UNIVERSITY OF CALIFORNIA SAN DIEGO

Disaggregated Data Structures: Sharing and Contention with RDMA and Programmable Networks

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

in

Computer Science

by

Stewart Steven Grant

Committee in charge:

Professor Alex C. Snoeren, Chair
Professor Amy Ousterhout
Professor George Papen
Professor Yiying Zhang

2024

The Dissertation of Stewart Steven Grant is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2024

DEDICATION

To my inner circle – Family, Loved ones and Friends. And to every belayer that caught me. Thanks for the proverbial and literal support.

EPIGRAPH

”If this is the best of possible worlds, what then are the others?”

-Voltaire *Candide*

“It must be considered that there is nothing more difficult to carry out, nor more doubtful of success, nor more dangerous to handle, than to initiate a new order of things.”

-Niccolo Machiavelli *The Prince*

”If every porkchop were perfect we wouldn’t have hotdogs”

-Greg Universe, *Steven Universe*

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

Chapter 2 is a partial reprint of work submitted to multiple USENIX and ACM conferences under the title ”SwordBox: Accelerating Shared Access in RDMA-based Disaggregated Memory. Stewart Grant, Alex C. Snoeren. This dissertations author was the primary investigator and author of this paper. Chapter 3 is a partial reprint of work submitted to multiple USENIX conferences under the title ”Cuckoo for Clients: Disaggregated Cuckoo Hashing. Stewart Grant, Alex C. Snoeren. This dissertations author was the primary investigator and author of this paper.

I would like to acknowledge my advisor Alex C. Snoeren for his dedication to his craft and guidance over the past 6 years. No piece of work within this dissertation would be possible without your collaboration. I would also like to thank my committee members Amy Ousterhout, Yiying Zhang, and George Papen for their feedback and guidance, and Srikanth Kandula for his mentorship during my time at MSR.

This dissertation has been extraordinarily influenced by Anil Yelam, my closest collaborator. Thank you for all the time you spent working on our collaborations, and the hours spent discussing and debating system designs and performance results. I’m forever grateful. To Maxwell Bland, your research energy is unmatched and without your help we would never have have acquired any SmartNICs. And Alex (Enze) Liu for his superior knowledge of Python and unmatched focus on research. Thank you to all of the members of the Systems and Networking group at UCSD, especially the optical networking group for your feedback and guidance during the first years of my PhD.

Thank you to Meta for funding my research and providing me with the opportunity to work on resource disaggregation, Cavium for the generous donation of two SmartNICs, and to ARPAe for funding my first years of research.

I’d like to thank all of the members of 3140 for their collaboration and friendship over the years. It’s truly the best office, Chez bob volunteers for keeping me fed, and to my friends for the support. Camille Rubel thanks for having the best climbing schedule in the world, Phillip Arndt for pushing my limits, and Camille Moore for keeping me on my toes.

VITA

2012-2016 Bachelor of Science, Computer Science University of British Columbia

2016-2018 Master of Science, Computer Science University of British Columbia

2018-2024 Doctor of Philosophy, Computer Science University of California San Diego

PUBLICATIONS

Deepak Bansal, Gerald DeGrace, Rishabh Tewari, Michal Zygmunt, and 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 . 2023. Disaggregating Stateful Network Functions. In proceedings of 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI '23). Usenix Association, Boston MA, USA, April 2018, 1469–1487.

Stewart Grant, Anil Yelam, Maxwell Bland, and Alex C. Snoeren. 2020. 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 (SIGCOMM '20). Association for Computing Machinery, Virtual Event, August 2020, 681–693.

Stewart Grant, Hendrik Cech, and Ivan Beschastnikh. 2018. Inferring and asserting distributed system invariants. In Proceedings of the 40th International Conference on Software Engineering (ICSE '18). Association for Computing Machinery, Gothenberg, Sweden, July 2018, 1149–1159.

ABSTRACT OF THE DISSERTATION

Disaggregated Data Structures: Sharing and Contention
with RDMA and Programmable Networks

by

Stewart Steven Grant

Doctor of Philosopy in Computer Science

University of California San Diego, 2024

Professor Alex C. Snoeren, Chair

Resource disaggregation proposes a next-generation architecture for data center resources. System components like compute, memory, storage, and accelerators are separated from one another by a fast network and composed dynamically into virtual servers when required. This paradigm promises to dramatically improved resource utilization, scalability, and flexibility, but introduces substantial challenges in terms of performance and fault tolerance. Memory is among the most difficult resources to disaggregate. CPUs currently expect DRAM to have ultra low latency, high bandwidth, and to share it's failure domain. In particular increased latency from network round trips dramatically shifts the performance of existing shared data structures

designed for local DRAM.

In this dissertation I demonstrate the challenges of sharing disaggregated memory and show that programmable network devices can be used to significantly improved system performance. I present two systems: First SwordBox which utilizes a centralized programmable switch to cache data structure state and dramatically improve key-value workload performance. Second I present a new key-value store RCuckoo which is designed to leverage RDMA and reduce round trips when accessed by CPUs over a network. Both systems demonstrate significant performance improvements over the existing state of the art.

Introduction

What should a data center server look like? Twenty years ago, a data center operator may have argued for the simplicity of homogeneity over optimal performance, reasoning that carefully picked commodity hardware at a given price point would yield the best cost-performance tradeoffs and that the performance gains of next-generation hardware would quickly erase any benefits made by specializing servers due to Dennard scaling [83].

Since then, both Moore's law and Dennard scaling have slowed down dramatically. CPU clock speeds and memory density improvements have stagnated, leaving operators to fight tooth and claw to enjoy the efficiency gains of prior decades. The effect is that new technologies are being introduced to achieve scaling. CPUs are now monstrously parallel. Custom accelerators are common for specialized workloads like video coding and machine learning. Indeed, the modern data center is a hodgepodge of heterogeneous hardware (GPUs, TPUs, DPUs, SmartNICs, and FPGAs) [12, 26, 43, 5], and various new memory offerings and tiers like NVMe [71]. Today, the number of server types in a data center, conservatively, is in the dozens. At the time of writing, EC2 has 84 listed instance types [4] for their customers to design their services. The trend is clear: in search of efficiency, data center and server design is increasingly heterogeneous, with servers being designed for specific applications and workloads.

Resource disaggregation is a new architectural paradigm for data center resources aimed at improving efficiency and managing increased heterogeneity. In the disaggregated model, a server's resources do not monolithically reside in a 1U, 2U, or 4U server form factor. Instead, each resource (i.e., compute, memory, storage) is deployed separately to a dedicated machine and interconnected via a fast network. Servers are composed dynamically from these resources, which

enables them to be provisioned for their exact purpose [10, 32, 58, 71, 84]. This model enables resource pooling and sharing, which in turn leads to higher efficiency [8, 9, 12, 81, 85, 86].

DRAM, in particular, has become a precious resource in data centers and is a focused target for resource pooling [66]. The benefits of pooling are clear when examining a simple bin packing problem. Consider two servers, each provisioned with 4GB of DRAM, and three jobs, each requiring 2.5GB of memory. In a monolithic design, a scheduler can only place one job per machine or risk swapping to disk. In a disaggregated model, the 4GB of memory could be placed in an 8GB pool, which could be easily subdivided into three 2.5GB partitions. In the monolithic case, the unused memory is stranded, while pooling reclaims stranded memory. More concretely, at data center levels, practitioners have to provision their servers for the sum of peak demand; when resources are pooled, they can be provisioned for the peak of the sum of demand, which can be significantly lower [12, 85].

Disaggregation, in general, is only possible because of new fast networks. Commodity NICs now offer 400Gbps with expectations for continued growth to 800Gbps and above [7]. Network and memory bandwidths are quickly approaching the same order of magnitude. At the same time, network stacks are becoming lighter-weight through kernel bypass and CPU bypass technologies like DPDK [27] and RDMA [38], enabling applications to more easily take advantage of the additional bandwidth. Network devices themselves are becoming increasingly programmable, with multiple vendors offering programmable SmartNICs, DPUs, and switches [40, 1, 76]. These two trends have led to intra-rack latencies of 1-2μs, with the ability to inspect, cache, and modify packets in flight at line rate.

Despite fast networks, memory disaggregation has remained elusive while storage, such as spinning disks and solid-state drives, has seen widespread disaggregation. The reason for this contrast is that storage device access latency is far higher than a network round trip. In the case of memory, the opposite is true. Memory access latency is approximately 20 times lower than a network round trip (50-100ns), effectively making it a separate tier of memory when placed across a network. While there is ongoing research demonstrating the advantages of tracking

cache lines over pages for remote memory [14], the cost of fully disaggregating all memory is deemed too high [84]. A common proposal for disaggregated memory is to have CPUs with a reasonably sized cache (e.g., 4GB) of DRAM attached to them, along with software to manage and reduce the cost of remote accesses [84].

A large body of literature exists on disaggregated memory systems with a significant local cache. Many of these systems intervene in the virtual memory system to decrease the cost of accessing remote memory by employing prefetching and eviction strategies aimed at minimizing blocking remote accesses [32, 9, 64, 51]. Similarly, object-based disaggregated systems utilize remotable objects with per-object tracking to mitigate the frequency of remote accesses [81, 100]. Additionally, compiler-based systems leverage static and dynamic analysis to identify large, small, and hot objects for cache optimization [33, 90]. These systems primarily focus on analyzing memory access patterns and reducing the number of remote accesses with decreasing volumes of local cache. However, they often overlook a critical aspect of memory access: sharing. When memory is shared, access patterns alone are insufficient to minimize round trips. Some degree of coherence must be maintained between remote caches, which is the primary focus of this dissertation.

At its core, the challenge of sharing in disaggregated memory is serialization. When a data structure is shared by multiple accessors, some mechanism must ensure the consistency of the data structure. In a monolithic system, this is often achieved with locks or atomic operations. Across machines, serialization is typically accomplished by either a centralized sequencer or a distributed protocol. As a point of comparison, consider the differences between serialization in a traditional RPC system and disaggregated memory. In an RPC system, requests arrive on a NIC and are delivered to one or more CPU cores for processing [56, 67, 45, 24, 65]. If all requests are routed through a single CPU core, it implicitly serializes operations by enqueueing the RPC requests and servicing them one at a time. If the RPC service has multiple cores, any shared structure can be protected via a lock or other synchronization mechanism in the RPC server's local memory. In contrast, in a disaggregated system, no such server-side CPU exists.

Or, if one does, it is assumed to be low power and incapable of handling significant traffic. In the absence of such a CPU, clients must enforce serialization amongst themselves.

In the absence of a serialization mechanism close to memory, clients must rely on the only mechanism available in commodity systems: RDMA atomics. Throughout this dissertation, RDMA atomics will be used as the backbone of all shared disaggregated data structures. They take the form of two operations: compare-and-swap (CAS) and fetch-and-add (FAA), which execute with the same semantics as their local counterparts but on the memory of a remote machine. While their semantics remain the same, their performance is dramatically different. As noted before, the cost of executing a remote operation is 20 times that of local memory. This latency inflates the size of critical sections built with remote locks and leads to stale caches and poor performance for optimistic data structures under contention [91, 93, 87, 102, 55].

Data structures can be optimized for disaggregated memory by leveraging network programmability.

This thesis statement is the core of the work presented in this dissertation. The challenges listed above, while fundamental to the problem domain can be practically alleviated in a variety of ways by exploiting modern network hardware. We provide evidence for the thesis statement above by addressing the following two questions. *Where and how should serialization occur?* The default answer to these questions is ”on the NIC” and ”with RDMA atomic verbs.” However, given the landscape of programmable in-network hardware, our options are flexible. At rack-scale, both TORs and NICs offer serialization points. The interface and mechanism for serialization can be customized using programmable hardware. For instance, a switch could be programmed to maintain locks [97], provide sequencing [78], or directly implement contended functions [50]. Simultaneously, NICs, though lower bandwidth and less centralized than TORs, have the potential to offer extended RDMA interfaces [22] and implement OS functionality for remote clients [34, 85]. *What data structures should be used?* Few data structures are currently designed for remote memory [91, 87, 102, 93, 55, 86]. While these systems resemble RDMA [67, 45, 68]

and NUMA [35, 36, 15] systems of the past, disaggregation requires special consideration for the network hardware it runs on. How can round trips and access latencies be reduced? What data structures are easy to build, and which are hard? These questions are a key focus of this dissertation.

This dissertation explores the design of shared data structures in disaggregated memory systems. I introduce two systems, SwordBox and RCuckoo, which address the challenges of sharing and contending access to remote memory. SwordBox (Chapter 2) adopts a middlebox approach to alleviate contention in shared data structures. Its key insight leverages the serialized view of traffic at rack-scale TORs, caching data structure state on a programmable switch. Additionally, I present RCuckoo (Chapter 3), a fully disaggregated key-value store specifically designed to enhance key-value locality. RCuckoo utilizes locality-sensitive hashing to improve performance in reads, writes, locking, and fault recovery by minimizing round trips. SwordBox demonstrates significant improvements in both throughput (up to 30x) and tail latency (up to 300x), while RCuckoo meets or surpasses the performance of all state-of-the-art disaggregated key-value stores on small key-value pairs and outperforms other systems on the most common data-center workloads [20, 73].

Chapter 1

Background

Disaggregated systems rely on a variety of state-of-the-art technologies. We begin this Chapter by describing disaggregation generally and technologies that enable it (Section 1.1). The disaggregated networks described in this dissertation are fast (100Gbps+) and rely heavily on one-sided RDMA operations. In Section 1.2, we describe RDMA, the connections which enable one-sided operations, their consistency guarantees, and the atomic operations which serialize operations across connections. In-network computation enables new serialization points for shared data structures – Section 1.3 describes the current landscape of programmable hardware, its strengths and limitations. Section 1.4 introduces the concepts behind and challenges of building shared data structures in disaggregated memory and how traditional structures can be adapted to remote memory.

1.1 Disaggregation

Disaggregation stems from shifting hardware trends, marking a paradigm shift within the systems community. Over decades, per-core access to memory bandwidth and capacity has steadily declined [58]. While CPU core counts have seen consistent increases, memory speeds and capacities have improved at a slower rate [66]. Consequently, CPU cores now have diminished access to memory bandwidth and capacity compared to a decade ago. With memory becoming an increasingly scarce resource, data center operators are exploring novel approaches

to enhance memory utilization. Disaggregation emerges as one such option. Monolithic servers typically allocate a fixed amount of RAM per machine, resulting in an uneven distribution of memory utilization across the data center. Some servers suffer from memory shortages, while others have surplus gigabytes. Disaggregation targets this spare, stranded memory, aiming to provide each server access to a shared pool of RAM. In essence, disaggregated resources, whether memory, FPGAs, or the network itself, can be provisioned for the peak-of-sums rather than the sum-of-peaks [34, 85, 12].

The primary challenge in disaggregation is network latency [28]. The difficulty of disaggregating a resource is directly related to its access latency relative to the network latency. Disaggregated storage has become commonplace, with academia and industry pooling SSDs and HDDs into shared storage pools for increased capacity and cost efficiency. This process is comparatively straightforward compared to memory disaggregation, given the higher access costs of storage relative to the network. For instance, intra-rack RDMA ping latencies typically range from 1-2 microseconds, whereas HDD latencies are 10-20 milliseconds and SSDs are often hundreds of microseconds. In these scenarios, the network overhead is usually a single-digit percentage or less [71]. In contrast, DRAM latencies are in the range of 50-100 nanoseconds. DRAM over RDMA incurs nearly a 20x overhead compared to local access. Despite this overhead, RDMA remains a prominent candidate for disaggregated transport. CXL, an emerging technology, promises NUMA-like latencies (200-300 nanoseconds) for remote memory access [21]. However, CXL's availability and performance scalability are not well-established at present [29, 52, 89]. Regardless of the interconnect used, the fundamental challenge of access latency persists. This dissertation primarily focuses on RDMA, although the algorithms and data structures presented herein are largely agnostic to interconnect specifics.

1.2 RDMA

Remote Direct Memory Access (RDMA) serves as the fundamental technology enabling disaggregation. This section delineates RDMA's attributes in facilitating disaggregated architectures via its one-sided verbs, alongside the intricacies of constructing shared data structures atop RDMA, particularly concerning serialization.

RDMA stands as a low-latency, high-bandwidth network protocol. It achieves superior performance by circumventing multiple overheads inherent in traditional networking stacks, such as Linux Sockets. Firstly, RDMA operates as a kernel-bypass technology, empowering user-space applications to directly engage with the network interface card (NIC), leveraging NIC-specific features like on-NIC caches [93], and sidestepping context switch overheads. Secondly, RDMA's foremost feature lies in offloading a significant portion of the network stack to NIC hardware, effectively bypassing the CPU entirely for data transfer. CPU bypass on the receiver end distinguishes RDMA as the pivotal technology for disaggregation. A receiver can expose its resources like memory without necessitating CPU intervention in managing the transfer.

To illustrate the contrast between traditional Unix API-based communication and RDMA, consider the following scenario: In a conventional networking stack, when a process sends a UDP message on an existing socket, the user initially marshals the data and submits it to the kernel through a *send* operation. Subsequently, the kernel copies the data from user-space, configures the packet header, and dispatches the packet to the NIC. Conversely, with RDMA, the user-space application pre-registers a memory region with the NIC before sending data. When an application intends to transmit data from this region to a remote machine, it furnishes a pointer to the data, the data's size, the remote machine's address, and the NIC's intended action (verbs). The application initiates transmission by invoking a dedicated *RDMA send* operation. This operation, non-blocking in nature, signals the NIC to perform a direct memory access (DMA) operation, extracting the memory from the process's address space, assembling a packet on the NIC (inclusive of managing transport state), and transmitting directly to another machine. In

the case of a write, the receiving NIC issues DMA to the remote machine's memory without engaging its CPU.

While the aforementioned example provides a high-level portrayal of CPU bypass for RDMA writes, it is imperative to recognize that the RDMA protocol is inherently complex, featuring various connection types and verbs. The extant InfiniBand RDMA specification spans over 1,700 pages [38]. Within this section, I briefly highlight the salient aspects of RDMA pertinent to this dissertation's objectives.

1.2.1 RDMA Connections

When run over Ethernet using the RoCEv2 (RDMA over Converged Ethernet) standard, RDMA NICs located on client and server can cooperate to implement congestion [57, 101] and flow control, reliable delivery, and at-most-once delivery semantics. Before exchanging data, RDMA endpoints establish a queue pair (QP) which defines the region(s) of memory each is able to access. Like Internet transport protocols, RDMA queue pairs provide a configurable set of semantics depending on the transport mode selected: UDP-like semantics are provided by unreliable datagram (UD) and unreliable connections (UC), while reliable connections (RC) are similar to TCP, ensuring reliable, in-order delivery. Moreover, reliable connections support so-called 1-sided verbs (e.g., read, write, and compare-and-swap) that are executed autonomously by the remote NIC without any remote CPU involvement.

The benefits of the various transport modes and 1-vs-2-sided verbs have been a topic of intense debate. While reliable connections provide enhanced guarantees, their implementation requires on-NIC memory, a precious resource, and researchers have observed scalability bottlenecks due to memory and cache limitations in the Mellanox ConnectX family of RDMA NICs [23, 47, 44, 92, 46]. Recent work has shown how to overcome limits in terms of the number of connections [74, 69], but the ordering guarantees provided by RC remain restricted to individual queue pairs. While unreliable transport modes can deliver superior performance and scalability [47], they require the use of 2-sided verbs—i.e., involvement of a CPU at the memory server—to

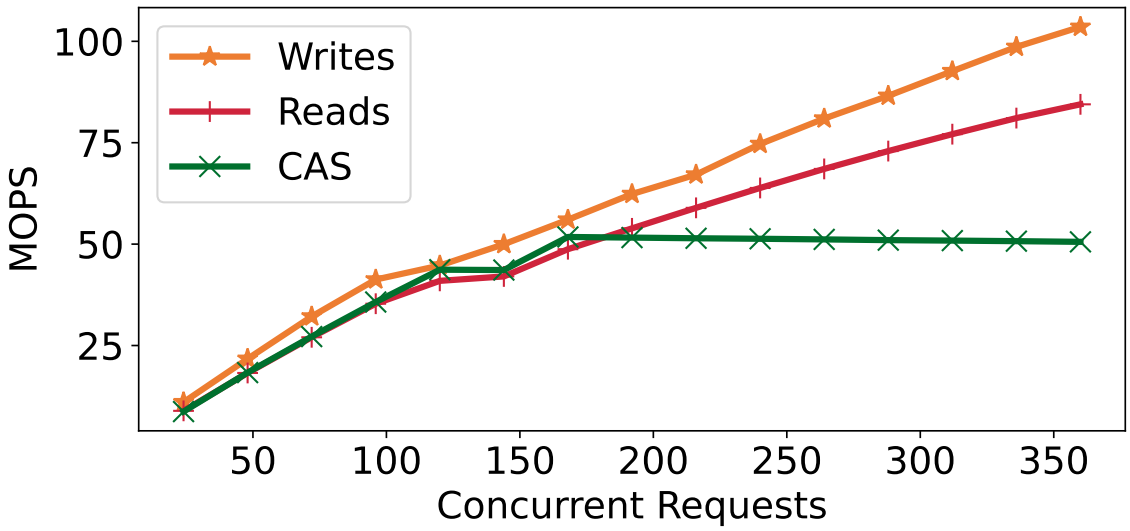

Figure 1.1. Achieved throughput of RDMA verbs across twenty queue pairs on data-independent addresses as a function of request concurrency. When using atomic requests, ConnectX-5 NICs can support approximately 2.7 MOPS per queue pair, up to about 55 MOPS in aggregate.

ensure ordering, a non-starter for passive disaggregated settings. Unless hardware support for more sophisticated 1-sided verbs [96, 88, 94] becomes available, another approach is required.

The memory semantics of one-sided RDMA are complex. While RC provides in-order delivery of messages, different verbs have their own ordering semantics. For instance, issuing a read prior to a write may see the results of the write. Across QPs, no ordering is guaranteed by default. The effect of these semantics is that system designers must be very careful with how they use RDMA. If a read is issued on the same address as a write before waiting for the write to complete, the user must specify a fence flag in the read operation. Across QPs, the lack of ordering means that partially written data is visible to other QPs; a common tactic is to accompany writes with CRCs to enable the readers to verify the data's integrity [67, 68, 91, 102]. When serialization is required across QPs, the only available mechanism are RDMA atomic operations.

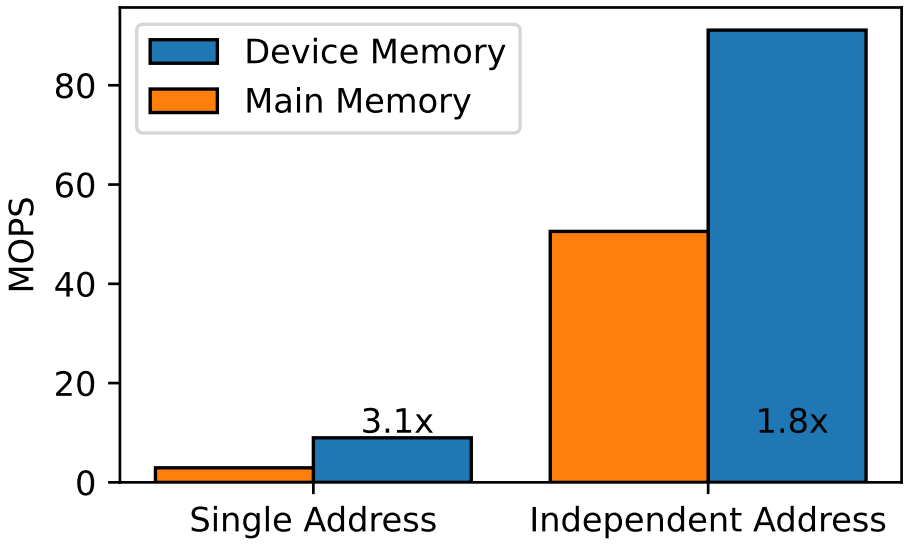

Figure 1.2. Compare-and-swap performance on device and main memory

1.2.2 RDMA Verbs on Mellanox NICs

While RDMA is a generic protocol, the Network Interface Cards (NICs) utilized throughout this dissertation are Mellanox ConnectX-5. These NICs implement the base specification of RDMA and also provide additional features outside the InfiniBand specification, which can be exploited for performance gains. In this section, I describe RDMA verbs and their performance characteristics on Mellanox NICs, with a specific focus on atomic operations.

Atomic verbs such as compare-and-swap (CAS) and fetch-and-add (FAA) are essential for implementing locks or opportunistic concurrency. Atomics are limited to 64-bit operations and bottleneck at lower rates than reads and writes because they block requests on data-dependent addresses while waiting on Peripheral Component Interconnect Express (PCIe) round trips [46, 93]. Figure 1.1 shows that the NICs in our testbed (100-Gbps NVIDIA Mellanox ConnectX-5s) are capable of serving many tens of millions of RDMA operations per second (limited only by link speed), but CAS operations to remote server memory top out around 50 MOPS.

While atomic operations are limited to 64 bits, read and write message sizes are bounded only by the link Maximum Transmission Unit (MTU). Figure 1.1 shows that on our testbed, NIC-

to-NIC round-trip times are similar for all message sizes less than about 128 bytes, and messages must exceed 1 KB before the latency of a single large operation exceeds two round-trip times of smaller ones. We leverage this observation in Chapter 3 by collapsing multiple small reads into a single larger one when appropriate. The optimal threshold trades off latency reduction against the amplified bandwidth cost of performing larger reads (read amplification).

Mellanox NICs include a small amount (256 KB in our case) of on-NIC memory that can be addressed by remote clients using RDMA operations [6]. Accesses to NIC memory avoid the need to cross the server's PCIe bus, decreasing latency and increasing throughput. The performance gain is particularly significant for atomic operations. Figure 1.2 shows the maximum aggregate throughput of concurrent CAS operations targeting the same single (i.e., contended) address or distinct, independent addresses in both main server memory (shown in orange) and on-NIC device memory (blue). CAS operations perform between 1.8 and 3.1 times faster on NIC memory. Chapter 3 illustrates the profound effect that utilizing NIC memory can have on data structure performance in a disaggregated setting by using NIC memory specifically for high contention locking operations.

1.2.3 RDMA Limitations

RDMA is a powerful technology; however, it has well-documented drawbacks that can make system design difficult and limit performance. One significant debate revolves around the limitations of the RDMA API. Certain operations are challenging with RDMA; for instance, allocating memory requires calls into the control path, and data indirection like pointers necessitates a round trip back to the sender to resolve [22]. As detailed in prior sections, atomic operations are slow and lead to performance bottlenecks [46].

Of particular note is the difficulty in achieving global ordering across queue pairs (QP). Even using fast NIC memory for RDMA operations yields only a few million operations per second (around 10M). Using this technique to implement a global sequencer is orders of magnitude slower than a global sequencer implemented with two-sided verbs and an efficient

RPC system (around 120M) [46]. In the next section, I overview the rise of programmable network devices and provide some background on how they can alleviate some of the limitations of RDMA.

1.3 Programable Networks

The past decade has seen the rise of programmable network devices. These devices are capable of executing users' code often at line rate. A huge variety of devices exist: SmartNICs [30, 61, 60, 77], DPUs [12], FPGAs [26, 34, 79, 85], and programmable switches [19, 41, 42, 97] from a wide variety of vendors. Often these devices provide thin operating systems which allow users to develop and deploy code written either in C or P4. These programmable devices are powerful and transformational tools for designing network systems as they can offer orders-of-magnitude performance improvements when deployed in the right context [78], such as sequencing where it has shown to offer huge benefits for consensus [53, 54]. In this dissertation, we leverage the power of programmable switches to exactly this benefit to get fast in-network serialization for RDMA based data structures (Chapter 2).

Most prior proposals for disaggregation consider rack-scale deployments where servers are partitioned into roles: compute, memory, and storage, all of which are interconnected by a top-of-rack switch [10, 25, 39, 48, 84]. The central role of the ToR in this architecture has not gone unnoticed, and researchers have observed that a programmable switch can offload a wide variety of traditional operating system services [10, 41, 42, 50, 97, 99]. The constraints in each case are similar: programmable switches have limited memory and processing capabilities. If the computational task is too large packets must be recirculated adding additional latency and reducing aggregate bandwidth. Ideal applications for programmable switches use little memory, require minimal processing and deliver outsized performance benefit.

Specifically, prior work has shown that programmable switches are able to provide rack-scale serialization at low cost [53, 54, 78], manage locks [97], and track the state required

to maintain an RDMA reliable connection [49]. Researchers have even used a programmable switch to implement a centralized memory controller including a unified TLB and cache for passive remote memory [50]. Their approach is limited, however, by the resource constraints of the switch. Inspired by performance-enhancing TCP proxies of old [11, 31], we consider a slightly different design point where the authoritative state and control logic remain at the endpoints.

In Chapter 2, we will show that a programmable switch can be used to great effect in accelerating a shared RDMA based data structure by caching a small amount of data and modifying operations in flight to reduce (or entirely remove) contention. In the following section we describe data structures in disaggregated systems and prior work on NUMA based data structures.

1.4 Disaggregated Systems and Data Structures

The aforementioned trends and technologies have enabled disaggregated systems to become a reality. A common model for disaggregated memory is that the remote memory of another machine can be used as a swap space or as a remote cache rather than disk. In this model, applications are apportioned a partition of remote memory for their pages [9, 32, 51, 58, 64, 84], objects [81, 100], or cache lines [14]. These systems focus on improving performance by reducing the number of remote memory accesses that an application has to make. In general, this is done by identifying hot and cold memory, then prefetching and evicting data to reduce the number of faults to remote memory. Additionally, these systems focus on fault-tolerance by replicating or erasure coding memory across replicated memory servers [51].

A commonality between each of these memory systems is the lack of sharing. Operations common to non-disaggregated systems like mmapping a shared page are not supported by these systems. In cases where shared access is supported using POSIX interfaces, performance is not considered to be a concern [8]. Disaggregated systems that share efficiently, at the time of writing,

are entirely custom-built data structures, the majority of which are key-value stores, transaction processors, or caches [55, 87, 86, 91, 93, 98, 102] . Each of these systems takes on a significant burden in terms of development. Most designers develop their own fault tolerance, replication, recovering, allocation, and serialization protocols. In nearly every case, the performance of these systems is determined by the techniques used to serialize writes on the index of the data structure.

In this section I describe the challenges of building a shared data structure in disaggregated memory. Pointers and pointer chasing are expensive in disaggregated memory. I describe the tradeoffs of using pointer-based structures (e.g. linked lists) vs dense structures (e.g. arrays) in Section 1.4.1, finally we describe these same tradeoffs in the context of hash tables, and opportunistic vs lock based concurrency schemes.

1.4.1 Sparse vs Dense Data Structures

Data structures in disaggregated systems can be broadly categorized as either sparse or dense. Sparse data structures are pointer-based, such as linked lists and trees. Dense data structures reside in a linear block of memory, such as an array or heap. These two categories form a spectrum as some structures, like hash tables, may have a dense index and a sparse data region formed by linked lists. The choice of data structure has a profound impact on the number of memory accesses required to perform operations. Sparse data structures typically require pointer chasing, which involves traversing some number of pointers to reach the data, for example, searching through a linked list or down a binary tree. Dense data structures are typically easy to index into but tend to require more data movement, such as inserting into a sorted array. Moreover, when designed for concurrency, sparse data structures are typically more amenable to optimistic approaches, where operations can be done out of place and committed via an atomic operation, while dense data structures typically use locks to implement a critical section while data is updated.

In the context of disaggregated memory, the choice of data structure is critical as each

Table 1.1. Cross section of systems and techniques. Full circles ● imply that a system uses the category, ◐ denotes when a system meets the qualification in spirit but not explicitly, and ○ when the technique is absent. Columns OC and CC stand for Optimistic Concurrency and Compute Coalescing respectively.

	project	read inflation	relaxed layout	pre-compute	OC*	metadata caching	CC*	self verifing
Multicore NUMA	Flat Combining [35]	○	○	○	○	○	●	○
	Hopscotch Hash [36]	●	○	○	○	○	○	○
	Blackbox NUMA [15]	◐	○	○	●	●	●	○
RDMA Key-Value	pilaf [67]	○	○	○	○	○	○	●
	farm [23]	●	○	○	●	●	●	○
	herd [45]	○	○	○	○	○	○	○
	cell [68]	○	○	○	◐	●	○	●
Disaggregated Datastructures	Clover [91]	○	◐	○	●	●	○	○
	RACE [102]	●	●	◐	●	●	○	●
	Sherman [93]	●	●	◐	●	●	●	●
	Mind [50]	○	◐	◐	○	●	○	○

pointer resolution or data move requires an RDMA round trip. Lock-based data structures can bottleneck quickly if locks are too coarse-grained, and client failure while holding a lock can lead to distributed deadlock. Optimistic approaches can lead to high amounts of wasted work if operations fail, although their failure cases may be easier to reason about as the effects of the operation are not visible until the operation is committed.

Disaggregated data structures are not the first to face these challenges. NUMA data structures, RDMA key-value stores, and preliminary work on disaggregated data structures each have their own combination of techniques for managing this tradeoff space.

Table 1.1 is the result of a literature review on the state-of-the-art for shared data structures. We noted a variety of techniques that cross-cut the systems we reviewed. Read inflation is a technique which takes advantage of data structure locality. Simply put, if a rough location of data is known, a big read can be issued to fetch a region containing the data. This reduces search time for data in terms of memory access (or round trips) and trades off bandwidth for latency.

Hopscotch hashing [36], detailed more in the next section, defines a range h in which data can be placed in its hash index. This technique goes hand in hand with relaxed data structure layout (e.g., associativity) which allows for data to be placed in any order within pre-defined bounds. RACE [102], a recent disaggregated key-value store, uses 8-way associative storage in its index and uses read inflation to grab each bucket in a single read. Sherman [93], a write-optimized B+Tree, uses associative, rather than sorted, leaves to reduce contention on writes.

Due to the high cost of reading far memory, a common trait among these systems is to push complexity to the client. We classify the act of pushing complexity to clients into three categories: pre-compute, metadata caching, and self-verifying. Pre-compute is the idea that additional work on the client can reduce the number of reads required in remote memory. As noted in the introduction, Clio [34] uses a flat precomputed page-table, RACE uses a local power-of-two choices decision to reduce contention, and Sherman uses a local lock table to reduce remote contention. Each disaggregated system makes use of some degree of metadata caching, where a component of the data structure's index is cached locally on the client to reduce the number of reads required to synchronize with the remote state. In nearly all cases, data structures are self-verifying, meaning that the data structure's integrity can be determined by a local calculation on the client. These are commonly CRC64s [67, 68, 87, 102].

The techniques used by these systems are largely the same as those used to design RDMA key-value stores for non-disaggregated memory over the past decade [23, 45, 67, 68]. The primary difference is that disaggregated systems use exclusively one-sided RDMA operations, while RDMA key-value stores typically route write operations through a CPU to serialize requests.

Despite the similarities between the two classes of systems, there is no agreement on whether data structures should be optimistic or lock-based as each have distinct benefits. For instance some data structures require large critical sections which are more amenable to locks [36], while others (such as linked lists) can have their critical sections reduced to a single pointer update with relative ease [91]. In Chapter 3, we design a lock-based hash table which

merges the qualities of Cuckoo and Hopscotch hashing to improve locality and enable efficient precomputation. In Section 1.4.3, we describe how different disaggregated systems have used locks and optimistic concurrency, and in Section 1.4.2, we describe the properties of Cuckoo and Hopscotch hashing. Our stance in Chapter 2 is that both locks and optimistic concurrency have their place in disaggregated systems, and that a middlebox can be used to accelerate both lock-based and optimistic approaches.

1.4.2 Hash Tables

Fully disaggregated key-value stores are essentially concurrent hash tables whose conflict-resolution strategy is implemented entirely by individual clients [55, 87, 102]. Like any hash table, the underlying hashing algorithm must have an approach to managing collisions. Cuckoo and hopscotch hashing are particularly attractive in this context because they both provide the property that the potential locations of an entry in the table, regardless of contention or collision, can be deterministically computed by clients based only upon the key itself [23, 24, 36, 56, 67, 75]. Moreover, the set of locations is limited. Hence, at least in theory, systems built around either cuckoo or hopscotch hashing hold the potential for single-round-trip reads.

Cuckoo hashing uses independent hash functions to compute two (or more) potential table locations for a key, a primary and a secondary, where each location corresponds to an associative row of entries. A key is always inserted into its primary location. If that row is full, an existing key is evicted (or "cuckooed") to its secondary row to make space. If the cuckooed entry's secondary row is also full, the process iterates (by selecting yet another entry in the secondary row to cuckoo) until an open location is found. The path of evictions is known as a *cuckoo path*. While insertions can be involved, reads can always be executed in a single round trip by reading the rows corresponding to both of a key's locations simultaneously [67].

Hopscotch hashing works in a similar fashion but provides a slightly different guarantee, namely that keys will be located within a bounded neighborhood. (While cuckoo hashing limits the number of locations in which a key may be stored, it does not provide any locality guarantees

regarding those locations.) It does so by finding the physically closest empty entry to the desired location and then, if that location is not within the neighborhood, iteratively moving other entries out of the way to make room for the new key. The hopscotch process is facilitated by maintaining a per-entry bitmask of nearby collisions. The entries are stored directly in the index at the location the entry hashes to, when updates are made during insertions or deletions the bitmask is updated to show that the collied entry has been inserted or removed. As with cuckoo hashing, clients can index entries in a hopscotch hash in a single round trip by reading a key's entire neighborhood at once.

The insert operation is expensive for both approaches, and prior systems have taken steps to mitigate its cost. In associative hashes like cuckoo hash tables, multiple entries can be chosen as eviction candidates, and breadth-first search (BFS) has been shown to minimize both cuckoo-path length and critical section time [24, 56]. Farm [23] and Reno [37], two systems based on hopscotch hashing, completely avoid executing long hopscotch chains due to their execution time and complexity. Moreover, under either approach, the insert operation can fail despite vacant entries in the table—they are just too far away to be reached by either the cuckoo path or hopscotch's neighborhood-bounded linear probing. The point at which inserts begin to fail, known as the *maximum fill factor*, is a function of the number of hash locations and row associativity in cuckoo hashing and desired neighborhood size for hopscotch hashing.

RCuckoo (Chapter 3) uses cuckoo rather than hopscotch hashing due to locking concerns. First, each step of a cuckoo insert process requires one update—to the entry being moved to its secondary location—rather than two. When an entry is relocated in a hopscotch table, the collision bitmask must also be updated. (Reno [37] uses one-sided atomics to sloppily update the bitmask but requires a server-side CPU to fix the bitmasks whenever concurrent inserts execute.) Second, keys exist in one of two locations in cuckoo hashing, so updates and deletes require locking only two rows, while hopscotch entries inhabit a range of locations, so a conservative locking strategy must lock the entire range. Yet, RCuckoo takes inspiration from hopscotch neighborhoods and employs dependent hashing to increase the spatial locality of key locations,

enabling clients to use local caches to speculatively compute cuckoo paths.

1.4.3 Locks vs Optimistic Concurrency

Locks and optimistic approaches both provide mechanisms for managing concurrent access to shared data structures. Locks provide a critical section which is executed atomically by a single thread, while optimistic approaches allow multiple threads to execute concurrently and resolve conflicts at the end of the operation. Both have their own tradeoffs which lead to different performance and fault-tolerance characteristics in disaggregated memory.

Sherman's B+ Tree [93] is augmented using entirely one-sided RDMA operations. Sherman improves performance under contention in two ways. First, it places locks for each node in the B+ Tree in a special region of NIC memory exposed by ConnectX-5 NICs. Second, Sherman's clients employ a hierarchical locking scheme to reduce the contention for server-hosted locks. This client-local optimization significantly improves performance in cases where clients are collocated; SwordBox seeks to achieve similar efficiencies at the rack scale.

Clover [91], RACE [102], and a successor to RACE called FUSEE [87] all use optimistic concurrency. They each support remote key-value stores through optimistic use of one-sided RDMA atomic operations and client-driven resolution protocols. In Clover, reads and writes for a given key are made to an append-only linked list stored in (persistent) remote memory [91]; clients race to update the tail of the list. In FUSEE, persistence is implemented through client-driven replication, so clients race to update a majority of replicas in an instance of distributed consensus [87]. In both cases, writes are guarded by CAS operations so clients can independently determine the outcome of the race. Because we are interested in the fundamental costs of contention—as opposed to the additional challenge of replication—we focus specifically on Clover in this paper, but SwordBox could equally well apply to a (degenerate) non-replicated instantiation of FUSEE. Indeed, we provide a performance comparison in the evaluation (Section 2.4).

In Clover, all RDMA requests are targeted at the (presumed) tail of the list and writes

are guarded by CAS operations. A client may not know the location of the tail as other writers concurrently push it forward. When an operation fails to land at the tail of the list, Clover traverses the structure until the tail is found. While this provides no liveness guarantees, in the common read-heavy case concurrent clients eventually reach the end of the list. To speed up operations, clients keep caches of the end of each key's linked list to avoid traversals. By implementing a shared cache at the ToR, SwordBox decreases the likelihood of stale accesses.

Chapter 2

Swordbox: Accelerated Sharing of Disaggregated Memory

Proposals for disaggregated memory systems often forgo sharing entirely in favor of partitioned regions [9, 32, 64, 84]. Each of these proposals cuts at a fundamental goal of remote memory, which is to increase capacity via pooling and reduce the need to increase CPU pin counts for increased memory bandwidth. The unfortunate result is that these systems do not meet the same expectations as local memory systems. For instance, any system requiring mmap with *Shared* semantics is not natively supported, and for those that do, the performance penalty is extreme with no tools to mitigate the cost. The issue with this approach is that it may have unforeseen consequences in terms of memory utilization that may actually be worse than those on monolithic servers.

One strategy to deal with the cost of synchronization is to simply have applications duplicate resources rather than share them. On a large system hosting hundreds of VMs, the cost of duplicating shared libraries is known to be high, and deduplication among VMs is already common. Further, on monolithic servers, the explicit nature of message passing systems has been studied extensively, and extremely high performance can be achieved by using explicit remote accesses [44, 45]. Given these two factors, monolithic servers with well-designed RPC systems could potentially share more effectively and achieve better resource utilization than a naive disaggregated system which exposes a transparent but slow interface to remote memory or

blindly replicates rather than sharing.

The key motivation behind SwordBox is to demonstrate that a centralized in-network device can effectively remove contention and enable line-rate performance for disaggregated shared memory. We noticed while investigating shared remote memory that the fastest RDMA key-value stores used mostly one-sided RDMA but still required a CPU to serialize writes. The moment that the CPU was removed entirely, the 99th percentile tail latency jumped dramatically [91]. One option we considered was to use a smartNIC to serialize writes rather than a CPU. Prior projects like Clio [34], SuperNIC [85], and Prism [22] suggested that network functions were a good fit for smartNICs and that the could be used to implement *close to memory* operations like pointer chasing. While we agreed, individual NICs could not provide rack-scale coherence—in order to provide a rack-scale uniform memory machine, we would need to think about a mechanism which could provide a global total order to all operations. A programmable switch is an ideal candidate for this role as it sees all of the traffic in a rack and can enforce global ordering. Our goal in this project was to design a system that could provide a full upper bound on the performance achievable by a disaggregated shared memory system as a benchmark for future systems to compare to.

The reason why sharing remote memory is hard is easy to identify: sharing remote memory requires coordinating access across multiple clients, yet the RDMA protocol—like TCP—provides a connection-based abstraction; while connection-less operation is possible, much like UDP it provides essentially no semantic guarantees. Fundamentally, remote memory operations must be ordered to provide coherent access, and the RDMA protocol provides two basic mechanisms to do so remotely (i.e., in a 1-sided fashion): reliable connections that ensure ordering and atomic operations that deliver mutual exclusion. While the performance of RDMA connection handling has received considerable attention [23, 74, 17], connections remain an end-to-end abstraction, and do not provide any guarantees regarding operations from distinct clients. For that, systems must rely on atomic operations like compare-and-swap (CAS), but their enhanced semantics dictate expensive implementation choices on the NIC, dramatically

restricting their performance compared to simple verbs like read and write [46]. Moreover, atomic operations are available only over reliable connections.

As a result, most existing systems that deliver scalable, high-performance shared remote access depend on the presence of computational resources collocated with the remote data [45, 68, 23, 67, 74]. In particular, a memory-local CPU can employ 2-sided RDMA operations to orchestrate operations between multiple clients [45, 47], avoiding the need for atomic operations. Unfortunately, such RPC-like approaches are infeasible in the passive memory setting. Alternatively, organizations with significant resources have considered redesigning the RDMA protocol itself to better support the needs of the disaggregated usage case—by, e.g., removing the connection abstraction and providing more powerful verbs [96, 88, 94]—but such hardware is not yet available.

In this chapter, we explore an alternative dimension: rather than relying exclusively on end-to-end solutions, we consider leveraging in-network resources—specifically programmable switches that are located between clients and the remote memory servers—to accelerate systems based on existing 1-sided RDMA verbs. Concretely, we observe that in rack-scale disaggregated settings, the top-of-rack (ToR) switch serves as a single serialization point for all RDMA requests. As a result, it is possible to transparently rewrite RDMA operations in flight to orchestrate requests from multiple clients to passive memory servers, sidestepping the fundamental bottlenecks present in the current connection-based RDMA protocol.

We present SwordBox, a top-of-rack switch that implements two separate yet complimentary approaches to accelerating RDMA-based passive memory. Client-driven schemes must rely either on mutual exclusion (i.e., locks) or optimistic concurrency control (which require multiple round trips to resolve conflicts). SwordBox removes the performance bottlenecks of both by 1) multiplexing multiple clients' RDMA operations onto shared connections to leverage the ordering semantics delivered by reliable connections [69], and 2) caching small amounts of metadata to dynamically steer in-flight RDMA updates to serialize concurrent operations to remote-memory indexing structures.

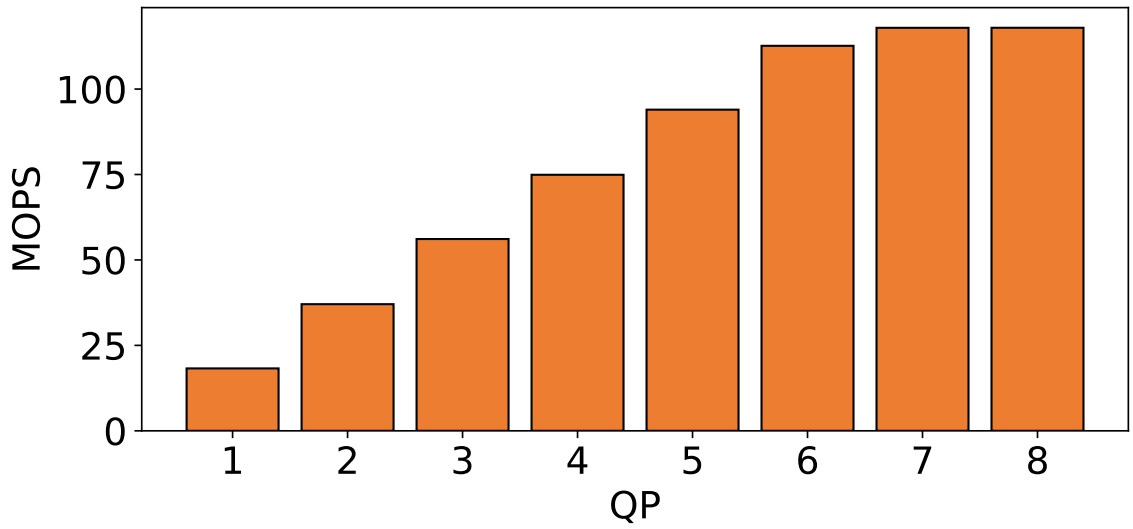

Figure 2.1. Max throughput as a function of the number of RoCEv2 RC connections. Each queue pair is managed by a separate core and issues in-lined writes.

We apply SwordBox to two remote memory systems that natively support sharing: Sherman [93], which uses locking, and Clover [91] that relies on optimistic concurrency. We show that both systems natively collapse under contention due to RDMA's limitations, but SwordBox can remove their bottlenecks. Concretely, by multiplexing all acquire and release operations for a shared lock in Sherman onto a single reliable connection, SwordBox can replace the client-issued compare-and-swap operations with a lightweight writes, delivering a potential 10× throughput gain. Performance gains are even higher in the case of Clover, where we resolve update conflicts to Clover's internal, append-only metadata index structure by steering requests to an advancing set of locations, as if they had been issued by a single serialized client. Our evaluation shows that under a 50:50 read-write workload, throughput rises by almost 35× while bandwidth usage and tail latency drop by 16 and 300×.

2.1 Serialization

The fundamental challenge faced by passive remote memory systems is ensuring consistency [70] by ordering accesses to any given location. RDMA reliable connections provide per-connection ordering, enabling clients to issue multiple outstanding requests; the NIC ensures

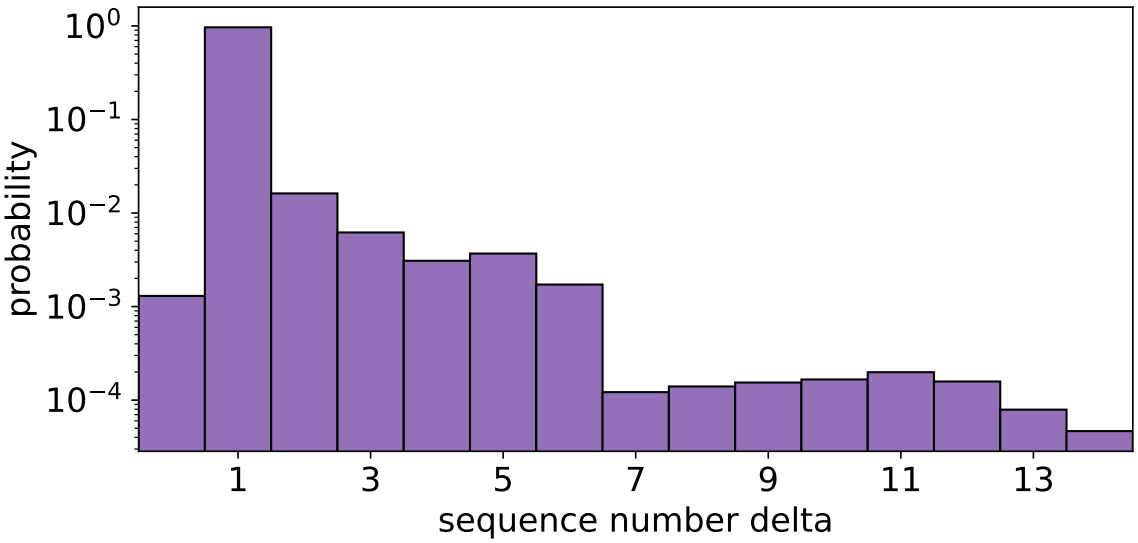

Figure 2.2. PDF of request reorderings. Retransmitted requests lead to reordering values of zero. 97% of requests retain their order (delta=1), however reorderings of up to 13 requests can occur. (Note logarithmic *y* axis.)

in-order delivery despite packet reordering and drops with sequence numbers and go-back-*n* retransmission. When clients are collocated, queue pairs can be shared by multiple cores through techniques such as flat combining [69, 93]. Unfortunately Figure 2.1 shows that the performance of individual queue pairs fall far short of line rate on our NICs; we do not observe full performance until at least seven queue pairs are used simultaneously. Moreover, as traditionally conceived, a QP is intended to be established between a single client and server—limiting its utility in a disaggregated setting. In the following subsections we experimentally illustrate the challenges to ordering across such clients.

2.1.1 Switch-Enforced Ordering

Packets processed by a programmable switch pipeline are sequenced in order: updates to switch registers are atomic as each state of a pipeline is occupied by exactly one packet at a time. Moreover, all packets destined to a given port must traverse the same egress pipeline. As a result, the ToR places packets from all flows destined to the same (single-homed) destination in a total order—not only with respect to their own flow, but others as well. In the context of

RDMA, however, switch-enforced packet ordering is insufficient. Even if packets (from various reliable connections) arrive at a server NIC in a given order, they may (appear to) be processed in arbitrary order due to contention at the NIC or PCIe bus [72].

Figure 2.2 shows that NIC and PCIe reordering is not merely an academic concern, but occurs with some frequency. In this example, we issue RDMA read, write and CAS requests at a rate of one million requests per second to 1,024 different memory locations according to a Zipf distribution and spread these requests across 32 different reliable connections. Each request is routed through a programmable switch that keeps a global request counter for each RDMA request (i.e., ground truth regarding request ordering). We track the order of responses relative to the order the corresponding request was issued from the middlebox. The plot shows the distribution of sequence-number gaps between responses. As expected, the vast majority differ by one (i.e., the same order they were dispatched from the switch), but a non-trivial number are out of order by one to five requests, and some by up to 15. Moreover, this experiment neglects the reality that some fames may be corrupted and/or lost by the link, necessitating retransmission and further cross-flow reordering.

2.1.2 Atomic RDMA Operations

In a remote-locking scheme, clients use an atomic RDMA operation to attempt to acquire a lock: because the operations are totally ordered at most one client will succeed at a time. Unfortunately, atomics are famously expensive [46], fundamentally because they require mutual exclusion across all RDMA queue pairs—concurrent read and write operations with a data dependency on the atomic address must stall until the atomic completes. Figure 1.1 considers the best-case scenario where clients attempt to access unique locks (i.e., each instruction is issued to an isolated cache line) in remote memory using an atomic operation in comparison to reads and writes. We confirm that the findings of prior studies [46, Fig. 14] with older hardware (i.e., ConnectX-3) remain true on our ConnectX-5 NICs, namely that atomic requests scale with

non-atomics only to a point.[1] CAS operations have a hard performance ceiling, while standard verbs (e.g., read and write) continue to scale with increased request concurrency. The situation is even worse when operations target the same address (i.e., lock contention; not shown).

One of the difficulties RDMA NICs face when implementing atomic operation is ensuring that there are no other conflicting memory operations at the server—even ones issued locally. More generally, any main-memory operation issued by the NIC must cross the PCIe bus and face potential contention. Modern nVIDIA Mellanox NICs like the ConnectX-5 provide a small region of on-NIC memory that can be mapped into the address space of RDMA applications, removing the remote PCIe overhead for frequently accessed data. (Indeed, Sherman employs this memory region to store its B+Tree locks.) Figure 2.3 compares the performance of serialized CAS and write operations to addresses in main vs. NIC-hosted device memory. CAS operations are issued across many queue pairs to achieve maximum throughput while the write operations are issued on a single queue pair to enforce serialization. While the use of NIC-hosted memory boosts CAS throughput from approximately 3 to around 9 MOPS, write operations remain dramatically more efficient in either case.

One way to avoid the overhead of remote lock acquisition in low-load situations is to attempt to directly modify the data (using an atomic RDMA operation) and recover if the operation fails due to a race; such schemes are known as optimistic concurrency control. While far more performant than lock-based approaches in the un-contended case, optimistic approaches can be prohibitively expensive when contention is common. As a concrete example we consider the chances of success in Clover. Figure 2.4 shows the percentage of requests which succeed in a 50:50 read-write workload as a function of the number of concurrent client threads. Success rate drops dramatically with concurrency.

At present RDMA has no support for addressing failed operations at the server, such as pointer chasing or operation retryi—although some have proposed such extensions [88, 63, 22]. Rather, clients resolve failures themselves at significant cost. In some systems, the retry is a

[1] Experiments with a ConnectX-6 exhibit similar behavior.

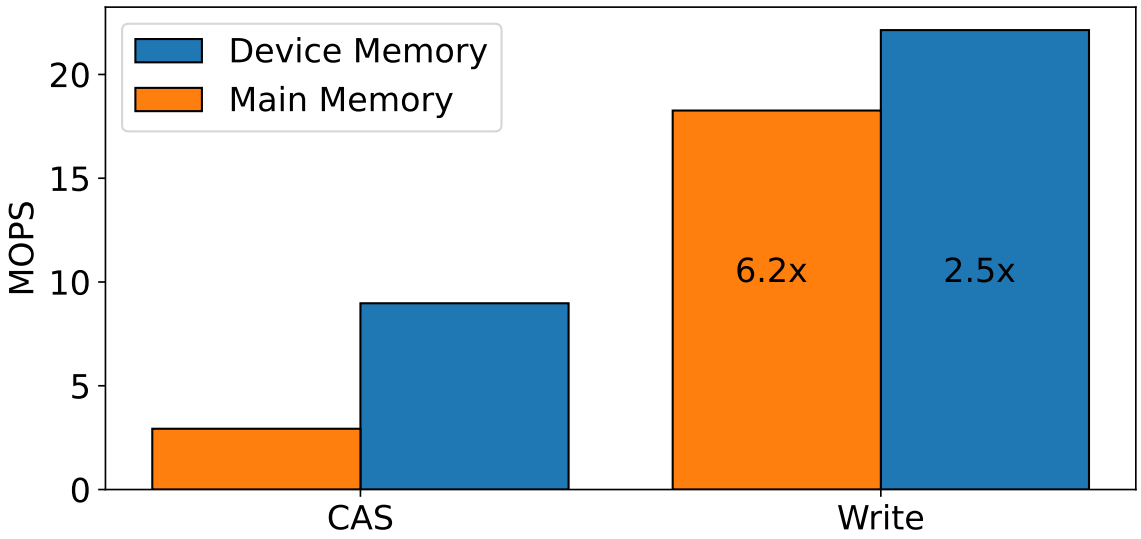

Figure 2.3. Throughput comparison of serialized RDMA operations in NIC-mapped device and main memory. Writes obtain 6.2× higher throughput than CAS in host memory and 2.5× higher in NIC memory. CAS values here are the same values as (Single Address) in Figure 1.2.

heavyweight, pessimistic operation, leading to a substantial—but fixed—overhead. In others, like Clover and FUSEE, subsequent attempts remain optimistic, resulting in a linear (per-retry) increase in costs. In the latter case, high rates of contention lead to congestion collapse, where retries are essentially doomed to fail, dramatically decreasing goodput.

Concretely, our measurements show that under contention the average bandwidth cost of Clover read and write operations can inflate by 16× (Figure 2.14) when compared to an optimal scenario in which all operations succeed on their first try. Perhaps even more significant than the overheads associated with the expected number of retries is the cost at the tail—namely the latency associated with those particularly "unlucky" requests that fail repeatedly. Note that these operations are precisely those for hot memory locations, so likely to be ones that matter. Under contention Clover's p99 tail latency increases by over 300× (Figure 2.15).

2.1.3 Implications

Systems that leverage RDMA atomics have hard performance limits because the aforementioned constraints. Locks located at a single address which use traditional lock, unlock

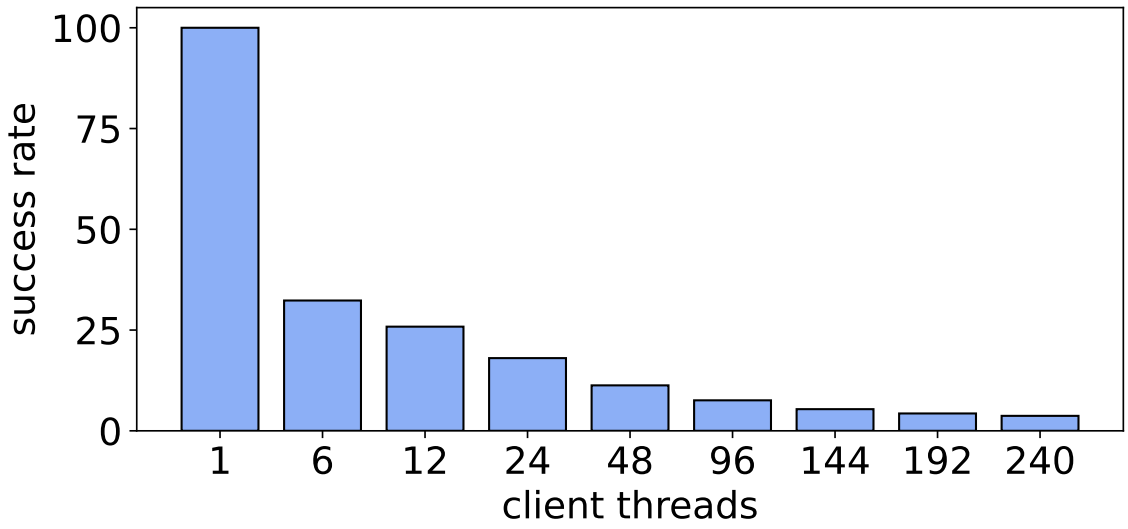

Figure 2.4. Percentage of successful operations in a 50:50 read-write workload spread across 1,024 keys according to a Zipf(0.99) distribution as a function of thread count. At 240 threads less than 4% of operations succeed.

operations are limited to around 500k accesses per second. This assumes perfectly coordinated requests, under contention requests which fail to acquire or release a lock still consume operation bandwidth. Under contention RDMA has poor support for traditional locking. In contrast optimistic data structures with locks scattered throughout, such as a linked list, are not rate limited by this single address restriction. However, they are fundamentally limited by the fact that any atomics have half the throughput of reads and writes. More critically, under contention optimistic data structures have no liveness guarantees.

2.2 SwordBox's Design

SwordBox is our general-purpose approach to accelerating RDMA-based applications like shared disaggregated memory that require operation ordering across clients. In this section we explain the functionality SwordBox provides and then apply it to two separate remote memory systems. At a high level, SwordBox is capable of 1) tracking on-going reliable connections, 2) parsing and caching their contents, and 3) modifying operations in flight. Because it defines a total order on outgoing RDMA requests, SwordBox can safely remap them between different

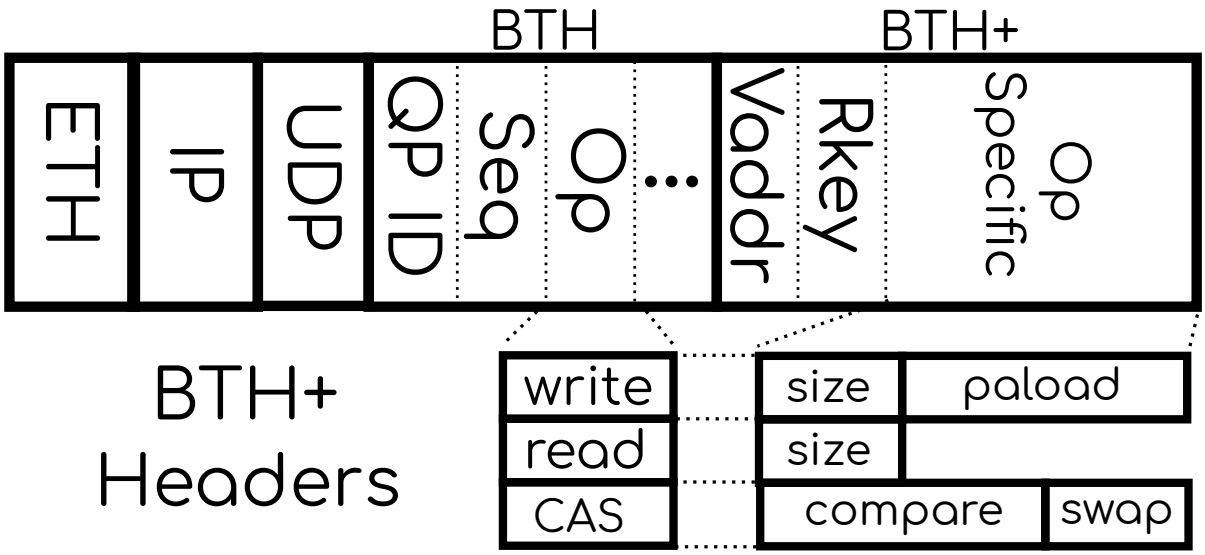

Figure 2.5. RoCEv2 packets consist of an Ethernet, IP, and UDP header. The RoCE BTH header stores QP data, sequence numbers, flags, and operations. The BTH+ header contains operation specific data: virtual addresses, DMA size, and atomic payloads.

connections as well as transform atomics into lightweight verbs. As we show in the context of Clover, by tracking a small bit of application-specific state, SwordBox can also use its knowledge of operation order to modify the target address or value of conflicting operations to resolve write/write conflicts before they occur.

2.2.1 Connection Multiplexing

RoCEv2 tunnels the original Infiniband-based RDMA protocol on top of UDP, using destination port 4791. RoCEv2 packets have two headers, BTH,and BTH+, shown in Figure 2.5, both of which SwordBox needs to parse. The BTH header indicates the operation, while the BTH+ headers contains the target virtual address and the operation payload. In cases where SwordBox wishes to enforce ordering across operations from different clients, it multiplexes them onto the same reliable connection. SwordBox does not establish or terminate connections itself—setup and teardown are handled end-to-end as usual by the RDMA NICs. Rather, SwordBox simply moves operations between existing connections.

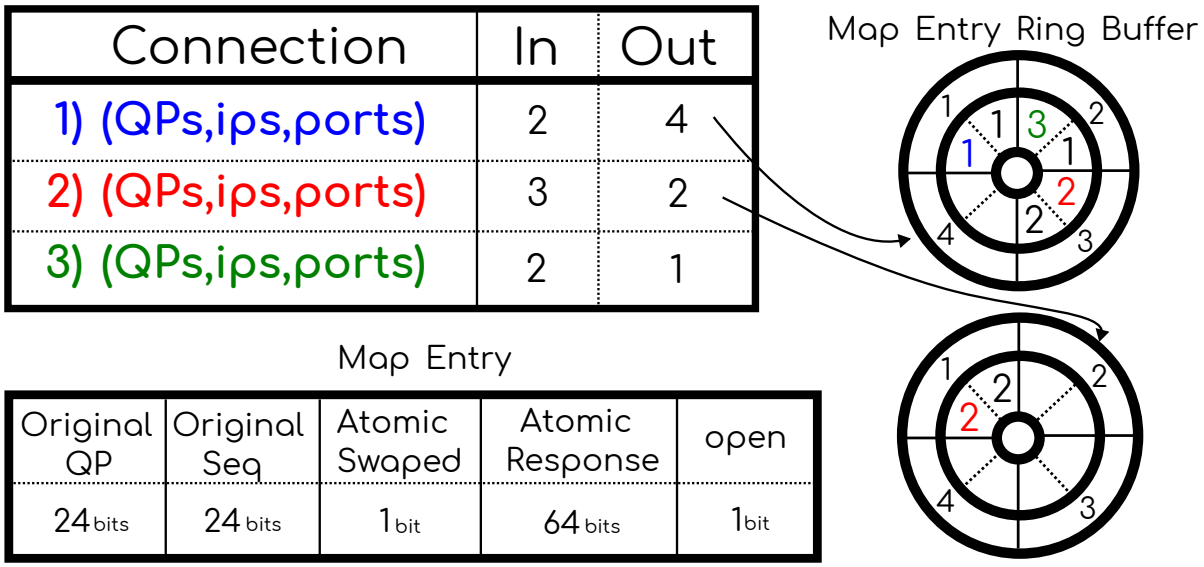

Figure 2.6. RC multiplexing in SwordBox. Per connection in and out sequence numbers are tracked to decouple sending and receiving QP. Map entries are stored in their outgoing connection's ring buffer; the ring diagram shows only the original QP row and sequence number.

Connection tracking

To facilitate connection multiplexing, SwordBox maintains a table of reliable connections transiting the switch, shown in Figure 2.6. Connections are uniquely identified by their source and destination IP address, source UDP port (the destination port is fixed for all RoCEv2 traffic), and queue pair ID. Entries are added to the table upon queue pair establishment and removed at teardown (or after a timeout). Each row in the table is associated with a ring buffer of *map entries* that track outstanding operations. As RDMA packets arrive and are placed into a total order at the switch, the row corresponding to the packet's incoming queue pair is updated with the current sequence number. Retransmissions (i.e., packets whose sequence numbers are no greater than the value already in the table) do not update the table. The table also records the highest sequence number used by an outgoing packet on each connection, which may not be the same as the incoming sequence number due to remapping.

Remapping

In general, RDMA packets will be forwarded using the same connection on which they arrived. In application-specific cases, however, SwordBox may wish to move them to a different

queue pair, i.e., to multiplex them onto a shared connection. In that case, the packet needs to be rewritten to use the new connection's source and destination addresses (both IP and Ethernet), UDP port, and an appropriate sequence number—which is computed to be one higher than the last packet transmitted on the outgoing connection. (Incoming retransmissions—detected due to their non-advancing sequence number—are always mapped to the same outgoing connection and sequence number as the original.) To facilitate ACK and retransmission handling, every packet creates a map entry in the ring buffer of its outgoing connection.

For efficiency, our DPDK implementation maintains the ring buffers as fixed-size arrays and use the packet's outgoing sequence number (modulo the buffer size) as an index. Each entry contains a reference to the packet's incoming queue pair, its original sequence number, and, in the case of atomics, space to record whether the operation was replaced by a write (Section 2.2.3) and, if so, the prior value of the target address—which is tracked using application-specific logic discussed below. If the packet was remapped, both the invariant CRC (ICRC) and the IP checksum must be updated.

ACK coalescing

Demultiplexing ACKs for remapped operations is non-trivial due to optimizations in the RDMA protocol. Specifically, an RDMA NIC may coalesce ACKs to reduce the number of packets transmitted and save bandwidth: like TCP, ACKs are cumulative. Coalescing presents a challenge when operations from multiple incoming connections are multiplexed onto another, as an ACK may correspond to operations issued by more than one client. Forwarding the ACK back to only the client who issued the (last) operation referenced in the ACK will cause the other clients whose operations were implicitly acknowledged by the server to timeout and retransmit. Conversely, forwarding ACKs to clients without outstanding operations could lead to unspecified behavior. Upon receipt of an ACK, SwordBox consults the map entries in the ring buffer for the relevant connection. SwordBox generates a separate ACK for each incoming connection with outstanding packets acknowledged by this ACK, setting the sequence number (and address and

queue pair information) according to their map entries.

2.2.2 State Caching

In addition to connection information, SwordBox can also parse RDMA operations to track the current state of memory locations of interest. The particular addresses are obviously application specific, but the mechanism is generic: SwordBox simply needs to apply the operations to its local cache in the same order it transmits the operations to the destination. We find that despite the large amounts of data transferred by passive memory systems, contention is typically localized to a few key addresses, such as those that are used to store locks, indexing datastructures, and other metadata. Moreover, these locations are typically accessed using atomic operations, limiting the data size to eight bytes a piece.

2.2.3 Atomic Replacement

When SwordBox tracks the state of address locations of interest, it necessarily determines outcome of atomic operations. Hence, SwordBox can be configured to multiplex all operations targeting specific addresses to the same connection and replace atomic operations with writes that simply store the outcome of the atomic, whether it be a compare-and-swap or fetch-and-add. Here we describe how SwordBox handles the former without loss of generality.

Replacing a CAS operation with a write is straightforward as the RoCEv2 headers differ by only a few fields (Figure 2.5). SwordBox transforms CAS requests to writes by swapping the BTH OP code, setting the BTH+ size field of the write to eight (recall all CAS operations are 64-bits long), and copying the appropriate value—either the "swap" value from the CAS operation on success or the current (cached) value on failure—into the payload. SwordBox indicates the operation has been transformed in its map entry (Section 2.2.1) as well as recording the prior value. Because a write's size field is only four-bytes long (as compared to the second 64-bit compare field in a CAS operation), the length of the packet shrinks by four bytes; SwordBox updates the IP length field accordingly.

After processing the write operation the destination NIC will respond with a regular, write ACK. As part of its ACK processing, SwordBox applies an inverse transformation to convert write ACKs to Atomic ACKs when necessary. RDMA Atomic ACK headers are very similar to regular ACKs with the only difference being that the atomic ACK contains the value that was overwritten, which SwordBox retrieves from the corresponding map entry.

2.2.4 Applying SwordBox to Disaggregated Memory

We now describe how we use SwordBox's techniques to accelerate the contention management techniques used by Sherman and Clover. In the case of Sherman, SwordBox multiplexes all operations (encoded as CAS operations) for a given lock on a single connection, caches lock state at the switch, and replaces acquisition attempts with writes. SwordBox's acceleration of Clover is more lightweight—in fact, entirely soft-state—but even more impactful. By caching a small amount of server state that clients manage through CAS operations, SwordBox is able to adjust these requests to ensure they succeed at the server, thereby avoiding expensive application-level retries for concurrent updates.

Shared locks

Sherman uses CAS operations to implement its node locks. Sherman's locks are simply specific (NIC-hosted) memory locations that store either a one (locked) or zero (free). Hence, lock requests are expressed as `CAS(0,1)`, which fail if the lock is unavailable (i.e., the stored value is currently not 0) or atomically set the value to 1—acquiring the lock—if successful. Unlock operations are the inverse. Presuming communication between the ToR and the memory server is reliable and in-order (as provided by an RDMA reliable connection), it is conceptually straightforward for SwordBox to cache the current value of the lock at the ToR.

In our design, SwordBox multiplexes all operations for a given address (i.e., lock) over the same connection to maintain ordering between the ToR and destination server. It can then use its local cache to determine whether an arriving CAS operation will succeed or fail. (If it

does not have the current value at the target address cached, it allows the atomic to pass through unmodified and populates its cache with the response.) Knowing the outcome, SwordBox is free to replace the CAS operation with a lightweight write in flight. When a CAS operation arrives for a lock address, SwordBox replaces it with a write for the specified value. (Releases will always succeed, setting the value to 0, while lock acquisition attempts always leave the value as 1; their success or failure is dictated by the prior state.) When the ACK comes back, SwordBox converts the ACK to an Atomic ACK before forwarding it back over the original connection to the client. Because lock values are always zero or one, it suffices to store a single bit—as opposed to eight bytes—to record the prior value in a lock operation's SwordBox map entry.

Even in the case when the lock is already held (and the acquire attempt is doomed to fail), SwordBox still forwards a write request to the memory server to ensure the client and server agree regarding the total number of RDMA verbs communicated between them. The ACK is replaced with a CAS "failure" so that the sender knows the lock acquisition failed. This is in keeping with SwordBox's performance-enhancing-proxy philosophy: it accelerates, but does not replace, the application's end-to-end semantics. Indeed, one could implement the lock server at the ToR itself [97], but that would require a redesign of the underlying system; our goal is to support selective deployment where SwordBox may not be on-path for all servers, dictating that we do not make any changes to the existing system. Moreover, our approach does not require terminating RDMA connections at the switch, which would require extensive buffering.

In general, the determination of which operations share state is application specific and requires inspecting each packet to extract the relevant pieces of metadata. In the case of Sherman, lock locations can be identified by inspecting CAS requests. Each CAS virtual address corresponds to a node lock in the Sherman B+Tree. While SwordBox must interpose on the full set of queue pairs terminated by a given (set of) server(s), this seems reasonable as the ToR is usually on-path for all servers in disaggregated rack settings.

SwordBox is designed for closed-loop clients. Connection remapping would require large amounts of buffering if clients had many in-flight requests spread across multiple QP.

Out-of-order requests would need to be buffered prior to delivering them as out-of-order packet delivery triggers RoCE's go-back-n retransmission protocol. Our aim is to enable rack-scale disaggregation where the total number of cores (clients) is less than $O(1k)$ where the few MB of available switch memory is more than sufficient.

Steering

Unlike Sherman, Clover does not implement locks. Instead, Clover attempts to append to a per-key linked list using atomic operations. Clover detects concurrent updates by breaking writes (i.e., list appends) into two RDMA operations: one write to create a new node, and a CAS operation to update the next pointer of the node at the tail of the list—the latter fails when another node was added concurrently. Concretely, it uses CAS operations to attempt to replace a `NULL` pointer (indicating the end of the list) with a pointer to a new element. To prevent such stale CAS requests from failing, SwordBox maintains a cache of the location of the (next pointer of the) node at the tail of each key's linked list. If a CAS request arrives at SwordBox destined for a stale virtual address (i.e., an address other than the one currently cached for that key), SwordBox *steers* the CAS operation by replacing its target address in the BTH+ header with the cached address. While SwordBox could multiplex these operations on a shared connection to enforce ordering (and replace them with writes), our evaluation shows the probability of reordering with a contending operation on a separate connection after departing the ToR is sufficiently low that the remaining cost of (clients) resolving such failures is minimal.

We implement steering by maintaining a cache of Clover's linked-list datastructures for popular keys. When a Clover packet arrives at the switch, it is parsed and passed to application-specific cache management code that extracts the salient information from the payload. Unfortunately, Clover RDMA CAS requests do not explicitly specify the write operation to which they correspond; SwordBox infers the operation by checking the size of the RDMA request and then extracts the Clover key from the appropriate the location in the packet. The key is used as an index into a lookup table to find the virtual address of the current tail node for that

key. Our strategy requires 64 bytes of data per key—the size of an RDMA virtual address.

While write steering suffices to avoid write/write conflicts, concurrent reads face a similar dilemma: Clover reads seek to access the current tail of the linked list, but the address may be stale if they "lose" a race with a concurrent write. To improve performance, SwordBox similarly steers reads to the correct tail address. Unfortunately, unlike writes (which are easy to identify by their use of the CAS operation), Clover reads are simply RDMA read operations for a virtual address and a length. As reads can be for arbitrarily old virtual addresses a naive solution that stored the lineage of each key would effectively require caching the entire contents of Clover's metadata server. Instead, SwordBox hashes the address of each write into an array somewhat larger than the size of the key space and stores the key along with the address. Collisions are resolved by replacing the old entry; keys with higher update rates maintain longer histories.

When reads arrive SwordBox looks up their destination address in the table; if the address has a hit the associated key is used to look up the current tail in the write cache and the RDMA read is steered to the cached location. Should a miss occur—either because the hash bucket was overwritten by another key, or because the tail address is not cached—the read is left unmodified. If it fails to arrive at the current tail, Clover's end-to-end recovery mechanism kicks in.

2.2.5 Failure Handling

SwordBox collocates functionality—and therefore shares fate—with the top-of-rack switch: if SwordBox fails, connectivity was already disrupted (i.e., the ToR is down). Hence, the fact that a SwordBox failure will reset all remapped RDMA queue pairs to the attached servers seems of little additional consequence. In the case of steering, however, we note that SwordBox does not maintain any hard state: failure simply results in a performance hiccup if packet-level connectivity can be maintained. The upshot is that a complete SwordBox failure does not introduce safety concerns in any event.

However, there are other failure scenarios to consider. In particular, we presume that the ToR sees the exact stream of packets that will be received—and processed—by attached

servers. Unfortunately, this may not be true due to packet loss (e.g., due to CRC failures or queue overflow) or even bugs on the server. Of course, these failure cases exist even without SwordBox, and Sherman and Clover both provide their own error handling. The key distinction, however, is that SwordBox maintains a cache that may become inconsistent with an attached server, which was previously the single authority of both application and connection state.

With respect to connection mapping, if a packet is dropped between SwordBox and a server and SwordBox maps a subsequent request from a different client onto the same QP, the server will generate a go-back-n response and any other in-flight requests on that QP will become invalidated. Hence, when SwordBox sees a go-back-n ACK, it triggers the same mechanism used for ACK coalescing but in reverse: it broadcasts a go-back-n ACK to all clients with outstanding messages. While this approach amplifies the performance impact of a lost packet, we expect such scenarios to be unlikely in practice. Indeed, no packet drops ever occurred between SwordBox and a server during our experiments because our clients issue only closed-loop operations.

While we do not employ connection mapping in Clover, SwordBox must still manage potential inconsistency between its cache and server state. Concretely, it is possible for a linked list to become "broken". If SwordBox sees a client issue a `CAS(A,B)` request (attempting to append node B to the list at A) before another issues `CAS(A,C)` (appending node C to the same—stale—tail), SwordBox will steer `CAS(A,C)` to `CAS(B,C)`. If the `CAS(A,B)` operation is lost between SwordBox and the server, `CAS(B,C)` will still succeed, causing a broken chain: the pointer $A \rightarrow B$ does not exist but $B \rightarrow C$ does, and SwordBox believes C to be the tail.

In the normal case, the client will timeout and retransmit `CAS(A,B)`, which SwordBox will identify as a retransmission and *not* steer to the "new" tail, thereby repairing the list. (In the mean time, the missing link is immaterial because subsequent requests are being steered by SwordBox.) If, however, the client were to fail prior to retransmitting the CAS the chain will remain broken. Here we use an out-of-band mechanism to repair the chain: on occasion our control plane queries the switch to check for outstanding CAS requests and simply retransmits them (spurious retransmissions are handled gracefully by the server). The trickiest case is if

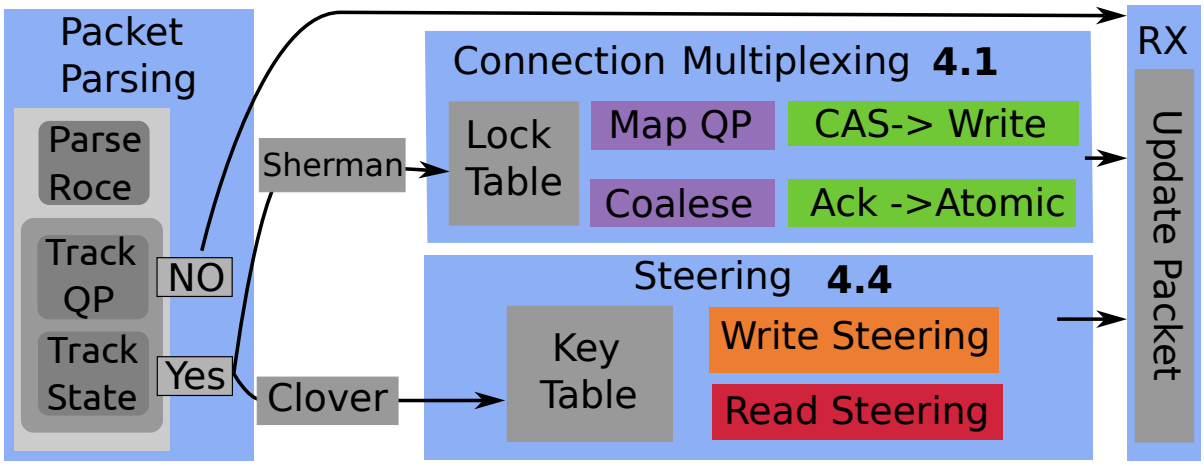

Figure 2.7. SwordBox's DPDK processing pipeline.

SwordBox itself also fails in the mean time: we defer protecting against this double-failure scenario to future work.

2.3 Implementation

We implement SwordBox in DPDK and P4. Our DPDK SwordBox implementation (shown in Figure 2.7) consists of 3,392 lines of C and includes all of the features described in the previous section—including ICRC recalculation—but is limited by single-core CPU performance. Our P4 prototype has more limited functionality, but operates at 100-Gbps line rate.

2.3.1 Connection Steering

Our P4 prototype implements connection steering (§2.2.4) by using registers to store connection state, virtual addresses, and outstanding requests. Switch registers are constrained to 32, 16 and 8-bit blocks, and are bound to specific switch pipeline stages [42]. Packets visit each stage exactly once, so register reads and writes must be pipelined correctly so that the same stage which stores a virtual address on a write, is the same that produces the address for CAS and read. Because registers are fixed width, some lookups take multiple stages. We use two-stage lookups for (64-bit) virtual addresses with two 32-bit registers, and a single stage for queue pairs, sequence numbers, and connection IDs. Prior work has demonstrated that RDMA ICRC's can

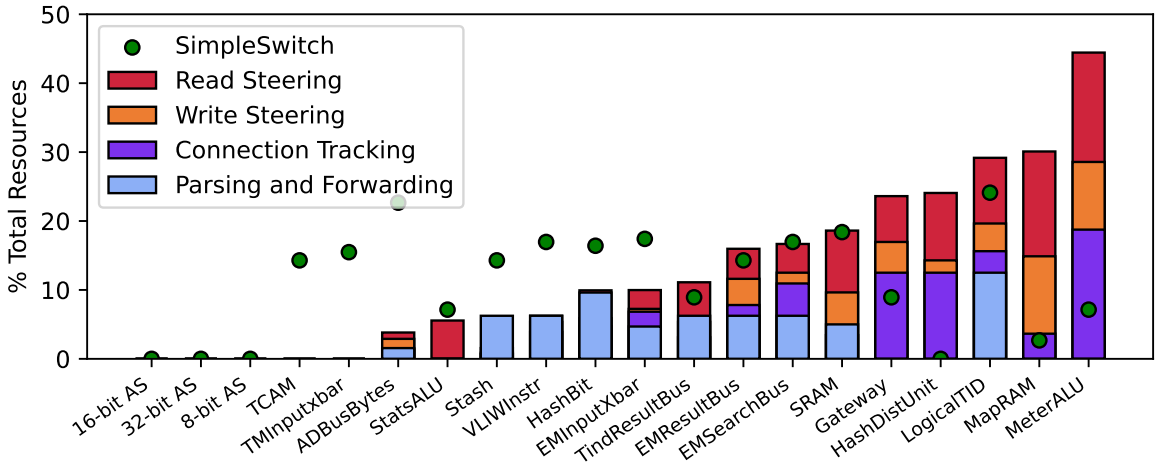

Figure 2.8. Breakdown of switch resource utilization by SwordBox component.

be implemented in a P4 switch, but are redundant with the Ethernet CRC [13, 95]. Hence, like previous authors [82], we disable ICRC checks at sender and receiver NICs and do not update them at the switch.

Figure 2.8 provides a breakdown of the resource consumption of our P4 SwordBox implementation as reported by the Barefoot SDE version 9.7.0. Each percentage is the average value across the total 16 switch pipeline stages. SwordBox fits into 8 stages, and is run entirely on the ingress pipeline. We use the header parser from the P4 simple switch to parse up to the UDP header and create our own header parser for RoCEv2 and Clover headers. SwordBox uses RoCEv2 header, write, and CAS payload information to identify traffic for steering. When new Clover traffic is identified the connection is added to the connection tracker.

2.3.2 Connection Multiplexing

While straightforward to implement in DPDK, our P4 prototype currently does not support connection multiplexing (and, hence, atomic replacement) due to the challenge of supporting ACK coalescing. In order to determine to which clients to return an ACK, a variable number—up to the number of clients—of entries must be matched against every packet. Yet, each stage of a P4 pipeline holds unique data and supports only a single lookup. Replicating entries across stages would allow for multiple lookups per packet but a server can coalesce an

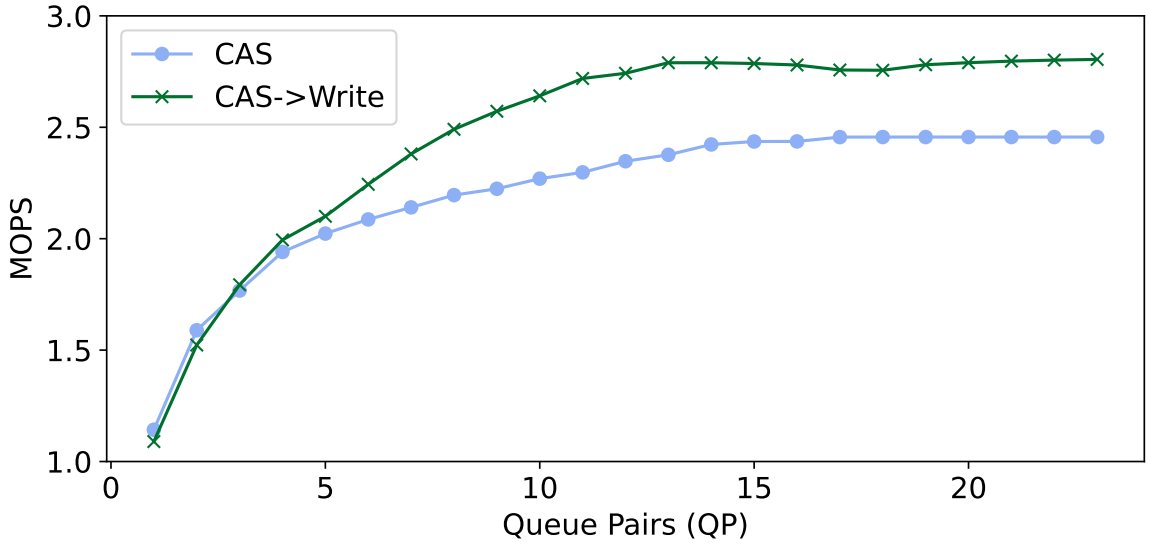

Figure 2.9. Throughput of conflicting CAS and rewritten CAS requests as a function of client threads/QPs.

arbitrary number of ACKs so no fixed number of duplications suffice (and we frequently observe coalescing of 10 or more requests). Recirculation is another alternative, but inflates bandwidth usage in the common case and causes responses to be delivered in reverse order. One obvious alternative is to disabling ACK coalescing at the server NIC, but we are unaware of a way to do so on Mellanox NICs.

2.4 Evaluation

We use our DPDK implementation to perform a micro-benchmark where we explicitly manage RDMA connections to remove the atomic operations used by Sherman's locking mechanism. We use the P4 implementation installed on a programmable switch to show the impact of in-flight conflict resolution at rack scale in Clover.

2.4.1 Testbed

Our testbed consists of a rack of nine identical machines equipped with two Intel Xeon E5-2640 CPUs and 256 GB of main memory evenly spread across the NUMA domains. Each

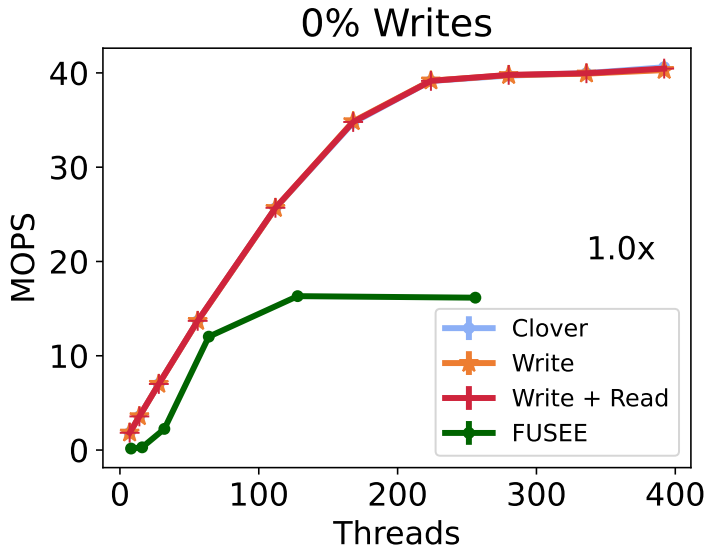

Figure 2.10. Swordbox workload performance on a read only workload. SwordBox's performance adds no observable overhead to clover.

server is equipped with an NVIDIA Mellanox ConnectX-5 100-Gbps NIC installed in a 16x PCIe slot and connected to a 100-Gbps ToR. Our DPDK-based micro-benchmarks use only three machines: a load generator, a memory server, and a machine hosting our DPDK implementation of SwordBox. The load generator is configured with default routing settings—it sends traffic directly to the memory server. We install OpenFlow rules on a Mellanox Onyx switch to redirect the traffic to the DPDK box. For the P4-based Clover experiments, we replace the Onyx switch with an Edgecore Wedge-100 programmable switch running SwordBox. We configure one server as a Clover memory server, one as a metadata server, and the remaining seven as Clover clients.

2.4.2 Atomic Replacement

We show that SwordBox is able to overcome the NIC hardware bottleneck by replacing CAS operations with writes serialized on a given RC by running a micro-benchmark that focuses exclusively on CAS performance. Specifically, we extract the CAS request from Sherman's lock operation and repeatedly generate it from one client to a single memory server (while routing it through SwordBox using OpenFlow rules). Each client thread is bound to its own queue pair,

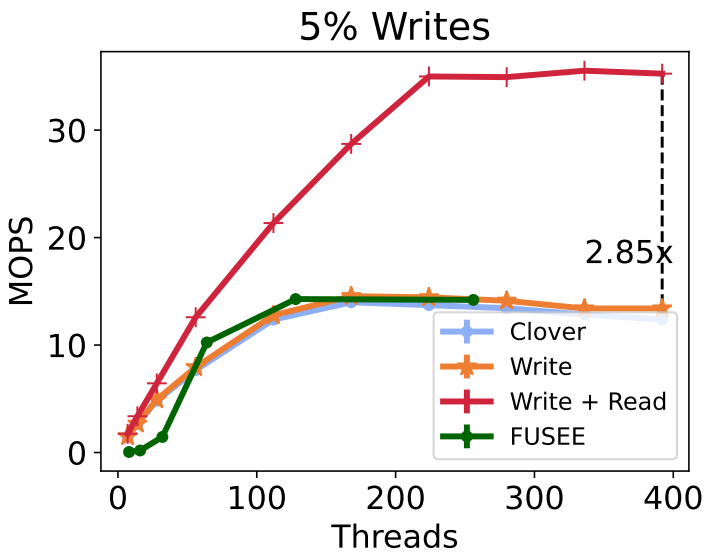

Figure 2.11. Swordbox workload performance on 5% write workload. Due to cache misses and contention SwordBoxgains nearly a 3x performance boost over Clover

and all client threads issue CAS requests to the same shared virtual address. We set the number of cores on the SwordBox middlebox to 24 so that in our maximal test case each client thread flows through exactly one middlebox core for the lowest degree of interference between QP.

Figure 2.9 shows the results when all requests are directed at the same address in the remote server's main memory. In the default case (labeled CAS in blue), SwordBox lets CAS requests flow through without modification, each on their own queue pair. In the CAS→Write (green) configuration SwordBox maps all client requests to the same QP at the server to ensure serialization and replaces the CAS operation with a simple write.

We see a significant increase in performance when SwordBox converts CAS-guarded requests to QP-serialized writes. Each configuration hits a distinct hardware limit: CAS requests bottleneck at the server NIC due to being applied to a single key (c.f. Figure 1.1). When converting CAS to serialized write operations, the bottleneck moves to the DPDK middlebox. Specifically, DPDK requires all TX for a destination QP to be done by the same core; hence, all requests must flow through a single core, capping the performance of our DPDK implementation to the maximum per-core throughput of our middlebox server: 2.8 MOPS.

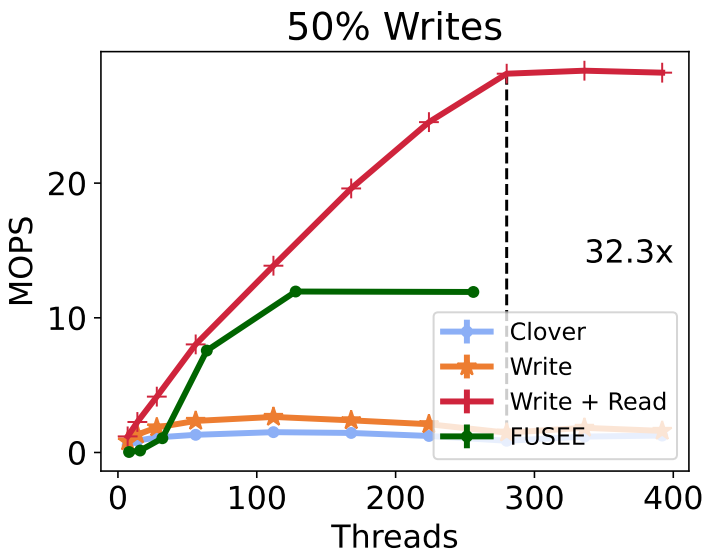

Figure 2.12. Swordbox workload performance 50% write workload (YCSB-A)

2.4.3 Steering in Clover

While atomic replacement is feasible, it requires SwordBox to explicitly manage and remap all the RDMA connections to a given (set of) server(s)—a resource-intensive task. Here, we consider the more general and lightweight case where SwordBox serves as a performance-enhancing proxy and attempts to avoid failed operations by steering requests in flight. We use workloads from the YCSB benchmark [20] to access 1,024 128-byte objects stored in Clover.

Throughput

Figures [2.10, 2.11, 2.12, 2.13] shows the impact of SwordBox's techniques at various levels of contention. A read-only workload exhibits no contention, so SwordBox simply passes through all operations unmodified achieving a maximum throughput of approximately 40 million operations per second in our testbed. As a point of comparison, we also plot (in green) the performance of a non-replicated instance of FUSEE, in which case their SNAPSHOT consensus algorithm degenerates to a lock-based approach. While FUSEE's absolute read throughput on our testbed is considerably higher than reported by the original authors on their own hardware,

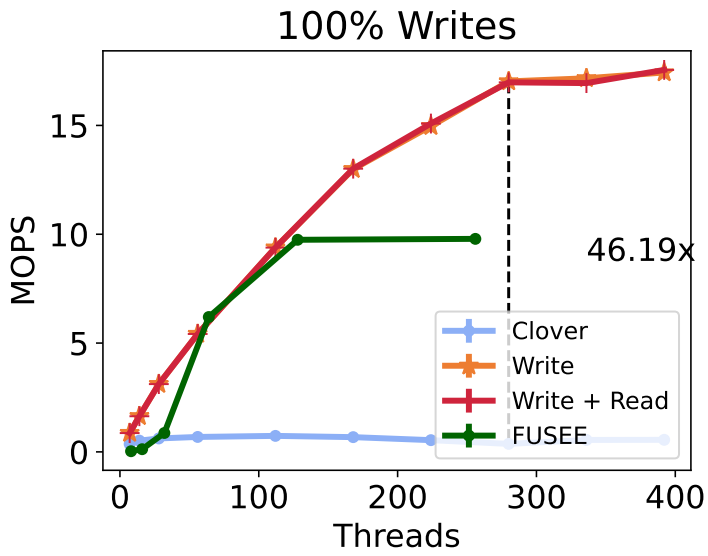

Figure 2.13. Swordbox workload performance on a write only workload

it is less than half that of Clover on this workload. While Clover clients can safely cache the linked-list location for popular keys (because any updates will cause the next pointer of the returned element to be non-NULL), FUSEE clients must always issue two seperate, dependant RDMA reads: one to obtain the current location for the desired key, and then one to read the value.[2]

Clover (shown in blue) performance decreases markedly with even 5% writes, nearly matching FUSEE; write steering alone (orange) provides minimal performance improvement as the vast majority of writes succeed on their first try—it is the reads that are failing. Steering both reads and writes (red) restores performance, although to a slightly lower overall throughput as even successful Clover writes require two RDMA operations instead of one. At 50% writes, over half of all write requests fail so applying write steering almost doubles performance. The steered writes, however, then out-pace reads causing the majority of reads to fail unless SwordBox also applies read steering. (The impact on tail latency is clearly shown in Figure 2.15.) Of course, in a 100% write workload write steering alone is sufficient. While FUSEE suffers less from

[2]While the results in the FUSEE paper suggest it outperforms Clover [87, Figs. 13–15], Clover's client cache is disabled in those experiments, forcing all reads to go through the metadata server. Moreover, in our experiments, FUSEE fails to scale beyond 256 clients—published results only go to 128 [87].

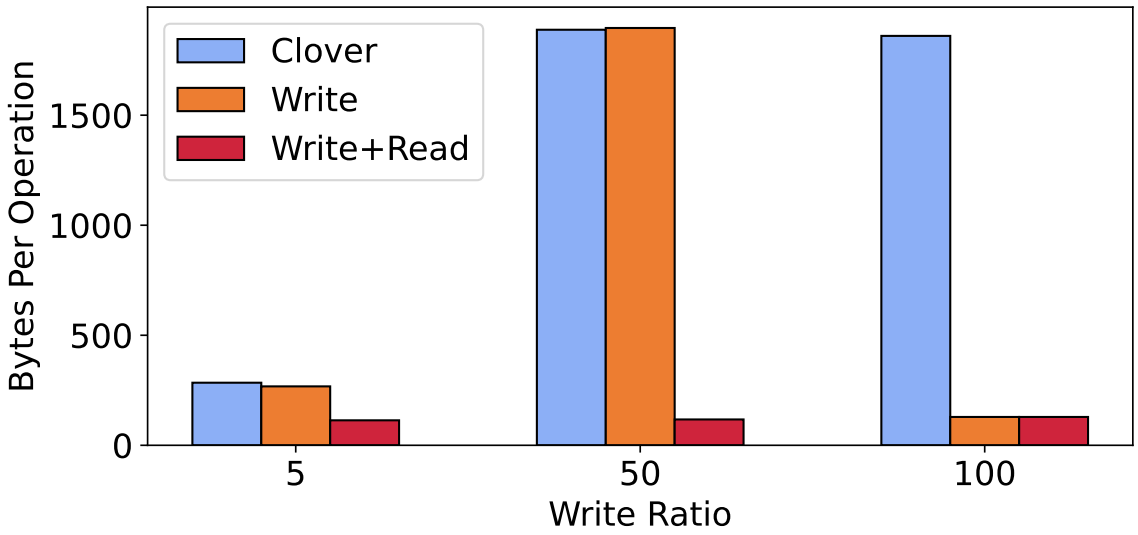

Figure 2.14. Average number of bytes required per Clover operation on 128-byte objects using each of the three techniques at various write intensities.

increased contention, its writes require three or more RDMA operations; as a result SwordBox pushes Clover to achieve 1.9–2.5× higher throughput than FUSEE.

Bandwidth reduction

Under contention, Clover's remote operations can require additional packet exchanges which inflate the bandwidth necessary to service the same number of memory accesses. SwordBox's steering algorithms remove the need for requests to retry, eliminating the overhead. Figure 2.14 plots the average bytes per operation for each strategy across the three workloads with writes. (The read-only workload, not shown, never needs to retry.) We calculate the value for each technique by summing the total bandwidth across a run and dividing by the number of operations. Clover's bandwidth usage increases with contention, growing by 2.5× at 5% and 16× at 50% writes—all of which is recovered by applying read and write steering. Write steering alone causes significant inflation in the cost of operations at 50% writes because many read requests fail as discussed above.

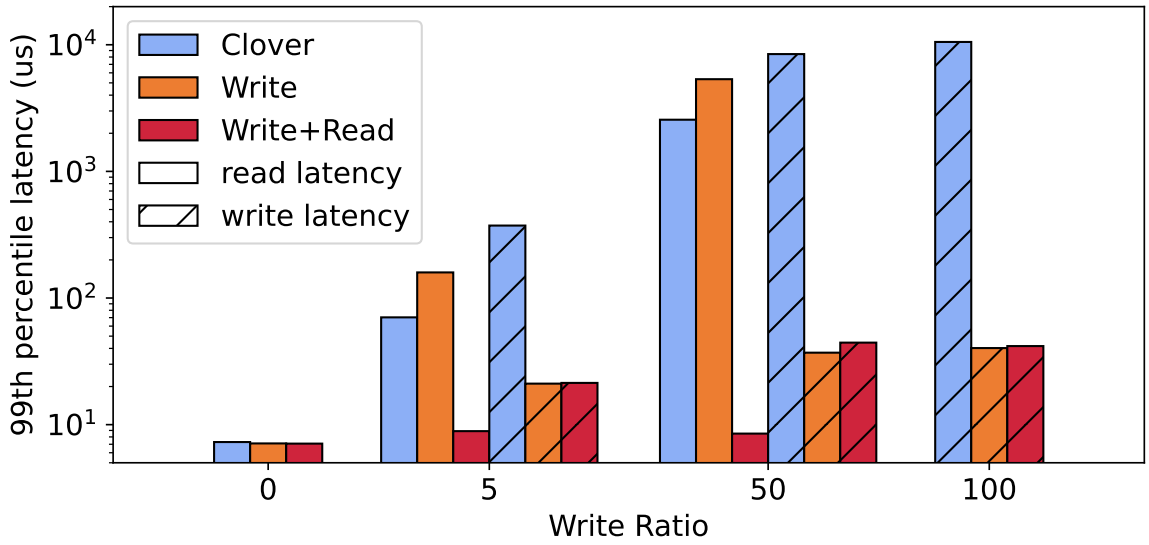

Figure 2.15. 99th-percentile tail latencies of read (solid) and write (striped) Clover operations at various write intensities. (Note logarithmic *y* axis.)

Tail latency

Optimistic concurrency is well known to exhibit poor tail latency under contention, and Clover is no exception. SwordBox significantly reduces latency as steering ensures that nearly all requests succeed on the first try. Figure 2.15 shows the 99th-percentile tail latencies associated with SwordBox's read and write steering in comparison to default Clover at varying write intensities. Clover's p99 read latency (solid blue) at 5% writes is 70 μs, around 10$\times$ its baseline our our testbed. With read and write steering (solid red) the read tail latency drops to 8 μs—a 8$\times$ improvement over Clover even in this low-contention regime. At 50% writes the performance increase from steering increases dramatically: p99 read latency drops by over 300$\times$. Writes (hashed) have slightly more than double the latency of reads as they require two round trips and atomics are slower to execute than other operations. Combined write and read steering provides a 17, 189, and 252$\times$ improvement in write tail latency, respectively, across 5, 50, and 100% write workloads. As one might expect, performing write steering alone privileges writes over reads, dropping their tail latencies slightly further—at the cost of a dramatic spike in read tail latency.

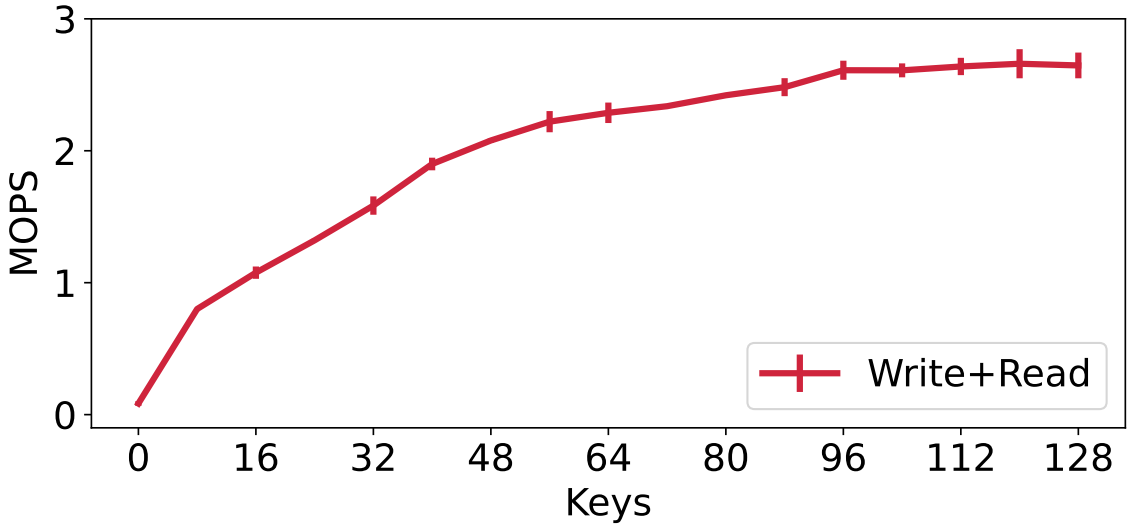

Figure 2.16. Per-client throughput as a function of the number of Clover keys SwordBox steers. 50:50 workload averaged across 6 hosts each running 56 threads.

Partial steering

One of most appealing aspects of SwordBox's steering is the fact that it need not be applied to all servers, or even memory regions (i.e., Clover keys) of a given server. Figure 2.16 shows per-client throughput as a function of the number of keys steered by SwordBox. To accentuate the impact, we use a Zipf parameter of 1.5—as opposed to 0.99 in prior experiments—to enhance the locality of requests. Steering requests for only the hottest-8 keys provides a 9.5× improvement while tracking the hottest 64 delivers 27×.

2.5 The Cost of Programmable Switches

As shown throughout its evaluation, SwordBox offers significant performance improvements over existing end-host solutions. The key insight behind SwordBox is that a small amount of programmability in the network combined with a massive amount of bandwidth can offer order-of-magnitude performance improvements for existing data structures, both lock-based and optimistic. However, SwordBox comes at a non-trivial cost. Programmable switches are

expensive, complex, and difficult to program. While SwordBox offers a solution to the problem of contention, it is likely that only the most performance-critical applications would be likely to qualify for the care and attention required for crafting a SwordBox-like solution. Further complicating the matter is the fact that the Tofino series of switches has been discontinued by Intel [59].

At a data structure level, the memory limitations of a switch pose a significant challenge for generalization. In the case of append operations made to a linked list, only the final value of the list needs to be cached to ensure the data structure's integrity. However, inserting into an arbitrary location in the list would require the entire list to be stored in cache. Given these operational complexities, high cost of development, and data structure limitations, we ask the question: *What other techniques can be used to improve the performance of disaggregated data structures?*

2.6 Acknowledgement to SwordBox Contributors

Chapter 2 is a partial reprint of work submitted to multiple USENIX and ACM conferences under the title "SwordBox: Accelerating Shared Access in RDMA-based Disaggregated Memory. Stewart Grant, Alex C. Snoeren. This dissertations author was the primary investigator and author of this paper.

Thank you to Alex C. Snoeren for his guidance and support throughout this project. Thank you to Rajdeep Das for his expertise in P4 and for supplying the initial P4 code and compiler configuration for our switch. Thank you to Anil Yelam for reviewing the figures and providing feedback on the initial draft of this work. Thank you to the reviewers at the *Workshop on Resource Disaggregation and Serverless (WORDS '21)* for your feedback and revision notes on this work early in its development. Thank you to Geoffrey M. Voelker and Yiying Zhang for your feedback on this project.

Chapter 3

Disaggregated Data Structure Design

SwordBox takes the stance that an additional piece of in-network equipment can be used to accelerate an existing data structure. But what if we could just make the data structure itself faster? In this chapter, we explore the design of a disaggregated key-value store, RCuckoo, which aims to answer exactly this question. Instead of adding a new piece of equipment to accelerate a data structure, with RCuckoo we aim to get better performance by co-designing itself with the network.

There are a variety of data structure-specific optimizations that have been leveraged to improve the performance of disaggregated data structures. Our position in this work is that locality-based optimizations can provide significant benefit due to the high cost of round trips to remote memory and the fact that network capacity continues to grow at an astonishing rate. In this chapter, we make the case for locality-optimized cuckoo hashing and show how it can be used to improve performance, reduce contention, and make use of the latest trends in network hardware.

3.1 A Case for Fully Disaggregated Cuckoo Hashing

In general, key-value stores rely upon a high-performance index structure to localize key operations and maintain values separately, necessitating multiple RDMA operations and network round trips even in the absence of contention. Moreover, given the dominance of read-heavy

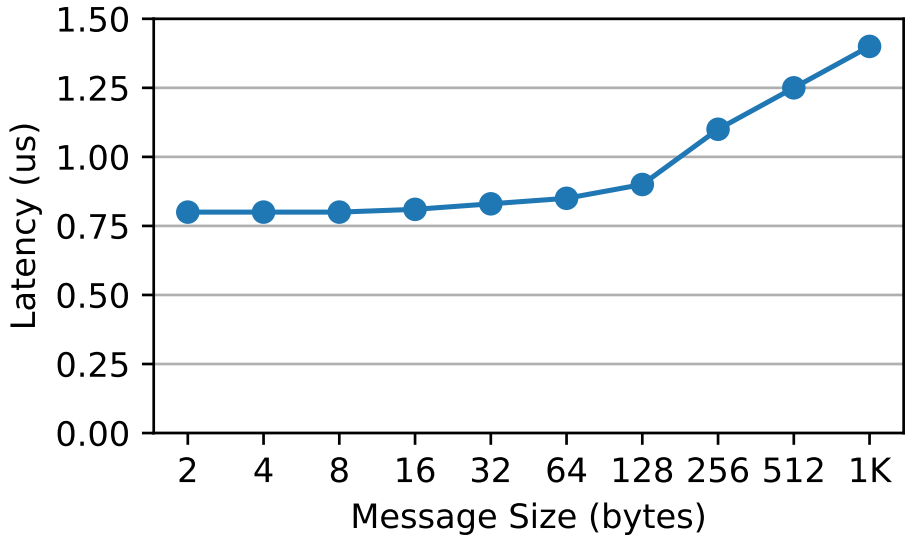

Figure 3.1. RDMA operation latency as a function of message size [3]

workloads [16, 73], most systems eschew locks in favor of optimistic update approaches that can lead to poor performance under contention. In this work we design an index datastructure tailored for the constraints of current RDMA hardware. Specifically, we facilitate lock-based updates by decreasing the number of round trips required to acquire locks and perform mutating operations.

We introduce RCuckoo, a fully disaggregated key/value store based on cuckoo hashing [75] that uses only one-sided RDMA operations. RCuckoo builds around a *dependent hashing* algorithm that makes spatial locality a tunable parameter. RCuckoo employs a set of complimentary techniques that leverage this enhanced locality to deliver higher performance than any prior disaggregated key/value store while gracefully handling client failures:

- **Deterministic lock-free reads.** Cuckoo hashing ensures an entry is always located in one of two locations which can be read in parallel.

- **Space-efficient locks** frequently allow clients to acquire necessary locks in a single RDMA operation.

- **Client-side caching** enables accurate cuckoo-path speculation to improve insert performance.
- **Leased lock acquisitions** allow clients to detect and recover from client failures using timeouts.

Combined with a datastructure design that facilitates aggressive batching of RDMA operations, these techniques enable RCuckoo to limit the number of round trips required for all table operations. In the common case, reads execute in one or two (for large values) round trips, uncontested updates and deletes require two round trips, and the median insert operation involves only two round trips—although the expected number increases as the table fills. On our testbed, RCuckoo delivers comparable or higher performance on small values across the standard set of YCSB benchmarks than all of the existing disaggregated key/value stores we consider. Concretely, with 320 clients RCuckoo delivers up to a 2.5× throughput improvement on read-intensive (YCSB-B) workloads and up to 7.1× their throughput on write-intensive (YCSB-A) workloads. Moreover, RCuckoo's performance remains high despite 100s of clients failing per second.

3.2 Design

In this section we describe the design of RCuckoo, a fully disaggregated lock-based cuckoo hash table in which clients communicate with passive memory servers over reliable RDMA connections using exclusively 1-sided operations. We first describe our table design and protocol to read and modify the contents of the table. In the common case, reads complete in one (for small values) or two (for large values) round trips while update and delete operations require two. Then we introduce a locality-enhanced hashing algorithm and show how it enables our protocol to perform inserts in a small number of round trips. Finally, we discuss lock-table practicalities. For simplicity, we describe our design in the context of a single memory server,

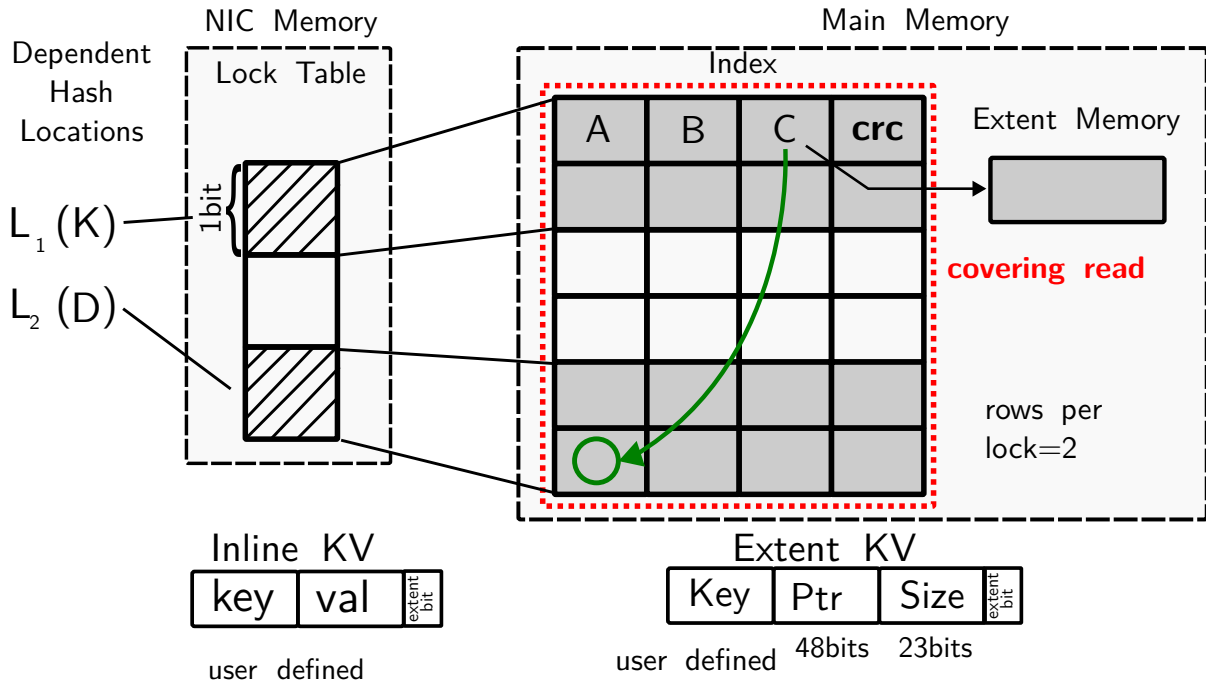

Figure 3.2. RCuckoo's datastructures showing insertion of key K as it displaces C, whose value is stored in an extent.

but it is straightforward to shard a large hash table across multiple servers (with a minor tweak, see Section 3.2.3).

3.2.1 Datastructures

Figure 3.2 shows RCuckoo's index and lock table (both maintained at the remote memory server) during an insertion of key K. The index table (right) is a single region of RDMA-registered main memory divided into rows of fixed-width entries. Each row contains n associative entries (3 in this figure; we use 8 in practice) and terminates with an 8-bit version number (not shown) and 64-bit CRC (that is computed over the entire row including version number). Clients access the index table using 1-sided RDMA reads and writes; CRCs facilitate lock-free reads as each row can be self-verified while version numbers enable clients to detect if a row has been modified.

Table entries contain either inlined key/value pairs or a key and 48-bit pointer to an extent to store larger values. The least-significant bit of an entry signifies its type; values are inlined by default. Extent entries use 23 bits to encode the value size (which can range from 2^3 to 2^{26}

bytes). Extents are located in separate, pre-allocated, per-client, RDMA-registered regions of server memory to avoid contention on inserts. Entry and key sizes are configuration parameters, but must be fixed at table creation.

Locks (stored in a bit vector in NIC memory, shown on the left) each protect a tunable number (here, two; 16 in our experiments) of index rows. Clients perform lock acquisition and release with RDMA compare-and-swap (CAS) operations. Specifically, RCuckoo leverages masked CAS (MCAS) operations [2, 93] to obtain multiple locks simultaneously while avoiding false sharing.

3.2.2 Operations

We detail the operations supported by RCuckoo below; Figure 3.3 visualizes the corresponding message exchanges.

Reads

RCuckoo is designed to facilitate lock-free, single round-trip reads for small values as they are the dominant operation for key/value stores in many data centers [16, 73]. To read the value associated with a given key clients calculate the potential table locations for the key's entry (using the hash functions described in the next subsection) and issue RDMA reads for both rows simultaneously. Because all operations between a client and a given server travel over a reliable connection, they are intrinsically ordered, but in our description we will only call out when a particular ordering among a batch of messages is required.

Moreover, as we discuss below, if the rows are located sufficiently close together, it can be beneficial for the client to issue a single *covering read* that returns the contents of both rows—as well as intervening and potentially surrounding ones—in a single request. In our experiments RCuckoo clients issue a single, large read rather than two small reads if the locations are within 148 bytes of each other (i.e., adjacent rows) as our parameter sweeps show it has a negligible increase in latency over smaller RDMA requests (Figure 3.1) and provides substantial

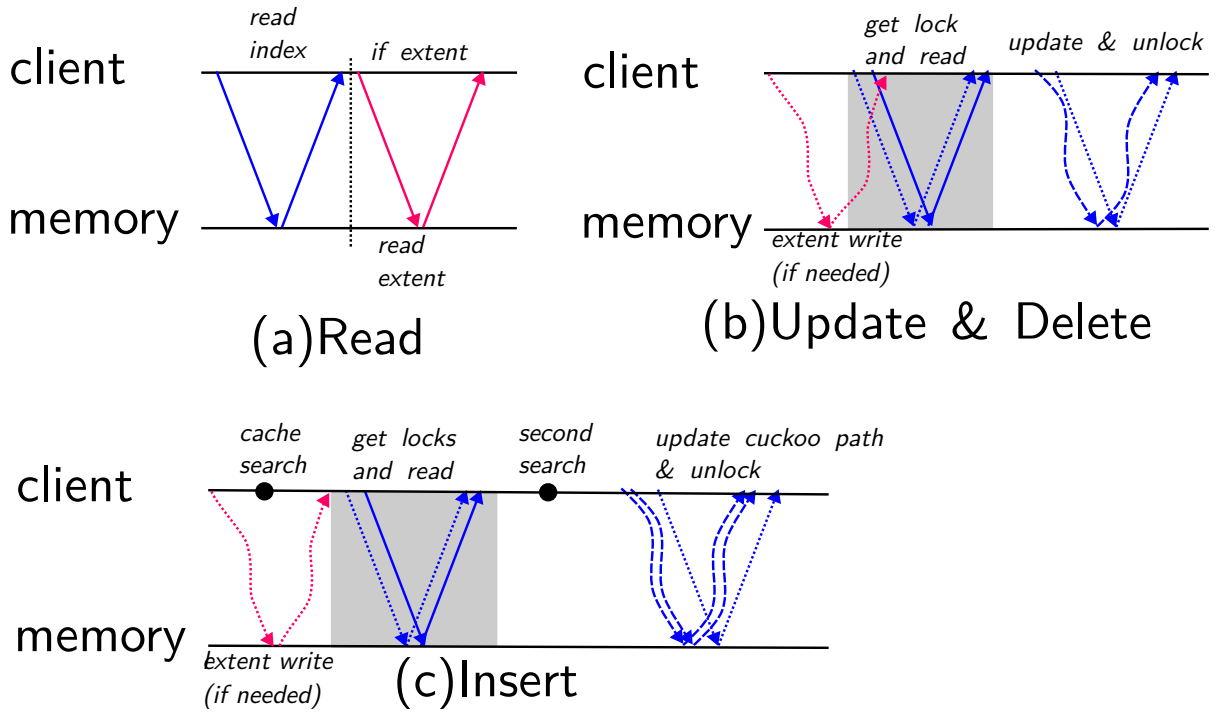

Figure 3.3. RCuckoo's protocol for reads, inserts, deletes and updates. Blue lines are index accesses, and red lines are extent accesses. Solid lines are reads, dotted lines are CAS, and curved dashed lines are writes.

improvement for insert operations. (If bandwidth is not a concern, we find that a 2-KB threshold can deliver a further 3% boost to insert performance, but the resulting read amplification is significant.)

A read is successful if either row contains an entry with the desired key and the row's CRC is valid. An invalid CRC indicates a torn write or rare failure case, in which case the operation is retried (see Section 3.3.1). As shown in Figure 3.3(a) inlined reads are complete after one round trip, while reads for large values require a second round trip to retrieve the extent.

Updates and deletes

Updates and deletes, like reads, access only two locations in the index table, but require a client to acquire the associated locks. Due to RCuckoo's locality enhancement, it is usually possible to attempt to acquire both locks in a single MCAS operation (Section 3.2.4). If so, the client issues read(s) for the corresponding rows of the index table immediately afterwards but in

the same batch of operations.[1] In the rare case that the locks must be acquired independently—necessitating an additional round trip—the reads are batched with the second lock request.

Assuming successful lock acquisition and valid reads, the operation can proceed if the key is present in either location. In a single (ordered) batch of operations, the client first writes the updated/freed entry and recomputed row version and CRC before releasing the locks. When updating values stored in extents, clients store the value to a new extent via an RDMA write that is sent in parallel with lock requests. On lock release clients write the first bit of the old extent to free it. Deletes operate identically save writing a new extent. Clients garbage collect their own extents by occasionally scanning their allocated region for freed extents. Figure 3.3(b) shows that most uncontested operations complete in two round trips; clients retry acquisitions until they succeed or detect a failed client.

Insert

Inserts are challenging because concurrent operations might result in cuckoo paths that collide. Rather than face the prospect of having to unravel a partially completed insert upon collision, RCuckoo clients compute a complete cuckoo path ahead of time and then acquire locks on all the relevant rows to ensure its success. Moreover, to facilitate recovery from client failures, an insert is performed by cuckooing elements one at a time, starting by moving the last entry in the path to the empty location, and then replacing it with the previous entry in the path, and so on until the new entry is inserted in its primary location.

To speed up cuckoo-path searches, RCuckoo clients keep a local, RDMA-registered cache of (relevant portions[2] of) the index table. Clients validate—and, if necessary, update—their cache at each step of the insert operation as explained below, so stale cache entries do not impact correctness, only performance.

At a high level an insert operation proceeds in three (or four) phases. For extent entries,

[1] Because the lock table is located in NIC memory, RCuckoo clients can employ `SEND_FENCE` on reads batched with lock acquisitions to ensure consistency without incurring a performance penalty.

[2] A small cache suffices; we use 64 KB in our experiments. Caching the entire index yields negligible additional benefit.

clients first write the value to a free extent—in parallel with the remaining three phases. Clients maintain a local slab allocator that manages their private extent region, so there is no contention. Regardless of whether the entry contains an inlined value or a pointer to an extent, RCuckoo clients start by identifying a potential cuckoo path using only the contents of their local table cache. Clients then simultaneously attempt to acquire the locks for and update their cache of the rows that comprise the candidate path. Using only the contents of their newly updated local cache, clients conduct a second search to confirm that a candidate path—either the initial guess or an alternative that similarly consists only of currently locked rows—exists. If so, the insert is performed; if not, the client releases its locks and retries.

Speculative local search: Identifying a viable cuckoo path for insertions requires multiple dependent reads. RCuckoo attempts to limit the number of remote operations by first conducting a speculative local search. Prior to contacting the server, RCuckoo clients search the contents of their local table cache to build a speculative cuckoo path. As with prior work, we use breadth-first search (BFS) to identify short paths in an attempt to minimize bandwidth and locking overhead [56]. If the client cache is empty the degenerate path is presumed, i.e., that the key will be inserted into its primary location without the need for any cuckooing. Obviously, speculative cuckoo paths are most useful when client caches are fresh—often due to a failed prior attempt to insert the same key.

Cache synchronization: Armed with a potential cuckoo path, clients identify the set of locks necessary to protect the relevant rows. Approximately 99% of paths can be locked with a single MCAS operation (Figure 3.6); longer paths acquire locks in groups (Section 3.2.4). Immediately after, but in the same batch of RDMA operations as each attempt to acquire (a subset of) the locks, clients synchronize their local cache by issuing reads for (at least) all of the rows covered by that set of locks. In general, locks cover multiple rows, so this will be a superset of the rows necessary for the identified path. Note that if lock acquisition is successful, the values returned by the read of the corresponding rows—and, thus, the client's local cache of those rows—will remain synchronized until the lock is released.

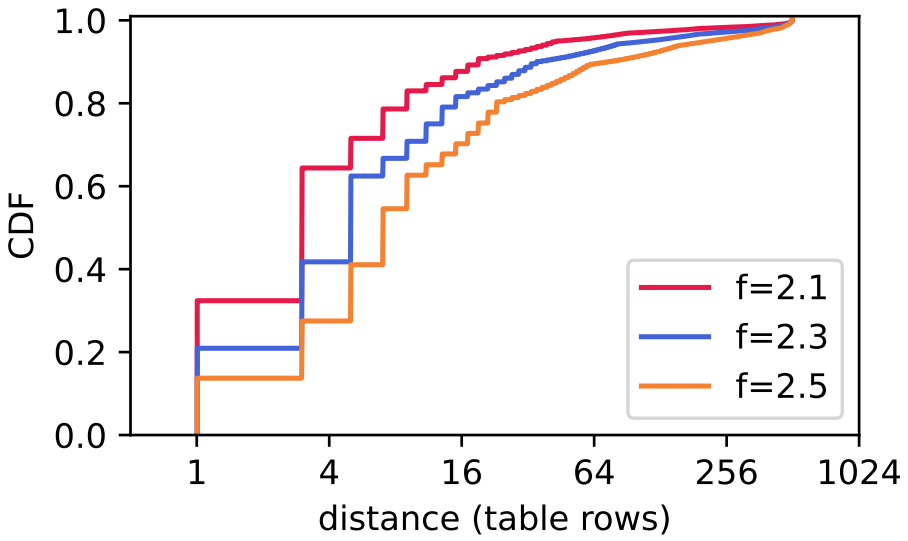

Figure 3.4. CDF of distances between cuckoo locations for different locality settings using RCuckoo's dependent hashing.

Figure 3.2 shows an example insert operation where the client updated its cache of the section of the index table outlined by the dashed red line using a single covering read. Here, the primary row for K is full and the entry for key C is being evicted to its secondary location. Hence, the client has acquired locks corresponding to the row where it hopes to insert the entry for key K as well as the row into which it plans to cuckoo the existing entry for key C. The rows shaded in gray are synchronized because they are covered by the acquired locks, while the intervening two (unshaded) rows are updated in the client's cache, but their contents cannot be depended upon without additional validation. Rather, they may increase the likelihood that a subsequent speculative search yields a valid result.

Second search: If all the lock acquisitions are successful, the client validates that the initial path remains valid. Under an insert-heavy workload, however, speculative cuckoo paths are frequently stale. Yet, a valid cuckoo path may still exist within the locked rows. Hence, if the initial path is no longer viable, clients perform a second search restricted to only the synchronized part of its cache, i.e., rows for which they currently hold the lock. In either case, if a valid path is found the series of swaps and version/CRC updates are calculated and issued as a batch of

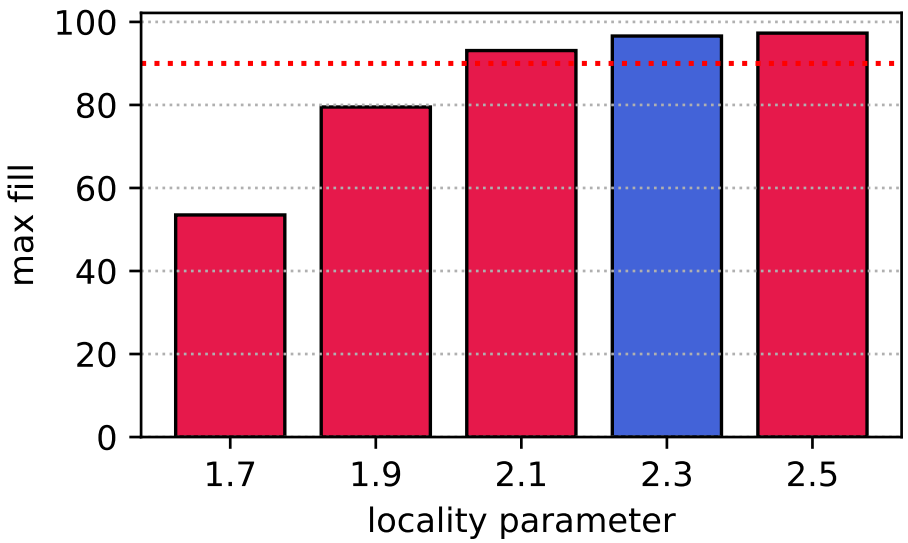

Figure 3.5. Achieved maximum fill percentage for different locality settings. 90% fill indicated by dashed red line.

RDMA writes, one row at a time, followed by (an ordered set of) lock releases. If no valid path exists the client releases its locks and tries again, conducting another speculative search on the updated cache contents.

This entire process repeats until success, a client determines there is no viable cuckoo path (at which point the insert operation returns an error indicating the table is full), or a failed client is detected. Given the fully-disaggregated context we assume the index table will be initially provisioned at its maximum size; we defer resizing to future work. If cuckoo paths were to randomly span the table it is unlikely that an alternate valid path would exist within the locked rows when speculation fails. In the next subsection, however, we describe how RCuckoo uses dependent hashing to dramatically increase the likelihood that an alternate path exists using rows surrounding those identified by the speculative search.

3.2.3 Locality

In a traditional cuckoo hash, the two locations for a given key are deliberately independent which allows the table to be filled quite full before inserts begin to fail. In RCuckoo the distance

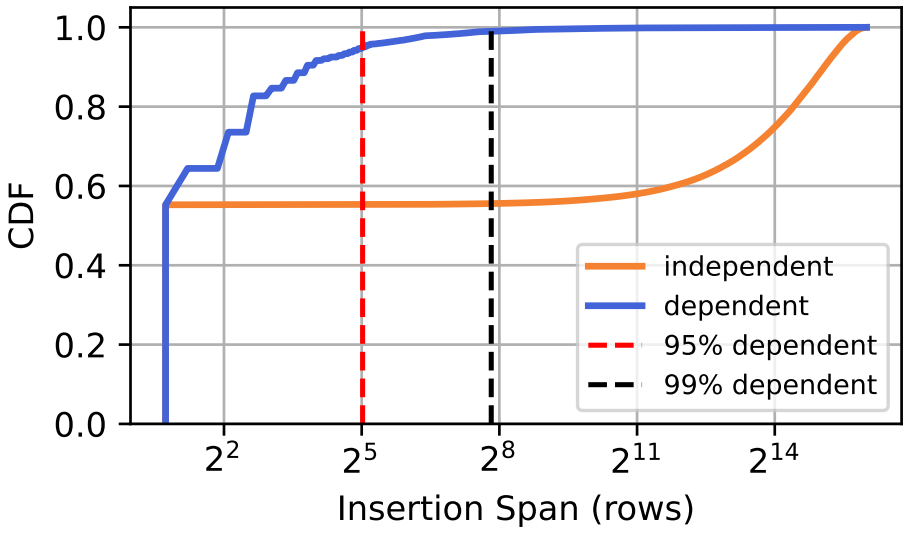

Figure 3.6. CDF of cuckoo spans for dependent and independent hashing. A cuckoo span is the distance between the smallest and largest index in a cuckoo path

between keys' two cuckoo hash locations is a tunable parameter. We show experimentally that an optimal locality setting can dramatically decrease the number of round trips required to perform inserts in RCuckoo while maintaining high (90%+) fill maximum factors.

Increased locality has two direct benefits: it increases the probability that both of a keys' locations can be read with a single covering read—which updates more of a client's local cache, improving the likelihood that failed insert operations will succeed upon retry—and decreases the number of MCAS operations necessary to acquire the relevant locks. It also reduces the region of the index table likely to be spanned by cuckoo paths, which speeds up inserts, but leads to hot spots that limit the table's expected maximum fill factor.

In RCuckoo, the primary location for a key is chosen uniformly at random, while the second is offset from the first by uniformly random value drawn from a probabilistically bounded range, where the range is likely to be relatively small. We start with a base hash[3], $h()$, and use it to implement three independent hash functions $h_1()$, $h_2()$, and $h_3()$. (In our implementation we use a different salt for each of the three functions.) We compute the two locations L_1 and L_2 for

[3]We use xxHash [18] in our implementation.

a key K as

$$L_1(K) = h_1(K) \mod T,$$

$$L_2(K) = L_1 + (h_2(K) \mod f^{f+\mathscr{Z}(h_3(K))}) \mod T,$$

where T is the size of the index table in rows, $\mathscr{Z}(x)$ is the number of trailing zeros in x, and f is a parameter that controls the expected distance between the two hash locations. (In a sharded deployment, the second location is restricted to the same shard by "wrapping around" the offset accordingly.) The particular formulation is not important, but the upshot is exponentially fewer keys have secondary locations at increasing distances from their primary. The latter aspect is crucial, as any fixed bound on the distance between hash locations leads to low maximum fill factors (on the order of 10–15% in our experiments).

Figure 3.4 shows the distance between hash locations as a CDF for different values of f, while Figure 3.5 shows that larger values of f enable practical fill factors for tables with 100 M entries. In our evaluation we set $f = 2.3$. As shown in the figures, for index tables with eight entries per row, RCuckoo delivers an expected max fill of greater than 95% with a 68% probability that a key's locations are located five or fewer rows apart.

Decreased distances between hash locations naturally lead to shorter cuckoo paths when combined with our breadth-first search approach. Using $f = 2.3$ and a table size of 100-K rows, Figure 3.6 shows that when filling the table to 95% full, slightly more than half of insertions do not require any cuckooing, and 95% of insertions require cuckoo paths that span 32 or fewer rows while nearly 99% span 256 or fewer. Conversely, with independent hashing, insertions that require any cuckooing at all almost always result in spans of 2 K rows or more.

3.2.4 Locking

While RCuckoo reads are lock free, updates and especially inserts depend critically on locking performance.

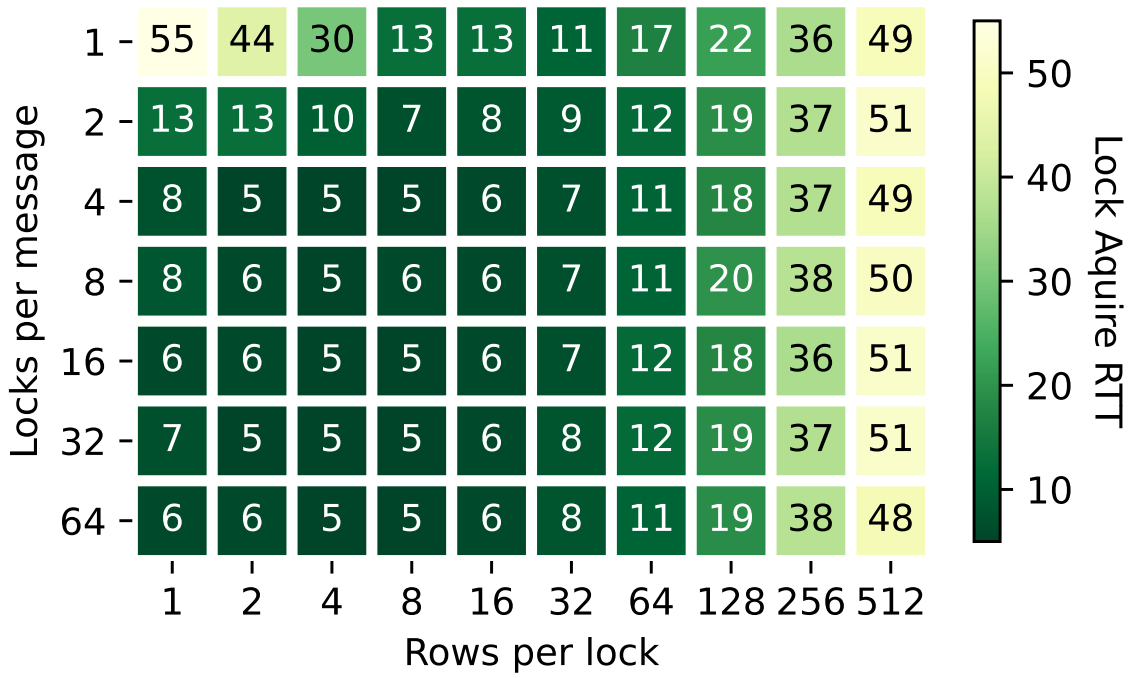

Figure 3.7. 99th-percentile round trips required per insert in a 512-row table when filling to 95%. 512 buckets per lock corresponds to a single global lock.

Lock granularity

Increased locality decreases the number of round trips required for lock acquisition. Recall the lock table is a linear array of lock bits and each bit locks one or more table rows. As mentioned previously, RCuckoo implements lock acquire and release using RDMA masked compare-and-swap (MCAS) operations that can update 64 bits at a time. To avoid deadlock RCuckoo acquires locks in increasing order. For any given operation, clients group the necessary locks into the smallest number of sets as possible (where each set is an attempt to acquire one or more locks within a single 64-bit span) and issue MCAS operations one at a time in order of their target address. Clients continuously spin on lock acquisitions and only move to the next MCAS operation after the current one succeeds. Due to this one-MCAS-per-round-trip acquisition procedure, lock granularity is critical to performance.

If, as in Figure 3.6, almost 99% of cuckoo spans are 256 rows or less, and each lock protects four rows, nearly 99% of insertions can have their locks acquired with a single MCAS. Of

course, increasing the number of rows covered by a single lock can lead to false sharing, forcing additional retries to acquire the necessary locks. Figure 3.7 shows the results of a representative experiment where 8 clients are concurrently filling a 512-row table to 95% full. We report the 99th-percentile (i.e., the most expensive inserts when the table is nearly full) number of round trips required to acquire the locks necessary to perform an insertion as a function of both lock granularity (i.e., number of rows per lock, on the x-axis) and lock size (i.e., the number of locks that can be accessed with a single MCAS operation, on the y-axis—RCuckoo's single-bit locks correspond to 64 locks per message).

While there is some noise due to experimental variance, the far-right column shows that a single global lock results in high contention (as there is only one lock in the system, it does not matter how many locks can be acquired per message). Conversely, the top left corner shows that, despite the lack of false sharing when a lock corresponds to exactly one row, the inability to acquire more than one lock at a time leads to a large number of round trips. Under these conditions, the sweet spot falls in the range of 2–16 rows per lock. In our experiments we use 16 rows per lock as, when combined with our choice of f, a single MCAS suffices for the vast majority of insertions.

Clients may fail while holding locks, preventing other clients from making progress. As described in the next section, clients independently detect such failures by setting a time out for lock acquisition. To avoid false alarms, i.e., clients timing out due to a slow-moving client (who might, for example need to acquire a long list of contended locks), RCuckoo clients are given a bounded time to acquire all their locks. If a client takes longer than this time to acquire all of its locks it releases all held locks and restarts the (timer and) acquisition process, hopefully allowing other clients to complete their own acquisitions.

Virtual locks

While Figure 3.7 suggests that RCuckoo could employ larger locks (e.g., 8–16-bits per) without increasing the number of round trip times required to acquire them, there is an additional

design constraint that drives our choice of single-bit locks. Specifically, to improve locking performance, RCuckoo locates the lock table in NIC device memory which delivers 3× higher throughout on contented addresses than host memory (see Figure 3.1). It is also lower latency as operations to device memory avoid a PCIe round trip. Unfortunately, NIC memory is limited (to 256 KB on our ConnectX-5s), so this choice bounds the size of the lock table and drives our single-bit design.

Moreover, to allow RCuckoo to support tables with more than 64 M rows, we implement a *virtual* lock table where multiple logical locks map to a single physical lock. Concretely, we map a logical lock l drawn from a table of size L to a physical location p in a bit-array of size P by computing $p = l \mod P$. Mapping multiple virtual locks to a single physical lock introduces yet another source of false sharing, but allows us to support arbitrarily large tables. When employing virtual locking, clients performing an insert first map rows to virtual locks, then to physical locks, then sort them into groups.

3.3 Fault Tolerance

In keeping with the rest of its design, RCuckoo handles failures in a fully disaggregated manner as well. We depend upon the RDMA hardware to handle network failures and focus exclusively on fail-stop client behavior. Server failure can be addressed by employing client-driven replication on top of RCuckoo. While there may be opportunities to integrate replication into RCuckoo itself, we defer such an exploration to future work. In RCuckoo, client failure is only of concern if the failure occurs in the middle of a mutating operation (i.e., update, delete, or insert); hence, RCuckoo detects client failures by noticing that a pending mutation does not complete in a timely fashion. Any client that encounters such a situation endeavors to recover the stranded lock and repair the impacted portion of the index table. The remainder of this section describes how RCuckoo clients detect faults, reclaim stranded locks, and, if necessary, repair the index table. Finally, we discuss additional measures that can be employed to prevent stale writes

if desired.

3.3.1 Failure detection

Clients detect failures by setting a timeout when attempting repeated lock acquisition or read requests. Because RCuckoo operations are designed to require only a few round trip times, a client performing a successful mutating operation will complete and release its locks extremely rapidly. Conversely, a client that is unable to acquire all the locks required for an insert operation releases those they do hold before trying again. Hence, it is extremely unlikely that repeated attempts to acquire a lock or perform an untorn read will fail continuously.

Of course, there is a possibility that a given row is highly popular, leading to high lock contention and/or repeatedly torn reads. Clients distinguish this case by consulting the CRC for the row they are unable to successfully read or lock. Because each mutation increases the version number, even updates that replace an entry with the same value will result in a different CRC. Clients declare a false positive and restart their failure timer if a CRC changes between attempts.

We expect client failures to be relatively rare, so set our fault timeout conservatively. Failure timers must allow for worst-case locking time, second-search time, and RDMA message transmission time. We bound locking time; search and message propagation are both measured in single-digit microseconds on our testbed. To guard against the possibility that network conditions lead to high rates of RDMA retries we set the maximum RDMA operation retry number to three. In this context, we set the failure timeout to 100 ms in our experiments, orders of magnitude above the 99th-percentile insert time of 50 μs.

3.3.2 Repair Leases

RCuckoo recovers stranded locks one lock at a time; if a client fails holding multiple locks recovery may be conducted by multiple clients at different times depending on their access patterns. RCuckoo's lock table does not maintain records of ownership, so there is no way to "transfer" lock ownership from the failed node to a recovery node in the table itself. Instead,

Figure 3.8. Format of a repair lease table entry

clients acquire a *repair lease* that grants exclusive permission to reclaim locks on a region of the index table. The index table is broken into n regions so repairs can be executed in parallel.

Figure 3.8 shows the format for entries in RCuckoo's lease table, maintained in RDMA-registered main memory on the server. Lease entries contain the lease holder's queue pair ID (which RDMA ensures is unique for a given server), a set bit, and a counter (incremented on each acquisition). A lease is considered free if the set bit of the current entry in the lease table is zero. Clients attempt to acquire the lease using RDMA CAS operations to ensure mutual exclusion. Upon successful acquisition, a client completes the repair (described below) and then relinquishes the lease by clearing the set bit. Leases are revoked (to handle the case of a failed recovery node) using a timeout mechanism similar to normal locks. If a client times out while attempting lease acquisition it claims it for itself (again, using CAS to resolve any races) and marks the lease holder as failed (see Section 3.3.4).

3.3.3 Table repair

All modification operations write new entries as a cuckoo path; updates and deletes have a path of length one. Cuckoo paths are executed by first claiming an open entry at the end of the path and proceeding backward along the path, cuckooing entries forward one-by-one until the new entry is written at the beginning of the path. Client failures can occur at any point along an insertion path; a failed client can leave the table in one of four distinct states based on how far along it was:

1. A duplicate entry exists and one has a bad CRC,
2. A duplicate entry exists and both have correct CRCs,
3. No duplicate exists but one row has a bad CRC, or

4. No duplicate exists and no rows have a bad CRC.

The last case can occur if a client fails prior to issuing any updates to the table or if it fails after updating all the rows but before releasing the locks. In either case recovery is trivial: a client with the repair lease can simply unlock the stranded lock. Recovery from the other three cases requires modification to the index table.

To repair the table a client first detects in which state the table is and then transitions the table forward through the states with a deterministic sequence of operations so that failures during recovery can be repaired by a subsequent client. A client determines the state by issuing reads to all rows protected by the stranded lock. It then proceeds one-by-one through each entry within the rows, checking both hash locations for the corresponding key (one of which may not be in the locked rows) for duplicates or a bad CRC. Because RCuckoo updates one row per RDMA write, there can be at most one duplicate or bad CRC.

Clients repair the table by moving though the states one step at a time. To move from state 1 to 2, the client writes a new CRC for the bad duplicate. The table can be transitioned from state 2 to 4 by clearing the duplicate entry in the second (i.e., pointed to by $L_2(K)$) location. Finally, the table can be transitioned from state 3 to 4 by recalculating and writing a new CRC to the impacted row. After a client has issued its repair sequence it unlocks the reclaimed lock and returns its lease.

From a correctness perspective, once all rows have a valid CRC and there are no duplicates, the table is usable again. Clearly the new value being inserted into the table by the failed client is lost, but this is indistinguishable from the case that the client failed before attempting the insert. If the client was in the middle of cuckooing values up the path, a subset of the values were moved from their primary cuckoo location to their secondary location, but reads check both locations in any case, so the entry will still be located. Finally, because duplicate entries are freed, no space in the table is lost.

3.3.4 Preventing Stale Writes

The one remaining concern is that a supposed-failed client could just be slow, and may yet attempt to complete its cuckoo path despite the fact that its locks were reclaimed. Our failure timeout is deliberately set many orders of magnitude larger than the expected operation completion time, but we cannot completely rule out the possibility. Ideally, RCuckoo could ensure that clients deemed to have failed by the failure detector are prevented from issuing further operations to the table without manual intervention (e.g. a reboot).

While clients that fail holding a recovery lease are easy to identify, our current implementation has no way to identify which client failed holding a stranded lock. There are two straightforward extensions to provide that functionality: increase the size of lock entries to include the queue pair of the client, or have a separate liveness datastructure that each client must update with some frequency so failed (or unreasonably slow) clients can be identified by their failure to update their entry in a timely manner. Our design supports the latter, but we have not found the need to implement it in our testbed—we have never seen a stale write that was delivered with a delay anywhere close to approaching our timeout value.

Once a failed client is identified, real-world deployments have many ways to ensure the client ceases operation, but it is interesting to consider providing such functionality within RCuckoo itself. Unfortunately, at the time of writing the Infiniband specification does not allow clients to modify each other's RDMA permissions.[4] To the best of our knowledge, the only current alternative is to reset a failed client's queue pair by crafting an invalid packet and sending it to their queue pair at the server [80, Attack 2]. For this attack to work the packet sequence number of the invalid packet must match expected sequence number at the receiver so 2^{24} packets must be sent to ensure the connection is corrupted successfully.

[4] Type-II memory windows enable clients to remove their own permissions using `SEND_WITH_INV`, but not another client's.

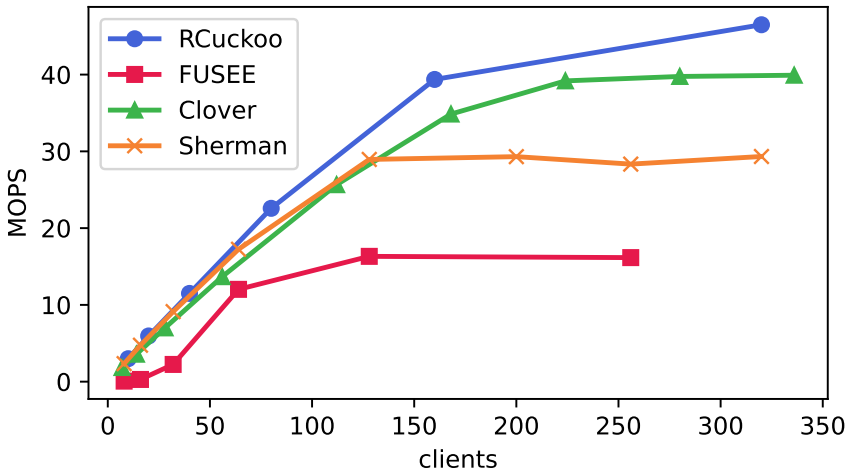

Figure 3.9. Throughput as a function of the number of clients for a read only workload (YCSB-C) (Zipf θ=0.99)

3.4 Evaluation

We evaluate RCuckoo by directly comparing its performance in terms of throughput and latency against representative state-of-the-art disaggregated key/value stores. When accessing large values, aggregate throughput is limited by link rate, so we focus on small key/value pairs where the table management overhead is most significant. When values are small enough to be stored inline, RCuckoo outperforms existing systems while delivering competitive insert latencies. Using fault injection, we show that our distributed approach to client failure detection and recovery enables RCuckoo to sustain high throughput even though 100s of clients are failing per second. Finally, we justify our design decisions through a series of micro-benchmarks. Specifically, we quantify the benefit RCuckoo extracts from its locality enhancement and speculative search strategy before measuring the impact of index-table entry size.

3.4.1 Testbed

We conduct our evaluation on an 9-node cluster of dual-socket Intel machines. Each CPU is an Intel Xeon E5-2650 clocked at 2.20 GHz. Each machine has 256 GB of RAM with 128 GB per NUMA node. All machines have a single dual-socket ConnectX-5 attached to a

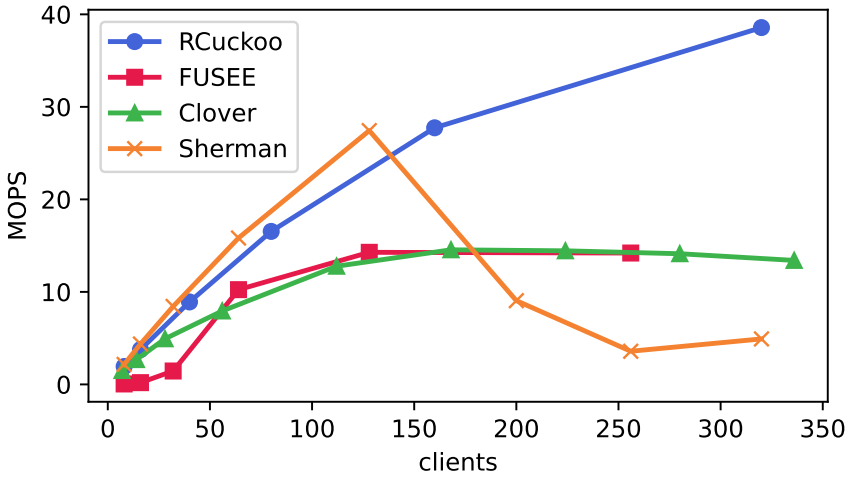

Figure 3.10. Throughput as a function of the number of clients for a read mostly workload (5% write) workload (YCSB-B) (Zipf θ=0.99)

100-Gbps Mellanox Onyx switch. In our RCuckoo experiments we use one sever as the memory server and the rest a client machines spreading threads evenly across machines.

We compare RCuckoo against three recent RDMA key/value stores with different designs, FUSEE [87], Clover [91], and Sherman [93]. While none have the exact same assumptions or feature set as RCuckoo, each represents an apt comparison point for different aspects. To avoid biasing our evaluation, we consider the same workloads (YCSB) as the authors of the previous systems.

FUSEE is a fully disaggregated key/value store that represents the closest available comparison point to RCuckoo. While both employ only 1-sided RDMA operations, FUSEE eschews locking in favor of optimistic insertions. FUSEE clients use CAS operations to manage fixed, 64-bit index table entries that contain pointers to values stored in extents. Due to its reliance on CAS operations, FUSEE is unable to support inlined storage of small values like RCuckoo, forcing all reads to require two round trips. Unlike RCuckoo, FUSEE is designed to support replication. To remove the overhead of replication, we deploy FUSEE with a single memory node.

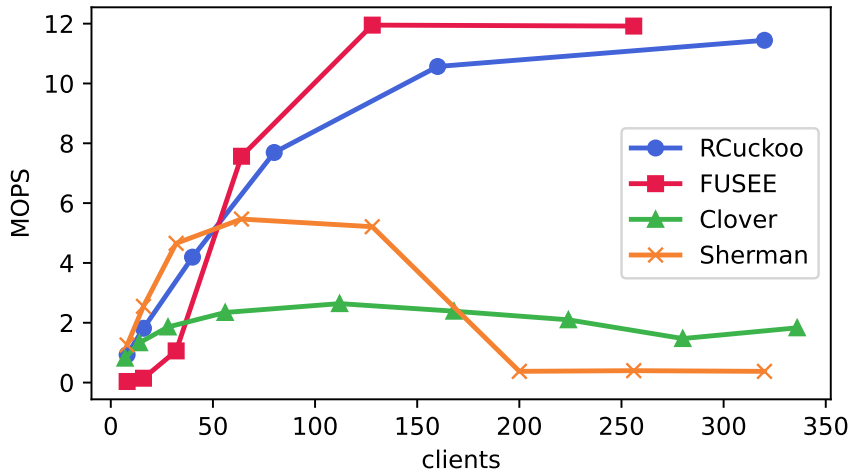

Figure 3.11. Throughput as a function of the number of clients for a write heavy workload (50% writes) (YCSB-A) (Zipf θ=0.99)

Clover is only partially disaggregated—it requires a metadata server to manage its index structure—but can deliver higher read performance than FUSEE on read-only workloads. Clover is designed to leverage remote persistent memory and implements both reads and updates using one-sided RDMA operations. Moreover, unlike FUSEE—and similar to RCuckoo—Clover reads are self verifying. In contrast to prior comparisons [87] that force clients to consult the metadata server on each read, we allow Clover to take advantage of its client caching to achieve maximum performance on read-heavy workloads.

Sherman is the highest-throughput distributed key/value storage system of which we are aware that employs locks. Sherman maintains a B-tree that spans multiple servers and supports range queries, a feature none of the other systems—RCuckoo included—provide. On the other hand, Sherman clusters are not fully disaggregated: each node in a cluster is a peer with many CPU cores and a single memory core that is responsible for servicing allocation RPC calls from clients. As such, Sherman does not encounter the same bandwidth bottlenecks as the other systems because requests are partitioned across machines.

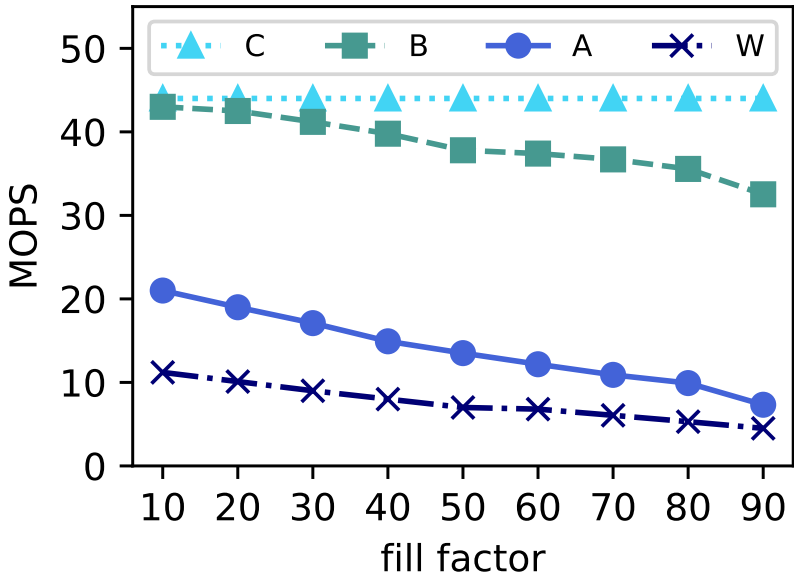

Figure 3.12. RCuckoo performance as a function of fill factor on each YCSB workload. Here updates are replaced with inserts. As the table fills inserts become more difficult thus reducing throughput.

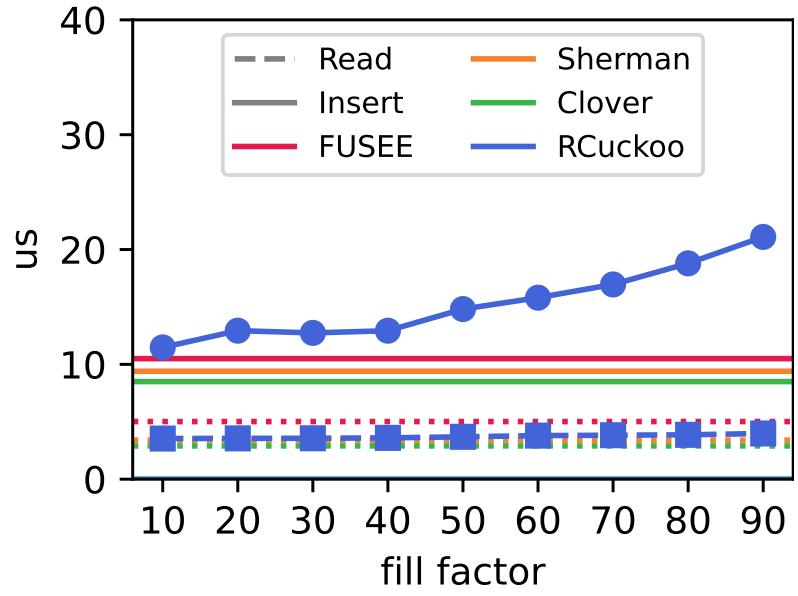

Figure 3.13. Read and insert latency as a function of fill factor. RCuckoo's read latency remains constant. Insert latency is proportional to the fill factor

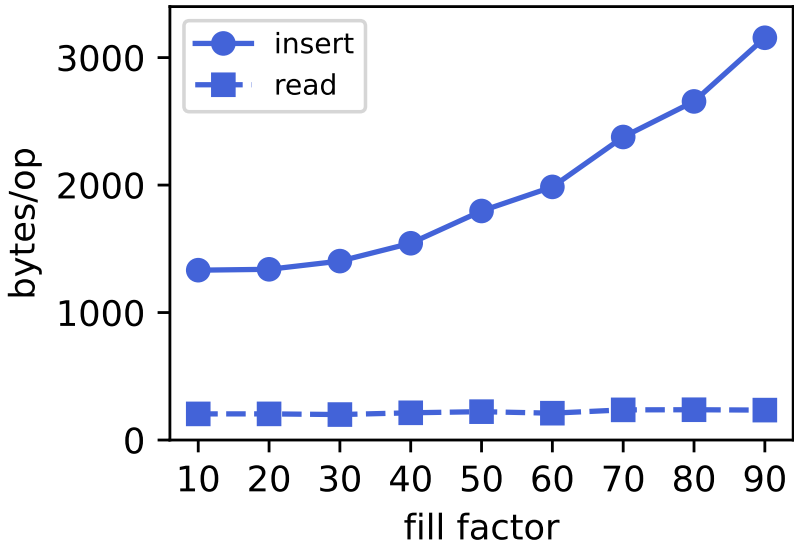

Figure 3.14. Bytes per operation as a function of fill factor. As the table fills inserts consume more bytes per operation due to additional round trips.

3.4.2 Performance

We start by considering throughput and latency on the classical YCSB workloads which employ varying mixes of read and update operations before turning to the more complex insert operation. RCuckoo delivers the highest performance on reads and updates across all settings, while insert performance varies as a function of table fill factor. Even in the worst case, however, RCuckoo limits I/O amplification to around 2×.

Throughput. [3.9, 3.10, 3.11] shows YCSB throughput for RCuckoo, FUSEE, Clover, and Sherman on three different YCSB workloads. For each system, we allocate a 100-M-entry table and pre-populate it with 90 M entries that each consist of a 32-bit key and 32-bit value (we consider larger sizes in Section 3.4.4). We plot the aggregate throughput of a variable number of clients concurrently accessing entries according to a Zipf(0.99) distribution.

In a read-only (YCSB-C) workload, FUSEE suffers from its extent-based value storage. RCuckoo, Clover, and Sherman perform similarly at low-to-moderate levels of concurrency, but they separate at scale. Sherman's read algorithm is more complex than RCuckoo's leading to

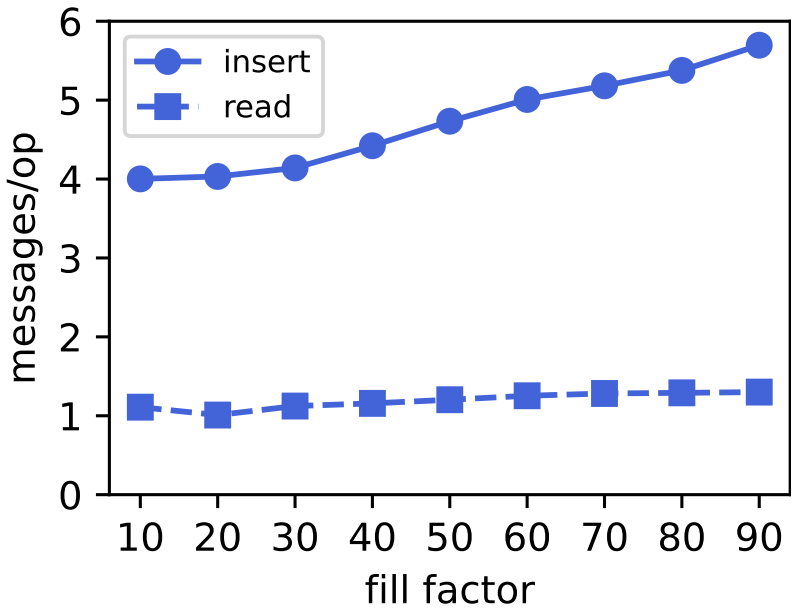

Figure 3.15. Messages per operation as a function of fill factor.

lower top-end performance. Clover's client-side caching shines under this skewed workload, where almost all reads hit in a client's index cache, requiring only a single read for the value; its performance degrades under a more uniform workload (not shown). RCuckoo, on the other hand, reads inlined values in a single round trip regardless of the distribution, leading to the highest performance.

Increased update rate slows all systems. Even with only 5% updates (YCSB-B), the picture changes dramatically. Sherman performs well at low levels of concurrency due to its single-round-trip reads, but hits a severe bottleneck due to lock contention on the skewed access pattern. (Sherman improves—but does not surpass RCuckoo—for uniform workloads, not shown, where lock contention is less of an issue.) Caching is less effective with updates, bringing Clover's throughput in line with FUSEE.

On the 50/50-mixed YCSB-A workload RCuckoo and FUSEE perform similarly, although we are unable to scale FUSEE past 250 clients in our testbed while RCuckoo continues to scale. Sherman begins to suffer from lock contention even earlier, topping out around 5 MOPS before collapsing. Clover performs worst under write-heavy workloads due to its inability to

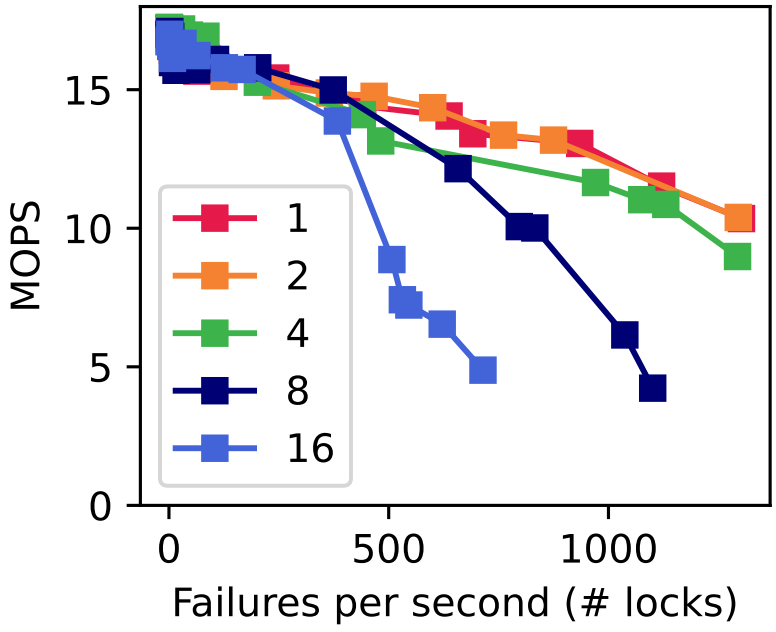

Figure 3.16. YCSB-A throughput vs. client failure rate

effectively leverage caching with a constantly changing index structure.

Inserts. Despite its complexity, RCuckoo's insert operation remains highly performant. To evaluate insert performance we run workloads with a mix of reads and inserts. Figures[3.12, 3.13, 3.14, 3.15] considers RCuckoo's performance on workloads that exclusively use inserts (rather than updates); as with YCSB nomenclature A is 50% insert and 50% read; W is insert only. Inserts become more expensive as the table fills, so we pre-populate the table with a varying number of entries and report insert performance as a function of the table's initial fill factor.

Figure 3.12 shows the aggregate throughput of 320 clients across four different workloads as a function of the table's fill factor. As the index table fills, cuckoo paths become longer leading to increased contention and additional bandwidth consumption from larger covering reads. In each case (except read-only C) RCuckoo's performance declines with fill factor. In the insert-only W case RCuckoo's performance drops from a high of 11.5 MOPS in a nearly empty table to 4.5 MOPS at a 90% fill factor. As a point of comparison, FUSEE's maximum insert-only

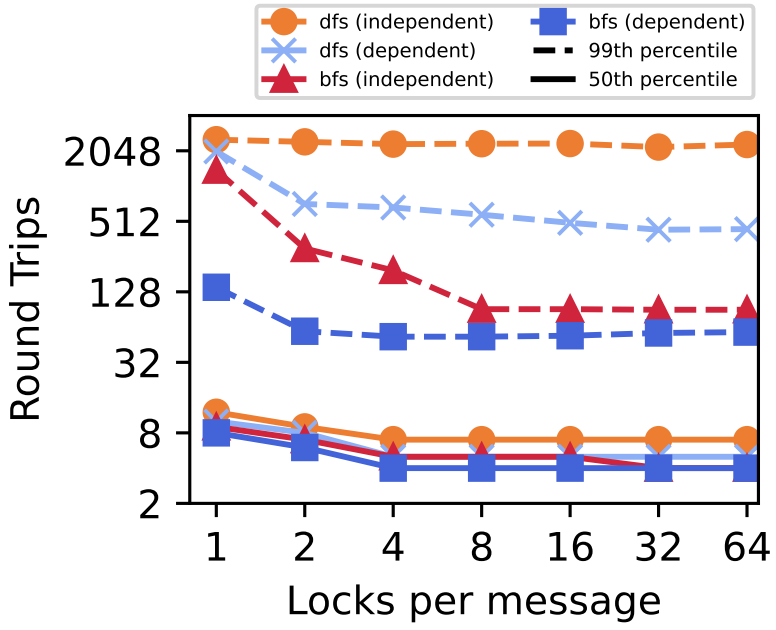

Figure 3.17. Round trip times required to acquire locks on insert

performance is 9.1 MOPS on our testbed, although it is independent of fill factor. While FUSEE out-performs RCuckoo at high fill factors, we observe that insert-only workloads are rare in practice [73].

Latency and overhead. We plot the latency of insert and read operations in Figures[3.13, 3.14, 3.15]. For comparison systems we report the best-case (lightly loaded) performance on our testbed. Read latency is nearly identical for all systems save FUSEE, as it requires an additional round trip. Insert times vary: Clover and Sherman use two-sided RDMA operations for insert and both need to perform allocations and set up metadata for the requesting client. FUSEE is slightly slower, roughly the same as RCuckoo's best case. As the table fills, however, cuckoo paths grow in length causing RCuckoo insert operations to require additional round trips to find valid cuckoo paths. At maximum fill, insert operations take roughly twice as long as in an empty table. This I/O amplification is reflected by the increase in both bytes (Figure 3.14) and messages (Figure 3.15) per operation. The cost and performance of reads, on the other hand, is insensitive to fill factor.

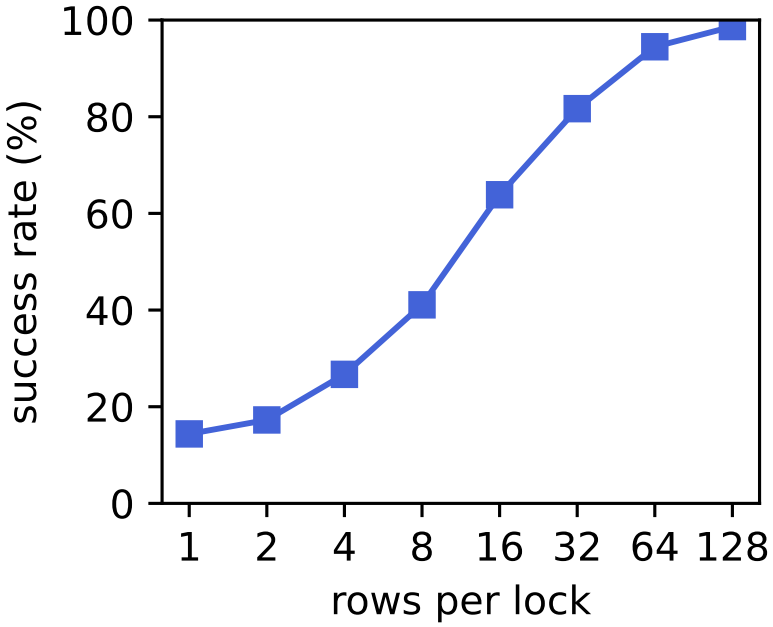

Figure 3.18. Insert second-search success rate as a function of lock granularity

3.4.3 Fault Tolerance Performance

RCuckoo runs at nearly full throughput during realistic failure scenarios and remains functional in the face of hundreds of failures per second. We emulate client failures by performing a partial insert operation that randomly truncates the batch of RDMA operations including lock releases, leaving the table in one of the states listed in Section 3.3.3. Figure 3.16 shows that throughput remains high until about 500 client failures per second, at which point lock granularity begins to play a significant role; finer-grained locks are easier to recover leading to less throughput degradation. As a point of reference, we observe that RDMA itself struggles to handle churn of this magnitude: a server can only establish approximately 1.4 K RDMA connections per second [62].

3.4.4 Microbenchmarks

Having established RCuckoo's superiority over prior systems and demonstrated its robustness to client failure, we now evaluate the impact of particular design choices.

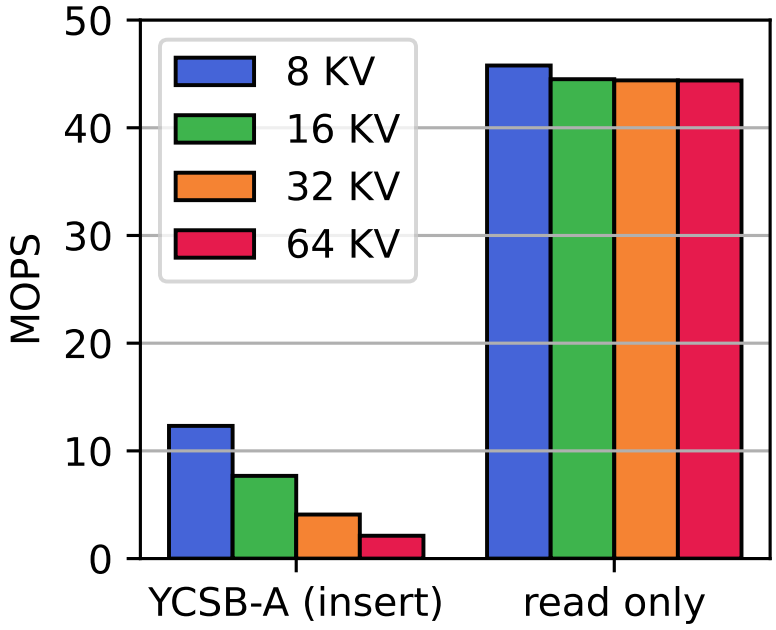

Figure 3.19. Throughput vs. key/value-entry size for YCSB-A (insert) and YCSB-C (read-only) workloads

Locality enhancement. Figure 3.17 illustrates the dramatic benefit RCuckoo extracts from its dependent hashing combined with a BFS cuckoo-path search strategy. To focus on longer cuckoo paths, we pre-populate a 100-M entry table to 85% and then report both the median and 99th percentile round trips per insert key/value pairs until the table is 95% full as a function of lock granularity. While median performance is on the same order, the 99th-percentile insert takes an order of magnitude fewer round trips with dependent hashing and BFS as opposed to independent hashing and DFS as used in prior cuckoo hash systems [56, 67, 75]. As before (c.f. Figure 3.7), performance is similar with four or more locks per message.

Secondary search. We measure second search success rate as a function of lock granularity under high contention. A table is filled up to 85% and then 320 clients concurrently run an insert-only workload. For this experiment only, we flush the client's cache before every insert, ensuring the initial speculative path will fail unless the entry does not require cuckooing (unlikely at this fill factor). Figure 3.18 plots the success rate of the second search as a function

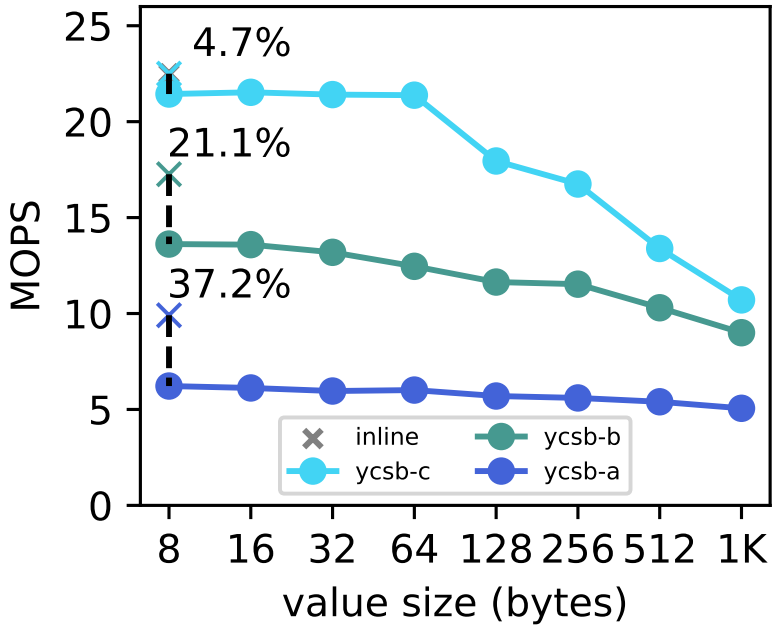

Figure 3.20. Extent performance value sizes up to 1KB. Dashed line marks the inline performance on 16 Byte entries. Overheads marked in black.

of lock granularity. Recall that if a second search fails, RCuckoo clients release their locks and retry the insert operation using the contents of their cache, so multiple speculative and secondary searches may be performed. At one row per lock, secondary search has limited effectiveness (as the only alternative is to cuckoo a different set of entries among the same rows), leading to multiple retries (approaching 4 in the 99th percentile, not shown). As locks cover additional rows, however, the second search becomes much more useful. At 64 rows per lock the second search succeeds 95% of the time.

Entry/value sizes. Inlined key/value entries enable single-round-trip reads. However large entries increase bandwidth consumption for inserts. Figure 3.19 shows the effect of entry size on throughput under 50% insert and read-only (YCSB-C) workloads. Insert is a bandwidth-limited operation, while reads (and update/delete, not shown) are largely unaffected by entry size. Extent entries are slightly slower.

Figure 3.20 shows YCSB throughput as a function of value size from 8 to 1024 B on 6

client machines with 120 cores using a Zipf(0.99) distribution. For comparison, we show the performance for 8-B inlined values on the same testbed and compute the difference. Inlined entries have two sources of performance gain: they avoid the overhead of reading and writing to extents which increases with value size, but, more importantly they avoid additional rounds trips on cache misses. YCSB-B sees a 21% performance improvement from inlining while YCSB-A gains 37% (YCSB-C has misses).

3.5 The Advantage of Locality

The key insight behind RCuckoo is that locality can be used to improve it's performance. Each aspect of it's design relies on it. Without locality spanning reads would span arbitrary ranges and consume unacceptable amounts of bandwidth. Lock acquisition, without locality, would involve random iterative reads throughout the table leading to many round trips and potential deadlock. Without locality the lock table may not easily fit into a linear block of NIC memory. And finally without locality the number of locks required for insertions would increase dramatically as the probability of a single lock spanning multiple relevant entries would significantly decrease. Locality enables us to take advantage of the fact that the network bandwidth is high and RDMA operations on linear regions of memory.

In general RCuckoo is a demonstration of the power of locality in disaggregated systems, but it is far from the final word. While many aspects of it's design are specific to Cuckoo hashing the concept of reducing the number of round trips to perform an operation by increasing the likelihood that all relevant information lies within a given range is general property we expect would provide benefit many data structures.

3.6 Acknowledgement to RCuckoo Contributors

Chapter 3 is a partial reprint of work submitted to multiple USENIX conferences under the title "Cuckoo for Clients: Disaggregated Cuckoo Hashing. Stewart Grant, Alex C. Snoeren.

This dissertations author was the primary investigator and author of this paper.

Thank you to Alex C. Snoeren for your tireless guidance and support throughout this work. Thank you to Dave Andersen for supplying an open source implementation of MemC3 [24], which was used as a reference for the search algorithm for RCuckoo. Thank you to Jiacheng Shen for our conversations at SOSP '23; without your confirmation that acquiring locks for a cuckoo hash *was indeed hard* and that the progress we had made on the problem seemed promising, this work would not have been possible. Again, thank you to Anil Yelam for always providing feedback on the technical aspects of this project. Finally, thank you to Geoffrey M. Voelker for providing feedback on the initial draft of this work.

Chapter 4

Conclusion

This dissertation explores the problems and solutions for sharing disaggregated memory. The high access latency and extreme cost of contention in far-memory over RDMA cause many data structures, which would otherwise be efficient, to experience performance collapse. Stale caches cause opportunistic data structures to fail under contention, and pointer-based data structures incur additional round trips when far-memory pointers need to be resolved. Using existing techniques, applications can be made to work, but they simply have to incur the penalties of sharing when under contention.

We have presented two systems, SwordBox and RCuckoo, that present two different strategies for improving the performance of disaggregated data structures. SwordBox uses a programmable switch as a centralized cache to remove contention from shared data structures and to accelerate the use of locks on lock-based structures. RCuckoo uses locality within a hash table to improve the performance of reads and greatly reduce the cost of acquiring locks stored on a NIC. Together these works provide strong evidence that:

Data structures can be optimized for disaggregated memory by leveraging network programmability

In this chapter we begin by summarizing the contributions of this dissertation and their relationship to our thesis claim, and we conclude with a discussion of this works limitations paired with future work directions in this area of research.

4.1 Contributions

In this dissertation, we have contributed to the state of the art in disaggregated memory systems by:

- Demonstrating the significant performance gains achievable by using a programmable switch to cache the contended state of a data structure and resolve the conflicts in the network.
- Showing that a programmable switch can reduce the instruction-level bottlenecks of RDMA atomic operations.
- Demonstrating that through locality optimizations locks can be fit into a small amount of NIC memory.
- Showing that RDMA-verbs are well suited for locality optimized data structures.

We believe that these contributions are significant stepping stones towards the design of future efficient disaggregated systems. In the case of SwordBox, which demonstrated acceleration on list appends, it could be adapted to other data structures with similar properties, for example, log-structured systems. We believe that the insight behind RCuckoo's locality-based optimizations is general and that many data structures could benefit from localizing their data in a similar fashion.

4.2 Future Work

Each of the works presented in this dissertation is a step towards more efficient disaggregated systems. In this section, we speculate on future directions for this area of research based on the limitations of the work presented here.

Many off-the-shelf ARM-based or FPGA-based SmartNICs could be used to implement SwordBox as well as more complex data structures. While SmartNICs lack a global view

of a rack, they have a global view of the machine they are attached to and could be used to implement similar caching strategies. SmartNICs typically have much more available memory than programmable switches and could likely handle much larger data structures than SwordBox. For example, inserting into a linked list (because the entire linked list needs to be stored in memory) is a *difficult* data structure for SwordBox to handle. This is unlikely to be the same on a SmartNIC with a few GB of memory. SmartNICs could be used to cache index structures for large structures like B-Trees and use their limited compute power to steer read and write requests similar to SwordBox. More generically, they could be used for simple functions that are difficult to implement in remote memory, such as an allocator or scheduler that needs to maintain centralized state.

A further and more generic option for NIC designers is to extend the interface for RDMA to better accommodate complex one-sided data structures. While calls for pointer chasing are common, calls for more complex atomic operations are less so [22, 34]. The algorithms community has designed many wait-free and lockless data structures such as binary trees and heaps that make use of multi-CAS. The ability to CAS multiple addresses simultaneously could open up disaggregation to many pointer-based data structures that would currently be difficult to implement with a single CAS, such as any structure which requires multiple pointers to be updated atomically, like a doubly linked list. Simultaneously CAS and FAA are of limited width. Extensions for larger CAS sizes would enable more efficient data structures, such as in the case of RACE and FUSEE, which must limit the size of their key-value pairs due to the lack of space in the 64-bit RDMA CAS.

Network-data structure co-design is a powerful but underexplored area. In the case of SwordBox, Clover was a good fit for the system as only the tail pointer of linked lists needed to be stored on the switch. Other data structures do not always exhibit this property. Log-based or append-only data structures have promise here, as the point of contention (the end of the log) can be managed with relatively little state. Future work could enable *append-mostly* data structures. Sherman, for instance, enjoys a degree of associativity at its leaf nodes by having associative

leaves. Other data structures may be able to take advantage of similar properties, like appending updates to a shared log and eventually combining them into a consolidated data structure.

Most proposals for disaggregated systems are focused on rack-scale deployments. As intra-rack latencies get lower and lower, and disaggregated technology gets better, intra-rack solutions will become more tenable. An early example is *Disaggregating Stateful Network Functions* [12] and *SuperNIC* [85], which take the stance that a large pool of dedicated accelerators can provide a significant portion of the network functions for a data center. In this line of work, routing is paramount and underexplored. SwordBox assumes a centralized model as it needs to track an entire data structure. However, a future distributed model could potentially scale to multiple racks if the data-dependent operations were routed through the correct network components.

The future for disaggregated systems is bright. This work has demonstrated that shared remote memory can be made efficient with programmable networking hardware and that, through careful design, data structures can be adapted to disaggregated memory. Hopefully, future work will build on these techniques and enable a shift towards mainstream disaggregated computing.

Bibliography

[1] NVIDIA BlueField-2 DPU. https://www.nvidia.com/en-us/networking/products/bluefield-2/, July 2020.

[2] Advanced transport (NVIDIA RDMA documentation). https://docs.nvidia.com/networking/display/MLNXOFEDv494170/Advanced+Transport, August 2022.

[3] RoCE vs. iWARP competitive analysis. https://network.nvidia.com/pdf/whitepapers/WP_RoCE_vs_iWARP.pdf, June 2023.

[4] Amazon EC2 instances. https://aws.amazon.com/ec2/instance-types/, April 2024.

[5] AWS Nitro system. https://aws.amazon.com/ec2/nitro/, July 2024.

[6] Device memory programming (NVIDIA RDMA documentation). https://docs.nvidia.com/networking/display/ OFEDv502180/Programming#Programming-DeviceMemoryProgramming, January 2024.

[7] Nvidia quantum-x800 infiniband platform. https://nvdam.widen.net/s/hbp8zz7fvt/solution-overview-gtcspring24-quantum-x800-3175164, March 2024.

[8] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novaković, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 775–787, Boston, MA, July 2018. USENIX Association.

[9] 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, New York, NY, USA, 2020. Association for Computing Machinery.

[10] Sebastian Angel, Mihir Nanavati, and Siddhartha Sen. Disaggregation and the application. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, held virtually online, July 2020. USENIX Association.

[11] Hari Balakrishnan, Srinivasan Seshan, Elan Amir, and Randy H. Katz. Improving TCP/IP performance over wireless networks. In *Proceedings of the 1st Annual International Conference on Mobile Computing and Networking*, MobiCom '95, page 2–11, New York, NY, USA, 1995. Association for Computing Machinery.

[12] 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.

[13] Rutger Beltman, Silke Knossen, Joseph Hill, and Paola Grosso. Using P4 and RDMA to collect telemetry data. *2020 IEEE/ACM Innovating the Network for Data-Intensive Science (INDIS)*, pages 1–9, Nov 2020.

[14] 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), page 79–92, New York, NY, USA, April 2021. Association for Computing Machinery.

[15] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. Black-box concurrent data structures for NUMA architectures. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, (ASPLOS '17), page 207–221, New York, NY, USA, April 2017. Association for Computing Machinery.

[16] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, modeling, and benchmarking RocksDB Key-Value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, Santa Clara, CA, February 2020. USENIX Association.

[17] Youmin Chen, Youyou Lu, and Jiwu Shu. Scalable RDMA RPC on reliable connection with efficient resource sharing. In *Proceedings of the Fourteenth EuroSys Conference 2019*, (EuroSys '19), Dresden, Germany, 2019. Association for Computing Machinery.

[18] Yann Collet. xxhash extremely fast hash algorithm, Dec 2023.

[19] P4 Language Consortium. P4 language specification. https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.html, Feb 2017.

[20] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, (SoCC '10), page 143–154, Indianapolis, Indiana, USA, June 2010. Association for Computing Machinery.

[21] CXL Consortium. CXL 3.0 specification. https://www.computeexpresslink.org/download-the-specification.

[22] Sowmya Dharanipragada, Shannon Joyner, Matthew Burke, Adriana Szekeres, Jacob Nelson, Irene Zhang, and Dan R. K. Ports. Prism: Rethinking the RDMA interface for distributed systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*, Virtual Event, Germany, October 2021. Association for Computing Machinery.

[23] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, April 2014. USENIX Association.

[24] Bin Fan, David G. Andersen, and Michael Kaminsky. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 371–384, Lombard, IL, April 2013. USENIX Association.

[25] Paolo Faraboschi, Kimberly Keeton, Tim Marsland, and Dejan Milojicic. Beyond processor-centric operating systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, March 2015. USENIX Association.

[26] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, Renton, WA, April 2018. USENIX Association.

[27] Linux Foundation. Data plane development kit (DPDK). http://www.dpdk.org, June 2024.

[28] 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)*, pages 249–264, Savannah, GA, November 2016. USENIX Association.

[29] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. Direct access, High-Performance memory disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (ATC '22)*, pages 287–294, Carlsbad, CA, July 2022. USENIX Association.

[30] 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, (SIGCOMM '20), page 681–693, Virtual Event, USA, 2020. Association for Computing Machinery.

[31] Jim Griner, John Border, Markku Kojo, Zach D. Shelby, and Gabriel Montenegro. Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations. RFC 3135, June 2001.

[32] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient memory disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 649–667, Boston, MA, March 2017. USENIX Association.

[33] Zhiyuan Guo, Zijian He, and Yiying Zhang. Mira: A program-behavior-guided far memory system. In *Proceedings of the 29th Symposium on Operating Systems Principles*, (SOSP '23), page 692–708, Koblenz, Germany, October 2023. Association for Computing Machinery.

[34] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: A hardware-software co-designed disaggregated memory system. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, (ASPLOS '22), page 417–433, Lausanne, Switzerland, Feburary 2022. Association for Computing Machinery.

[35] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, (SPAA '10), page 355–364, Thira, Santorini, Greece, June 2010. Association for Computing Machinery.

[36] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch hashing. In *Proceedings of the 22nd International Symposium on Distributed Computing*, (DISC '08), page 350–364, Berlin, Heidelberg, September 2008. Springer-Verlag.

[37] Rulin Huang, Kaixin Huang, Jingyu Wang, and Yuting Chen. Reno: An RDMA-enabled, non-volatile memory-optimized key-value store. In *2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS '21)*, pages 466–473, December 2021.

[38] Infiniband Trade Association. Infiniband specification. https://www.afs.enea.it/asantoro/, December 2007.

[39] Intel. Intel rack scale architecture: Faster service delivery and lower TCO. https://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html, July 2018.

[40] Intel. Intel Tofino 2 P4 programmability with more bandwidth. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series/tofino-2.html, August 2020.

[41] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-Free Sub-RTT coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 35–49, Renton, WA, April 2018. USENIX Association.

[42] 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, page 121–136, New York, NY, USA, 2017. Association for Computing Machinery.

[43] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, and Jonathan Ross. In-datacenter performance analysis of a tensor processing unit. In *Arxiv*, April 2017.

[44] 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)*, pages 1–16, Boston, MA, February 2019. USENIX Association.

[45] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA efficiently for key-value services. *SIGCOMM Comput. Commun. Rev.*, 44(4):295–306, August 2014.

[46] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 437–450, Denver, CO, June 2016. USENIX Association.

[47] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 185–201, Savannah, GA, November 2016. USENIX Association.

[48] Sagar Karandikar, Albert Ou, Alon Amid, Howard Mao, Randy Katz, Borivoje Nikolić, and Krste Asanović. FirePerf: FPGA-accelerated full-system hardware/software performance profiling and co-design. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, (ASPLOS '20), page 715–731, Lausanne, Switzerland, March 2020. Association for Computing Machinery.

[49] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. TEA: Enabling state-intensive network functions on programmable switches. 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), page 90–106, Virtual Event, USA, September 2020. Association for Computing Machinery.

[50] 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*, (SOSP '21), page 488–504, Virtual Event, Germany, October 2021. Association for Computing Machinery.

[51] Youngmoon Lee, Hasan Al Maruf, Mosharaf Chowdhury, Asaf Cidon, and Kang G. Shin. Hydra : Resilient and highly available remote memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 181–198, Santa Clara, CA, February 2022. USENIX Association.

[52] Huaicheng Li, Daniel S. Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, 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 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '23)*. Association for Computing Machinery, March 2023.

[53] Jialin Li, Ellis Michael, and Dan R. K. Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proceedings of the 26th Symposium on Operating Systems Principles*, (SOSP '17), page 104–120, Shanghai, China, October 2017. Association for Computing Machinery.

[54] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just say NO to Paxos overhead: Replacing consensus with network ordering. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, pages 467–483, Savannah, GA, November 2016. USENIX Association.

[55] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. ROLEX: A scalable RDMA-oriented learned Key-Value store for disaggregated memory systems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 99–114, Santa Clara, CA, February 2023. USENIX Association.

[56] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems*, (EuroSys '14), Amsterdam, The Netherlands, April 2014. Association for Computing Machinery.

[57] 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), page 44–58, Beijing, China, August 2019. Association for Computing Machinery.

[58] 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), page 267–278, Austin, TX, USA, June 2009. Association for Computing Machinery.

[59] Hugo Lin. Impacts of Intel's decision to stop Tofino development on P4. https://forum.p4.org/t/impacts-of-intels-decision-to-stop-tofino-development-on-p4/669/2, March 2023.

[60] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the ACM Special Interest Group on Data Communication*, (SIGCOMM '19), page 318–333, Beijing, China, August 2019. Association for Computing Machinery.

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

[62] Teng Ma, Tao Ma, Zhuo Song, Jingxuan Li, Huaixin Chang, Kang Chen, Hai Jiang, and Yongwei Wu. X-rdma: Effective RDMA middleware in large-scale production environments. In *2019 IEEE International Conference on Cluster Computing (CLUSTER ' 19)*, pages 1–12, September 2019.

[63] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Mike Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Mike Ryan, Erik Rubow, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: a microkernel approach to host networking. In *In ACM SIGOPS 27th Symposium on Operating Systems Principles (SOSP '19)*, Huntsville, Ontario, Canada, 2019. Association for Computing Machinery.

[64] Hasan Al Maruf and Mosharaf Chowdhury. Effectively prefetching remote memory with Leap. In *2020 USENIX Annual Technical Conference (ATC '20)*, pages 843–857. USENIX Association, July 2020.

[65] Memcached. Memcached: a Distributed Memory Object Caching System. http://www.memcached.org/, March 2024.

[66] Micron. Micron's perspective on impact of CXL on DRAM bit growth rate. https://www.micron.com/content/dam/micron/global/public/products/white-paper/cxl-impact-dram-bit-growth-white-paper.pdf, August 2023.

[67] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *2013 USENIX Annual Technical Conference (ATC '13)*, pages 103–114, San Jose, CA, June 2013. USENIX Association.

[68] Christopher Mitchell, Kate Montgomery, Lamont Nelson, Siddhartha Sen, and Jinyang Li. Balancing CPU and network in the Cell distributed B-Tree store. In *2016 USENIX Annual Technical Conference (ATC '16)*, pages 451–464, Denver, CO, June 2016. USENIX Association.

[69] Sumit Kumar Monga, Sanidhya Kashyap, and Changwoo Min. Birds of a feather flock together: Scaling RDMA RPCs with flock. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, (SOSP '21), page 212–227, Virtual Event, Germany, October 2021. Association for Computing Machinery.

[70] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A Read/Write Peer-to-Peer file system. In *5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, pages 911–929, Boston, MA, December 2002. USENIX Association.

[71] Mihir Nanavati, Jake Wires, and Andrew Warfield. Decibel: Isolation and sharing in disaggregated rack-scale storage. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 17–33, Boston, MA, March 2017. USENIX Association.

[72] 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), page 327–341, Budapest, Hungary, August 2018. Association for Computing Machinery.

[73] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, April 2013. USENIX Association.

[74] 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), page 97–108, Haifa, Israel, June 2019. Association for Computing Machinery.

[75] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, April 2004.

[76] Pensando. Pensando infrastructure accelerators, 2020.

[77] 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, Carlsbad, CA, October 2018. USENIX Association.

[78] Dan R. K. Ports and Jacob Nelson. When should the network be the computer? In *Proceedings of the Workshop on Hot Topics in Operating Systems*, (HotOS '19), page 209–215, Bertinoro, Italy, May 2019. Association for Computing Machinery.

[79] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, Eric Peterson, Aaron Smith, Jason Thong, Phillip Yi Xiao, Doug Burger, Jim Larus, Gopi Prashanth Gopal, and Simon Pope. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*, pages 13–24. IEEE Press, June 2014.

[80] Benjamin Rothenberger, Konstantin Taranov, Adrian Perrig, and Torsten Hoefler. ReDMArk: Bypassing RDMA security mechanisms. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 4277–4292. USENIX Association, August 2021.

[81] 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)*, pages 315–332. USENIX Association, November 2020.

[82] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with In-Network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 785–808. USENIX Association, April 2021.

[83] Richard Schaller. Moore's law: past, present and future. *IEEE Spectrum*, 34(6):52–59, 1997.

[84] 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)*, pages 69–87, Carlsbad, CA, October 2018. USENIX Association.

[85] Yizhou Shan, Will Lin, Ryan Kosta, Arvind Krishnamurthy, and Yiying Zhang. Disaggregating and consolidating network functionalities. https://arxiv.org/abs/2109.07744, September 2021.

[86] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Yuxin Su, Jiazhen Gu, Hao Feng, Yangfan Zhou, and Michael R. Lyu. Ditto: An elastic and adaptive memory-disaggregated

caching system. In *Proceedings of the 29th Symposium on Operating Systems Principles*, (SOSP '23), page 675–691, Koblenz, Germany, October 2023. Association for Computing Machinery.

[87] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R. Lyu. FUSEE: A fully Memory-Disaggregated Key-Value store. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 81–98, Santa Clara, CA, February 2023. USENIX Association.

[88] 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), page 708–721, Virtual Event, USA, 2020. Association for Computing Machinery.

[89] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. Demystifying CXL memory with genuine cxl-ready systems and devices. In *56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23. ACM, October 2023.

[90] Brian R. Tauro, Brian Suchy, Simone Campanoni, Peter Dinda, and Kyle C. Hale. TrackFM: Far-out compiler support for a far memory world. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, (ASPLOS '24), page 401–419, La Jolla, CA, USA,, April 2024. Association for Computing Machinery.

[91] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *(ATC '20)*, pages 33–48, July 2020.

[92] 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)*, page 306–324, Shanghai, China, October 2017. Association for Computing Machinery.

[93] Qing Wang, Youyou Lu, and Jiwu Shu. Sherman: A write-optimized distributed B+Tree index on disaggregated memory. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD ' 22)*, page 1033–1048, Philadelphia, PA, June 2022. Association for Computing Machinery.

[94] Xizheng Wang, Guo Chen, Xijin Yin, Huichen Dai, Bojie Li, Binzhang Fu, and Kun Tan. StaR: Breaking the scalability limit for RDMA. In *Proceedings of the 29th IEEE International Conference on Network Protocols (ICNP '21)*, November 2021.

[95] Jiarong Xing, Kuo-Feng Hsu, Yiming Qiu, Ziyang Yang, Hongyi Liu, and Ang Chen. Bedrock: Programmable network support for secure RDMA systems. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2585–2600, Boston, MA, August 2022. USENIX Association.

[96] Jian Yang, Joseph Izraelevitz, and Steven Swanson. FileMR: Rethinking RDMA networking for scalable persistent memory. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*, pages 111–125, Santa Clara, CA, Feburary 2020. USENIX Association.

[97] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. NetLock: Fast, centralized lock management using programmable switches. In *Proceedings of the 2020 ACM SIGCOMM 2020 Conference*, (SIGCOMM '20), page 126–138, Virtual Event, USA, August 2020. Association for Computing Machinery.

[98] Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu. FORD: Fast one-sided RDMA-based distributed transactions for disaggregated persistent memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 51–68, Santa Clara, CA, February 2022. USENIX Association.

[99] Wei Zhang, Timothy Wood, and Jinho Hwang. NetKV: Scalable, self-managing, load balancing as a network function. In *2016 IEEE International Conference on Autonomic Computing (ICAC '16)*, pages 5–14, January 2016.

[100] Yang Zhou, Hassan M. G. Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E. Culler, Henry M. Levy, and Amin Vahdat. Carbink: Fault-Tolerant far memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22)*, pages 55–71, Carlsbad, CA, July 2022. USENIX Association.

[101] 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. *SIGCOMM Comput. Commun. Rev.*, 45(4):523–536, August 2015.

[102] 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 (ATC '21)*, pages 15–29. USENIX Association, July 2021.