and can handle concurrent get/set requests even under contention. These results are similar to or better than previous FPGA-based and RDMA-based key-value stores that are fine-tuned for just key-value

workloads (Table 3 in [104]), while we got our results without any performance tuning.

Multi-version data store. We evaluate Clio-MV by varying the number of CNs that concurrently access data objects (of 16 B) on an MN using workloads of 50% read (of different versions) and 50% write under uniform and Zipf distribution of objects (Figure 3.19). Clio-MV's read and write have the same performance, and reading any version has the same performance, since we use an array-based version design.

Data analytics. We run a simple workload which first `selects` rows in a table whose field-A matches a value (*e.g.*, gender is female) and then calculates `avg` of field-B (*e.g.*, final score) of all the rows. Finally, it calculates the histogram of the selected rows (*e.g.*, score distribution), which can be presented to the user together with the avg value. Clio executes the first two steps at MN offloads and the final step at CN, while RDMA always reads rows to CN and then does each operation. Figure 3.20 plots the total run time as the select ratio decreases (*i.e.*, fewer rows selected). When the select ratio is low, Clio transfers much less data than RDMA, resulting in its better performance.

3.7.3 CapEx, Energy, and FPGA Utilization

We estimate the cost of server and Clio Memory Board using market prices of different hardware units. When using 1 TB DRAM, a server-based MN costs 1.1-1.5× and consumes 1.9-2.7× power compared to Clio Memory Board. These numbers become 1.4-2.5× and 5.1-8.6× with OptaneDimm [132], which we expect to be the more likely remote memory media in future systems.

We measure the total energy used for running YCSB workloads by collecting the total CPU (or FPGA) cycles and the Watt of a CPU core [14], ARM processor [131], and FPGA (measured). We omit the energy used by DRAM and NICs in all the calculations. Clover, a system that centers its design around low cost, has slightly higher energy than Clio. Even though there is no processing at MNs for Clover, its CNs use more cycles to process and manage memory. HERD consumes 1.6× to 3× more energy than Clio, mainly because of its CPU overhead at MNs.

Surprisingly, HERD-BF consumes the most energy, even though it is a low-power ARM-based SmartNIC. This is because of its worse performance and longer total runtime.

3.8 Discussion and Conclusion

We presented Clio, a new hardware-based disaggregated memory platform. Our FPGA prototype demonstrates that Clio achieves great performance, scalability, and cost-saving. This work not only guides the future development of Memory Disaggregation solutions but also demonstrates how to implement a core OS subsystem in hardware and co-design it with the network. We now present our concluding thoughts with several open questions.

Security and performance isolation. Clio's protection domain is a user process, which is the same as the traditional single-server process-address-space-based protection. The difference is that Clio performs permission checks at MNs: it restricts a process' access to only its (remote) memory address space and does this check based on the global PID. Thus, the safety of Clio relies on PIDs to be authentic (*e.g.*, by letting a trusted CN OS or trusted CN hardware attach process IDs to each Clio request). There have been researches on attacking RDMA systems by forging requests [141] and on adding security features to RDMA [151, 159]. How these and other existing security works relate and could be extended in a memory disaggregation setting is an open problem, and we leave this for future work.

There are also designs in our current implementation that could be improved to provide more protection against side-channel and DoS attacks. For example, currently, the TLB is shared across application processes, and there is no network bandwidth limit for an individual connection. Adding more isolation to these components would potentially increase the cost of Clio Memory Board or reduce its performance. We leave exploring such tradeoffs to future work.

Failure handling. Although memory systems are usually assumed to be volatile, there are still situations that require proper failure handling (*e.g.*, for high availability or to use memory for storing data). As there can be many ways to build memory services on Clio and many such

services are already or would benefit from handling failure on their own, we choose not to have any built-in failure handling mechanism in Clio. Instead, Clio should offer primitives like replicated writes for users to build their own services. We leave adding such API extensions to Clio as future work.

CN-side stack. An interesting finding we have is that CN-side systems could become a performance bottleneck after we made the remote memory layer very fast. Surprisingly, most of our performance tuning efforts are spent on the CN side (*e.g.*, thread model, network stack implementation). Nonetheless, software implementation is inevitably slower than customized hardware implementation. Future works could potentially improve Clio's CN side performance by offloading the software stack to a customized hardware NIC.

3.9 Acknowledgement

Chapter 3, in full, is a reprint of Zhiyuan Guo, Yizhou Shan(co-first authors), Xuhao Luo, Yutong Huang, Yiying Zhang, "Clio: A Hardware-Software Co-Designed Disaggregated Memory System", ASPLOS, 2022. The dissertation author was the primary investigator and author of this paper.

Chapter 4

NetPool: A Network Functionality Disaggregation and Consolidation System

4.1 Introduction

Host servers in today's data centers spend significant computing resources on network processing. With CPUs meeting their scaling limits, more network functionalities are offloaded to networking hardware (*e.g.*, SmartNICs) to keep up with the ever-increasing network speed that is projected to reach 400 Gbps soon [135, 137, 111, 64, 113, 139]. As such, several issues arise. First, each end-host needs to provision network resources for all its anticipated network functionalities and its peak network traffic, involving various hardware accelerators like encryption engines and AI accelerators. However, only a small amount of network functionalities [173] and bandwidth [142] are usually needed at a time. As a result, networking resources today are over-provisioned, resulting in cost wastage [112] and unnecessary carbon emissions [69, 114]. Meanwhile, applications can incur long tail latency when traffic surges go beyond a single SmartNIC's processing capacity [56, 100, 113].

We propose to solve these cost and performance problems with one idea: *disaggregating network functionalities from end-hosts and consolidating them into a network resource pool* at the rack scale. Our idea is based on our insight that *the peak of sums is significantly lower than the sum of peaks*, where the former represents the highest of aggregated traffic in a rack and the latter represents the summation of the highest traffic at individual end-hosts. As shown in

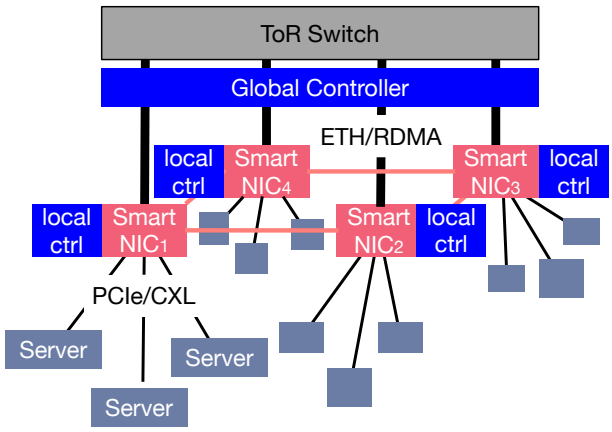

Figure 4.1. NetPool Design Overview.

Figure 4.1, each SmartNIC in the pool connects to N end-hosts via a high-speed interconnect like PCIe or CXL. Each also connects to the ToR switch via Ethernet or Infiniband. Because of the N-to-1 end-host to SmartNIC ratio, network resource pooling reduces the cost of network devices roughly by a factor of N.

Notably, the peak traffic the pool can handle is N times lower than what today's per-end-host SmartNIC can handle. However, if the peak of sums is no bigger than $1/N$ of the sum of peaks, the rack can still handle all the traffic. For this goal to be practical, our approach is to allow all SmartNICs in the rack-level pool to handle an end-host's traffic. For this purpose, we connect the SmartNICs in the pool with each other via Ethernet or RDMA, *e.g.*, in a ring topology (we call these connections *peer links*). We redirect traffic that exceeds what a SmartNIC can handle to other SmartNICs via the peer links while leaving the links to the ToR switch to outgoing/incoming traffic.

Based on this basic architecture, we build *NetPool*, a rack-level network solution for disaggregated and pooled SmartNICs. NetPool aims for the effective, efficient, and fair allocation of various network resources in the pool to multiple tenants on different end-hosts. Unlike existing disaggregation and pooling solutions for non-network resources like memory and storage, network resources present unique challenges. First, different SmartNICs in a network resource pool cannot be treated equally because of their different proximity to an end-host. Second,

network resource needs (in the form of traffic load) change more frequently and less predictably than memory and storage resource needs. Finally, network resources imply multiple types of SmartNIC resources, such as different hardware accelerators like compression, encryption, and AI engines.

To address the above challenges, NetPool's key design centers around separating global and local control planes by leveraging their respective strength and mitigating each other's limitations. Our global control plane, running at a centralized location, assigns pool resources to the bulk chunk of traffic with multi-facet considerations. It delivers globally optimal resource assignments by carefully choosing the right amount of resources at each SmartNIC based on the fair sharing of multiple types of network resources and the proximity of SmartNICs from an end host. Because of the well-rounded considerations and the need to collect per-NIC resource utilization, the global controller adjusts its assignments relatively slowly and can miss short traffic fluctuations. To mitigate this issue, our local control plane, running at each individual SmartNIC, quickly handles traffic spikes using the collective power of the whole pool. Without the overhead of pinpointing other SmartNICs' loads, the local control plane distributes a traffic spike happening at one SmartNIC to all the SmartNICs. We reserve a small amount of resources at each SmartNIC for the local control plane to handle traffic spikes and the majority of resources for the global control plane. Together, global resource assignment ensures fairness and high resource utilization, and reserved resources help deliver strong application performance.

Under NetPool's control planes, we design a data plane that efficiently executes the control planes' resource allocation decisions. It models the time and space sharing of a resource as a number of "units" and establishes a work queue for each unit of resource. Work queues can point to a local resource or a resource at a non-local SmartNIC, in which case NetPool steers the traffic to the destined SmartNIC via peer links.

We prototype NetPool with two Nvidia BlueField-2 SmartNICs [35] under one ToR switch. We separate each SmartNIC as two virtual SmartNICs, each connecting to eight virtual end-hosts. The virtual SmartNICs are connected to each other in a ring topology. We implemented

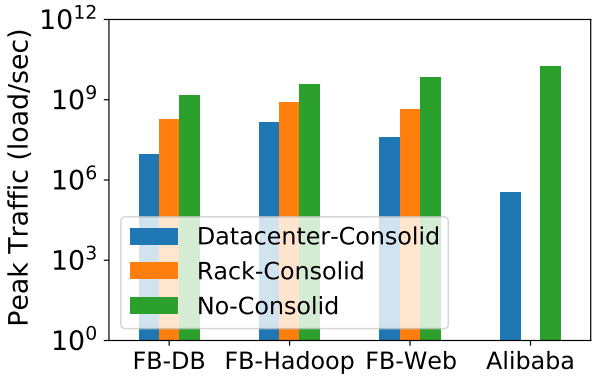

Figure 4.2. Consolidation Analysis of Datacenter Traces.

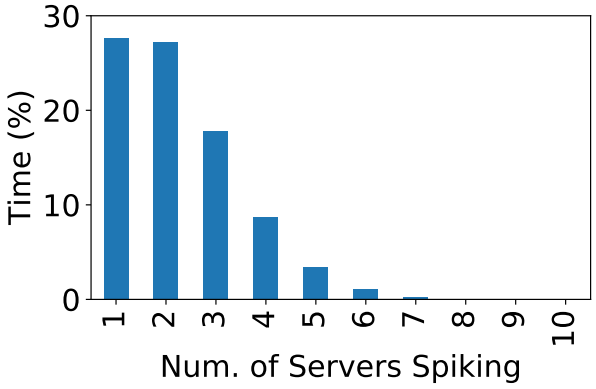

Figure 4.3. Load Spike Variation across Endhosts in Facebook Trace.

three types of applications on NetPool: application-level data encryption offloaded to NetPool, a key-value store with data compression and encryption offloaded, and virtual private cloud (VPC) with firewall, encryption, and NAT offloaded. We evaluate NetPool and the ported applications with micro- and macro-benchmarks and compare NetPool with no network disaggregation and different resource allocation algorithms applied to the NetPool architecture. Overall, compared to today's data center architectures, NetPool reduces datacenter network resource requirements by over 7.4× and improves application throughput by up to 44% in SmartNIC-bottlenecked cases, all while ensuring fairness in a multi-tenant environment.

4.2 Motivation

This section motivates the overall idea of network resource disaggregation and presents a data-center traffic analysis that drives NetPool's control plane design.

4.2.1 Benefits of Network Disaggregation

Although less explored than computing, memory, and storage resources, disaggregating network resources has several key benefits for data centers. First, network disaggregation and consolidation largely cut data center spending on network resources. A consolidated network pool only needs to collectively provision for the peak aggregated traffic and the superset of network accelerators used by the whole rack at any single time. In contrast, today's data centers often equip each end-host with a SmartNIC to handle an end-host's anticipated peak traffic and all network accelerators. At non-peak times or when not all types of accelerators are needed, network resources are largely wasted. On the other hand, when an end-host's traffic goes beyond what an equipped SmartNIC can handle, applications experience performance degradation. Second, deploying and managing network task offloads on SmartNICs is a demanding task, especially when there is a need to upgrade installed SmartNICs with new ones. A consolidated and separated network pool makes it easier to manage and change network resources, as replacing SmartNICs in the pool could be performed incrementally while maintaining connectivity to the end-hosts.

4.2.2 Data Center Traffic Analysis

To understand network behavior in real data centers, we analyze two sets of traces: a Facebook trace that consists of Web, Cache, and Hadoop workloads [142], and an Alibaba trace that hosts latency-critical and batch jobs together [68].

We first perform a consolidation analysis where we calculate the sum of peaks in each individual end-host's traffic (sum of peak) and the peak of aggregated traffic within a rack

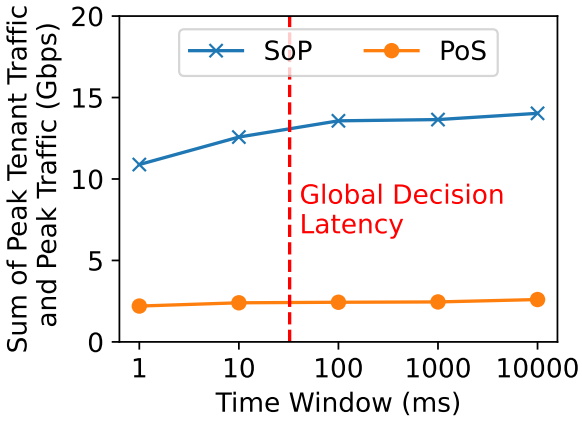

Figure 4.4. Peak of Sum and Sum of Peak at Different Time scales.

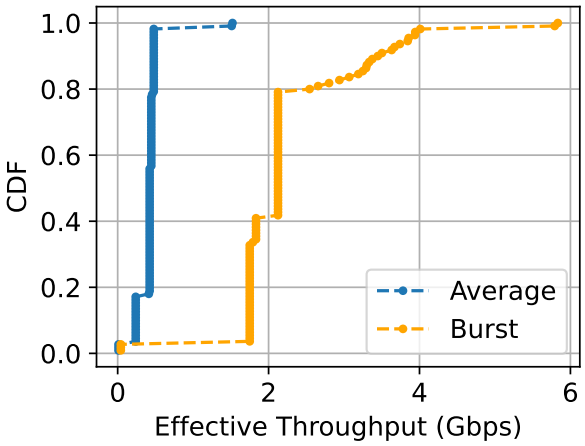

Figure 4.5. Traffic Bursts in 1-millisecond windows.

and across the entire data center (peak of sum). These calculations model the cases of no disaggregation and disaggregation and consolidation at the rack level and the data center level. As shown in Figure 4.2, for both data centers, a rack-level consolidation can potentially reduce network resource usage by an order of magnitude than no consolidation.

We then analyze the load spikes in these traces by comparing different end-hosts' spikes and analyzing whether they spike at similar or different times, which implies how much chance there is for efficient consolidation. Specifically, we count how much time in the entire 1-day trace X number of end-hosts spike together. Figure 4.3 shows that 55% of the time, only one or two servers spike together, and only 14% of the time, four or more servers spike together.

This result shows that servers mostly spike at different times, confirming the large potential for consolidation benefits.

We further analyze the difference between peak-of-sum and sum-of-peak at different time scales. As the Facebook and Alibaba traces do not provide fine-grained time-scale information, we analyze a set of fine-grained, packet-level traces collected from university data centers [88]. We calculate the sum of peak and peak of sum traffic inside a time window of different time scales, ranging from 1 millisecond to minutes. Figure 4.4 shows the average results from 100 randomly sampled windows of the trace. As seen, the peak of sum at all granularity as low as 1ms is always significantly lower than the sum of peak, indicating the benefit of network resource pooling at both coarse and fine time granularity.

Traffic at finer granularity is hard to handle with a global controller when scaling to a rack's servers. For example, to achieve resource fair sharing, running a typical algorithm like DRF [61], HUG [41] and DRFQ [60] at a centralized location takes more than 23 milliseconds (shown as the red vertical line in Figure 4.4. A naive approach to managing pooled network resources is to ignore traffic spikes that happen at the fine granularity and only have a global controller perform resource assignment at a coarse granularity (*e.g.*, allocating resources for each flow instead of packets). To understand the implication of such naive approaches, we analyze how significant traffic spikes are within a small time window of the above traces. Figure 4.5 shows the average traffic load and the peak load at 1ms granularity of our sampled windows. At the 1ms granularity, the peak minus the average can be viewed as traffic spikes. As seen, the peak load is significantly higher than the average load, indicating that spikes account for a large amount of traffic and cannot simply be ignored. This result aligns with other existing works of data-center traffic observations [88, 146, 31].

Note that all existing traffic analysis is based on a non-disaggregated network architecture. With network resource pooling and aggregation, spikes can occur more frequently because applications can use network resources beyond what a single NIC can handle. Moreover, spikes have a greater impact on application performance because a significant spike can block all

end-hosts under a NIC rather than a single end-host. Therefore, it is crucial to manage network resource pooling at all time scales.

4.3 NetPool Overview

NetPool is a disaggregated SmartNIC system that consolidates and shares the SmartNIC resources inside a rack, including programming ARM cores, hardware accelerators, and network bandwidth, to increase the SmartNIC resource utilization. In NetPool, multiple hosts in a rack connect to a multi-ported SmartNIC. These SmartNICs connect to each other using peer links, and each connects to the data center network through the Top-of-Rack (ToR) switch. NetPool allows each host to utilize the SmartNIC resources on its directly attached SmartNIC (we call it the home NIC), as well as resources on other NICs in the pool.

NetPool achieves several critical goals simultaneously: **G1)** effective network resource pooling with significant cost cut and high resource utilization; **G2)** delivers high application performance even when traffic load changes quickly; and **G3)** isolation and fair sharing of all resources in the pool.

Achieving each of them is relatively easy, but these goals combined in the pool setup present unprecedented challenges. For example, G1 implies that NetPool right-provisions at the pool but not the SmartNIC level, and a NIC in the pool can often be over-committed. When that happens, to fulfill G2, NetPool needs to quickly redirect the overflow traffic to a non-home NIC, especially for short-term, ephemeral spikes. Traffic redirection can be quick if each NIC always makes its own decision without considering other NICs' resource usage. However, that violates G3, as fair sharing of the pool's resources requires a global picture of all NICs in the pool.

To confront these challenges, we propose a two-layer approach that incorporates a global controller for handling the bulk of the traffic load and local controllers for handling short-term traffic spikes. Our observation from Section 4.2.2 is that data-center traffic could be decomposed into longer-term patterns and ephemeral traffic spikes. Longer-term patterns make up the majority

of traffic. Thus, ensuring their fair and efficient network resource utilization ensures fair sharing and high utilization in the long run. Meanwhile, their relatively slower changes allow for a more sophisticated resource allocation algorithm to be executed at a global level. NetPool performs global *tenant-level resource partition* for long-term traffic. NetPool's global control plane, running at a centralized location in a rack, first figures out the amount of resources to be reserved for ephemeral traffic at each NIC and each hardware resource type, then allocates a total, pool-level amount of different hardware resources to a tenant in a fair way, and finally determines how the allocated total resources map to different NICs in the pool.

Ephemeral traffic, often manifested as microbursts and traffic spikes, happens quickly and accounts for only a small portion of total traffic, but not handling them timely could highly impact application performance because of the link congestion they cause. NetPool performs local *packet level per resource timeshare* through **resource reservation**. NetPool handles them by reserving a small portion of resources at each NIC, detecting ephemeral spikes at individual NICs, and evenly distributing the traffic to all NICs in the pool. This NIC-local control plane allows for fast reaction to traffic load changes, as no global information is needed when making decisions. Meanwhile, the amount of potentially unutilized resources (reserved but no spikes) and the potential degree of sub-optimal fairness only account for a tiny portion of the entire pool of resources. NetPool's global control plane still ensures end-to-end fairness and resource utilization in the long run.

Underneath NetPool's global and local control plane, NetPool's data plane represents each type of resource as a set of virtual units, each being the smallest piece of time- or space-sharing entity. NetPool associates a work queue for each unit. If the unit represents a non-local resource, NetPool steers the traffic to the corresponding destination NIC. NetPool assigns work-queue units of different types to a network flow based on the assignment of the control plane, thereby achieving fairness in a hardware-agnostic way.

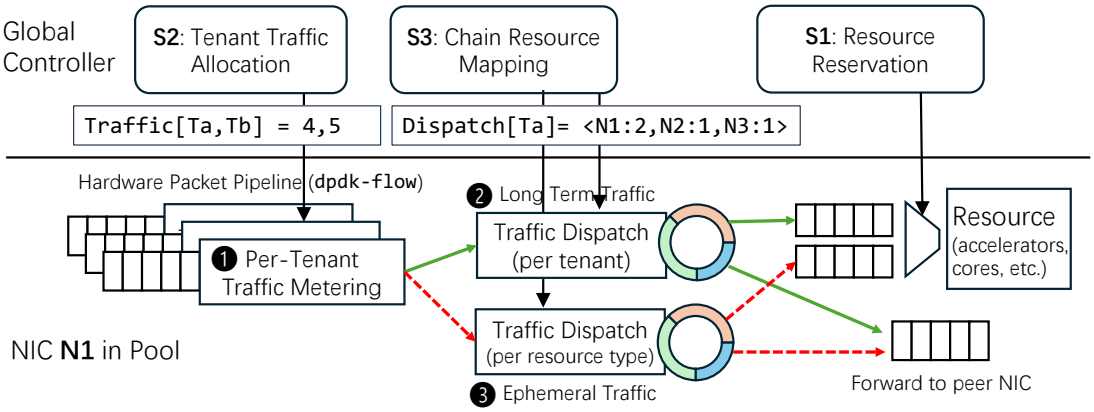

Figure 4.6. The NIC Control and Data Plane Design.

4.4 NetPool Design

This section presents the control plane and data plane of NetPool and discusses how we achieve good performance, high utilization, and multi-tenant fairness. We defer implementation details to Section 4.5.

To use NetPool, each tenant specifies the chain of offloading accelerators/cores for their application (*e.g.*, general-purpose core, followed by encryption accelerator, followed by compression engine). They also specify an estimated load relationship across the offloading types as a vector of values between 0 and 1, $V_k =< R_1^k, R_2^k, ..R_n^k >$, where k represents tenant k and n is the total number of resource types (*e.g.*, $< R_1^k = 1, R_2^k = 0.8, R_3^k = 0.5 >$ to represent all traffic goes through a general-purpose core, 0.8 of it goes to encryption, and 0.5 to compression). At runtime, applications send traffic via our DPDK-based library at each end-host.

4.4.1 Traffic Separation and Resource Reservation

Before introducing NetPool's resource allocation algorithms used at the global and local controllers, we first describe how NetPool determines the amount of load that belongs to long-term traffic and the amount that is considered ephemeral. NetPool reserves the latter amount for the local control plane and leaves the remaining amount under the control of the global controller.

Traffic monitoring. At every monitoring window (set to 1 *ms* by default), NetPool collects two

Algorithm 1. Resource Reservation At Each Adjustment Period

Input: Resource vector $V_k =< R_1^k, R_2^k, ..R_n^k >$ for tenant k
Output: Total reserved amount for resource type i in pool

for $w = 0$ **to** $adjust - period - length - 1$ **do**
    $D_k^w \leftarrow$ monitored demand traffic for tenant k
    $T_i^w \leftarrow$ monitored traffic handled at resource type i
end for
$P_i \leftarrow \max_w T_i^w$
$AvgD^k \leftarrow_w D_k^w$
$AvgD_i^k \leftarrow \text{AvgD}^k \times R_i^k$
$AvgD_i \leftarrow \sum_k AvgD_i^k$
return $P_i - AvgD_i$

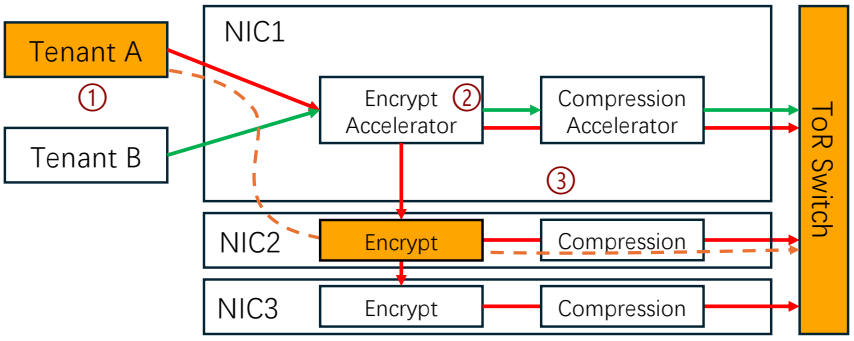

Figure 4.7. Example Tenants Sharing Resource in NetPool.

types of information at the pool (rack) level. The first is the average traffic demand from each tenant (hereafter, we use tenant and application interchangeably). NetPool monitors the outgoing average traffic amount from each tenant at each end-host. It monitors the incoming traffic for each tenant at the ToR switch. Adding the incoming and outgoing traffic across, we get the total average traffic demanded by tenant k for window w, D_k^w. As these values are not limited by the network pool's processing capacity, they reflect true demand from the tenant. The second type of information NetPool collects per window is the traffic handled by each offloading hardware across all tenants. NetPool aggregates this value across the whole pool to get traffic amount T_i^w for window w and resource type i.

Resource reservation. NetPool adjusts the amount of long-term and ephemeral traffic in the pool periodically using Algorithm 1. We set the monitoring window length based on how fast

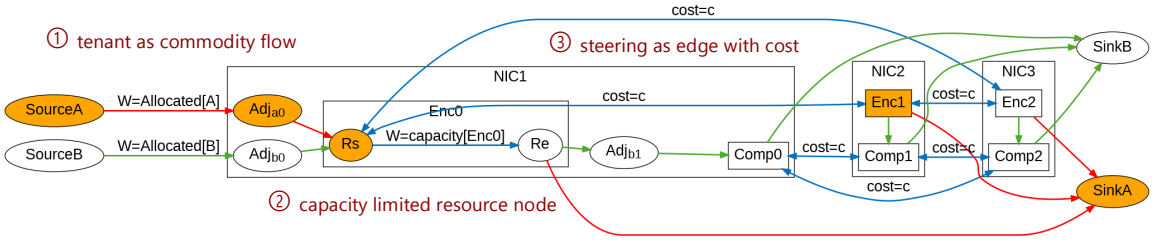

Figure 4.8. Constructed Network Flow. Orange components marks the same path in hardware and in constructed network flow graph. Blue edges represent traffic steering between one NIC and another, which is associated with a cost, c. W represents the resource amount on an edge.

we can collect traffic information (default to 1 *ms*) and the length of the resource *adjustment cycle* based on how fast the global controller can collect global information and use it to assign resource allocation (*e.g.*, 10 *ms* with our testbed, or every ten monitoring windows). At the end of each adjustment cycle, NetPool collects all the per-window traffic information for the past cycle across all servers in a rack. It calculates the peak traffic handled by each resource type across all the SmartNICs in the rack pool as $P_i = max(T_i^w)$ over all the windows w in the cycle. This value represents the peak of sum traffic that the pool has to handle in the past cycle, which we use to estimate the peak of sum in the next cycle. NetPool then calculates the average demanded traffic by each tenant for each resource type i. Specifically, we get the average monitored traffic demand for a tenant k across the adjustment cycle, $AvgD^k = avg(D_k^w)$ over all the windows w. We then multiply this value with the resource vector value for each resource type i to get per-resource and per-tenant average demand, $AvgD_i^k = AvgD^k \times R_i^k$. This is the estimated longer-term, stable traffic of tenant i. Afterward, we sum this value across all tenants to get the sum of long-term traffic across all tenants, $AvgD_i = sum(AvgD_i^k)$. The difference between P_i and this value is the short-term traffic. Thus, for each resource type i, NetPool reserves the corresponding amount, $Reserve_i = P_i - AvgD_i$ to handle the potential short-term traffic spikes in the next adjustment cycle.

4.4.2 NetPool Global Resource Allocation

For long-term traffic, our goal is to perform global resource allocation of the non-reserved resource amounts, *i.e.*, total physical resource amount of a type i minus the reserved amount, $Reserve_i$. Furthermore, we aim to achieve fair resource allocation while being work-conserving. To achieve this goal, we design a global controller that runs at a centralized location and assigns per-tenant, per-type, and per-NIC resources periodically (every adjustment cycle).

Fair per-tenant resource allocation. NetPool fairly allocates the available resource of each type to each tenant based on its demanded resource amount. From Section 4.4.1, we have $AvgD_i^k$ as the demanded resource type i from tenant k. We determine the allocated resource amount A_i^k for type i to tenant k using an adapted Dominant Resource Fairness (DRF) algorithm [61], with the output being $\vec{Allocated}$, a vector of allocated traffic per tenant. DRF allocates resources of different types based on the *dominating* resource type. We adapt DRF for the global controller to scale to process a rack's network resource changes promptly (*i.e.*, finishing within one resource adjustment cycle). As an optimization, we adopt an approximate and iterative version of DRF algorithm, DC-DRF [89], which computes tn approximate allocation within a deadline and improve when deadline increases. We stop the resource allocation exploration when it finds the perfect solution or runs beyond the preset deadline (20 ms), to get timely allocation results. As NetPool adjusts resource assignments frequently, the slight degradation in a particular cycle's assignment is acceptable.

Resource placement. The above step determines the amount of resources (of a type) in the entire pool. Unlike single NICs, assigning an amount to a pool of SmartNICs involves dividing the amount across different SmartNICs under a topology. Ideally, all allocated resource amounts for a tenant are placed on its home NIC (*i.e.*, the NIC directly connected to it), as doing so avoids incurring additional latency to go through a non-local NIC. However, the global controller's allocated amount targets the whole pool's resource availability and can go beyond the available resources in a single NIC, especially when multiple tenants on the same server have a high

demand for the same type of resource. Thus, our goal for resource placement is to achieve minimal steering latency globally across all tenants.

Our idea is to treat every traffic steering from one NIC to another as adding a cost to the whole system and the amount of resources available at a hardware accelerator or as assigned to a tenant as "flow". Minimizing the impact of traffic steering while allocating all the resources of $\vec{Allocated}$ to tenants then becomes a multicommodity min-cost max-flow problem [76, 77, 157]. We now illustrate how we construct the min-cost max-flow graph for a NIC pool step by step.

First, we model each tenant as the source (or sink) graph node and, similarly, the ToR switch as the sink (or source) graph node. We connect the tenant graph nodes to their home NIC with edges having weights of the tenant's allocated resource amount. We connect the ToR graph nodes to all NICs with weights equal to the link bandwidth. Within each NIC, we use a pair of graph nodes to represent one accelerator (resource type) (*e.g.*, R_s and R_e for the encryption engine in NIC1 in Figure 4.8). The weight of the edge between the pair of nodes represents the total capacity of the accelerator.

Second, we introduce the peer links between NICs to represent the possibility of steering traffic from one to another. As there can be multiple types of resources in each NIC, we connect the graph nodes representing the same resource type of two peer NICs (blue edges in Figure 4.8). To represent the performance overhead of sending traffic to more NICs, we give each such edge a constant cost c.

Finally, we add directed edges between two accelerators or between the tenant source node and an accelerator to represent the tenant's demanded chain of acceleration (*e.g.*, tenant b first accesses encryption and then compression). For each such directed edge, we add a special "Adjust" graph node to represent the ratio between the previous node and the subsequent node, as a tenant can be assigned different amounts of resources for different types (accelerators).

With the full graph constructed, we then solve the min-cost max-flow problem. The result shows which steering edges to enable and how many resources to assign for each resource-capacity edge. This information is labeled as $Dispatch$ array in Figure 4.6 and enforced with the

traffic dispatcher at runtime.

Constructing and solving the above min-cost max-flow problems can take long [144, 46]. By default, we perform such problem-solving at every resource adjustment cycle at the global controller. To reduce this scheduling overhead, we apply an optimization to skip re-running the full algorithm when no new tenants are added or removed in a cycle. As our cycle granularity is fine enough for the allocation change to be incremental, we could checkpoint the solver state of the previous cycle and continue the iterative optimization with the incremental traffic difference.

4.4.3 NetPool Local Controller

After the global resource allocation (Section 4.4.2), we get the reserved amount for each resource type i at each NIC to be the amount of resources unallocated by the global controller. The NetPool local controllers running at every NIC use these reserved resources to handle traffic spikes. Specifically, when the received traffic amount exceeds the globally allocated resource at a type, the local controller evenly distributes the exceeded amount across all NICs in the pool. We use this simple algorithm to handle traffic spikes instead of attempting to get to a global optimum so that each NIC can quickly adapt to traffic spikes.

4.4.4 NetPool Data Plane

SmartNIC data path enables packets to be taken from hosts or datacenter network, to be forwarded between NICs in the pool to finish chain offloading, and sent back to datacenter network or target hosts. In this section, we focus on three important things: First, how does the data plane enable sharing of resources over the pool; Second, how does the data plane separate the long-term and ephemeral traffic and enforce different policies on them.

Sharing SmartNIC Resource in a pool. Today's SmartNICs contain various types of computing units, such as general-purpose CPU cores, encryption engines, ML accelerators, FPGA, etc. Some of them can only be time-shared (*e.g.*, a single CPU core), while others can be shared both temporally and spatially (*e.g.*, FPGA). NetPool abstracts different interfaces and sharing

methods into one sharing model. Regardless of the form of the units, we abstract them as *per-packet accelerators*. For each type of resource on one NIC, regardless of the actual hardware communication method, we abstract it as a single work queue on NIC. The accelerator will process one packet from the queue at each time. All queues are lossless in the implementation and will backpressure the sender if the queue is full.

For resources on multiple SmartNICs, NetPool also allows one packet to be processed on multiple SmartNICs in its chain offloading. NetPool enables sharing by forwarding the request packet to different SmartNICs. As we will introduce in Section 4.5, Each packet header will encode the information about its associated tenant and its current step in the chain. On each SmartNIC, we implement a *hardware-accelerated* switch and forward infrastructure to dynamically route one packet to different NICs through the peer links. The forwarding target SmartNIC will finish the chain offloading based on the information within the packet header without extra metadata attached or communication between NICs.

Separating the traffic. Now, we discuss how we identify and enforce the policy for two types of traffic. As shown in Figure 4.6, the first step is to identify and separate different patterns. We achieve this by using a hardware-offloaded packet processing pipeline with traffic metering and packet coloring. When any packet enters the NIC, a decoder determines its tenant and forwards it to a specific traffic metering unit. The unit use token bucket to meter the performance, and mark the packet as green if it is within the traffic rate limit calculated by control plane, we see this packet as part of long term traffic; otherwise, we mark it as red to indicate that it belongs to ephemeral traffic. The rate limit is updated every decision interval based on the allocated traffic vector.

This separation only happens at the start of the chain. The marked color will be persisted in the header packet to keep the information. The assigned color will be restored from the header at the decoder and reused in later offloading steps, even on another SmartNIC.

Enforcing the fair share and placement for long-term traffic. We achieve fairness and low latency by enforcing the global controller's allocation and mapping results. After being

separated, the long-term traffic will be dispatched to its next physical resource based on the resource allocation vector. For example, in Figure 4.6, the allocation vector shows that it is configured to use 2 units of accelerator unit on NIC 1 (local NIC) and 1 unit both on NICs 2 and 3, then the dispatcher will be configured to forward 50% of packets to the local accelerator and 25% each to NIC 2 and 3.

As the given share is allocated to the tenant, the dispatch could safely dispatch the packets without worrying about offloading resources' capacity. This dispatching happens after each step of the offloading chain. The weighted dispatching algorithm is implemented in a hardware-accelerated module using ring hash, as will described in Section 4.5.

Sharing and isolating ephemeral traffic. Different from long-term traffic, ephemeral traffic is fast-changing and unpredictable and may show very different patterns across different flows. It changes quickly (usually at μs level), and random distribution requires it to be handled at the packet level. In NetPool data path, we prioritize packet-level high utilization for ephemeral traffic with tenant-level isolation.

After separation, the ephemeral traffic (red flows in Figure 4.6) will be handled by a per-resource dispatching unit and be dispatched to its next required resource according to the reserved resource mapping, similar to the long-term traffic. Although the dispatcher works locally, it evenly dispatches packets onto reserved resources, ensuring that each packet in ephemeral traffic will only be steered once for a single resource type. As the amount of ephemeral traffic is unpredictable for each tenant, we need a data path mechanism to ensure the isolation between different tenants. The isolation is enforced at the entrance of each ephemeral dispatcher. To implement the isolation, we insert a work-conserving weighted fair queuing (WFQ) scheduler between the tenant queues and the dispatcher to fairly admit tenants into the queue based on the weight of the allocated resource amount. The dispatcher will try to dispatch packets until all destinations fully utilize the resources allocated.

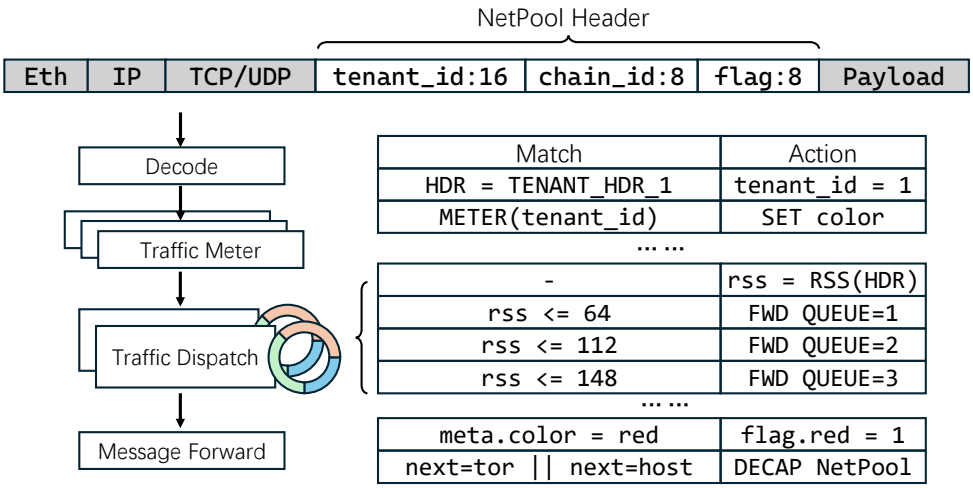

Figure 4.9. Hardware Accelerated Implementation of NetPool Datapath.

4.4.5 NetPool Reliability

NetPool's reliability implications are in two facets. First, our proposed rack-scale architecture connects multiple servers to one SmartNIC. Thus, if a SmartNIC loses its connectivity, all servers connected to it could be disconnected from the rest of the data center. Note that a SmartNIC can lose its accelerator processing but still maintain basic packet forwarding. In this case, NetPool controller treats this failed SmartNIC as containing zero accelerator resource and steers traffic to other SmartNICs in the pool. Also, a potential mitigation is to connect each server to two SmartNICs in the pool, which allows for one to fail.

Second, software components in NetPool can fail, including local controllers and the global controller. When the former fails, NetPool degrades to a global-control-plane-only solution by not handling traffic spikes but still handling long-term traffic. When the global controller fails, NetPool degrades to a local-only control plane, where traffic exceeding a SmartNIC would be directed to the remaining SmartNICs in the pool without any global coordination. Future works can investigate the possibility of adding a high-availability shadow global controller to take over the global control plane when the primary global controller fails.

4.5 Implementation

We implement our design on a Mellanox BlueField-2 SmartNIC with 3.4K C++ LOC. The NetPool data path is implemented as a platform-independent, hardware-offloaded packet processing pipeline, leveraging the power of *dpdk* generic flow [1], traffic metering [3], and traffic manager [2] libraries. The control path is implemented as user-space processes that fetch metrics and send decisions through assigned high-priority DPDK ports. We implement the deadline-constrained DRF algorithm together with a check-point-enabled simplex method solver for resource allocation. We implement three offloading services with NetPool resource interface, including RegEx matching, compression acceleration, and packet encryption. The former two utilize hardware accelerators, and the latter one utilizes the ARMv8 cryptographic extension and runs on ARM cores.

Host and datacenter network Interface. We provide the interface to hosts as a kernel-bypassed networking library, similar to the DPDK interface. We provide a thin host driver layer, mostly a set of ring buffers that connect to its home NIC's ring buffers for each tenant. When a new tenant is added/removed, the corresponding rules about tenant identification and resource requirements are provided to the driver and inserted into NetPool's data and control path. To enable packets to be exchanged between hosts, as shown in Figure 4.9, we modify the packet format and insert a header after the IP and transport headers to encode the tenant ID, offloading chain progress, and other metadata that need to be preserved when passing across different NICs. The header is automatically added and removed when it enters and leaves NetPool by the NetPool data path. NICs are connected using peer links built on top of reliable Ethernet.

Datapath implementation. The major challenge in the implementation is implementing a full hardware accelerated data path. Notably, we implement a fully offloaded weight-based packet dispatcher, as multiple dispatches are required in chain offloading for both long-term and ephemeral traffic. DPDK-provided software dispatcher will bring significant performance overhead (up to over 400 μs). The main idea that makes the implementation possible is converting

a weighted dispatch to a ring hash lookup. As shown in Figure 4.9, we leverage the capability of hardware RSS to compute a hash value (RSS value). Multiple upper bounded flow match-action entries are inserted; each would match multiple hash values within a range whose size is proportional to its weight, and the action would steer the packet to the target queue. The rules cover all possible values of hash and effectively form a hash ring that forwards packets based on the hash value. The design greatly improves both updating and lookup latency (within 1 μs).

Portability. The NetPool system could be separated into two parts: the NetPool framework, which covers the host driver, data plane, and global controller, and the NetPool offloading accelerators. We implement the NetPool framework using standard DPDK $flow$ and $traffic\ management$ libraries, ensuring broad portability across various platforms. Resources within the framework are abstracted as generalized software or hardware components. These components retrieve packets from DPDK queues and process them, posting the completed packets into another queue. This design allows flexibility for offloading tasks across both CPU cores and hardware accelerators.

NetPool framework is built upon standard DPDK libraries, making it inherently portable. Additionally, hardware-specific offloading resources can be easily integrated through our resource interface. For instance, while the initial implementation of NetPool was done on BlueField NICs, we were able to port the code to Octeon TX2 NICs [5] with minimal changes— 334 lines of code, primarily involved invoking the hardware accelerators specific to the new architecture.

Furthermore, since the steering packet exchanged between NICs is architecture-independent, it is feasible to incorporate different types of SmartNICs within the same resource pool. A key advantage of mixing different SmartNICs is the ability to leverage varied hardware acceleration capabilities, enabling a more flexible and utilization-optimized system.

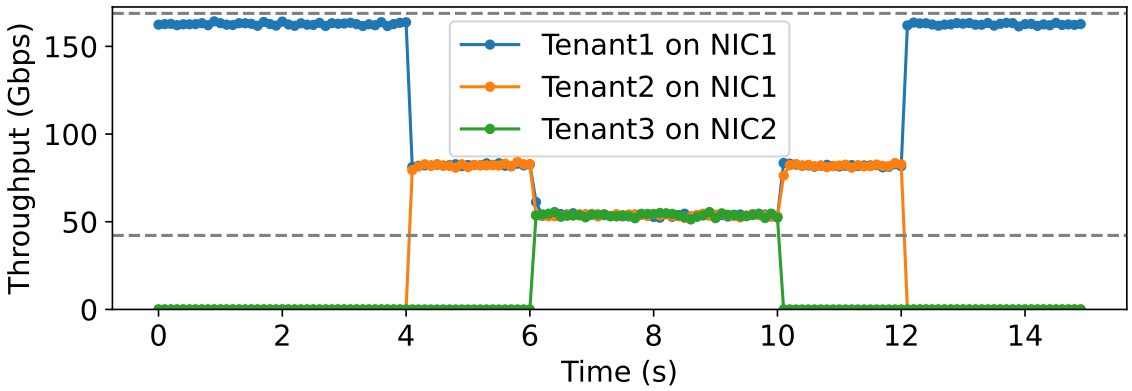

Figure 4.10. Fairsharing of Domain Resources (Accelerator) Across Three Flows.

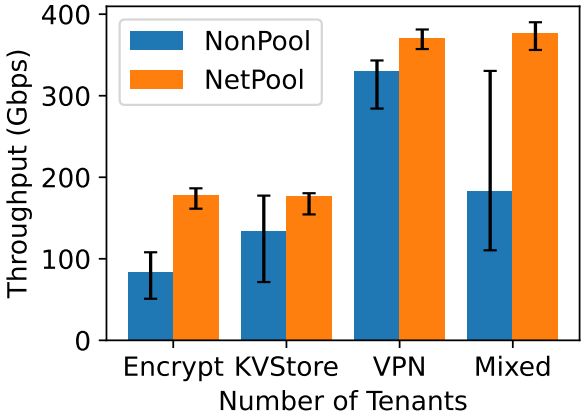

Figure 4.11. Overall Application Throughput.

4.6 Evaluation Results

This section presents the evaluation results of NetPool. We first describe our testbed and our implemented applications. We then present the overall cost-saving benefits and performance implications of NetPool. Afterward, we present deep dives to understand NetPool's performance and sensitivity with microbenchmark tests.

4.6.1 Testbed Setup and Baselines

Our evaluation testbed consists of two server machines and two 100Gbps BlueField-2 [35] SmartNICs that we use to emulate a rack-level cluster. Each BlueField-2 SmartNIC

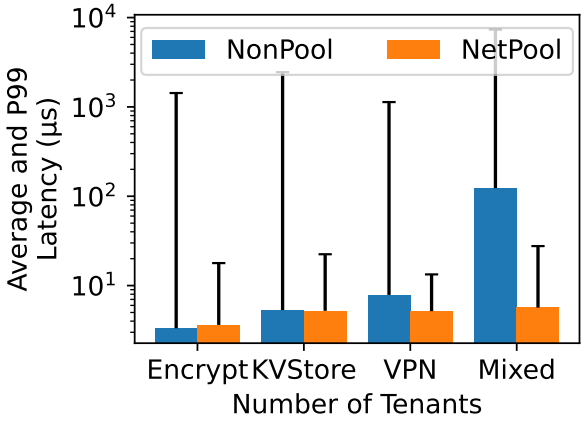

Figure 4.12. Overall Application Latency.

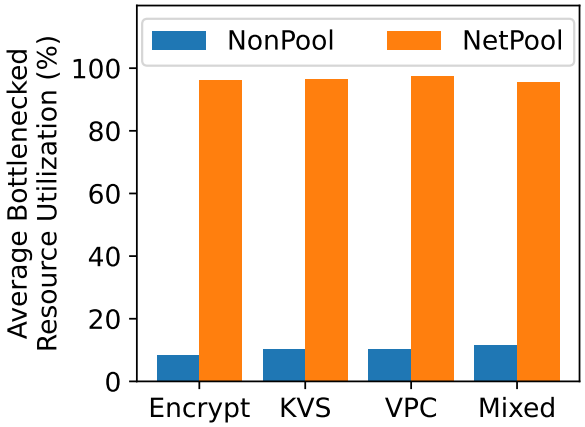

Figure 4.13. The utilization with different aplications.

contains 8 ARM cores and 16 GB of memory, an encryption accelerator, and a compression accelerator. The NICs are connected to a 100Gbps Mellanox Ethernet switch. Additionally, they are connected to each other with a 100Gbps peer link. We partition the resources of each physical BlueField-2 NIC into two virtual NICs (vNICs), each assigned an isolated PCIe physical function (PF) to connect to the host server. Each vNIC independently ran a complete instance of NetPool NIC. Each vNIC communicated with the other vNIC on the same physical NIC using a hardware-accelerated Open vSwitch (OVS) [134, 166] bridge.

Each server is equipped with dual Intel Xeon Gold 6258R CPUs and 256 GB of DDR4-2933 memory. Each server emulates 16 virtual servers, each assigned with an isolated PCIe

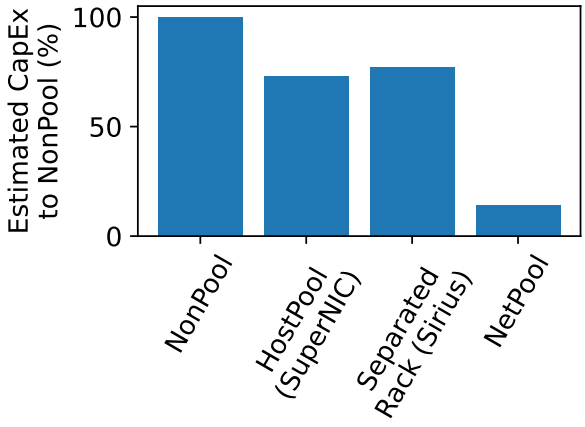

Figure 4.14. The CapEx of different SmartNIC deployment methods.

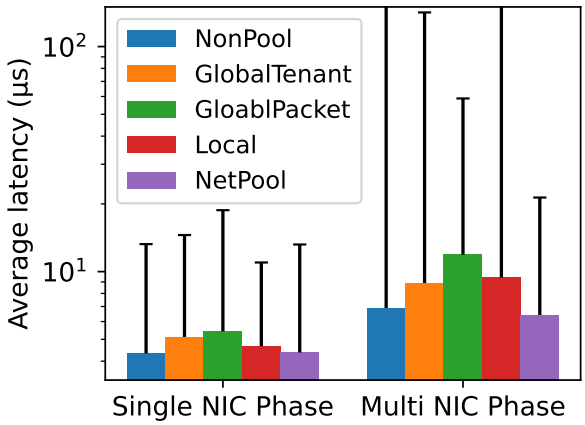

Figure 4.15. Average latency on L7 encryption application.

virtual function (VF). Eight of them are connected to one vNIC, and the rest of them are assigned to the other. This allowed us to evaluate the system using 32 distinct virtual servers spread across four virtual SmartNICs.

4.6.2 Application Workloads

We implement three representative applications to run on NetPool, each one demonstrating a unique access pattern.

L7 encryption. We use this application to demonstrate that NetPool has the ability to handle resource requirements beyond one SmartNIC's capacity and utilize the resources in the pool. We

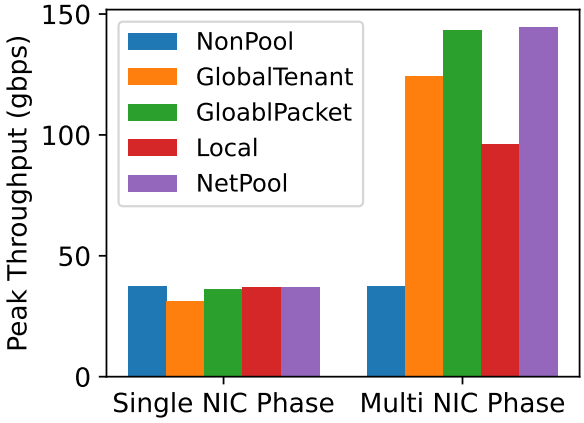

Figure 4.16. The throughput of L7 encryption application.

generate a flow with increasing traffic with tenants on a single NIC, increasing from 0 to the pool's processing capacity. We generate background traffic evenly to other tenants at the same time and decrease them accordingly to keep the total traffic amount within the pool's capacity.

Key-value store with compression and encryption. We use this application to demonstrate NetPool's ability to handle highly skewed workloads across different servers. The skewed workloads are commonly used for evaluating key-value stores [82] and are supported by observations from real-world deployments. In the experiment, we generate workloads with a fixed total traffic bandwidth (our assumed pool-level max utilization), and the distribution of each individual server follows a Zipf distribution with varying skewness parameters. As illustrated by Figure 4.17, the traffic requirements could go higher than a single NIC's capability in both pooling and normal provisioned cases.

Virtual private cloud with firewall, encryption, and NAT. We use this chain offloading application to demonstrate NetPool's ability to handle short-term bursts and traffic changes. This sudden traffic change could cause GBs of traffic change within a second, which is common in networking and is well evaluated in network systems [82]. We generate traces with sudden traffic changes at the boundary of every second, with different source and target rates, to evaluate NetPool's ability to handle short-term traffic changes.

Mixed workload. Finally, we test a case that mixes all three above traffics together. We randomly

assign the traffic to different tenants. The three different applications feature different offloading chains and share (compete for) multiple types of resources, including encryption engines and in-out bandwidth. We evaluate the ability to share fairly and isolate the resources in this setup.

4.6.3 Network Resource Consolidation Benefits

We first demonstrate that NetPool's capability of consolidating and fair sharing of network resources as a pool. We deploy three single resource tenants in NetPool; two of them are connected to NIC1, and the last one is connected to NIC2. We launch them in order and log the throughput change over time in Figure 4.10. After we launch tenant 1, although it is only connected to one SmartNIC, it quickly takes utilization of resources beyond one SmartNIC, up to 97% of the total resources of all 4 NICs. Later, after the other two tenants are launched, they fair share the resources in the pool, utilizing 95.2% of total resources. Each tenant stably utilizes resources beyond a single NIC's limit, thanks to NetPool's fair resource allocation and low latency data path design.

4.6.4 Overall Application Performance

Baselines. We compare NetPool with today's data-center architecture, where each server is attached to a SmartNIC with different capacities (*NonPool*).

First, we present the benefits and performance overhead under different access patterns with the overall performance of applications. The total amount of generated traffic is set to the capacity of the pool, which aligns with our design goal. We compare NetPool to non-disaggregated baselines, where we attach a SmartNIC with half the capacity as the SmartNIC in NetPool to each end-host (*i.e.*, total rack-level capacity is four times that of NetPool).

Figure 4.11 shows the aggregated throughput of the testbed. The error bars show the best and worst cases under different patterns for the same application. NetPool outperforms the NonPool solution, even as NonPool has four times more resources than NetPool. Encryption and mixed workload benefit more, as sharing resources beyond a single NIC happens more

frequently in these cases. Figure 4.12 presents the latency results. Similarly, NetPool largely reduces application average and p99 latencies compared to the baseline.

Figure 4.13 shows the overall resource utilization. NetPool handles different traffic patterns, including those with extreme traffic changes or ephemeral bursts, while keeping constantly high resource utilization (over 87.3% under all scenarios) across different applications. In contrast, the non-disaggregated solution suffers from low utilization and throughput, as it can only utilize the resources within the directly attached SmartNIC.

Overall, NetPool achieves a CapEx cost and energy saving of around K, where K is the ratio of servers to SmartNICs (eight in our default configurations). Figure 4.14 shows the estimated capital expenses (CapEx) of different SmartNIC deployment model that provides similar capacity. Considering the cost of SmartNIC hardware, as well as datacenter resources (including ToR port density, network cabling, and rack space), NetPool features a disaggregated sharing model and brings high resource utilization without needing to change the existing network topology.

In summary, compared to today's data-center that provisions a SmartNIC for each individual server, NetPool reduces total network resources by multiple times while maintaining comparable application performance. Under cases that reach per NIC resource limitation (which is common with SmartNIC offloading), NetPool could bring up 44% higher throughput and over 72% latency reduction by sharing the resources within a pool to mitigate congestions.

4.6.5 Performance Breakdown

In this section, we examine NetPool's performance across various traffic patterns, highlighting its ability to manage different traffic and sharing requests. We compare NetPool with three alternative resource pooling implementations that use the same hardware architecture as NetPool but with different resource allocation algorithms and mechanisms: an ad-hoc steering system (*Local*) that redirects tasks to peer NICs when local resources are exhausted, a global resource allocation approach that performs tenant level resource allocation (*GlobalTenant*),

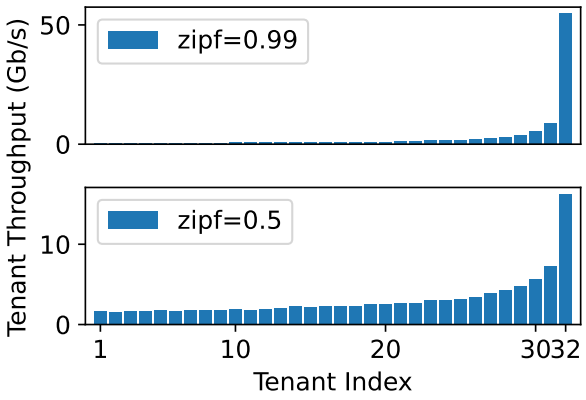

Figure 4.17. The Skewed Traffic Distribution Pattern.

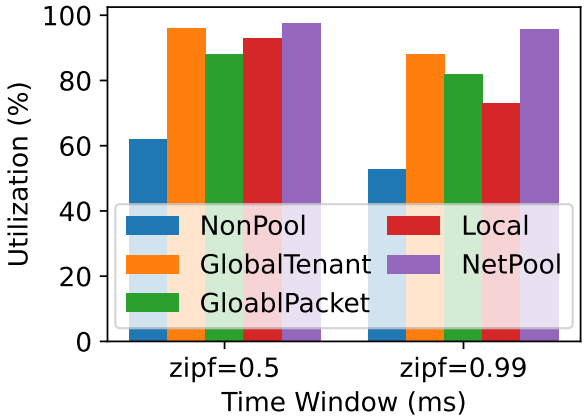

Figure 4.18. The Utilization under Zipf Traffic Distribution.

and a centralized resource-fair packet scheduling system (*GlobalPacket*) that fairly distributes resources at the packet level. All three alternatives are implemented by using NetPool's hardware-accelerated data path to ensure a fair comparison.

Handling Traffic Scaling Beyond Single NIC

We evaluate NetPool's resource scaling ability using an L7 encryption application. In this experiment, a single tenant is placed on one NIC, and we gradually increase the encryption unit's resource requirements from zero to the full capacity of the resource pool. The results are divided into two phases: in the first phase, we scale the traffic from zero to the capacity of a single NIC; in the second phase, we scale the traffic from one NIC's capacity to the full capacity

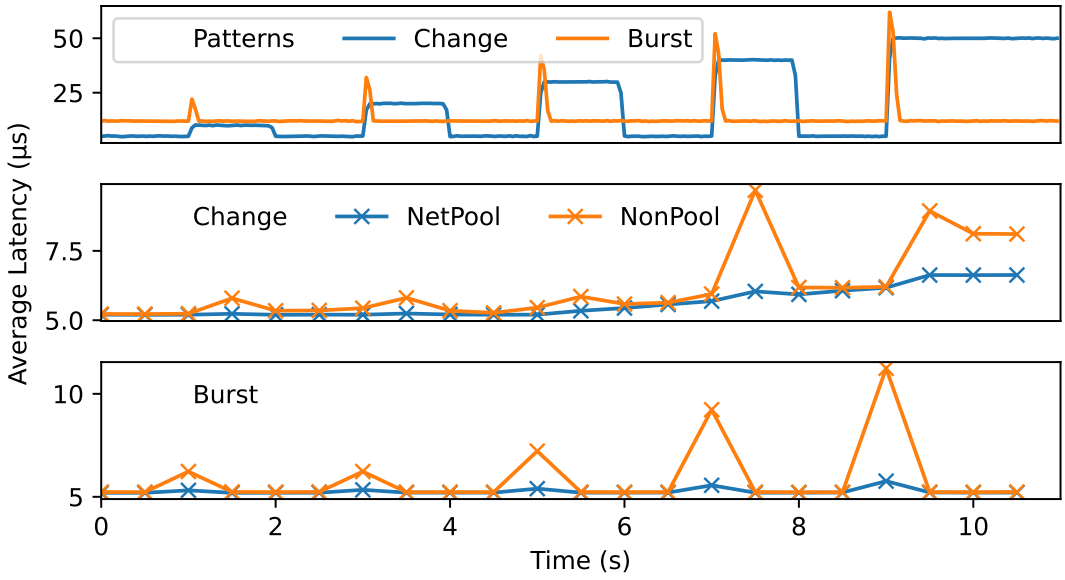

Figure 4.19. The Latency Changes under Ephemeral Patterns.

of the resource pool.

Figure 4.15 shows the performance overhead, measured by latency in different phases. NetPool keeps one of the lowest latency. In the local NIC phase, NetPool's global allocator adds negligible (up to 1.2%) overhead compared to a NonPool solution. GlobalPacket has the highest latency as all the decisions need to go through a centralized scheduler, even when all packets are from a single NIC. In the Multi-NIC phase, both Local and GlobalPacket methods' latency spikes as multiple packet steerings are required to find the available unit. Local performs worse, as a lack of global view could cause one packet to be redirected multiple times before it could be processed. By avoiding the unnecessary steering (from global tenant-level allocation) and handling the ephemeral spikes efficiently, NetPool keeps stable latency and tail latency.

Figure 4.16 evaluates the total throughput in different phase. This challenges the systems' ability to utilize the available resources inside a pool. GloablTenant's utilization drops in multi-NIC phases as its allocation is not fast enough to capture the swift ephemeral traffic changes. Local methods' lack of global view causes low utilization on far-side processing units, as well as long latency that triggers packet drop. NetPool have the highest resources and over 93% resource

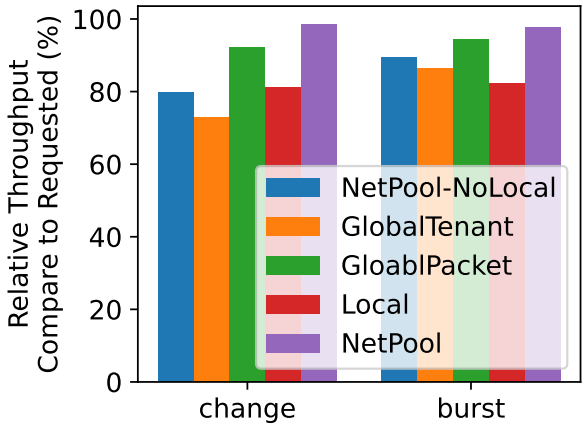

Figure 4.20. Throughput under Different Ephemeral Changes.

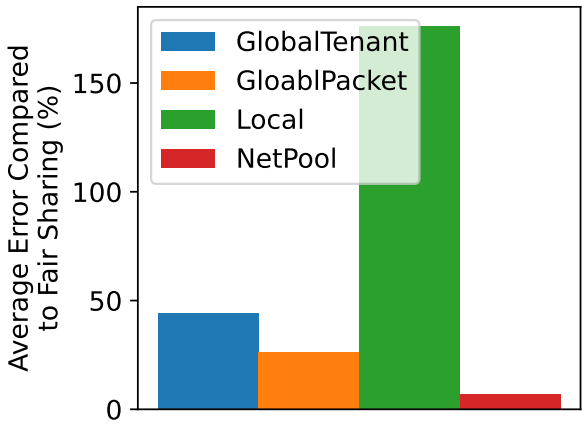

Figure 4.21. Average fairness of mixed pattern.

utilization on its peak throughput.

Handling Skewed Traffic Distribution

In this section, we evaluate NetPool's performance when traffic amount distribution over different servers is skewed. With a non-pooling solution, the traffic will flood on servers and likely cause performance downgrade and service failure.

Figure 4.18 shows the throughput and utilization comparison of different allocation methods under different skewness. NetPool handles the long-term skewed traffic in the pool by allocating the resource in the pool to adapt the hot points in the pool and gets the best performance

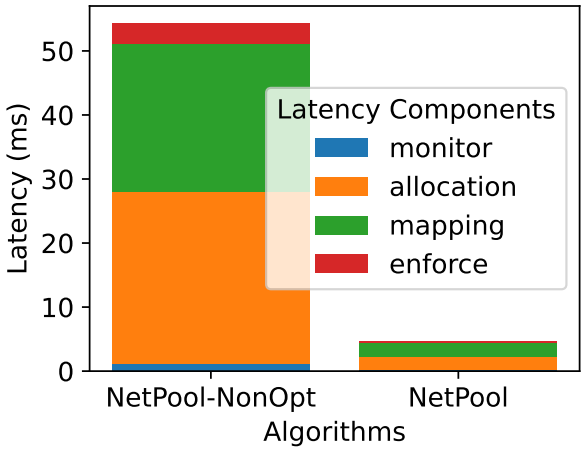

Figure 4.22. The Breakdown of NetPool Controller.

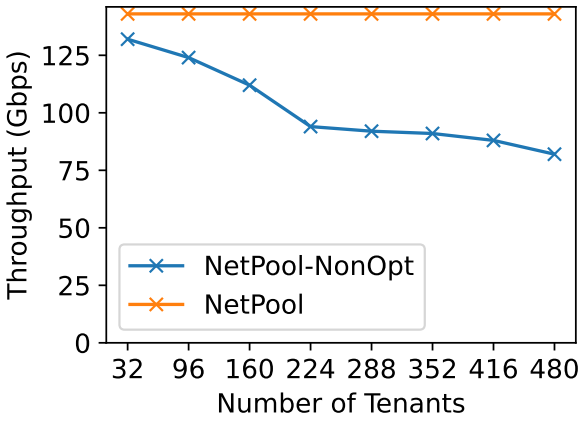

Figure 4.23. NetPool Scalability.

over different solutions, especially under the more extreme ($zipf = 0.99$) distribution patterns. NetPool achieves both high utilization and fair allocation of resources.

Handling Ephemeral Traffic Changes

We evaluate two different cases of ephemeral traffic. First, we change the traffic load every second (*Change*). Second is the burst traffic, where we keep the average throughput unchanged and introduce a burstiness with different rates at the beginning of each second (*Burst*).

Figure 4.19 shows the traffic pattern and corresponding latency changes under different methods. Similar to latency, the drawbacks of the solutions mentioned above also affect the

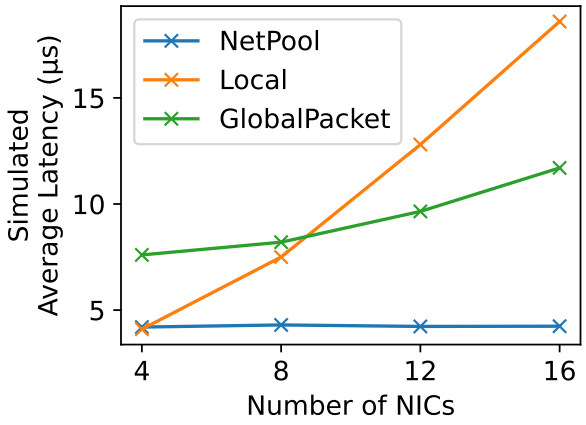

Figure 4.24. Comparing different packet-level steering methods.

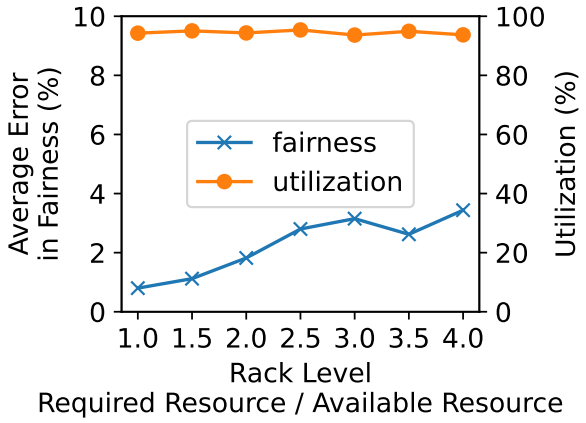

Figure 4.25. Utilization and Fairness under Overcommitment.

throughput, as shown in Figure 4.20. NetPool achieves lower latency than GlobalTenant solution over time for both workloads. Due to the slow response and full utilization, the GlobalTenant solution fails to respond to spikes and is slow to change with the changing pattern. The staled resource allocation could sub-allocate the resource (leading to increasing latency) and over-allocate resources (leading to lower throughput and utilization) on different changing patterns. And, as a spike could easily be as large as the total capacity of the cluster in a short time, in the GloablPacket solution, a centralized scheduler without enough resources (which is the consolidation goal) will be easily flooded by the packets and results in long schedule time and lower throughput.

Handling Mixed Resource Requirements

Lastly, we evaluate the utilization of mixed patterns. We randomly create flows with the three applications we mentioned above, whose total resource requirement will be the total pool capacity. This challenges NetPool's ability to keep multiple resources fairly allocated.

Figure 4.21 shows the average fairness of different pooling methods, compared to an offline computed, fully domain resource fair allocation method. As seen, NetPool achieves optimal fairness with mixed resources and fast-changing requirements. The unfair resource allocation behavior is observed in both GlobalTenant and GlobalPacket methods due to the high centralized decision overhead, which cannot match the speed of flow traffic change and flow creation. NetPool's optimized centralized scheduler ensures real-time fair allocation. With per-resource isolation, NetPool's local traffic handling only causes up to 11% unfair traffic, compared to over 37% error when using other methods.

4.6.6 Microbenchmark Results

In this section, we take an in-depth view of NetPool's technical decisions with synthetic traces and simulations.

Global Controller Scalability

The design goal of NetPool is to support tens of SmartNICs to be deployed in the cluster with hundreds of tenants. We simulated the traffic pattern with up to 480 tenants and evaluated NetPool performance. Figure 4.22 shows the breakdown of controller decision time. NetPool's deadline-constrained implementation and optimized resource allocation greatly reduce the search time for finding a fair resource allocation while keeping strong fairness between tenants. Figure 4.23 demonstrates that the optimizations enable NetPool's global controller to scale as more tenants are added to the system. NetPool could keep a relatively stable latency (up to 14.7 μs at rack scale) when scaling to support hundreds of tenants. In comparison, the unoptimized algorithm takes up to 243 milliseconds to finish the cycle for 480 hosts. Also, the

utilization will be largely dropped to 63.4% due to the long decision cycles, while NetPool keeps a constant high utilization of over 93%.

Ephemeral Traffic Steering

Figure 4.24 shows the latency comparison with different methods that handle ephemeral traffic at packet level. NetPool outperforms other scheduling methods, thanks to the design of resource reservation and fully local traffic steering. When the number of NICs in the pool increases, Local suffers from frequent packet re-schedule while GlobalPacket is threshed by its centralized global controller.

Performance with Overcommitment

By design, NetPool should have enough resources for peak requirements in the resource pool. However, when the total required traffic is larger than the capability of the resource pool, we could still maintain fairness and isolation between tenants. Figure 4.25 shows that throughput and fairness when overcommitting. While NetPool cannot saturate the required bandwidth in this rare case, it could still keep strong performance isolation and near-optimal resource utilization without being affected by the scarcity of resources.

4.7 Related Works

Unlike other types of resources, network disaggregation has only received limited attention. One of the early forms of network disaggregation is multi-host NICs [4, 78, 62]. One multi-host NIC connects to multiple servers (*e.g.*, 2 or 4) via PCIe connections and provides traditional Layer-1/2 functionalities. These works only provide consolidation at a single NIC level. When traffic goes beyond the NIC, they have to slow down traffic, impacting application performance. In contrast, NetPool pools network resources at the rack level and allows one end host to use resources of the whole pool.

Recently, Bansal *et al.*, proposed Sirius, a system to disaggregate network function

accelerators into a separate rack [29]. Although sharing the network resource disaggregation and consolidation idea with NetPool, NetPool differs in several key aspects from Sirius. First, Sirius disaggregates network accelerators to a separate rack from compute racks, while NetPool's pooling happens within the same rack as where applications run. Second, servers in Sirius are still equipped with regular NICs connected to ToR switches in the compute racks, while NetPool's servers directly connect to SmartNICs in our pool. Third, all network devices in the Sirius pool are treated equally, as they all have the same distance from compute servers, while NetPool treats SmartNICs in our pool differently as they have different proximity to servers. Finally, NetPool supports multiple types of network resources and focuses on the fair sharing of these resources across tenants, which is not addressed in Sirius.

4.8 Conclusion

We propose NetPool, a rack-level network-resource-pooling solution. Via the combination of a global control plane, a NIC-local control plane, and a uniform data plane, NetPool significantly reduces data-centers' network resources while maintaining performance on par with isolated SmartNICs and ensuring fairness in a multi-tenant environment.

4.9 Acknowledgement

Chapter 4, in part, is a reprint of Zhiyuan Guo, Arvind Krishnamurthy, Yiying Zhang, "Network Functionality Disaggregation and Consolidation with NetPool", has been prepared for publication. The dissertation author was the primary investigator and author author of this paper.

Chapter 5

Conclusion and Future Work

The challenge of data-center resource management is intensifying as applications evolve rapidly and as heterogeneous, domain-specific accelerators proliferate. Traditional monolithic servers—where compute, memory, and storage are statically bound—restrict innovation and lead to poor utilization. Hardware resource disaggregation, which decomposes servers into independent, network-attached resource pools, promises improved manageability, higher utilization, and stronger failure isolation. Yet, when this work began, it was unclear how disaggregation could be deployed in practice or whether it could sustain acceptable performance.

Resource disaggregation addresses the data center's dual need for efficiency and flexibility. By exposing a uniform interface for both local and remote resources, a disaggregated system lets applications consume capacity beyond a single server boundary. However, early prototypes suffered from prohibitive overheads because the "vertical" layers (hardware, network, OS, and application) were designed in isolation.

In this dissertation, we traced those overheads to semantic mismatches across layers, then introduced the design principle of semantic-guided co-design. We translated that principle into several concrete systems that jointly optimize disaggregated hardware, network fabrics, runtimes, and application abstractions. Collectively, the projects demonstrate that *feasibility, performance, and scalability* can be achieved simultaneously. They also validate the practical benefits of disaggregation: tighter resource packing, improved failure isolation, and elastic growth.

By co-designing a disaggregated memory cache with application-level insight, chapter 2 presents **Mira**, which delivers *transparent and efficient* far-memory to diverse applications without invasive data-structure refactoring or opaque swap-based paging. Mira uses static program analysis at compile time to classify objects, predict access phases, and orchestrate placement and prefetching across local DRAM and RDMA-backed memory. It adapts online to workload and system dynamics. On data-intensive workloads, Mira improves execution time by up to 18× and halves tail latency versus the best swap-based and API-driven approaches, proving that intelligent cross-layer insight is key to near-local performance.

By co-designing specialized hardware at the memory node with unified management and networking support at the compute node, chapter 3 shows that **Clio** enables disaggregated memory that is *dynamic* yet still offers local-like latency and full line-rate bandwidth. The redesign collapses the traditional read/write path—from CPU instructions, through page tables, to DIMMs—into a streamlined networking protocol that eliminates duplicated state and communication-centric overheads. Clio introduces a hardware virtual-memory engine, a 100 Gbps packet network, and an offload framework inside an FPGA memory blade, paired with a lightweight client library. Across micro-benchmarks and real workloads, Clio sustains a 2.5 µs median load-to-use latency, scales throughput linearly with additional blades, and reduces energy by up to 3.4× compared with CPU-centric alternatives.

Further when we shift our focus from a single disaggregated application to cluster level, how to coordinate applications and manage resources become a new challenge. By co-designing the network resource managemer with data-center traffic patterns, chapter 4 introduces **NetPool**, which serves network bandwidth and in-network computation as a disaggregated, rack-scale service that reacts at microsecond time scales. NetPool moves packet-processing offloads from every host NIC into a shared pool of SmartNICs. A control plane elastically replicates chains of network functions, while a highly parallel data plane multiplexes many tenants. The design boosts overall utilization, and maintains low-latency packet handling—showing that disaggregation principles extend naturally beyond compute and memory to the network datapath.

5.1 Future Work

With ever-growing application demands, we expect resource disaggregation to emerge as a new *standard abstraction* for datacenter resources. Much like Infrastructure as a Service (IaaS) and Function as a Service (FaaS), a "Resource as a Service" model would let applications acquire capacity from datacenter-wide pools, independent of the underlying hardware topology, and thereby enjoy on-demand utilization, flexible resource ratios, and effectively unlimited scalability.

5.1.1 Boosting Disaggregation Research with Composable Components

Progress in resource-disaggregation research has been sluggish because the community still lacks essential tooling. Without firm foundations, advances remain hard to reproduce, compare, or compose. A particularly impactful direction is the creation of *configurable and portable components*. Useful items include:

1) **A comprehensive benchmark suite for resource disaggregation.** Today, evaluations rely on ad-hoc microbenchmarks that reveal only fragments of system behavior and could not provide a comprehensive view on disaggregated systems. The community needs an open, versioned suite that spans: *Microbenchmarks* for latency, bandwidth, and consistency under varying topologies; *Application kernels* such as key–value stores, analytics pipelines, and ML training loops; *End-to-end workloads* that mix CPU, memory, storage, and accelerator pressure. A shared suite would enable fair comparisons, surface trade-offs, and accelerate iterative improvement.

2) **Composable disaggregation components.** Consider remote memory: kernel-space RDMA networking and remote paging have been explored, yet no stable, reusable toolkit exists. Neither a maintained kernel module nor a user-space path built on `userfaultfd` [24] is available across kernel versions. An understandable, well-tested data path—packaged like a library—would let prototypes migrate across kernels, foster broader experimentation, and let

other subsystems (e.g., live migration, checkpointing) reap the same benefits.

3) **A separated, declarative policy interface.** Current prototypes entangle mechanism with hard-wired, implementation coupled policies. As an example, for disaggregation memory placement, replication, and admission control are usually hard coupled with datapath implementations. A small, declarative API expressed as, for example, eBPF hooks or a domain-specific language, would let researchers plug in new scheduling or consistency policies without touching the data path. Decoupling policy from mechanism reduces code churn, broadens participation, and ultimately shortens the path from idea to reproducible result.

5.1.2 Clean-Slate Redesign of the Resource-Disaggregation Stack

Native Programming Languages for Disaggregated Resources. Contemporary languages presuppose a uniform memory hierarchy; they cannot describe, for example, one object residing in remote HBM while another streams through a SmartNIC. Rich type-system features, such as *substructural types* and *generational references*, can expose location and mobility information of in-memory objects, allowing the compiler to reason explicitly about placement, prefetching, and replication. Embedding these resource-aware types would keep programs safe, verifiable, and portable even as the underlying pool scales from a laptop-sized cluster to an entire data center.

Portable Resource Behaviors via a Resource Contract. This dissertation treats application and resource behaviors on a case-by-case basis: for each resource, behavior, and system layer, we discover behaviors independently and embed bespoke protocols and message formats. A unified *resource contract*, capturing lifetime, admissible reordering, and inter-resource dependencies, would separate policy from mechanism and enable reuse across memory, storage, accelerators, and the network. Such a contract would also clarify intent in non-disaggregated deployments.

5.1.3 Beyond Efficiency: Leveraging Disaggregation for New Capabilities

Fault Tolerance. Resource disaggregation proposes both challenges and oppotunities for building fault tolerance systems. Now applications could be executed on multiple, physically isolated regions, each could be considered an isolated failure domain. New capabilities could be brought, including partitioning memory pools by application region can confine faults and enable graceful degradation. However, co-locating applications on single resource could be more common and couples the fault handling across them. Future placement algorithms must balance latency constraints, heterogeneous devices, and energy budgets while maximizing isolation.

Energy-Efficiency Datacenter. Resource disaggregation could optimize for energy efficiency with the proposed co-designed approach, through considering the resources' energy characteristics combined with performance. Because resources are no longer captive to a single server, idle pools can be power-gated independently. Coordinated pool-level resource request steering and demand-driven activation could yield substantial energy savings.

Edge and IoT Environments. A semantics-driven disaggregation model extends naturally to personal computing, robotics, and IoT: edge devices would expose fragmented pools of compute, memory, and sensors. The same contracts that optimize data-center workloads can adapt applications to volatile, bandwidth-limited links and diverse failure modes.

I ultimately envision a world in which a single program could be built for a *unified, semantically rich resource pool*. It could be written once and runs efficiently on a laptop, a home cluster of smart appliances, or an entire cloud data center adapting to the resource demands changes. Realizing this vision will require shared baselines, modular runtimes, resource-aware languages, and portable contracts. The work presented in this dissertation constitutes one set of stepping stones toward that goal.

Bibliography

[1] DPDK - Generic flow API (rte_flow) - Documentation. https://doc.dpdk.org/guides-20.11/prog_guide/rte_flow.html.

[2] DPDK - raffic Management API - Documentation. https://doc.dpdk.org/guides-21.08/prog_guide/traffic_management.html.

[3] DPDK - Traffic Metering and Policing API - Documentation. https://doc.dpdk.org/guides-24.07/prog_guide/traffic_metering_and_policing.html.

[4] Facebook Multi-Node Server Platform: Yosemite Design Specification. https://www.opencompute.org/documents/multi-node-server-platform-yosemite-v05.

[5] OCTEON TX2 Infrastructure Processor Family. https://www.marvell.com/content/dam/marvell/en/company/media-kit/infrastructure-processors/marvell-octeon-tx2-press-deck.pdf.

[6] YCSB Github Repository. https://github.com/brianfrankcooper/YCSB.

[7] MLIR SPIR-V Dialect, 2020.

[8] Writing dataflow analyses in mlir, 2021.

[9] Llama.cpp 30B runs with only 6GB of RAM now, 2023.

[10] MLIR llvm Dialect. https://mlir.llvm.org/docs/Dialects/LLVM/, 2023.

[11] MLIR memref Dialect. https://mlir.llvm.org/docs/Dialects/MemRef/, 2023.

[12] ONNX-MLIR. https://github.com/onnx/onnx-mlir, 2023.

[13] Open Neural Network Exchange. https://onnx.ai/, 2023.

[14] Intel Xeon Gold 5128. https://ark.intel.com/content/www/us/en/ark/products/192444/intel-xeon-gold-5218-processor-22m-cache-2-30-ghz.html.

[15] Marcos K. Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. Designing Far Memory Data Structures: Think Outside the Box. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '19)*, Bertinoro, Italy, May 2019.

[16] Hasan Al Maruf and Mosharaf Chowdhury. Effectively Prefetching Remote Memory with Leap. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference (ATC '20)*, Virtual, 2020.

[17] Alibaba. "pangu – the high performance distributed file system by alibaba cloud". https://www.alibabacloud.com/blog/pangu-the-high-performance-distributed-file-system-by-alibaba-cloud_594059.

[18] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*.

[19] Amazon. Amazon elastic block store. https://aws.amazon.com/ebs/?nc1=h_ls, 2019.

[20] Amazon. Amazon s3. https://aws.amazon.com/s3/, 2019.

[21] Combinatorial optimization Single-depot vehicle scheduling, 2011.

[22] Sebastian Angel, Mihir Nanavati, and Siddhartha Sen. Disaggregation and the Application. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '20)*.

[23] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in high-speed nics. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*.

[24] Linux Kernel Archieves. *Userfaultfd — The Linux Kernel documentation*, 2018.

[25] ARMv8. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/armv8-a-architecture-2016-additions.

[26] Krste Asanović. FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers, February 2014. Keynote talk at the 12th USENIX Conference on File and Storage Technologies (FAST '14).

[27] Grant Ayers, Nayana Prasad Nagendra, David I. August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp Moseley, and Parthasarathy Ranganathan. Asmdb: Understanding and mitigating front-end stalls in warehouse-scale computers. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, Phoenix, Arizona, 2019.

[28] Abhiram Balasubramanian, Marek S Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamarić, and Leonid Ryzhyk. System programming in rust: Beyond safety. In *Proceedings of the 16th workshop on hot topics in operating systems*, HotOS '17, Whistler, Canada, 2017.

[29] Deepak Bansal, Gerald DeGrace, Rishabh Tewari, Michal Zygmunt, James Grantham, Silvano Gai, Mario Baldi, Krishna Doddapaneni, Arun Selvarajan, Arunkumar Arumugam, Balakrishnan Raman, Avijit Gupta, Sachin Jain, Deven Jagasia, Evan Langlais, Pranjal Srivastava, Rishiraj Hazarika, Neeraj Motwani, Soumya Tiwari, Stewart Grant, Ranveer Chandra, and Srikanth Kandula. Disaggregating stateful network functions. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1469–1487, Boston, MA, April 2023. USENIX Association.

[30] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Translation caching: Skip, don't walk (the page table). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, 2010.

[31] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280, 2010.

[32] Daniel Berlin, David Edelsohn, and Sebastian Pop. High-level loop optimizations for gcc. In *Proceedings of the 2004 GCC Developers Summit*, 2004.

[33] Christopher Branner-Augmon, Narek Galstyan, Sam Kumar, Emmanuel Amaro, Amy Ousterhout, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 3PO: Programmed Far-Memory Prefetching for Oblivious Applications, 2022.

[34] Peter Braun and Heiner Litz. Understanding memory access patterns for prefetching. In *International Workshop on AI-assisted Design for Architecture (AIDArc), held in conjunction with ISCA*, 2019.

[35] Idan Burstein. Nvidia data center processing unit (dpu) architecture. In *2021 IEEE Hot Chips 33 Symposium (HCS)*, pages 1–20. IEEE, 2021.

[36] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. Rethinking Software Runtimes for Disaggregated Memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*, Virtual, USA, 2021.

[37] Irina Calciu, Ivan Puddu, Aasheesh Kolli, Andreas Nowatzyk, Jayneel Gandhi, Onur Mutlu, and Pratap Subrahmanyam. Project PBerry: FPGA Acceleration for Remote Memory. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '19)*, Bertinoro, Italy, 2019.

[38] Gautam Chakrabarti and Fred Chow PathScale. Structure layout optimizations in the open 64 compiler : Design , implementation and measurements. 2008.

[39] Dehao Chen, Tipp Moseley, and David Xinliang Li. AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications. In *2016 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '16, Edinburgh, UK, 2016.

[40] Brian Cho and Ergin Seyfe. Taking advantage of a disaggregated storage and compute architecture. In *Spark+AI Summit 2019 (SAIS '19)*, San Francisco, CA, USA, April 2019.

[41] Mosharaf Chowdhury, Zhenhua Liu, Ali Ghodsi, and Ion Stoica. {HUG}:{Multi-Resource} fairness for correlated and elastic demands. In *13th USENIX symposium on networked systems design and implementation (NSDI 16)*, pages 407–424, 2016.

[42] CloudLab. https://www.cloudlab.us/.

[43] CXL Consortium. https://www.computeexpresslink.org/.

[44] CXL Consortium. https://www.computeexpresslink.org/.

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

[46] Nina K Detlefsen and Stein W Wallace. The simplex algorithm for multicommodity networks. *Networks: An International Journal*, 39(1):15–28, 2002.

[47] Chen Ding and Ken Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. *Journal of Parallel and Distributed Computing*, 64(1):108–134, 2004.

[48] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. *SIGPLAN Not.*, 33(5):106–117, May 1998.

[49] DPDK. https://www.dpdk.org/.

[50] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI '14)*, Seattle, WA, USA, April 2014.

[51] 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 the 25th Symposium on Operating Systems Principles (SOSP '15)*, Monterey, California, 2015.

[52] Facebook. Introducing bryce canyon: Our next-generation storage platform. https://code.fb.com/data-center-engineering/introducing-bryce-canyon-our-next-generation-storage-platform/, 2017.

[53] Paolo Faraboschi, Kimberly Keeton, Tim Marsland, and Dejan Milojicic. Beyond Processor-centric Operating Systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS '15)*, Kartause Ittingen, Switzerland, May 2015.

[54] Guanyu Feng, Huanqi Cao, Xiaowei Zhu, Bowen Yu, Yuanwei Wang, Zixuan Ma, Shengqi Chen, and Wenguang Chen. TriCache: A User-Transparent Block Cache Enabling High-Performance Out-of-Core Processing with In-Memory Programs. In *16th USENIX*

Symposium on Operating Systems Design and Implementation (OSDI' 22), Carlsbad, CA, jul 2022.

[55] Alex Forencich, Alex C. Snoeren, George Porter, and George Papen. Corundum: An Open-Source 100-Gbps NIC. In *28th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM '20)*, Fayetteville,AK, May 2020.

[56] Michael Galles and Francis Matus. Pensando distributed services architecture. *IEEE Micro*, 41(2):43–49, 2021.

[57] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[58] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, Fei Feng, Yan Zhuang, Fan Liu, Pan Liu, Xingkui Liu, Zhongjie Wu, Junping Wu, Zheng Cao, Chen Tian, Jinbo Wu, Jiaji Zhu, Haiyong Wang, Dennis Cai, and Jiesheng Wu. When cloud storage meets RDMA. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021.

[59] Gen-Z Consortium. https://genzconsortium.org.

[60] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. Multi-Resource Fair Queueing for Packet Processing. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '12)*, Helsinki, Finland, 2012.

[61] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. 2011.

[62] Dan Gibson, Hema Hariharan, Eric Lance, Moray McLaren, Behnam Montazeri, Arjun Singh, Stephen Wang, Hassan M. G. Wassel, Zhehua Wu, Sunghwan Yoo, Raghuraman Balasubramanian, Prashant Chandra, Michael Cutforth, Peter Cuy, David Decotigny, Rakesh Gautam, Alex Iriza, Milo M. K. Martin, Rick Roy, Zuowei Shen, Ming Tan, Ye Tang, Monica Wong-Chan, Joe Zbiciak, and Amin Vahdat. Aquila: A Unified, Low-Latency Fabric for Datacenter Networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI '22)*, Renton, WA, April 2022.

[63] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. Direct Access, High-Performance Memory Disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, Carlsbad, CA, July 2022.

[64] Stewart Grant, Anil Yelam, Maxwell Bland, and Alex C Snoeren. Smartnic performance isolation with fairnic: Programmable networking for the cloud. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 681–693, 2020.

[65] Albert Greenberg, Gisli Hjalmtysson, Dave Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. A clean slate 4d approach to network control and management ˆ—. *ACM SIGCOMM Computer Communication Review*, October 2005.

[66] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang Shin. Efficient Memory Disaggregation with Infiniswap. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*, Boston, MA, USA, April 2017.

[67] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces. In *2019 IEEE/ACM 27th International Symposium on Quality of Service (IWQoS)*, pages 1–10. IEEE, 2019.

[68] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces. In *Proceedings of the International Symposium on Quality of Service*, IWQoS '19, 2019.

[69] Yibo Guo. *Towards Leaner Data Centers: Energy Efficiency and Carbon Savings through Network Optimization*. PhD thesis, University of California, San Diego, 2024.

[70] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*.

[71] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[72] Hewlett Packard. The Machine: A New Kind of Computer. http://www.hpl.hp.com/research/systems-research/themachine/, 2005.

[73] Hewlett-Packard. Memory Technology Evolution: An Overview of System Memory Technologies the 9th edition, 2010. https://support.hpe.com/hpesc/public/docDisplay?docId=emr_na-c00256987.

[74] Hewlett Packard Labs. Memory-Driven Computing. https://www.hpe.com/us/en/newsroom/blog-post/2017/05/memory-driven-computing-explained.html, 2017.

[75] C++ DataFrame for statistical, Financial, and ML analysis., 2023.

[76] T Chiang Hu. Multi-commodity network flows. *Operations research*, 11(3):344–360, 1963.

[77] Ho Van Hung and Tran Quoc Chien. Extended linear multicommodity multicost network and maximal concurent flow problems. *Available at SSRN 3407884*, 2019.

[78] Intel. https://ark.intel.com/content/www/us/en/ark/products/codename/63546/red-rock-canyon.html.

[79] Intel Corporation. Intel Rack Scale Architecture: Faster Service Delivery and Lower TCO. http://www.intel.com/content/www/us/en/architecture-and-technology/intel-rack-scale-architecture.html.

[80] ITRS. International Technology Roadmap for Semiconductors (SIA) 2014 Edition.

[81] Saba Jamilan, Tanvir Ahmed Khan, Grant Ayers, Baris Kasikci, and Heiner Litz. Apt-get: Profile-guided timely software prefetching. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, Rennes, France, 2022.

[82] 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 the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 121–136, 2017.

[83] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for usecond-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019.

[84] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*, Boston, MA, USA, February 2019.

[85] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA Efficiently for Key-value Services. In *Proceedings of the 2014 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '14)*, Chicago, IL, USA, August 2014.

[86] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design Guidelines for High Performance RDMA Systems. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC '16)*, Denver, CO, USA, June 2016.

[87] Mahmut Kandemir, Ismail Kadayif, and Ugur Sezer. Exploiting scratch-pad memory using presburger formulas. In *Proceedings of the 14th international symposium on Systems synthesis*, ISSS 2001, Montrél, Canada, 2001.

[88] Rishi Kapoor, Alex C Snoeren, Geoffrey M Voelker, and George Porter. Bullet trains: A study of nic burst behavior at microsecond timescales. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 133–138, 2013.

[89] Ian A Kash, Greg O'Shea, and Stavros Volos. Dc-drf: Adaptive multi-resource sharing at public cloud scale. In *Proceedings of the ACM symposium on cloud computing*, pages 374–385, 2018.

[90] Navdeep Katel, Vivek Khandelwal, and Uday Bondhugula. Mlir-based code generation for gpu tensor cores. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, CC 2022, Seoul, South Korea, 2022.

[91] Linux Kernel. Red-black trees (rbtree) in linux. https://www.kernel.org/doc/Documentation/rbtree.txt.

[92] Tanvir Ahmed Khan, Nathan Brown, Akshitha Sriraman, Niranjan K Soundararajan, Rakesh Kumar, Joseph Devietti, Sreenivas Subramoney, Gilles A Pokam, Heiner Litz, and Baris Kasikci. Twig: Profile-guided btb prefetching for data center applications. In *54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, Virtual, 2021.

[93] Tanvir Ahmed Khan, Ian Neal, Gilles Pokam, Barzan Mozafari, and Baris Kasikci. Dmon: Efficient detection and correction of data locality problems using selective profiling. In *15th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '21, Virtual, 2021.

[94] Tanvir Ahmed Khan, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. I-spy: Context-driven conditional instruction prefetching with coalescing. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '20, Virtual, 2020.

[95] Thomas Kistler and Michael Franz. Automated data-member layout of heap objects to improve memory-hierarchy performance. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(3):490–505, 2000.

[96] Sohyang Ko, Seonsoo Jun, Yeonseung Ryu, Ohhoon Kwon, and Kern Koh. A New Linux Swap System for Flash Memory Storage Devices. In *2008 International Conference on Computational Sciences and Its Applications*, 2008.

[97] Teemu Koponen, Keith Amidon, Peter Balland, Martin Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Paul Ingram, Ethan Jackson, Andrew Lambeth, Romain Lenglet, Shih-Hao Li, Amar Padmanabhan, Justin Pettit, Ben Pfaff, Rajiv Ramanathan, Scott Shenker, Alan Shieh, Jeremy Stribling, Pankaj Thakkar, Dan Wendlandt, Alexander Yip, and Ronghua Zhang. Network virtualization in multi-tenant datacenters. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, Seattle, WA, April 2014.

[98] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, 2020.

[99] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '21, Virtual, 2021.

[100] Yanfang Le, Hyunseok Chang, Sarit Mukherjee, Limin Wang, Aditya Akella, Michael M Swift, and TV Lakshman. Uno: Uniflying host and smart nic offload for flexible packet processing. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 506–519, 2017.

[101] Gyusun Lee, Wenjing Jin, Wonsuk Song, Jeonghun Gong, Jonghyun Bae, Tae Jun Ham, Jae W. Lee, and Jinkyu Jeong. A case for hardware-based demand paging. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, 2020.

[102] Junghee Lee, Chanik Park, and Soonhoi Ha. Memory access pattern analysis and stream cache design for multimedia applications. In *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*, ASP-DAC 2003, Kitakyushu, Japan, 2003.

[103] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. Mind: In-network memory management for disaggregated data centers. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 488–504, 2021.

[104] 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 the 26th Symposium on Operating Systems Principles (SOSP '17)*, Shanghai, China, October 2017.

[105] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '23)*, Vancouver, Canada, March 2023.

[106] Jie Li, Laiping Zhao, Yanan Yang, Kunlin Zhan, and Keqiu Li. Tetris: Memory-efficient serverless inference through tensor sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, Carlsbad, CA, July 2022.

[107] Pengcheng Li, Hao Luo, Chen Ding, Ziang Hu, and Handong Ye. Code layout optimization for defensiveness and politeness in shared cache. In *43rd International Conference on Parallel Processing*, ICPP 2014, Minneapolis, MN, 2014.

[108] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: High

Precision Congestion Control. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19)*.

[109] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*, Austin, Texas, 2009.

[110] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. System-level implications of disaggregated memory. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture (HPCA '12)*, New Orleans, LA, USA, February 2012.

[111] Jiaxin Lin, Kiran Patel, Brent E Stephens, Anirudh Sivaraman, and Aditya Akella. {PANIC}: A {High-Performance} programmable {NIC} for multi-tenant networks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 243–259, 2020.

[112] Will Lin, Yizhou Shan, Ryan Kosta, Arvind Krishnamurthy, and Yiying Zhang. Supernic: An fpga-based, cloud-oriented smartnic. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 130–141, 2024.

[113] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3:{Energy-Efficient} microservices on {SmartNIC-Accelerated} servers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 363–378, 2019.

[114] Yanan Liu, Xiaoxia Wei, Jinyu Xiao, Zhijie Liu, Yang Xu, and Yun Tian. Energy consumption and emission mitigation prediction based on data center traffic and pue for global data centers. *Global Energy Interconnection*, 3(3):272–282, 2020.

[115] Yuanwei Lu, Guo Chen, Zhenyuan Ruan, Wencong Xiao, Bojie Li, Jiansong Zhang, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Memory efficient loss recovery for hardware-based transport in datacenter. In *Proceedings of the First Asia-Pacific Workshop on Networking*, APNet'17, 2017.

[116] Mellanox. Bluefield smartnic. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf, 2018.

[117] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-based Congestion Control for the Datacenter. *ACM SIGCOMM Computer Communication Review (SIGCOMM '15)*.

[118] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting network support for rdma. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, 2018.

[119] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*.

[120] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding pcie performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, 2018.

[121] Vlad Nitu, Boris Teabe, Alain Tchana, Canturk Isci, and Daniel Hagimont. Welcome to zombieland: Practical and energy-efficient memory disaggregation in a datacenter. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, 2018.

[122] Vlad Nitu, Boris Teabe, Alain Tchana, Canturk Isci, and Daniel Hagimont. Welcome to zombieland: Practical and energy-efficient memory disaggregation in a datacenter. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*, Porto, Portugal, April 2018.

[123] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. Scale-out numa. *ACM SIGPLAN Notices*, 49(4):3–18, 2014.

[124] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, and Marcos Aguilera. Storm: A fast transactional dataplane for remote data structures. In *Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR '19)*.

[125] Diego Novillo. Memory ssa—a unified approach for sparsely representing memory. In *Proc of the GCC Developers' Summit*, 2007.

[126] OpenAI. GPT-2: 1.5B release. https://openai.com/research/gpt-2-1-5b-release, 2019.

[127] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019.

[128] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. The ramcloud storage system. *ACM Transactions Computer System*, 33(3):7:1–7:55, August 2015.

[129] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. Bolt: a practical binary optimizer for data centers and beyond. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '19, Washington DC, 2019.

[130] Jongsoo Park, Maxim Naumov, Protonu Basu, Summer Deng, Aravind Kalaiah, Daya Khudia, James Law, Parth Malani, Andrey Malevich, Satish Nadathur, Juan Pino, Martin Schatz, Alexander Sidorov, Viswanath Sivakumar, Andrew Tulloch, Xiaodong Wang, Yiming Wu, Hector Yuen, Utku Diril, Dmytro Dzhulgakov, Kim Hazelwood, Bill Jia, Yangqing Jia, Lin Qiao, Vijay Rao, Nadav Rotem, Sungjoo Yoo, and Mikhail Smelyanskiy. Deep Learning Inference in Facebook Data Centers: Characterization, Performance Optimizations and Hardware Implications. *arXiv:1811.09886*, 2018.

[131] P. Peng, Y. Mingyu, and X. Weisheng. Running 8-bit dynamic fixed-point convolutional neural network on low-cost arm platforms. In *2017 Chinese Automation Congress (CAC)*, 2017.

[132] Intel Optane persistent memory. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-dc-persistent-memory-brief.html.

[133] 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.

[134] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. The design and implementation of open vswitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015.

[135] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: A programming system for {NIC-Accelerated} network applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 663–679, 2018.

[136] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. In *Proceedings of Machine Learning and Systems*, MLSys '23, Miami, FL, 2023.

[137] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*, Minneapolis, MN, USA, June 2014.

[138] Yifan Qiao, Chenxi Wang, Zhenyuan Ruan, Adam Belay, Qingda Lu, Yiying Zhang, Miryung Kim, and Guoqing Harry Xu. Hermit: Low-Latency, High-Throughput, and Transparent Remote Memory via Feedback-Directed Asynchrony. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, Boston, MA, April 2023.

[139] Yiming Qiu, Jiarong Xing, Kuo-Feng Hsu, Qiao Kang, Ming Liu, Srinivas Narayana, and Ang Chen. Automated smartnic offloading insights for network functions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 772–787, 2021.

[140] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE micro*, 30(4):65–79, 2010.

[141] Benjamin Rothenberger, Konstantin Taranov, Adrian Perrig, and Torsten Hoefler. ReDMArk: Bypassing RDMA security mechanisms. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[142] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, 2015.

[143] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-performance, application-integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*.

[144] Khodakaram Salimifard and Sara Bigharaz. The multicommodity network flow problem: state of the art classification, applications, and solution methods. *Operational Research*, 22(1):1–47, 2022.

[145] Mohit Saxena and Michael M. Swift. FlashVM: Virtual Memory Management on Flash. In *Usenix ATC*, 2010.

[146] Danfeng Shan, Fengyuan Ren, Peng Cheng, Ran Shu, and Chuanxiong Guo. Observing and mitigating micro-burst traffic in data center networks. *IEEE/ACM Transactions on Networking*, 28(1):98–111, 2019.

[147] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. Legoos: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*, Carlsbad, CA, October 2018.

[148] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed shared persistent memory. In *Proceedings of the 8th Annual Symposium on Cloud Computing (SOCC '17)*, Santa Clara, CA, USA, September 2017.

[149] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. StRoM: Smart Remote Memory. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*, Heraklion, Greece, April 2020.

[150] Arjun Singhvi, Aditya Akella, Maggie Anderson, Rob Cauble, Harshad Deshmukh, Dan Gibson, Milo M. K. Martin, Amanda Strominger, Thomas F. Wenisch, and Amin Vahdat. Cliquemap: Productionizing an rma-based distributed caching system. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, 2021.

[151] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F. Wenisch, Monica Wong-Chan, Sean Clark, Milo M. K. Martin, Moray McLaren, Prashant Chandra, Rob Cauble, Hassan M. G. Wassel, Behnam Montazeri, Simon L. Sabato, Joel Scherpelz, and Amin Vahdat. 1rma: Re-envisioning remote memory access for multi-tenant datacenters. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '20)*.

[152] Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. Elastic cuckoo page tables: Rethinking virtual memory translation for parallelism. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, 2020.

[153] Irene Solaiman, Miles Brundage, Jack Clark, Amanda Askell, Ariel Herbert-Voss, Jeff Wu, Alec Radford, Gretchen Krueger, Jong Wook Kim, Sarah Kreps, Miles McCain, Alex Newhouse, Jason Blazakis, Kris McGuffie, and Jasmine Wang. Release strategies and the social impacts of language models, 2019.

[154] Shixin Song, Tanvir Ahmed Khan, Sara Mahdizadeh Shahri, Akshitha Sriraman, Niranjan K Soundararajan, Sreenivas Subramoney, Daniel A. Jiménez, Heiner Litz, and Baris Kasikci. Thermometer: Profile-guided btb replacement for data center applications. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, New York, NY, June 2022.

[155] SpinalHDL. SpinalHDL. https://github.com/SpinalHDL/SpinalHDL.

[156] Alexandru E Şuşu. A vector-length agnostic compiler for the connex-s accelerator with scratchpad memory. *ACM Transactions on Embedded Computing Systems (TECS)*, 19(6):1–30, 2020.

[157] Wayne Szeto, Youssef Iraqi, and Raouf Boutaba. A multi-commodity flow based approach to virtual network resource allocation. In *GLOBECOM'03. IEEE Global Telecommunications Conference (IEEE Cat. No. 03CH37489)*, volume 6, pages 3004–3008. IEEE, 2003.

[158] Akshitha Sriraman Joseph Devietti Gilles Pokam Heiner Litz Baris Kasikci Tanvir Ahmed Khan, Dexin Zhang. Ripple: Profile-guided instruction cache replacement for data center applications. In *Proceedings of the 48th International Symposium on Computer Architecture (ISCA '21)*, Virtual, June 2021.

[159] Konstantin Taranov, Benjamin Rothenberger, Adrian Perrig, and Torsten Hoefler. sRDMA – efficient NIC-based authentication and encryption for remote direct memory access. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020.

[160] Jon Tate, Pall Beck, Hector Hugo Ibarra, Shanmuganathan Kumaravel, Libor Miklas, et al. *Introduction to storage area networks*. IBM Redbooks, 2018.

[161] TECHPP. Alibaba singles' day 2019 had a record peak order rate of 544,000 per second. https://techpp.com/2019/11/19/alibaba-singles-day-2019-record/, 2019.

[162] Po-An Tsai, Nathan Beckmann, and Daniel Sanchez. Jenga: Software-defined cache hierarchies. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*, Toronto, Canada, June 2017.

[163] Shin-Yeh Tsai, Mathias Payer, and Yiying Zhang. Pythia: Remote oracles for the masses. In *28th USENIX Security Symposium (USENIX Security 19)*.

[164] Shin-Yeh Tsai, Yizhou Shan, , and Yiying Zhang. Disaggregating Persistent Memory and Controlling Them from Remote: An Exploration of Passive Disaggregated Key-Value Stores. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC '20)*, Boston, MA, USA, July 2020.

[165] Shin-Yeh Tsai and Yiying Zhang. LITE Kernel RDMA Support for Datacenter Applications. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, Shanghai, China, October 2017.

[166] William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. Revisiting the open vswitch dataplane ten years later. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, 2021.

[167] Sumesh Udayakumaran and Rajeev Barua. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 276–286, 2003.

[168] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[169] Haris Volos, Kimberly Keeton, Yupu Zhang, Milind Chabbi, Se Kwon Lee, Mark Lillibridge, Yuvraj Patel, and Wei Zhang. Memory-Oriented Distributed Computing at Rack Scale. In *Proceedings of the ACM Symposium on Cloud Computing, (SoCC '18)*, Carlsbad, CA, USA, October 2018.

[170] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building An Elastic Query Engine on Disaggregated Storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*, Santa Clara, CA, February 2020.

[171] 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 *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*.

[172] Chenxi Wang, Yifan Qiao, Haoran Ma, Shi Liu, Yiying Zhang, Wenguang Chen, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Canvas: Isolated and Adaptive Swapping for Multi-Applications on Remote Memory. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, Boston, MA, April 2023.

[173] Tao Wang, Hang Zhu, Fabian Ruffy, Xin Jin, Anirudh Sivaraman, Dan R. K. Ports, and Aurojit Panda. Multitenancy for fast and programmable networks in the cloud. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020.

[174] Xingda Wei, Fangming Lu, Rong Chen, and Haibo Chen. KRCORE: A microsecond-scale RDMA control plane for elastic computing. In *2022 USENIX Annual Technical Conference (ATC '22)*, July 2022.

[175] Wikipedia. "jenkins hash function". https://en.wikipedia.org/wiki/Jenkins_hash_function.

[176] Emmett Witchel and Krste Asanovic. The span cache: Software controlled tag checks and cache line size. In *Workshop on Complexity-Effective Design, 28th ISCA*, 2001.

[177] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH Computer Architecture News*, 23(1), March 1995.

[178] Xilinx. Zynq UltraScale+ MPSoC ZCU106 Evaluation Kit. https://www.xilinx.com/products/boards-and-kits/zcu106.html. Accessed May 2020.

[179] Idan Yaniv and Dan Tsafrir. Hash, don't cache (the page table). In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, SIGMETRICS '16, 2016.

[180] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, 2021.

[181] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys '09)*, Nuremberg, Germany, 2009.

[182] Zhiyuan Guo and Yizhou Shan and Xuhao Luo and Yutong Huang and Yiying Zhang. Clio: A hardware-software co-designed disaggregated memory system. In *the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*, Lausanne, Switzerland, March 2022.

[183] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*.

[184] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. One-sided RDMA-Conscious extendible hashing for disaggregated memory. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021.