Systems for Using Far Memory in Datacenters

Emmanuel Amaro

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2021-268
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-268.html

December 23, 2021

Systems for Using Far Memory in Datacenters

by

Emmanuel Amaro Ramirez

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Scott Shenker, Chair
Professor Sylvia Ratnasamy
Professor Aurojit Panda
Professor Ion Stoica

Fall 2021

Systems for Using Far Memory in Datacenters

Abstract

Systems for Using Far Memory in Datacenters

by

Emmanuel Amaro Ramirez

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Scott Shenker, Chair

Datacenter efficiency has become increasingly relevant, as the end of Moore's Law and Dennard scaling have caused CPU and memory performance to begin plateauing. Resource disaggregation is a recent datacenter design point, where server nodes share remote resources through a fast (usually RDMA-based) network, enabling greater execution flexibility and performance in datacenters. Remote or far memory–an instance of resource disaggregation–increases flexibility because nodes can access more memory than locally available. And performance in distributed applications can improve as RDMA provides high-performance access to shared state. This dissertation describes two networked systems that allow server nodes in a data center to leverage far memory.

First, WICkit is a framework and runtime for Where-Independent Code. WICs are a location-independent abstraction representing complex remote memory accesses, *e.g.* accessing a value in a hashmap. Without code changes, the WICkit runtime can execute WICs at the client, server, and SmartNIC CPU locations. As different locations provide different performance and resource tradeoffs, WICkit allows users to flexibly choose the location when execution begins while obtaining comparable performance to location-specific systems.

Second, Cluster Far Memory is a system that transparently allows existing jobs to access far memory. CFM includes a fast swapping mechanism and a far memory-aware job scheduler that enable far memory support at rack scale. Using CFM for memory-intensive workloads, a rack can improve its throughput on the order of 10% or more without increasing the total amount of memory in it.

To my family:
My Parents, Ariel, Abril and Alonso.

Contents

Acknowledgments

I feel lucky and proud of having been a member of the NetSys lab at Berkeley. My Ph.D. has been as collaborative as one could imagine, and I feel grateful to have learned from many wise teachers in the process, but I want to recognize four people in particular:

Scott Shenker. More than an advisor, he was my mentor. I vividly remember one interaction with him that has influenced me beyond research: I had just joined NetSys as a second-year graduate student, and I felt anxious by my lack of progress. He noticed and asked me to talk to him much more often, even for a few minutes, not necessarily about research, but anything I felt talking about. Spending time with him talking about research and various aspects of life helped me overcome my anxiety and enjoy the Ph.D. process. Thank you, Scott, for the vast amounts of time you dedicated to me, for your patience, your support and candor.

Aurojit Panda. He was my unofficial co-advisor. I want to thank him for believing in me, listening to my messy ideas, and helping me shape them. His patience and feedback allowed me to believe in myself. During my first few years, when I still struggled to discuss research during meetings (partially due to the language barrier), somehow, he usually understood what I meant and would translate my ramblings into something coherent. Thank you, Panda, for advising me and being a true friend.

Sylvia Ratnasamy. Although she was not my advisor, she made time for me every time I asked her to chat. I want to thank her for listening to me and giving me advice when I needed it the most. She was the person I thought I needed to talk to whenever I felt lost in research. I hope to one day emulate her capacity of asking the fundamental questions that help find clarity in everyone's messy research.

Amy Ousterhout. Thank you for all the feedback you gave me whenever I asked; whether I was giving a talk or attempting to write a paper, you were always willing to provide me with strong concise feedback. Thank you for showing me the importance of distilling research ideas into their fundamental pieces.

Aisha Mushtaq, Amin Tootoonchian, Anwar Hithnawi, Christopher Branner-Augmon, Lloyd Brown, Michael Chang, Murphy McCauley, Narek Galstyan, Peter Gao, Sarah McClure, Silvery Fu, Vivian Fang, Wen Zhang, Yotam Harchol, Zhihong Luo. The friends and collaborators I met at NetSys have each taught me something unique I will always keep with me. Thank you all for the conversations, the laughs, the lunches, the culture, the arguments about which food is best, and everything in between.

Last but definitely not least, I would not be finishing my Ph.D. without the support from my family. Thank you, **Abril**, my partner, for living my Ph.D. with me through the ups and downs. Thank you, **Dad** and **Mom** and **Ariel**, my brother, for all your understanding and always having my back.

Chapter 1

Introduction

Although Dennard scaling ended by 2010 and Moore's law continues to slow down, there is an ever-increasing demand for more storage and computing power in large scale clusters. Predictions say the amount of data created from 2020 to 2023 will be more than the data created over the previous 30 years. Similarly, new workloads that did not exist a decade ago, such as ML training, have impressively grown in popularity and the compute resources they require. For example, a cloud company has claimed their training requirements for their largest models grew an order of magnitude each year from 2012 to 2018 [65].

Datacenters and their users have reacted to these technological trends by breaking monolithic applications into distributed ones across server nodes. In turn, this has exacerbated the challenge of data center resource efficiency. In 2019, Google reported an average resource utilization of less than 70% for CPU and memory in their datacenters [117]. Given current technological trends, maximizing the utilization of available resources would seem ideal, so why is improving utilization challenging? The reason is twofold: first, because compute jobs are bin-packed onto the rack's physically available resources; thus, if any of the resources are lacking, the server node cannot admit the job. Second, because job schedulers usually allocate resources for a job's maximum predicted usage; hence, whenever a job uses fewer resources than its maximum, utilization decreases.

Resource disaggregation is a recent datacenter design point where server nodes share access to remote resources through a fast (usually RDMA-based) network [52, 57, 111]. Remote or far memory–an instance of resource disaggregation–enables greater execution flexibility and performance in datacenters. Flexibility increases because far memory allows server nodes to use more memory than locally available when required. At the same time, distributed applications' performance can improve as RDMA provides high-performance access to shared state.

In this dissertation, we address the question of, how do we build systems that use far memory in datacenters? We answer this question from two different perspectives; first, by looking at how new distributed applications can best use remote memory to share state; and second, by exploring mechanisms that allow existing unmodified applications to execute efficiently in them. We expand on these two perspectives next.

1.1 Where-Independent Code for New Distributed Applications

Many datacenter applications distribute both state and processing logic across multiple servers. Consequently, a client executing processing logic for a distributed application often needs to access and alter the state on a remote server. In many cases, accessing this state requires executing complex memory-access logic, *e.g.*, accessing a value in a hashmap. RDMA networks allow memory-access logic to be executed either at the client or the server. The emergence of SmartNICs means that memory-access logic can also be executed on the server's SmartNIC. However, it is not easy to determine which of these locations to use because they provide different performance and cost trade-offs. Worse, these trade-offs can vary over time, and as a result, no single location is always best for an application.

In Chapter 3, we argue that users should be able to change memory-access logic execution locations without changing the code. To this end, Chapter 3 proposes a new Where-Independent Code (WIC) abstraction that allows users to decide where an application's memory-access logic executes flexibly. We implement the WIC abstraction in a system called WICkit and demonstrate that WICkit applications have performance comparable to applications whose memory-access logic executes in fixed locations.

1.2 Rack-scale Far Memory for Existing Applications

Chapter 4 presents Cluster Far Memory (CFM), a faster swapping mechanism, and a far-memory aware cluster scheduler that enables using far memory at rack scale. The chapter examines the conditions under which using far memory with CFM can increase job throughput. We find that, while far memory is not a panacea, for memory-intensive workloads it can provide performance improvements on the order of 10% or more even without changing the total amount of memory available.

1.3 Previously Published Work

In this dissertation, the material in Chapter 3 is based on [5] but the content presented here significantly extends the programming model and the runtime design. The material in Chapter 4 is an adaptation from [4].

Chapter 2

Background

In this chapter we provide a brief overview of RDMA and SmartNICs.

2.1 RDMA

Remote-direct memory access (RDMA) is a set of network technologies where most of the network stack is implemented in the network interface card (NIC) or within the network. As a result, in RDMA networks, no CPU cycles are spent on providing reliable message delivery, implementing congestion control, or packetizing data, thus reducing network I/O overheads. Over the last decade, RDMA networks have seen wide adoption and use in datacenters [96, 100, 131]. A variety of applications, including key-value stores [40, 80, 83, 98], machine learning [129], and graph processing [112], have been shown to benefit from the use of RDMA.

To benefit from RDMA, applications need to use an specific network API, *e.g.* the IB Verbs API [95]. The logic for many of the functions exposed by these RDMA-specific network APIs, *e.g.* functions that send or receive data, are implemented entirely in the NIC hardware. This allows greater flexibility when designing networked applications. RDMA applications can be structured either as traditional client-server applications or as *one-sided applications*, where application logic is implemented entirely at the client.

Concretely, RDMA provides two types of operations: 1-sided and 2-sided operations. Client-server RDMA applications are implemented using **2-sided operations** and they require processes to be run on both at the client and the server. The client and server processes communicate with each other by sending and receiving messages using 2-sided operations. On the other hand, application logic in one-sided RDMA applications is implemented entirely at the client. The client process uses **1-sided RDMA operations** to read or modify data stored in the server's memory. These operations are executed by the server's RDMA NIC, and as a result, the NIC hardware dictates what operations can be performed on remote memory. Consequently, as prior work [67, 71] has observed, implementing complex application logic in a one-sided application might require several network round-trips.

When deciding whether to structure an application as a client-server application or a one-sided

application, a developer needs to navigate a resource and performance trade-off: client-server applications require more CPU cores but fewer network round-trips while one-sided applications require fewer CPU cores but more network round-trips. Furthermore, how an application should be structured depends not on the functionality that it provides but on the algorithms and data structures used, workloads considered, and where it is deployed. For example, several prior projects have described transaction key-value stores built using RDMA. Some of these projects are structured as one-sided applications (*e.g.* early versions of FaRM [40] and Pilaf [98]), some are structured as client-server applications (*e.g.* eRPC [67] and FaSST [71]), while yet others are hybrid applications combining both options (*e.g.* Dr. TM+H [125] and newer versions of FaRM [41]). In each case these systems are designed to provide high performance, and the core difference lies in whether they use existing off-the shelf datastructures and algorithms [91] or develop specialized RDMA specific algorithms.

2.2 SmartNICs

SmartNICs are NICs that contain programmable processing elements, usually FPGAs or embedded CPU cores. These processing elements are commonly used to implement virtual switches [47] or other systems responsible for enforcing network policies. SmartNICs have been adopted by most cloud providers [6,47,127] because they reduce the isolation overheads in multi-tenant datacenters. Chapter 3, assumes the use of CPU-based SmartNICs.

Chapter 3

WICkit

In this chapter we present WICkit, a framework and runtime for Where-Independent Code. WICs are a location-independent abstraction to represent complex remote memory accesses; *e.g.* accessing a value in a hashmap. WICs are flexible, as they can be executed at client (similarly to 1-sided RDMA operations), at the server (similarly to 2-sided RDMA operations), and at new locations such as SmartNICs. Our design makes the cost of the WIC abstraction low, as client and server locations provide performance comparable to previous location-specific systems.

3.1 Where-Independent Code for Remote Memory Access

Most datacenter applications, including transactional data stores, in-memory storage, and machine learning systems, are distributed applications whose state and processing logic spans multiple servers [40, 82, 126]. Many operations in these applications require accessing state on remote servers, *e.g.* a transactional datastore [40, 123] might need to acquire locks and read and update tuples on multiple servers. In many cases these remote memory accesses are not just simple loads and stores, and as a result these accesses can require executing complex application-dependent *memory-access logic*.

Today this memory-access logic can run in many different locations. Traditional RPC based applications send messages and execute memory-access logic on a remote CPU core [67]. On the other hand, applications built using 1-sided RDMA, can execute memory-access logic at a local CPU using memory-access primitives provided by the RDMA NIC.

Which location is best depends on the application and workload. Previous research has shown that for some applications [40, 71, 123], running memory-access logic on client CPUs and using 1-sided operations can improve resource efficiency and performance. The resource efficiency gains are from not using CPU cores on the server, while the performance gains are because there is no notification delay (*e.g.* via interrupts) or processing cycles spent at the server. On the other hand, other work [40, 71] has shown that using 1-sided RDMA to implement memory-access logic that performs multiple dependent memory accesses yields suboptimal latency because each memory access requires a network round-trip. As a result, the choice of where memory-access logic should

be executed depends on the complexity of the memory-access logic: client-based implementations perform best when the memory-access logic accesses a single memory location, and server based implementations are better otherwise. Where memory-access logic is implemented has an impact not just on performance but also on deployment costs. However, when implementing complex memory-access logic a 1-sided application can require more network bandwidth. Thus, the relative cost of client-based memory access-logic and server-based memory-access logic varies depending on the relative cost of CPU cores and network bandwidth. Thus, the location that provides the best performance and lowest cost varies depending on workload, application logic, and deployment environment. Indeed, prior work [125], has shown the benefit of carefully choosing between these options for different operations.

Modern datacenters also include SmartNICs and other locations where memory-access logic can be executed. These new locations offer different performance and cost trade-offs. While SmartNIC cores generally have lower clock frequencies than host cores, a SmartNIC located on the server can access server memory without needing to traverse the network. Therefore, in many cases memory-access logic executed on the SmartNIC can outperform the same logic when executed by the client. Similarly, most cloud providers [47] do not sell SmartNIC cores, while they do offer host CPU cores to tenants. As a result, providers such as Azure have previously stated that running compute on SmartNIC cores is cheaper than running compute on host cores, and consequently executing memory-access logic on SmartNIC cores is currently cheaper than executing on server CPU cores.

We can thus observe that the best location for memory-access logic varies depending on the application, workload, and deployment environment (which dictates resource costs). Furthermore, all of these factors change over time: application logic and workload vary as a result of changes in user demands, while resource costs vary depending on what other applications are executing in the datacenter. As a result, in this chapter, we argue that it is better to determine where memory-access logic is executed at *application start-up time* rather than when designing an application. Additionally, users should be able to choose the location from a wide array of options, including client CPUs, server CPUs, and even SmartNIC CPUs.

Unfortunately, current RDMA network APIs [95] and frameworks [67, 71] require applications to choose where memory-access logic is executed at development time rather than at start-up time. Building a framework that allows this presents two challenges: First, how do we represent memory-access logic in a location-independent manner? Second, how do we provide reasonable throughput and latency comparable to existing location-specific approaches?

We address the first challenge, by proposing *Where-Independent Code* (WIC), a location-independent abstraction that encapsulates remote memory accesses and their surrounding logic. WICs access memory using a unified API, and a *WIC runtime* adapts the underlying memory-access mechanism, leveraging 1-sided RDMA or local memory accesses, depending on where the WIC executes. Thus, our runtime allows WICs to be written once and then executed in any of the supported locations: client, server, or SmartNIC CPUs.

We address the second challenge, *i*.e., providing latency and throughput comparable to location-specific approaches by designing a new runtime. This runtime builds on the observation that WIC execution time is dominated by the cost of memory access and that these access costs vary de-

pending on where the WIC is run. For example, accessing memory from the server CPU takes about 100*ns*, while accessing the same memory location from the client can take between 1.5-10μs. Memory access latencies affect how WICs access memory: to maximize throughput WICs running on the server should access memory synchronously, while those running at the client should access memory asynchronously. Therefore, the runtime needs to support both synchronous and asynchronous memory accesses, and must be able to switch between WICs performing asynchronous memory accesses. As a result, the WIC runtime needs to support both asynchronous and synchronous memory accesses and rapidly switch between different WICs when performing asynchronous memory access.

In this chapter, we describe WICkit, a framework and runtime that provides applications with the WIC abstraction. WICkit supports both synchronous and asynchronous memory accesses and implements WICs using C++ stackless coroutines to minimize switching costs. We demonstrate the efficacy and generality of WICkit by implementing and evaluating two distributed applications: a key-value store and a remote-shared log. Additionally, we also evaluate WICkit's performance using microbenchmarks. We show, using a linked list traversal microbenchmark (§3.6) that when memory-access logic is executed on the server, WICkit can achieve comparable throughput and latency to an application written using eRPC. Similarly, we show for the same benchmark that when memory-access logic is run on the client, WICkit achieves throughput and latency comparable to that achieved by an application that directly uses 1-sided RDMA operations.

We thus demonstrate that WICkit provides an efficient mechanism for deciding the location of an application's memory-access logic when it is first executed. In §3.6, we demonstrate WICkit's utility by showing that there are performance and cost benefits to executing memory-access logic at all three locations. We currently do not address the question of how to decide where memory-access logic should be located, nor the question of how to move WICs between locations at runtime. We plan to address these questions in future work.

3.2 The Desire for Location Independence

WICkit is designed so that programs can be written once and configured to operate in three distinct modes: (a) a client-only configuration in which WICs execute on client cores; (b) a SmartNIC configuration in which WICs execute on cores on the server's SmartNIC; and (c) a server configuration in which WICs execute on server cores (WICs are invoked from client cores in all configurations). As we noted above, existing work on RDMA has largely focused on comparing the client-only configuration (one-sided RDMA applications) and server configuration (client-server applications) and in showing that one or the other is more desirable for different applications. Given this, one might wonder: why is the ability to choose between the three at deployment time desirable?

As prior work has shown [67, 71, 125], the relative performance and resource requirements for these configurations vary depending on application logic, workload, and deployment environment. However, beyond these factors, deployment costs can also vary significantly for these different configurations. This is because clients and servers generally face different scaling requirements and can be deployed on different instance types with different costs. Additionally, many cloud

Function	Description
`Awaitable Backend::read_host(uintptr_t raddr, uint32_t sz)`	Read `sz` bytes from host's address `raddr`.
`Awaitable Backend::write_host(uintptr_t raddr, void *data, uint32_t sz)`	Write `sz` bytes from `data` to host's address `raddr`.
`uintptr_t Backend::get_rbaseaddr()`	Gets remote base address.
`void Backend::reply(void *data, uint32_t sz)`	Send `data` of size `sz` as reply to the client.
`WICCoro WICLock::lock(Backend &b)`	Lock WICLock using Backend `b`.
`WICCoro WICLock::unlock(Backend &b)`	Unlock WICLock using Backend `b`.

Table 3.1: WICkit's location-agnostic API.

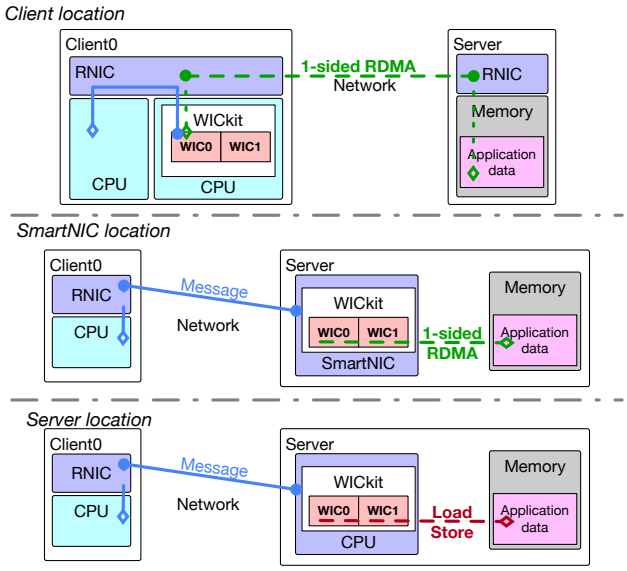

Figure 3.1: The three supported locations for WICkit and how each location accesses application data in server memory.

providers, including Microsoft [47], argue that processing resources on a SmartNIC cost less than on host CPUs because host CPUs can be more easily sold to tenants. Furthermore, instance costs vary over time, and we expect that SmartNIC core costs are also likely to vary over time. When deploying an application, developers and administrators need to consider not just performance but also costs, and it is thus easy to see that the best configuration can vary over time and across deployment environments. A framework such as WICkit provides developers and administrators with greater flexibility when responding to changing workload, application logic, or costs and thus enables more efficient application deployments.

3.3 Programming Model

The core abstraction provided by WICkit is Where-Independent Code (WIC). A WIC encapsulates the logic surrounding and including related memory accesses. For example, when implement-

ing a key-value store application, a user might decompose their application logic into WICs for `GET(K)`, `UPDATE(K,V)`, `SCAN(K)`, etc. With a tree-based index [123], the `GET(K)` WIC would traverse the index, specifying how to select each subsequent node to read in the tree, and issuing memory accesses to read each node. While WICs can execute at client CPUs, server CPUs, or SmartNIC CPUs, we assume that the main *application data* always resides at the server, as shown in Figure 3.1. Thus, when WICs issue memory accesses, these are primarily to server memory.

The WIC programming model specifies how a user expresses a WIC and the APIs between WICs and the WICkit runtime (§3.4). The runtime's job is to invoke and manage WICs and provide two APIs: the API the runtime uses to invoke WICs and the API functions that WICs can call (*e.g.* to access application data). The programming model should provide four key properties:

1. The code in a WIC should remain the same, regardless of where the WIC is executed.
2. The WIC abstraction should be amenable to good performance (low latency and high throughput).
3. WICs should be convenient to program.
4. WICs should provide the same functionality (e.g., locks) as existing frameworks.

WICkit provides these properties using two techniques. First, WICkit expresses WICs using a thread-like abstraction. This enables WICs to support both synchronous and asynchronous memory accesses (for high performance regardless of where a WIC executes) in a programmer-friendly way (§3.3.1). Second, WICkit provides a single unified API for WICs to use to access application data and features such as locks, regardless of where the WIC executes. The runtime then uses a different underlying mechanism depending on where the WIC is currently executing (§3.3.2).

3.3.1 Representing WICs

There are several possible ways to represent WICs. For example, each WIC could be represented as a single run-to-completion function [21, 37, 104] or as a thread that can suspend and resume. However, not all approaches enable high performance for WICs, regardless of where they execute.

The key challenge is that the latency to access application data differs significantly depending on where a WIC executes. For example, when WICs execute at the server, each local access to application data can complete in about 100ns. In contrast, when WICs execute at a client or SmartNIC CPU, they access application data over RDMA, which can take 1.5-10μs. While a run-to-completion approach can perform well at the server, it would yield poor throughput at the client and SmartNIC locations, because every access to application data would stall a CPU core for a few microseconds while waiting for the data to arrive. At those locations, WICs with outstanding memory accesses must yield the CPU to other WICs. Thus the WIC programming model must support both run-to-completion and suspend-and-resume models.

There are two main ways to implement a suspend-and-resume programming model: callbacks and threads. To implement WICs with callbacks, a WIC would need to specify which callback

function to call after each memory access is completed; this is similar to eRPC's completion handler functions, which execute when a recursive RPC request completes [67]. In WICkit, this would mean that a loop that iterates over n nodes, reading each from memory, would need to be split into $n+1$ chained callbacks. Furthermore, the programmer would need to explicitly specify what execution state to pass from callback i to callback $i+1$. As this is not a convenient programming model for users, WICkit instead represents each WIC using a thread-like abstraction. Threads work well because they support both run-to-completion and suspend-and-resume models for accessing application data.

As we describe in more detail in §3.4.3, WICkit implements WICs using C++20 stackless coroutines [106]. Coroutines behave similarly to threads, except they are able to suspend and be resumed later, transparently allowing the state before suspension to be available after the resumption. In addition, coroutines yield control at least an order of magnitude faster than existing thread implementations, enabling better performance.

3.3.2 WIC API

WICkit provides a single unified API for tasks such as accessing application data and acquiring locks, so that users can write each WIC once and execute it at any location without modification. WICkit's API is summarized in Table 3.1.

A distributed application that uses WICkit is comprised of mainly three components: a server component, the runtime, and the application that sends WIC request executions to the runtime. The server component holds and initializes the application data that will be accessible to the runtime. When the runtime begins execution, it connects to the server, and they exchange information about the available memory regions.

WICCoro. WICCoro is the type our runtime uses to define WICs [93]. Internally, it defines the suspension and resumption behavior of our coroutines, and it keeps the internal coroutine handle and any other per-WIC internal state [66, 106].

Invoking WICs and replies. Clients in a distributed application request WIC executions by sending an execution message with a specific WIC id to the current runtime's location. Besides the id, requests include parameters that must be passed to the requested WIC. When a WIC completes, it can send a reply back to the client that requested the execution by using `reply(void *data, uint32_t sz)`.

Memory Accesses. The WIC API provides two basic low-level functions for accessing application data: `read_host()` and `write_host()`. Rather than providing functions for fixed-size accesses, our functions accept variable buffer sizes. As shown in Table 3.1, both functions return an Awaitable object type, which means the type supports the `co_await()` operator [17]. In other words, Awaitable types define points at which a WIC could be suspended. For example, this allows us to suspend WICs when they access memory through RDMA and not suspend if the access is made synchronously to local memory. The memory-access functions take server virtual addresses, and the base remote address can be obtained by a WIC with `get_rbaseaddr()`. Both functions access application data on the server, but the WICkit runtime implements them differently depending on

```
1  WICCoro traverse_linkedlist(Backend &b, void* param0) {
2    int num_nodes = *(reinterpret_cast<int*>(param0));
3    uintptr_t addr = b.get_rbaseaddr();
4    LLNode *node = nullptr;
5
6    for (int i = 0; i < num_nodes; ++i) {
7      node = co_await reinterpret_cast<LLNode *>
8          (b.read_host(addr, sizeof(LLNode)));
9      addr = node->next;
10   }
11
12   b.reply(&node->value, sizeof(node->value));
13 }
```

Listing 1: A WIC that performs traverses a remote linked list.

where the WIC is executed. As described in Section 3.4.1, WICkit supports three different backends for accessing memory. Backends are selected at compile time and passed to every WIC as an argument. When WICs call `read_host` or `write_host`, they call these functions on the specified backend.

Locks. WICkit provides simple exclusive locks to WICs through a `WICLock` class. The class provides two methods that can be used regardless of the runtime's location: `lock(Backend &)` and `unlock(Backend &)`. We describe the design of locks and why they return `WICCoro` in Section 3.4.2.

Listing 1 shows a WIC that performs a remote linked list traversal. The number of nodes to traverse is specified as an execution argument by the client (*i*.e., `param0` in Line 1). The WIC gets the base remote virtual address by calling `get_rbaseaddr()` and then proceeds to traverse the linked-list by issuing `read_host()` calls. When the WIC has traversed the number of requested nodes, it issues a reply back to the client that requested the execution.

3.4 Runtime Design

3.4.1 Memory-Access Backends

WICkit's memory-access APIs (§3.3.2) are implemented by location-specific backends. Our implementation provides three backends: a 1-sided RDMA backend that is used in the client and SmartNIC configurations, a synchronous local-memory backend used by the server configuration when running WICs to completion, and an asynchronous local-memory backend which is also used by the server configuration. The asynchronous local-memory backend prefetches cache lines before accesses and provides better performance for some applications. We describe each of these backends below.

1-sided RDMA

The main challenge of the 1-sided RDMA backend is providing high memory-access throughput for client and SmartNIC locations. Our design addresses this challenge using two techniques: first, our design uses multiple cores allowing multiple simultaneous RDMA memory accesses, and second, our design batches RDMA operations, thus reducing the cost of communicating with the RDMA adapter. Multiplexing improves throughput by 8.4-12.1× relative to not multiplexing, and batching can improve throughput by 2.1× and decrease latency at high loads (§3.6.4).

Multiplexing memory accesses. The RDMA backend uses multiple cores and queues simultaneously, and can thus issue multiple memory accesses at a time. As we explain below our design avoids cross-core synchronization but does not balance load across cores. We will investigate approaches [102, 104] to balance load in the future.

RDMA's *queue pairs* are used by CPUs to submit work requests to the RDMA adapter. In Bluefield-2, each queue pair has a limit of 16 outstanding RDMA reads.[1] Conversely, *completion queues* are used by the RDMA adapter to signal cores that work requests have completed. Using 1 queue pair per core would only allow WICkit to maintain 16 outstanding reads per core at most.

The RDMA backend uses multiple queue pairs per CPU to increase memory-access multiplexing and improve throughput (§3.6.4). However, we configure the queue pairs to aggregate all completions into a single completion queue. This enables each core to have more outstanding operations and improves efficiency when polling for completion, since each core can poll from 1 CQ. By default we use 4 QPs per core at the SmartNIC and 16 QPs per core at the client; we evaluate the tradeoffs of different numbers of QPs in §3.6.4.

Reducing CPU cost of RDMA. We batch RDMA operations to reduce CPU cycles spent issuing new RDMA requests. Previous research has shown that reducing communication frequency to the RDMA adapter can result in better CPU efficiency and RDMA throughput [69]. This is of particular concern when using SmartNIC cores which can be significnatly slower than x86 cores. For example, when comparing the performance of a 2.6GHz Haswell core[2] to the performance of the 2GHz ArmV7 A7 core on a Bluefield-2 [115] NIC using STREAM [94] and Coremark [51] we found that the ARM core achieves at most 64% of an x86 cores performance (Table 3.3).

Batching requests allows a core to issue multiple work requests with a single MMIO write, reducing the number of cylces spent issuing RDMA requests. Our runtime creates batches of memory accesses for WICs by using the new verbs work request API `ibv_wr_start()` and `ibv_wr_end()`. These two functions establish a code region in which a core can more efficiently post work requests, compared to the previous `ibv_post_send()` API [116]. Once a code region is open, RDMA operations can be posted inside, for example, by using `ibv_wr_rdma_read()`. These operations are buffered and only delivered to the RDMA adapter when `ibv_wr_end()` is called. Each batch is submitted to a single queue pair.

Listing 2 shows how WICkit implements batching. We first select the queue pair we create the batch for and call `ibv_wr_start()` on it (Line 3). Next we pop a WIC from the `runqueue` (Line

[1] To the best of our knowledge other RDMAs impose the same limit.

[2] Intel Xeon CPU E5-2640 v3.

```
1  void execute_interleaved() {
2    for q in qps {
3      ibv_wr_start(q.qp);
4      while (len(runqueue)>0 && q.outstanding<MaxInFlight) {
5        wic = runqueue.pop();
6        wic.resume();
7        if (wic.issued_mem_access)
8          q.pending.push(wic);
9        ...
10     }
11     ibv_wr_end(q.qp);
12   }
13 }
```

Listing 2: Pseudocode showing how application data accesses are batched in RDMA backend using multiple queue pairs.

Stateful execution context	BF2 latency	x86 latency
pthread	1583.2ns	342.4ns
user thread [102]	-	52.0ns
boost coroutine	48.7ns	17.4ns
boost coroutine2	40.2ns	7.2ns
stackless coroutine	3.0ns	1.9ns
function call	1.5ns	0.9ns

Table 3.2: Yield latencies of different stateful execution contexts.

4) and resume it. The resumed WIC eventually issues a new application data access on queue pair `q`, which causes the WIC to yield back to Line 7. We keep selecting WICs from the `runqueue` and resuming them, until the `runqueue` is empty, or the current queue pair has `MaxInFlight=16` outstanding requests (Line 4). Lastly, we end the current batch by calling `ibv_wr_end()`, which posts the work requests to the RDMA adapter (Line 11). Therefore, our batching is dynamic as it does not wait for a fixed batch size to be reached before posting the batch.

We further reduce the number of CPU cycles needed by using *unsignaled RDMA work requests*. RDMA-based systems typically use signaled requests, where each request generates a completion when it finishes. Unsignaled RDMA requests do not generate completions, and leveraging them for some requests can reduce the number of PCIe transactions [69] performed by the RDMA adapter and the number of completions the CPU polls for. Thus our backend only uses a signaled work request for the last access in each batch.

The challenge to only signaling the last request in a batch is identifying what unsignaled WICs have completed. Signaled work requests in RDMA include an 8-byte identifier that is included with the completion event, and this can be used to identify the corresponding coroutine to resume. However, the RDMA adapter provides no information about which unsignaled requests have com-

Microbenchmark	Armv8 A72	Xeon E5-2640 v3	Armv8/x86
STREAM copy (MB/s)	7119.3	13494.1	53%
STREAM scale (MB/s)	7116.5	8735.5	81%
STREAM add (MB/s)	6392.3	9986.6	64%
STREAM triad (MB/s)	6400.8	10180.3	63%
Coremark	11426.6	18995.4	60%

Table 3.3: Two microbenchmarks showing the performance ratio achieved by 1 ARM CPU in Bluefield-2 relative to a Haswell x86 core. The ARM CPU offers only 64% performance of the x86 core.

pleted or when.

Fortunately, our runtime can leverage the fact that queue pairs complete work requests in order to identify which unsignaled request have completed. To do this, the runtime maintains a per-queue pair software `pending` queue to track WICs that have issued a memory access and suspended (Line 8 in Listing 2). When we submit a batch of memory accesses, we use the last memory access' identifier to encode the id of the corresponding `pending` queue and the batch size. When we receive a completion, WICkit can pop `batch_size` WICs from the corresponding `pending` queue and add these WICs to the current core's `runqueue`.

Local memory

The WICkit's server location uses one of two local memory backends to access application data directly with CPU instructions. The first backend accesses local memory synchronously, and a second one behaves asynchronously and uses prefetching instructions before load-stores.

As we discuss in more detail in §3.4.3, WICkit uses coroutines to implement logical threads. Such coroutines need to be initialized before they are invoked; thus, the runtime pays for additional CPU-memory traffic and CPU cycles for every invoked WIC. Our asynchronous memory backend attempts to compensate for the additional latency and CPU-memory traffic by multiplexing local memory access using `prefetchnta` instructions [64], similar to our RDMA backend. As we show in §3.6.3, the asynchronous backend can significantly improve throughput relative to run-to-completion in microbenchmarks where every access generates a CPU last level cache miss.

3.4.2 WICLocks

The runtime provides exclusive locks to WICs through a `WICLock` class with two methods: `lock(Backend &)` and `unlock(Backend &)`. WICLocks are designed so they can work at all three locations. Doing so requires ensuring that the runtime does not block when a WIC cannot acquire a lock. To achieve this, the `lock` function yields to the runtime when a lock cannot be acquired.

Additionally, we need to ensure that the lock itself is accessible to all executing WICs. One approach to doing so would be to always place the lock on the server, and we adopt this approach when using the server or client backend. In this case the client backend acquires and releases locks

using 1-sided RDMA atomic verbs, while the server backend uses atomic instructions. When using the SmartNIC backend we optimize the lock further by placing its state in SmartNIC memory. This allows us to use atomic instructions to acquire and release locks when using the SmartNIC backend. We plan to investigate approaches to further optimizing these locks and implement other synchronization primitives in the future.

3.4.3 Scheduling WICs

Thus far, we have described the design of our backends and the design of our where-independent WICLocks. We now focus on a crucial aspect of our design: our coroutine-based logical threads. Context switch latency of logical threads is a major concern for WICkit's performance, and our runtime uses C++ stackless coroutines to minimize these costs; we refer to these simply as coroutines. As Table 3.2 shows, switching coroutines in SmartNICs is three orders of magnitude faster than switching pthreads, and in x86, it is 27× faster than switching user threads.

Writing a WIC is similar to writing a function: statements are executed sequentially, and local variables can be defined at any point inside the WIC body. However, coroutines extend functions by adding support for preemption, or *suspension points*, defined inside the coroutine body by using the `co_await` operator; see Listing 1. After suspending, coroutines can either return control to the caller or can transfer control to another coroutine;name makes use of both kinds of control transfer. For example, the runtime returns control to the scheduler (*i*.e., the caller) after issuing an asynchronous memory access and transfers control to another coroutine when taking a lock. The runtime can also choose to not transfer control at a suspension point, and WICs do not yield control when performing synchronous memory accesses. In both cases, after a coroutine is resumed by the scheduler, execution resumes at the statement following the previous suspension point. Coroutines provide very fast switch latencies as the compiler generates the switching code and stores the coroutine's local variables and execution context on the heap rather than on the stack. Our implementation uses a custom allocator to reuse previously allocated WICs, but a recycled buffer must still be initialized every time a new WIC is instantiated.

Although all backends invoke WICs in FCFS order, they must also decide when to resume a suspended WIC. For example, a WIC that issued a 1-sided RDMA asynchronous operation should not be resumed until after its memory access completes. Thus, to improve WIC throughput, each memory access backend includes a scheduling policy to better target the memory performance characteristics available at each location. We describe them next.

RDMA scheduler. The scheduler of the 1-sided RDMA backend executes WICs as follows. First, a client sends a WIC execution request which is received on a SmartNIC core's `request` queue. The runtime polls the request queue and uses the received request's WIC id to instantiate a specific WIC. Newly created WICs are immediately suspended when they are created and added to the back of the `runqueue`. Next, the scheduler resumes WICs from the runqueue's front, and every resumed WIC runs until it finishes or suspends. The runtime continues removing and resuming WICs from the runqueue as long as the runqueue is not empty and the current queue pair has less than 16 outstanding memory accesses. Eventually, WICkit checks for completed memory accesses

by polling the `completions` queue, and all WICs that have a completed memory access are added back to the front of the runqueue (in order to ensure FCFS processing).

Local memory schedulers. The synchronous local memory backend runs individual WICs to completion; therefore, memory accesses do not suspend a WIC, and when an WIC finishes, another one is selected from the runqueue in FCFS order.

In our asynchronous local memory backend, the scheduler first selects a *window* of WICs from the runqueue in FCFS order. Instead of running a single WIC to completion, the scheduler executes the window-to-completion (similar to [64]). When WICs access application data, the backend issues `prefetchnta` instructions to bring the cache line close to the CPU and suspends. Once prefetch instructions have been issued for every WIC in the window, the scheduler resumes WICs in the order in which prefetches were issued. This time, the backend issues load-store instructions, completing the memory access for each. The same process repeats until all WICs within a window finish. We evaluate the effect different window sizes have in §3.6.4.

3.5 Implementation

We implemented our WICkit prototype in 4770 lines of C++20. We use GCC 10.2, and we cross compile to generate ARM binaries for Bluefield-2. After a developer writes a WIC, they use the WICkit build system to produce 5 binaries: a server binary that holds the application data for the RDMA backends, and 4 runtime binaries. One of the runtime binaries targets the client location thus comes with the RDMA backend; the SmartNIC binary is equivalent, except cross compiled for ARM. The last two binaries target the server location, one using the synchronous local memory backend, and the last one the asynchronous one.

Limitations. We now discuss the limitations of our prototype. These limitations are not fundamental, and we plan to address them in the future. First, our prototype assumes no failures. Second, our implementation requires one request and reply queue per connection. This is because we use 2-sided RDMA RC queues rather than raw ethernet queues as is done by eRPC [67]. Third, our prototype runtime currently assumes the client and SmartNIC have sufficient memory to hold all application data and the RDMA backend maps server memory one-to-one. A better implementation would not make this assumption, and instead allow WICs to manage their location-local memory.

3.6 Evaluation

Our evaluation of WICkit focuses on three main questions:

1. Are there scenarios where users would prefer each of the locations supported by WICkit? (§3.6.2)

2. How does the performance of WICs compare to systems that are customized to run at a specific location? (§3.6.3)

3. What impact do the design decisions described in §3.4.1 have on WIC performance? (§3.6.4)

3.6.1 Experiment Setup

We use two load generators in our experiments: a closed-loop load generator to understand maximum WIC throughput, and an open-loop load generator with exponentially distributed inter-arrival times to analyze latency under load. Our testbed system consists of two two-socket servers with Intel Xeon E5-2640 v3 CPUs. We disable hyperthreading and turbo boost and fix the CPU frequency at 2.6Ghz. Our servers are connected through 100GbE Bluefield-2 MBF2M516A SmartNICs which include a ConnectX-6 Dx RDMA adapter and 8 Armv8 A72 cores running at 2Ghz. We refer to the NIC simply as BF2. The BF2s are connected through an Arista 7160S-32CQ 100GbE switch. Both hosts and SmartNICs use Linux kernel 5.4 and we use `isolcpus` to isolate CPUs from kernel SMP balancing and scheduling disturbance. We also customize the SmartNIC's kernel to enable `nohz_full` [75] which prevents interrupt timers in cores the runtime uses, as we found these induced significant jitter. We use three workloads to evaluate WICkit:

Linked-list traversal. Our first workload creates a randomized linked-list in application data and uses a `traverse(num_nodes)` WIC to walk the linked-list starting from a random node. num_nodes is a request parameter, and each linked-list node is 16-bytes. Traversing a linked-list is equivalent to issuing multiple-dependent memory accesses.

Remote shared log. The second workload is a remote shared log with two WICs: `readtail()` and `append(value)`; we use a request composition of 50% readtail and 50% append. Shared logs are a core component in distributed protocols such as consensus and leader election [18]. Our shared log's application data is a buffer that holds 8-byte values, and its WICs maintain a head and tail pointer. As the names suggest, readtail reads the value currently pointed by tail, and append increments the tail pointer and sets a new value. We use a single exclusive WICLock to protect the pointers and the buffer's consistency under concurrent requests.

Key value store. For our last workload we evaluate a key value store based on a cuckoo hash table that we ported to WICkit [25]. We implement two WICs: `query(key)` and `update(key, value)`, and use a request composition of 50% queries and 50% updates. We use an exclusive WICLock to protect the table during updates and queries.

Unless specified otherwise, to serve the load, all experiments use one server x86 core when WICkit is located at the server, and one client x86 core when WICkit is located at the client. Depending on the experiment, for SmartNIC locations we show results for 1, or 1 and 7 cores.

3.6.2 Benefits of Different Locations

The main benefit of using WICkit in a distributed application is the ability to write the WIC once, and decide at application start-up time whether to execute it at client, server, or SmartNIC CPUs, where each location offers specific performance and resource trade-offs. In this subsection we use our three workloads to explore such trade-offs, and we synthesize our observations in §3.6.2.

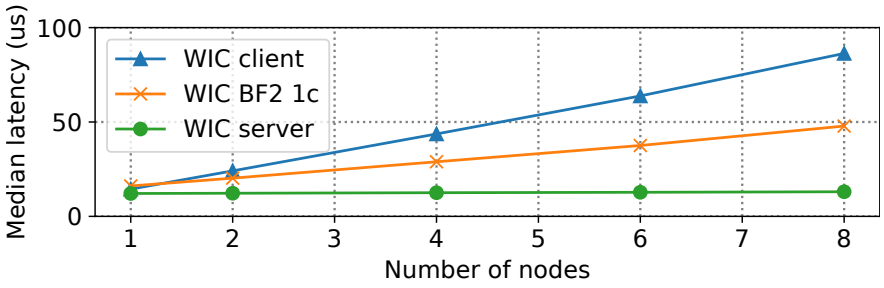

Figure 3.2: Unloaded latency of linked-list traversal as we vary the number of traversed nodes at all WICkit locations.

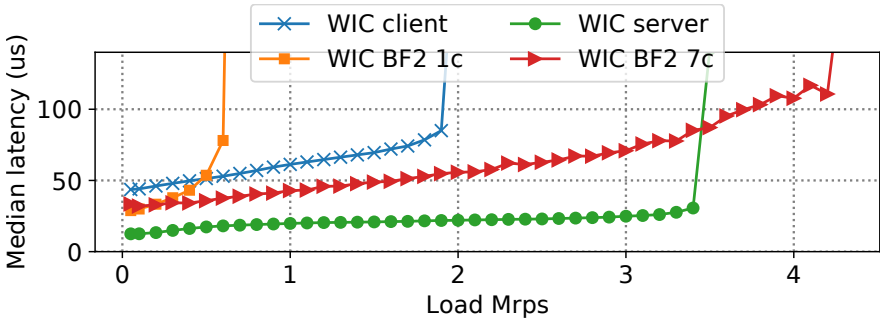

Figure 3.3: 4-node linked-list traversal latency under load at all WICkit locations. The server and client locations use 1 x86 CPU to serve load and the BF2 location uses 1 and 7 SmartNIC CPUs.

Linked-list Traversal

To understand the location trade-offs for this workload, we first evaluate how unloaded latency across locations changes as the number of traversed nodes in a WIC increases. Then, we measure latency across locations when traversing 4 nodes.

As shown in Figure 3.2, across WICkit's locations, the server provides lowest latency, and BF2 performs better than client when traversing two or more nodes. When traversing one node, a WIC located at the server takes 12.1μs to complete, whereas at the client it takes 14.4μs, and at BF2 it takes 16.1μs. Client is 19% slower than server because the client location pays for a loopback message from the load generator to its runtime on the same machine, plus the network round trip for reading the node with RDMA (see Figure 3.1). BF2 latency is higher than client when reading 1 node as the location pays for the latency of a request message over the network to the SmartNIC, plus the latency of an RDMA read over PCIe. However, once each WIC traverses 2 nodes, BF2 achieves a latency of 20.2μs and client's latency is 19% higher as it pays the full network round trip latency for both RDMA reads. As the benefits of reading nodes directly through PCIe apply for every memory access, reading 8 nodes is 93% faster in BF2 relative to client.

Next we analyze loaded latency while traversing 4 nodes. As Figure 3.3 shows, different locations saturate at distinct loads. The server location provides lowest latency until its maximum

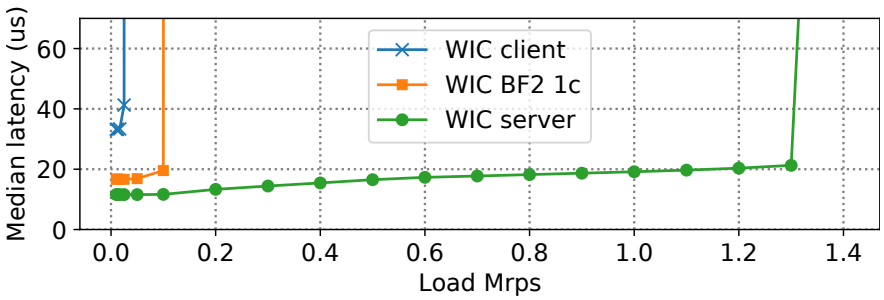

Figure 3.4: Remote shared log latency under load at all WICkit locations. To serve load, client and server locations use 1 x86 core, and BF2 location uses 1 SmartNIC CPU.

throughput of 3.4M WIC/s. The better choice between client and BF2 depends on the load and the number of cores used by BF2. At low loads, BF2 using 1 core achieves lower latency than client, but at 0.5M WIC/s, the software overheads accrue in the one BF2 core and client latency is better beyond this point. However, by leveraging 7 BF2 cores the SmartNIC location can serve a load of 4.2M WIC/s, 7× the throughput achieved with 1 SmartNIC core.

Remote Shared Log

We now explore the benefits of each location in our remote shared log application by evaluating latency under load. As Figure 3.4 shows, server provides the lowest unloaded latency with a median request latency of 11.6μs. The workload is amenable to the server location, as WICLock uses CPU atomics to local memory, and the shared log's array of values provides good locality for the server CPU's cache. BF2 also keeps the WICLock in NIC-memory, but must access values through RDMA, resulting in 44% higher latency than server. Client has the highest latency because, besides having to read the values through the network, WICLock uses RDMA atomics to lock. Using an exclusive WICLock to implement critical sections that issue RDMA memory accesses significantly reduces the throughput of the client and SmartNIC locations (§3.4.2). This is because the WICLock is kept locked while RDMA memory accesses complete. Since the server location accesses memory locally, the time a lock is taken significantly reduces. Thus, throughput is highest at the server and achieves 1.3M WIC/s. BF2 achieves a throughput of 100K WIC/s, and additional BF2 cores do not improve throughput as the contended lock prevents maintaining multiple outstanding memory accesses at once.

Remote Key-value Store

In our last location trade-off exploration we evaluate our key value store's latency under load. Similarly to the shared log application, this cuckoo hash table uses one exclusive WICLock to protect the table during updates and queries, so the same throughput limitations apply. However, as shown in Figure 3.5, overall latencies are higher than in shared log, because both update and query need to compute the hash of the requested key, which is compute intensive.

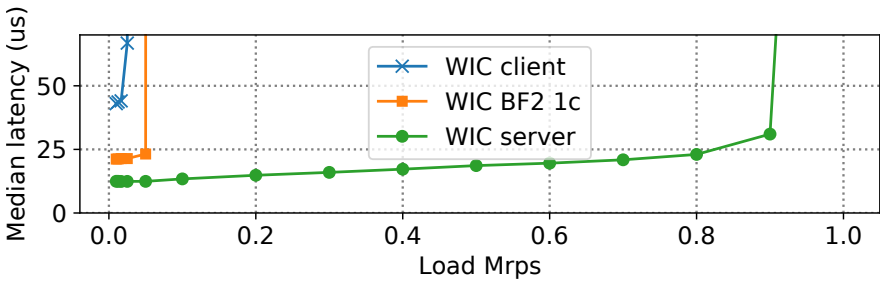

Figure 3.5: Remote cuckoo hash table latency under load at all WICkit locations. To serve load, client and server locations use 1 x86 core, and BF2 location uses 1 SmartNIC CPU.

The Best Location

For each location we support, are there scenarios in which users would prefer to run WICs in that location? Yes, but there are different factors that play a role in deciding which location is best for a given workload at a given time. The type and quantity of resources an application might want to use has an effect in the possible locations to consider. For instance, although the server location provides low latency and high throughput, it might not be ideal in scenarios where CPU availability is low, or when CPU cost is high. As we showed, the number of memory accesses a WIC issues can also be a factor in choosing a best location for latency. Similarly, offered load is a workload-specific factor that should be considered as well. A distributed application using WICkit is able to navigate the trade-off between remote memory access locations by writing WICs in a where-independent manner and choosing whichever is best at application start-up time.

3.6.3 WICkit vs Location-dependent Systems

We now evaluate how WICkit's performance compares to location-specific approaches using the linked-list traversal workload. We first evaluate the performance of server-located approaches by comparing WICkit's asynchronous local memory backend and eRPC, the state-of-the-art RPC system for communicating between hosts [67]. To maximize eRPC performance, we set session credits to 2048, set the session request window size to 2048, and the max inline message to 512. We then compare WICkit's client location and an optimized 1-sided RDMA linked-list traversal implementation. Finally, no current system uses SmartNICs and we instead compare WICkit's SmartNIC location to both the eRPC and the 1-sided RDMA optimized implementation.

Server Location

We evaluate server-located approaches when traversing a 4-node linked-list as shown in Figure 3.6. WICkit uses the asynchronous local memory backend with a window size of 8 (§3.4.3).

We find that eRPC achieves best latency under load until the CPU serving the load saturates, and WICkit provides better throughput due to prefetching and very fast context switching of WICs. eRPC achieves an unloaded latency of 9.6μs per request, whereas WICkit takes 12.5μs per WIC.

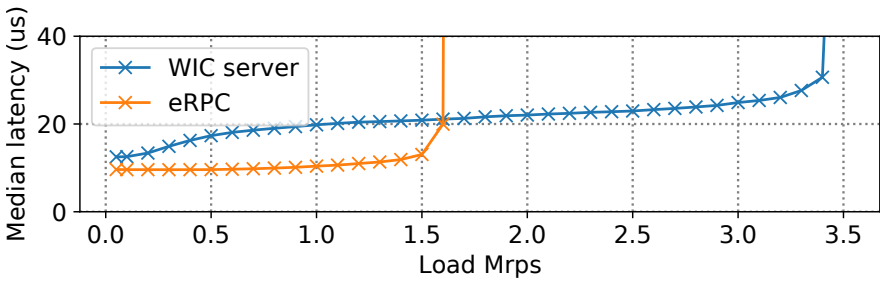

Figure 3.6: Comparison of server locations on a 4-node linked-list traversal. Both approaches use one x86 server core. WIC server uses the asynchronous backend.

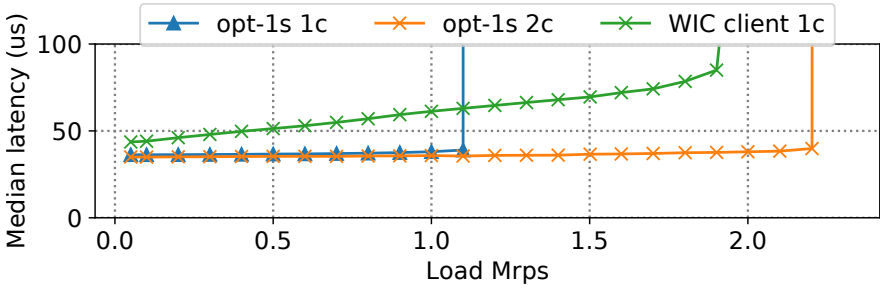

Figure 3.7: Comparison of client locations on a 4-node linked-list traversal.

The reason for eRPC's lower latency is due to its use of call backs, which are cheaper to invoke than coroutines since they do not have to be initialized. WICkit's asynchronous backend is able to achieve a throughput of 3.4M WIC/s, 2.1× higher than eRPC. However, we note that the improvements of the asynchronous backend brought by prefetching local memory depends on the specific workload. Linked-list traversal is amenable to prefetching because every read node produces a last-level cache miss. We leave for future work exploring in more detail the benefits of the asynchronous backend.

Client Location

We now compare client-located approaches: WICkit's client, and an optimized 1-sided RDMA linked-list traversal implementation, which we refer to as opt-1s 1c when using one core. opt-1s is both a load generator and a location where memory-access logic executes. In contrast, WICkit's client location uses one core for the runtime, and another core for the load generator, both in the same machine (see Figure 3.1). Although WICkit's load generator does not access remote memory, we report opt-1s 2c as well. To maximize RDMA throughput, both systems 16 QPs in total to remote application data, where opt-1s 2c uses 8 QPs per-core.

Overall, as Figure 3.7 shows, opt-1s achieves better latency while WICkit's client location achieves higher throughput for one core. The unloaded latency to read 4-nodes through the network with opt-1s 2c is 35μs whereas WICkit takes 43.6μs, a 25% increase. WICs have a higher latency as

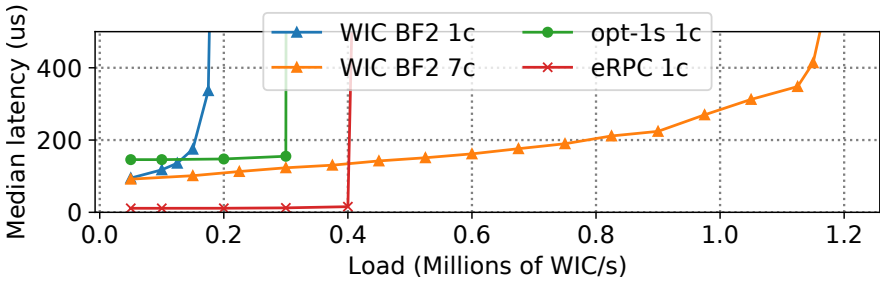

Figure 3.8: Latency for a 16-node linked-list traversal under load using WICs at BF2, an optimized 1-sided implementation, and eRPC.

opt-1s does not pay for loopback requests and replies between load generator and WICkit runtime, which take 1.4μs each way, and because it uses call backs instead of coroutines. The client location with one core achieves a throughput of 1.9M WIC/s, 72% higher than opt-1s using one core due to WICkit's batching mechanism described in §3.4.1. Finally, the client location using one core achieves 15% lower throughput than opt-1s 2c.

SmartNIC Location

We now compare the SmartNIC location to eRPC and opt-1s for context. Figure 3.8 shows the comparison where each request, or WIC, traverses 16 nodes.

eRPC provides lowest unloaded latency while BF2 1c is faster than opt-1s (similarly to Figure 3.2). However, BF2 1c saturates quickly with increased load whereas using more cores allows BF2 to maintain lower latency for larger loads. Although BF2 1c takes 94.7μs per WIC when reading 16 nodes, it is 54% faster than opt-1s, and represents a software overhead of only 1.5μs per memory access on top our baseline hardware latencies. In our hardware, each request from client to NIC takes 4.9μs, each RDMA read takes 3.8μs, and each message from NIC to client takes 4.9μs as well. In terms of throughput, Blufield-2 using 1 core (*i*.e., BF2 1c) achieves 58% of opt-1s's throughput, however, by using 7 cores BF2 achieves 6.5$\times$ higher throughput than when using 1.

3.6.4 Backends Design

We now evaluate how our backend designs contribute to WICs performance across locations. We first explore the 1-sided RDMA backend performance by focusing on SmartNICs, as no previous system has leveraged them as a memory-access logic location. We evaluate our design decisions to (§3.4.1): multiplex memory accesses, use multiple RDMA queue pairs, use coroutines-based logical threads, and issue memory accesses in dynamic batches. Then, we evaluate our local memory backends in §3.6.4.

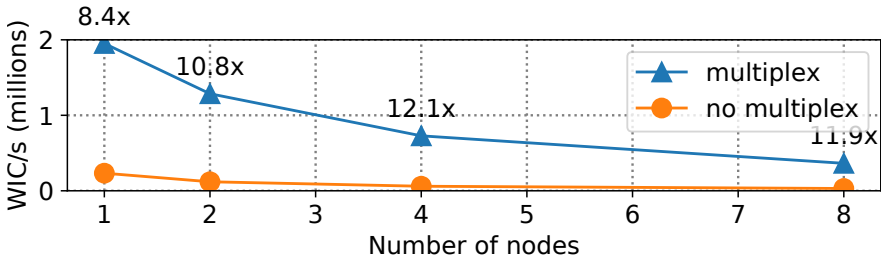

Figure 3.9: Comparison of multiplexing and not multiplexing RDMA accesses in a linked-list traversal WIC located at BF2. The labels above the marks show improvement of multiplex over no multiplex.

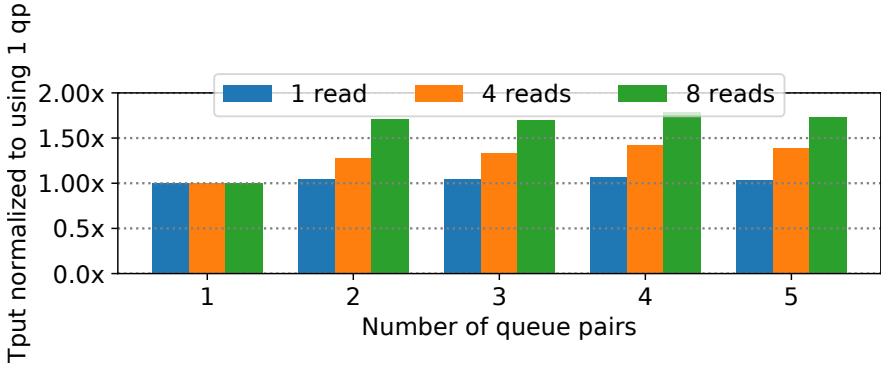

Figure 3.10: Throughput improvement as we increase the number of queue pairs, relative to using 1 queue pair. We show improvements for 1, 4 and 8 linked-list node traversals per WIC.

Multiplexing RDMA Application Data Access

We use our linked-list workload to measure WIC throughput as we increase the number of traversed nodes with two backends: our baseline 1-sided RDMA backend that multiplexes memory accesses, and a backend that does not multiplex accesses and runs WICs to completion. As shown in Figure 3.9, multiplexing memory accesses improves WIC throughput by 8.4-12.1× relative to not multiplexing when using one BF2 core. By using fast-yielding contexts, WICs efficiently relinquish the CPU after they issue an RDMA access and switch to a ready-to-run WIC, enabling multiple outstanding memory accesses at a time. We explore the impact of using slower yielding contexts later in this section.

RDMA Multiple Queue Pairs

We evaluate how the use of multiple queue pairs impacts maximum throughput and latency in our linked-list workload. As discussed before, our BF2 RDMA adapter has a limit of 16 outstanding RDMA reads per queue pair. Using multiple queue pairs per-core provides the potential for WICkit to maintain additional outstanding reads, increasing throughput. However, it's unclear if

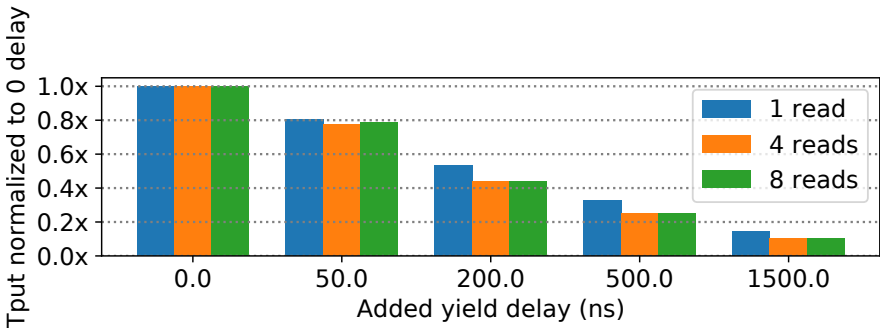

Figure 3.11: Throughput degradation when we add x-axis delay to coroutine yield, relative to baseline coroutine yield of 3ns.

our SmartNIC cores will be able to handle additional queue pairs and whether more queue pairs will negatively impact latency.

As Figure 3.10 shows, using multiple queue pairs per-core increases throughput. Using 4 queue pairs per-core improves throughput by 6% when traversing 1 node, and by 78% when traversing 8 nodes, relative to using 1 queue pair. Across 1, 4 and 8-node traversals, using 4 queue pairs increases latency at most by 1.7%, relative to using 1 queue pair. Therefore, our SmartNIC WICkit runtime uses 4 queue pairs per-core to increase memory access throughput to application data.

Performance Impact of Yield Latency

We now evaluate the performance impact of using alternative execution contexts to show coroutines, and their fast yielding latencies, are necessary for WICkit. We simulate the performance impact of execution contexts with higher yield latency by adding delays on top of coroutine yields. We use our linked-list traversal workload to traverse 1, 4 and 8 nodes.

We find that if our logical threads experienced yield latencies similar to user threads, SmartNIC throughput would degrade by 24% at best, and pthreads would cause the throughput to degrade at least 6.8×. Figure 3.11 shows simulated throughput degradation in BF2 when matching alternative execution contexts' yield latency; see Table 3.2. We make three observations. First, a 50*ns* delay represents a best case for user thread's yield latency in BF2, and we find the throughput in such a case would degrade by 24-29%. This represents a best case since user threads [102] take 52*ns* to yield in x86 cores. Second, a more realistic user thread yield latency in BF2 would be about 200*ns*. This would correspond to 4× the x86 latency, similarly to the BF2-pthread-yield latency being 4.5× the pthread-x86-yield latency (we measured both). In such a case, throughput would degrade by 1.87-2.27×. Finally, the 1500*ns* delay corresponds to the cost of using pthread yields and this would degrade throughput by 6.8-9.8×. Therefore, coroutines enable high WIC and memory access throughput by providing very fast yielding contexts. Alternative logical threads with higher yield latencies would lead to reduced throughput.

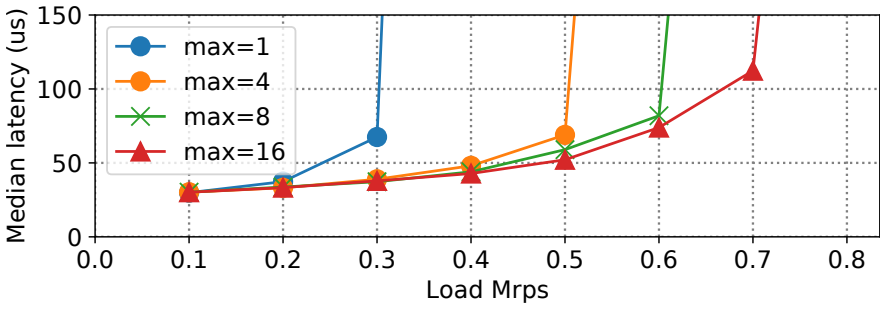

Figure 3.12: 4-node linked-list traversal WIC latency under load using 1 SmartNIC core and 4 QPs while varying the memory access maximum batch size.

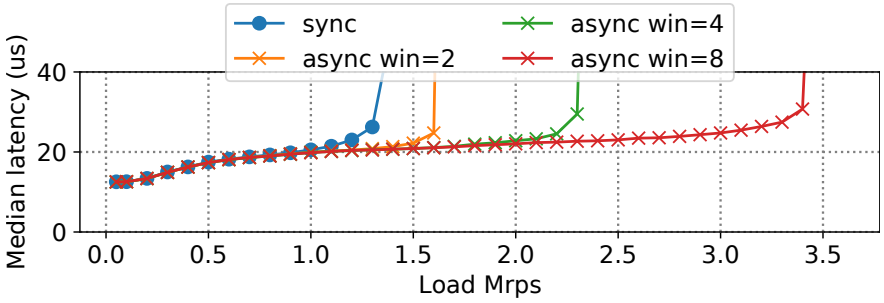

Figure 3.13: Local memory backends comparison for server location.

RDMA Dynamic Unsignaled Batching

We evaluate our batching design described in §3.4.1. Traditionally, systems that batch observe increased latency as elements within a batch wait for longer before being submitted for processing. Thus, here we explore the effect different batch sizes have on latency under load for 4-node linked-list traversal WICs. As our batching is dynamic, a maximum batch size of m only guarantees that no batch will be larger than m, but does not guarantee that all batches will be of size m.

Counter intuitively, as Figure 3.12 shows, a batch size of 16 improves latency under load (*e.g.* at 0.3Mrps) relative to no batching. In addition, max=16 improves throughput relative to max=1 by 2.3×. This is for three reasons: First because larger batch sizes reduce communication frequency between CPU and the RDMA adapter. Second, due to the dynamic nature of our batching, accesses within a batch stay pending for short enough time such that median latency does not increase. And third, because given our use of intra-batch unsignaled accesses, signaling a completion of a large batch costs the same for the RDMA adapter, and for the CPU, than a completion for a smaller batch.

Local Memory Backends

We evaluate our two local memory backends described in §3.4.1 with our linked-list traversal workload. sync refers to the synchronous backend that runs WICs to completion, and async win=x refers to the asynchronous backend using a window of size x. As shown in Figure 3.13, we find that async achieves higher throughput than sync without affecting latency at low loads. In particular, a window of 8 provides 2.6× higher throughput than sync. However, we also observe that larger windows increases latency at high loads.

3.7 Related Work

Accessing remote state. Many existing systems enable applications to access and modify state on a remote server. Some systems such as FaRM [40], Pilaf [98], and DrTM [126] achieve this by executing memory-access logic on clients using 1-sided RDMA. Other systems such as FaSST [71], HERD [68], and eRPC [67] execute memory-access logic on servers using 2-sided RDMA or messages over commodity network hardware. Finally, DrTM+H [125], XStore [124], and Octopus [88] combine both 1-sided and 2-sided RDMA in order to implement a specific application, such as a distributed transaction system or a key-value store. However, none of these approaches provide a general abstraction that enables arbitrary applications to execute at different locations without modification; this is what WICkit aims to do.

More expressive RDMA. Several prior works have attempted to bridge the gap between 1-sided and 2-sided RDMA by augmenting 1-sided RDMA with more expressive operations, such as chained operations. These efforts propose implementing more complex operations directly in NIC hardware [3,26,38], in FPGAs [80,113], or in SoC-based SmartNICs [5], or extending the IOMMU to trigger handler functions on the CPU [22].

WICkit expands on the proposal in [5] by designing a programming model and runtime system, and showing that these can be used to execute memory-access logic at different locations throughout a datacenter. WICkit can be deployed today on existing hardware, and does not require deploying additional FPGAs or SmartNICs or waiting for new NICs to be developed. In addition, if these new technologies are adopted, future work could explore compiling WICs to execute on them as well.

SmartNICs. In recent years, many systems have explored utilizing SoC-based, FPGA-based, or ASIC-based SmartNICs in distributed applications. These systems typically focus on offloading application logic [43, 56, 87, 103] or network processing [47, 61, 101, 105] to a server-attached SmartNIC. Other systems have proposed using SmartNICs for load balancing [35] or disaggregated storage [97]. These efforts focus primarily on offloading computation from servers to SmartNICs, and none provides a general framework that allows application logic to be executed at different locations throughout a datacenter, as WICkit does.

Shipping compute. Several prior works have studied questions of when to move compute to memory or storage and when the alternative—shipping data to compute—is preferable. For example, near-memory compute [114] considers CPU architectures that push computation to memory,

AIFM enables users to register specific code to execute on remote memory servers [108], and other work proposes adding compute capabilities to SSDs [39].

Splinter [23, 76] proposes a system that enables transactions to either execute at a datastore or be "pushed back" to execute on clients instead. It is designed to ensure serializability and targets latencies in the 100s of microseconds. In contrast, WICkit focuses on accessing memory rather than storage, with target latencies of 10s of microseconds. This imposes new challenges in designing a high-performance runtime, and yields a different runtime design (§3.4). Kayak [130] builds on Splinter to dynamically adjust what fraction of requests are executed locally vs. on a remote storage device; in the future, WICkit may be able to adopt a similar approach to dynamically decide which location to run WICs at.

Computation migration. Prior work has also looked at migrating computation running in distributed shared memory (DSM) clusters [24,62]. These efforts are similar to ours in that they focus on developing mechanisms that allow memory-access logic to run at different locations, and operate on a common pool of application memory. However, these works assume that all processors access memory using a common DSM API. By contrast, our setting requires the use of different logic for accessing application memory from different locations. Additionally, the difference in latencies also requires us to assume synchronous accesses at some locations and asynchronous accesses at others. As we showed in §3.6, not doing so can carry a performance penalty. As a result, approaches to migrating computation in DSM clusters cannot directly be used to build WICs.

Prior work on code mobility [27, 50] also looked at moving computation between devices that access application memory using different APIs. However, these works focused on moving computation between servers and mobile devices, and target very different performance requirements. Therefore, these techniques cannot be directly used for WICs.

3.8 Conclusion

This chapter studies WICkit, a framework, and runtime for Where-Independent Code. WICs allow distributed applications to choose their remote memory access execution location flexibly at application start-up time while enabling new locations such as SmartNICs. We demonstrate the low cost of the WIC abstraction by showing that WICkit's client and server locations achieve performance comparable to previous location-specific systems.

Chapter 4

Cluster Far Memory

In this chapter we describe CFM, a networked system that allows existing jobs to access far memory transparently. CFM includes a fast swapping mechanism and a far memory-aware job scheduler than enable far memory support at rack scale. In contrast to WICkit in the previous chapter, CFM only uses one-sided RDMA to access remote memory without the need for a server CPU.

The rising popularity of in-memory workloads such as machine learning applications and key-value stores is causing memory demands in compute clusters to grow rapidly [20]. At the same time, because of the end of Moore's law, DRAM manufacturers are struggling to achieve higher storage densities and lower per-storage-unit costs [72,78]. Taken together, these two developments cause main memory to increasingly be the bottleneck when operating compute clusters [42,77].

Memory disaggregation, which has been the subject of both academic research [2, 52, 57, 73, 85, 86, 111] and commercial projects [10, 32, 33], is one way to address the memory bottleneck, as it allows compute nodes to access memory at remote nodes; we will call this *far memory*. While far memory does not reduce the total amount of memory needed to run individual jobs, nor does it make memory cheaper or more dense, far memory does mean that jobs need not be restricted to *local* memory but instead can utilize memory that is located elsewhere in the cluster. This works around the "memory capacity wall" [85] and increases the extent to which memory can be shared efficiently across jobs.

While there has been previous work investigating how far memory can impact individual jobs [57], there has been limited work on how effective far memory is in actually increasing job throughput, or equivalently, in reducing job makespans for a cluster workload: that is the issue we address in this chapter. There are many ways one can support far memory, including making the local memory in each server available for remote use. However, for specificity and simplicity, we consider one particular model of memory disaggregation: using one or more "memory servers" to support all far memory, while all other servers in the cluster use their memory to support local jobs.

There are two main barriers to making far memory practical. The first barrier lies in how one designs the swapping mechanisms needed to access far memory, as existing systems [52, 57, 84] that swap over RDMA suffer from poor latency and throughput due to head-of-line blocking, and to handling interrupts and page reclamation on the critical path of page fault resolution. We have

designed a Linux swap system, called **Fastswap**, that is optimized to use far memory through RDMA. Like other swap systems, it is transparent both to applications and developers. In addition, it interacts directly with Linux control groups [9], allowing Fastswap to enforce local memory allocations. Fastswap prevents head-of-line blocking by steering page fetches on the critical path to a separate queue. Further, it reduces delays on the critical path by polling for completions of critical page operations and by offloading memory reclaim to a dedicated CPU. As a result, Fastswap achieves remote page access latencies of $< 5us$, enabling applications to access far memory at 10 Gbps with one thread, and 25 Gbps with multiple threads. Infiniswap [57] is the closest related work, and Fastswap's bandwidth is 1.51× (with one thread) and 2.54× (with multiple threads) higher than Infiniswap with disabled backup disk.

The second barrier lies in how one decides how to split each job's memory demands between local memory and far memory. The use of far memory is, to some extent, a bin-packing problem: how do you process a workload most quickly with a given amount of local memory on each server and a large pool of remote memory, and where each job must be assigned a total amount of memory (local and remote) that fully satisfies its requirements? To this end, we designed a **f**ar memory-aware cluster scheduler that leverages far memory to improve job throughput. When a new job arrives, the scheduler can place the job on a server that initially has insufficient available local memory to handle all jobs assigned to it. Our scheduler then reduces the local memory used by some of the existing jobs on that server, and uses far memory to ensure that all jobs have access to enough total memory. It is far from clear that such a strategy is beneficial, as using far memory inevitably slows down individual jobs (since accessing far memory is significantly slower than accessing local memory). However, using far memory can also enable more jobs to simultaneously run on a single server, albeit more slowly, which might increase overall throughput. We have studied this trade off extensively, and report on when the use of far memory increases overall throughput, and how this compares to merely increasing the amount of local memory. To the best of our knowledge, this is the first systematic exploration of these questions.

The combination of the improved swap system Fastswap and the cluster scheduler provides support for cluster-wide far memory, which we call CFM. While we do not have a large devoted cluster available to us, we used a cluster simulator (validated with runs on a real nine-node cluster) to explore what happens on a rack of 40 servers. We find that far memory is not a panacea; if the memory demands are substantially larger than available memory, then better performance is achieved by increasing the local memory per-server rather than by adding an equivalent amount of memory to a shared far memory server. However, we find that far memory provides significant benefits in two key scenarios on a single rack: (1) If the workloads are memory intensive (*i*.e., memory availability rather than core availability is the bottleneck), converting a compute node into a far memory server can result (for the case we studied) in roughly 10% improved throughput compared to the original rack, even though *both rack configurations have the same amount of total memory*. (2) If an operator wishes to moderately increase memory capacity in a rack, adding memory to a memory server allows for finer granularity increases which still result in significant performance improvements, whereas upgrading the local memory in each server can only be done in much larger (and therefore more expensive) increments (as we discuss in the next section).

We have made available our Linux kernel modifications and drivers, far memory-aware sched-

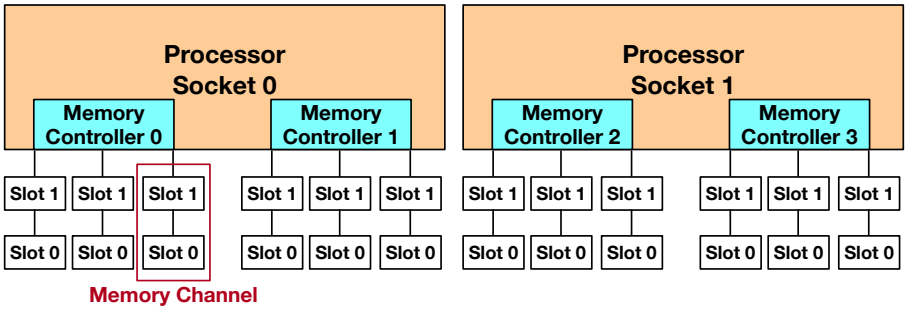

Figure 4.1: An example server platform with two sockets; each has two memory controllers, six memory channels, and twelve memory DIMM slots [30].

uler, and simulator at https://github.com/clusterfarmem.

4.1 Context

As memory requirements for datacenter applications increase steadily, memory comprises an ever larger fraction of the costs of operating a datacenter [77, 78, 89]. As a result, cluster operators face difficult choices in how to provision memory in their datacenters. In this section we provide context about memory provisioning, and state the assumptions of the deployment scenarios we consider.

4.1.1 Memory Provisioning

It is important to remember that local memory can only be provisioned at a coarse granularity. As an example, consider the memory configuration of a recent 2-socket Intel platform [30]. As Figure 4.1 shows, each socket has 2 memory controllers, each memory controller has 3 channels, and each channel has 2 slots for memory modules, or DIMMs, making a total of 24 possible DIMM slots. DIMM sizes are typically powers of 2: 4 GB, 8 GB, 16 GB, and so on. However, one cannot efficiently use arbitrary numbers and combinations of DIMMs in the same server. Because of the way that processors interleave memory accesses, configurations that are "unbalanced" yield significantly lower memory bandwidth. For example, using 1 DIMM on each memory controller yields only 35% of the maximum system memory bandwidth [8, 30, 48].

The guidelines for balancing memory to achieve full memory bandwidth are the following: (1) all memory channels should have the same total memory capacity and (2) all memory controllers should have the same configuration of DIMMs (number and sizes of DIMMs [1]) [30]. Therefore, in Figure 4.1, all slot 0s must have the same memory capacity and all slot 1s must have the same memory capacity.

[1] In addition, all populated memory channels should have the same number of ranks. A rank is a block of memory and memory controllers interleave accesses to different ranks.

Achieving balanced configurations limits the granularity at which memory can be initially provisioned, and also constrains the granularity at which memory can be upgraded if memory requirements change over time. For example, if we originally installed 192 GB of memory with twelve 16 GB DIMMs (one per channel), the smallest memory upgrade we could perform would be to add twelve 4 GB DIMMs (one per channel), representing a 25% increase in total memory capacity. If we then wanted to increase our memory capacity again at a later point, the smallest memory upgrade would involve *discarding* the 4 GB DIMMs and replacing them with 8 GB DIMMs (purchasing 40% more memory to increase our capacity by 20%). Worse, if we had originally installed 192 GB of memory by filling all twenty-four DIMMs in our platform with 8 GB DIMMs, the next memory upgrade would require us to discard 96 GB and buy twelve 16 GB DIMMs, resulting in 288 GB, a 50% upgrade from the original.

Thus, if you can only provision memory per-machine, the constraints on memory configurations require that you operate at significantly reduced memory bandwidth (which results in unacceptable performance) or else only increase your memory capacity at a very coarse granularity, yielding over or under-provisioning as memory demands change gradually over time [107, 120]. If we measure the granularity of memory upgrades in memory per core in the cluster, far memory can be upgraded at much finer granularity than local memory, despite obeying the same memory balancing constraints, because the added memory is shared by a much larger number of cores.

4.1.2 Deployment Scenarios

In this chapter we are not primarily considering green-field deployments where an operator can assess their workload's memory requirements and then determine the most cost-effective way to meet them. Our results do shed some light on green-field scenarios – in particular showing that for memory-intensive workloads on a single rack, it is better to convert one compute server into a far memory server – but our main focus is on incrementally upgrading existing deployments. Based on what we have heard from operators (these are not operators of hyperscale datacenters, but more reflective of smaller commercial datacenters found in many businesses), their existing datacenters tend to have all DIMM slots currently filled. This makes economic sense, because the cost per unit of memory increases with DIMM capacity, so the cheapest way to provision a given amount of memory is to use all available DIMM slots.

Considering a single rack that has been provisioned this way, how should an operator respond when the memory demands of their workloads exceed what the rack can accommodate? As described above, upgrading the memory in all the servers in a rack can be done only at a coarse granularity (in terms of the ratio of memory to cores), whereas far memory can be added at a finer granularity. Our goal, in this chapter, is to explore how these options compare.

To the best of our knowledge, datacenter operators ranging from small to medium prefer upgrading whole fleets at once. We do not consider the case where local memory is upgraded on a subset of the rack for two reasons: (1) we believe it would be challenging to manage memory-heterogeneous racks for operational reasons, and (2) we believe far memory would be better at avoiding memory over and under-provisioning due to static partitioning of memory across machines.

While far memory has the potential to improve cluster throughput, this comes at the cost of slower runtimes for individual jobs. As such, we believe that far memory is best suited for applications whose primary metric is job throughput, rather than customer-facing or latency-sensitive applications; that is our focus in this chapter.

4.2 CFM Overview

The goal of CFM is to enable clusters to improve their end-to-end job throughput by leveraging far memory on dedicated memory servers. Though prior work has explored mechanisms to enable far memory and has demonstrated performance benefits of individual jobs when they swap (e.g., [57]), to the best of our knowledge, no prior work has demonstrated performance improvements with far memory at rack scale. We focus on improving end-to-end makespan, or the time it takes to finish executing a list of jobs. At a high level, CFM's approach (§4.2.1) bears similarities to prior work, but CFM overcomes several key challenges (§4.2.2) that make it difficult to reap cluster-scale benefits from far memory today.

4.2.1 Approach

In CFM, applications utilize far memory via swapping over RDMA. CFM enforces memory allocations using Linux control groups.

Swapping. Applications can leverage far memory in broadly two ways: transparently (without application modifications), or via explicit and potentially custom APIs [1,40,99,118]. Though specialization has the potential to offer better performance, large-scale cluster operators have claimed that specialization is impractical [77], as their clusters execute thousands of heterogeneous workloads, and porting them to a different memory hierarchy would take significant effort. Instead, similar to Infiniswap [57], CFM realizes far memory with swapping, an existing mechanism that extends virtual memory beyond physically available memory. When a CPU accesses a memory address in a page that is not present in physical memory, a page fault is raised and the page fault handler transparently fetches the contents of the page from swap space into local memory. Traditionally swap space resided on disk and the resulting millisecond-scale access latencies induced large and poorly understood performance overheads on workloads. However, swapping itself does not fundamentally entail millisecond-scale latencies, and with today's microsecond-scale network latencies, swapping to far memory over the network has the potential to yield good performance.

Cgroups. CFM enforces per-job limits on local memory consumption using Linux control groups (cgroups) [9]. Cgroups control the amount of physical memory allocated to a group of processes and CFM uses the swap system to keep the excess in far memory.

RDMA. CFM leverages RDMA for low-latency access to memory on remote servers. CFM uses one-sided *read* and *write* operations, which enable access to memory without using the remote CPU. In general, RDMA operations are submitted to local *queue pairs* and are then processed by the local RDMA NIC. Once an operation has completed, the NIC posts a *completion* to a *completion queue*; completion queues can be configured to raise interrupts when completions arrive, or to

remain silent with the expectation that they will be polled. Traditionally RDMA bypasses both the remote and local operating systems, but RDMA also offers a kernel API for drivers to use; CFM leverages this API for swapping pages over the network.

4.2.2 Challenges and Contributions

There are two main challenges to realizing cluster-scale benefits from far memory: enabling far memory to be swapped in quickly (§4.2.2) and deciding how to schedule jobs across local and far memory (§4.2.2).

Fast Swapping

RDMA swap devices have been explored in prior works such as Infiniswap [57] and HPBD [84]. However, these approaches are unable to sustain the high performance required by applications today, for three main reasons:

1. To hide the I/O latency of future page faults, operating systems typically implement page prefetching by fetching several pages on each page fault. Unfortunately, in Linux the *faulted page*—the page currently required by the application—may lie anywhere in the aligned window of pages to be prefetched. Existing systems fetch all pages using a single queue pair per CPU (or worse, a single queue pair for the whole swap system [84]), so the faulted page may queue behind prefetch pages. Processing each page to fetch can take a few microseconds due to memory allocation, so with Linux's default prefetch window size of 8, head-of-line blocking may delay fault handling by tens of microseconds.

2. In existing systems that swap over RDMA, the CPU is notified that an RDMA operation (e.g., a read of a remote page) has completed via interrupts. This interrupt-handling occurs on the critical path—before the page fault handler is able to return to the application—and can add 10 μs or more [19] to page fault handling.

3. After the contents of a faulted page are read into local memory, the operating system charges the new page to its cgroup by increasing its memory counter. If the cgroup memory limit is exceeded, excess pages need to be reclaimed. In contrast to system-wide reclaim in Linux, memory reclaim in cgroups is always done directly, that is, before leaving the page fault handler and returning to the application. Thus the entire process of reclamation (finding pages to reclaim, writing them to the swap device, and returning the pages to the kernel for reuse) delays page fault resolution.

CFM introduces a faster swapping system called Fastswap (§4.3) that overcomes all three of these challenges, enabling CFM to achieve lower latency and higher throughput for remote swap than existing systems such as Infiniswap (§4.5.4).

Cluster Scheduling

Many existing schedulers enable efficient sharing of cluster resources by scheduling jobs across cores, memory, and other resources [53–55, 59, 90, 109, 120]. However, existing schedulers do not consider far memory; that is, they do not provide support for scheduling jobs whose memory can be dynamically split across local and far memory and they do not specify how to best allocate local memory across multiple jobs sharing the same machine. CFM proposes a centralized far memory-aware scheduler (§4.4) that considers far memory when assigning jobs to machines and decides how to partition local memory across different jobs in order to optimize makespan.

4.3 Fastswap

In this section we describe Fastswap, our RDMA swap system. Figure 4.2 shows its overall architecture and how it improves upon existing components of the operating system. While previous research efforts [52, 57, 84] simply expose their RDMA backend as a swap device, we found that enabling higher swapping performance required modifications to the page fault handler, swap system, and the cgroup memory controller. We implemented Fastswap by modifying approximately 300 lines of kernel code, and with a new device driver in 1200 lines of code for Linux 4.11.

Improving paging performance is challenging. While many systems focus on making improvements at millisecond time scales [58, 81], our system strives to enable microsecond-scale swapping. Most of the mechanisms we discuss in this section occur while program execution is *paused*. Therefore, every microsecond we save is a microsecond of compute time given back to the application.

4.3.1 RDMA Backend

In Fastswap, the operating system interacts with the RDMA NIC using the RDMA backend. As shown in Figure 4.2, the backend is used by all swap operation types: page faults, prefetches, and memory reclaim. While prior research has exposed an RDMA backend as a block device [57, 84], Fastswap uses the *Frontswap* interface [7]. Frontswap is designed for swapping at page granularity rather than supporting general block I/O operations, and strives to minimize context-switches to other tasks while swap operations complete.

Queue pairs. RDMA requests in a given queue pair are processed in-order by NIC Processing Units [70]. If different classes of swap operations share a queue, critical operations—*e.g.*, reads for faulted pages and writes for evicted pages—will queue behind less urgent prefetch reads. Fastswap avoids this head-of-line blocking by using two RDMA queue pairs per CPU, one for operations on the critical path and one for prefetches. Separating these operations into two queue pairs enables Fastswap to handle their completions differently. Our RDMA backend configures interrupt completions for prefetches, and disables them for critical operations (indicated by *int* and *poll* in Figure 4.2); Fastswap polls for completions of critical operations instead.

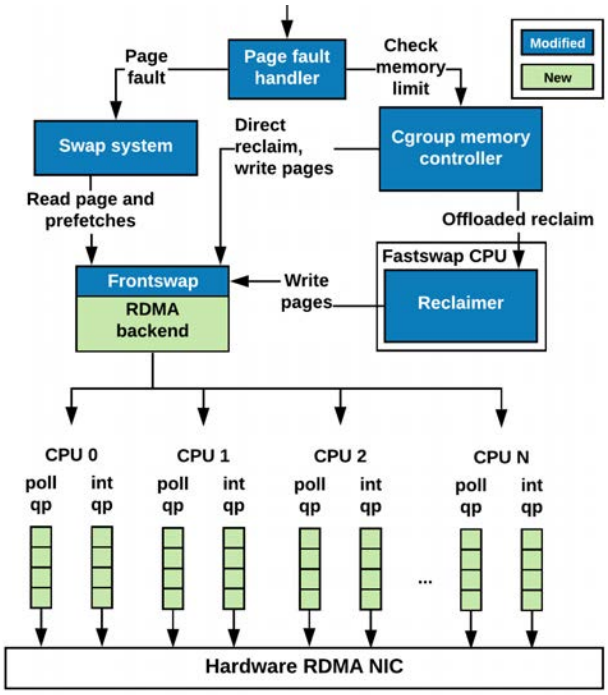

Figure 4.2: Architecture of Fastswap.

Frontswap interface. Frontswap assumes its operations complete synchronously [7], that is, control of execution returns to the swap system only after the Frontswap operation has completed. Therefore, it provides no mechanism to distinguish between operations that are on the critical path and those that are not. Thus, Fastswap enhances the Frontswap interface to distinguish between critical and non-critical operations, enabling the RDMA backend to steer requests to their appropriate queue pair. In our modified Frontswap interface, both types of operations return immediately after initiating their RDMA request. The modified swap system then polls for completion of critical path operations, while non-critical path operations trigger interrupts on completion.

4.3.2 Page Fault Handler

Fastswap modifies the page fault handler in two key ways. First, it instructs the swap system to handle faulted pages and prefetched pages differently, as described above. Second, Fastswap modifies the swap system to *first* read the faulted page, followed by the remaining pages within the prefetch window (a related approach is proposed in [45]). After issuing all reads, Fastswap poll waits for the faulted read to complete. By issuing the faulted read first, we overlap the latency of allocating physical pages for prefetch reads and the latency of issuing the prefetch RDMA reads, with the RDMA read for the faulted page. Figure 4.3 shows how Fastswap services a page fault and associated prefetches.

Handling faulted pages and prefetched pages separately minimizes the cost of missed prefetches.

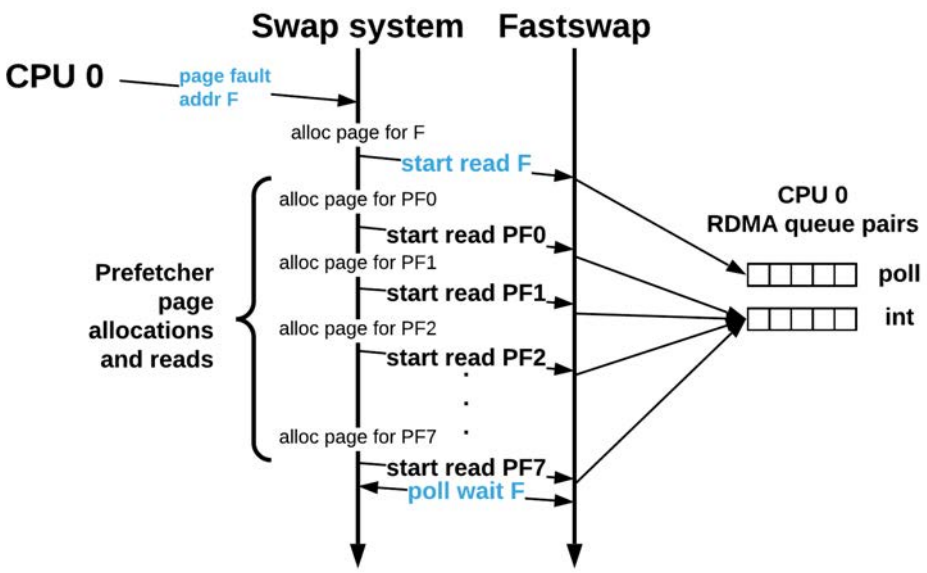

Figure 4.3: Page fault and prefetching with Fastswap.

For example, assume page fault 1 occurs on address F_1, with an associated set of prefetch pages P_1. Our swap system will issue reads for F_1 and P_1, and poll until the read for F_1 completes. At this point, the page fault handler will return to user space. Then, suppose page fault 2 occurs on address F_2 where $F_2 \notin P_1$ (*i*.e., a prefetch miss). Fastswap can fetch F_2 without waiting for any page in P_1, whereas previous systems would need to wait for all pages in P_1 before the F_2 read could finish [57, 84].

4.3.3 Memory Reclaim

We have covered how the Fastswap swap system brings pages from far memory into local memory. Now we describe how Fastswap reclaims memory so that processes do not use more local memory than their allowed share. Fastswap moves reclamation off the critical path of page fault handling by modifying the cgroup memory controller.

In general, memory reclaim is needed when memory in a cgroup grows beyond its allowed limit, or conversely, when a cgroup's memory limit shrinks. In Fastswap, memory in a cgroup increases when page faults bring pages from far memory into local memory, or when a process allocates additional memory. Meanwhile, memory limits shrink when our far memory-aware scheduler chooses to carve out memory to fit additional processes (see §4.4).

Traditionally, after reading a faulted page, the memory controller charges the page to its cgroup. Then, the controller checks if the cgroup has more memory than its allowed share. If there are excess pages, they are *directly reclaimed* and possibly evicted to far memory. Direct reclaim takes place in the context of the page fault handler, so it prevents the CPU from returning to user space and continuing workload execution.

As we show in §4.5.4, memory reclaim in Linux is surprisingly expensive, consuming 62-85% of the kernel time when our applications have 50% of their memory in far memory. To reduce these costs, whenever a node is using far memory, our modified memory controller offloads memory reclaim to a dedicated reclamation CPU (Figure 4.2); we call this process *offloaded reclaim*. Offloading memory reclaim allows the CPU that caused a page fault to return to user space without spending time on direct reclaim. Recent efforts have used a similar approach for offloading cold memory compression [77] and packet processing [36, 92] to dedicated CPUs.

Offloaded reclaim is not suitable for all situations, *e.g.*, to reclaim pages in response to large memory allocations, or in response to a large limit shrink. In these situations, offloaded reclaim can become a bottleneck because the reclaimer is shared across CPUs.

To prevent cgroups from significantly exceeding their memory allocation, Fastswap gives each cgroup a small threshold α of memory above its limit. When a cgroup first reaches its memory limit, the memory controller requests offloaded reclaim. If the reclaimer is busy and cannot service reclamation requests fast enough, memory in the requesting cgroup will keep increasing. Once the cgroup exceeds its memory limit by α, page fault handlers for the cgroup will perform direct reclaim as well. This guarantees that the cgroup does not exceed its limit by more than α. In our implementation we use α =8MB. If a node is not using far memory, the reclaimer is idle, so the CPU can be used to execute jobs.

Regardless of whether memory reclaim is direct or offloaded, when evicting pages to far memory, we poll for their completions. A page can be fully reclaimed only after its write to far memory finishes. At this point, the cgroup memory allocation decreases and the kernel can reuse the page. Since memory reclaim is done in batches, using interrupts for writes would delay observable reclaim and could cause more page reclaims than required. With polling, the cgroup memory counter decreases immediately after the write completes.

4.4 Far Memory-Aware Scheduler

In this section we describe how CFM makes scheduling decisions for a cluster equipped with far memory. Our scheduler uses bin-packing to allocate memory between jobs, and gains greater flexibility by allowing jobs to use far memory in addition to local memory. Intuitively, this can improve job throughput by allowing each node to fit additional jobs when memory is the constraining resource. However, the use of far memory slows down individual jobs, and as a result it is unclear how this impacts the overall makespan.

We define the memory request mem_i of job i to be the maximum it uses during execution; if allocated mem_i of memory, job i would not incur any hard page faults. Given a set of jobs and their CPU and memory requirements, there is a maximum number of jobs we can fit onto a single server. By using cgroups and CFM, we can rebalance the local memory that jobs use in a node, and free up enough local memory to fit additional jobs. When a cgroup is shrunk, pages are evicted to far memory such that memory of the cgroup does not exceed the new limit. In general, the cost of shrinking is the slowdown jobs experience, and the network bandwidth and latency to move memory from local to far memory.

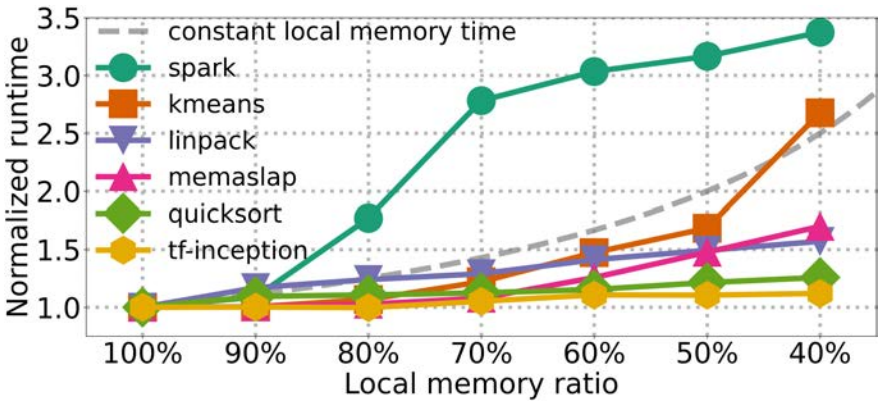

Figure 4.4: Performance degradation of applications from Table 4.1 using Fastswap. The constant local memory time line depicts A=B (§4.4.2).

4.4.1 Job Degradation Profiles

The performance degradation applications experience when they trade local memory for far memory is application dependent (see §4.5.1 for details on the applications we use). Figure 4.4 shows how the runtime of several jobs (each normalized by its runtime with 100% local memory) increases as we decrease the local memory ratio, or the fraction of the job's memory that is local. Some applications such as tensorflow-inception experience very little slow down (at most 10.5%) when using far memory, while others experience significant slowdowns (*e.g.*, spark's runtime triples when using 40% far memory).

Therefore, a scheduler that uses far memory cannot treat every job the same—it needs additional information that allows it to model the job slowdown in order to optimize workload makespan. Thus, for each application that we use in the rest of the chapter, we create a *degradation profile* [2] that estimates the runtime $f_i(r_i)$ at different local memory ratios r_i. To create a degradation profile, we measure the application's runtime at several discrete local memory ratios using Fastswap, and then use polynomial fitting to create a continuous function that the memory policy uses (explained in the next subsection).

Using job profiles has limitations; in particular, applications must finish so their profiles can be computed, and applications must have similar performance degradation when using far memory across different executions. It may be possible to use page fault frequencies to model job slowdown instead of pre-computed profiles; we leave exploring this to future work.

4.4.2 Far Memory Scheduling Policies

Our scheduler is simple and follows conventional designs [120], except when dealing with memory. When new jobs arrive, they are added to a pending queue which keeps jobs in arrival order.

[2] We note that our degradation profile is similar to Miss Ratio Curves [122], except in MRC the y-axis is miss ratio, whereas in our profile it is job execution slowdown.

Whenever the pending queue is not empty, the scheduler tries to assign jobs to nodes in the job-arrival order. For each job, the scheduler iterates through all nodes in order to find one that has a sufficient number of cores and memory available for the job. We iterate through the nodes in random order to improve the average runtime for finding such a node, and the job is placed on the first such node found. If no node has sufficient resources to execute a job, we leave it in the pending queue. In this work we do not consider job preemption.

When scheduling jobs, the scheduler needs to make two decisions. First, in the loop above it must decide if a job "fits" on a node (*i*.e., perform job admission control); in order to do so, we rely on a `fit` function provided by the memory policy. The `fit` function begins by checking if sufficient cores are available to execute the job; if not, we declare that the job does not fit on the node. If sufficient cores are available, we use a heuristic to determine whether sufficient local memory can be made available at the node, and enough free far memory remains to execute the job. If so, we say that the job fits on the node; otherwise, we say that it does not.

Second, once a node with sufficient resources has been found, we need to decide how much local memory to give the job. This is done by the `rebalance` function provided by the memory policy. The `rebalance` function revises the memory allocations of jobs executing on the node, so it is called *before a new job is started* to free enough local memory for the new job, and *after a job finishes*, in order to distribute local memory back across the remaining jobs. `rebalance` does not use far memory unless it needs to.

Next, we discuss several memory policies that we considered in terms of the `fit` and `rebalance` functions they provide.

Uniform policy. When a set of jobs on a machine requires more memory than is available locally, this policy shrinks all jobs uniformly up to a minimum ratio α. For example, if $\alpha = 0.75$, then the policy will trade up to 25% of local memory for far memory on every job. This policy uses this minimum ratio both to determine if jobs fit and to rebalance memory allocations. Although simple, this policy does not take into account the fact that different jobs slowdown differently when they are shrunk to the same ratio. In addition, the same ratio will mean different amounts of far memory depending on how much memory each job uses.

Variable policy. This policy improves upon uniform by allowing per-job minimum ratios. We chose minimum ratios for jobs that correspond to a 20% slowdown—we empirically determined 20% to be a good trade off between job slowdown and improved makespan. The `fit` function returns true if 1) there is enough memory on the machine for all jobs, including the incoming job, to have at least the amount of local memory specified by their minimum ratio, and 2) the cluster has sufficient far memory for the residuals. The `rebalance` function adjusts memory allocations by reducing local memory proportionally for each job according to its minimum ratio. Therefore, the minimum ratio of each job determines its friendliness for using far memory. Rebalancing jobs proportionally based on their minimum ratios requires users to define it for each job. Since this policy scales local memory linearly for each job, it performs best if performance degradation is also linear up to the minimum ratio.

Memory-time policy. Using insights gained from the previous policies, we designed a policy that directly captures the fact that jobs that use far memory experience nonlinear slowdowns. As

a result, rather than relying on manually specified minimum ratios, this policy makes use of the memory-time product, which we explain next, to determine the best local memory ratio for each job in a node when `rebalance` is called.

Given a set of memory-constrained jobs, the asymptotic makespan for these jobs is given by [54, 55, 120]:

$$makespan \approx \frac{memorytime}{local\_mem \cdot utilization}$$

where *memorytime* is the sum of all the jobs' memory requirements multiplied by their runtime, *local_mem* is the total available local memory in the cluster, and *utilization* is the average utilization of local memory in the cluster. Intuitively, the product of memory requirement and runtime captures how much memory a job consumes during its execution. Without far memory, *memorytime* is fixed because the amount of local memory a job uses and its duration are both fixed. The total local memory in the cluster is similarly fixed. Therefore, previous research on scheduling to lower makespan, or improve throughput, could only focus on increasing the denominator by improving *utilization* [54].

With far memory, we can decrease local *memorytime* and increase *utilization* to further improve makespan. Since we use far memory only when local memory is fully utilized, *utilization* is commonly very high. In addition, *memorytime* is no longer fixed at $\sum_{i=1}^{N} mem_i \cdot f_i(1)$ but is instead flexible:

$$memorytime = \sum_{i=1}^{N} mem_i \cdot r_i \cdot f_i(r_i)$$

As we increase far memory usage, the product $r_i \cdot f_i(r_i)$ depends on how gracefully a job's performance degrades. Figure 4.5 conceptually shows A, the original memory-time product for a job using only local memory, and B+C, its new memory-time product when using far memory (*i*.e., B is the local memory-time product while C is the far memory-time product). As long as the area of B is less than A, a job's local *memorytime* can be reduced by trading some of its local memory for remote memory. When this is not the case, using additional far memory would increase a job's local memory-time, so our scheduler must not reduce a job's memory ratio below this point. As illustrated in Figure 4.4 with the *constant local memory-time* curve, for many jobs the slowdown is graceful enough that they can be shrunk significantly without reaching the point where A and B have the same area. For example, we could shrink local memory at least 60% for all six workloads except for spark and linpack. Therefore, our memory-time policy can reduce local *memorytime* by finding appropriate local memory ratios for each job. For example, for $r_i = 0.5$, if $f_i(0.5) < 2 \cdot f_i(1)$, then we reduce the local memory-time product because we save half of the job's local memory while incurring less than twice the slowdown.

We turn this insight into a better local memory rebalancing policy by considering three factors. First, to optimize for makespan, we should pick local memory ratios for each job to minimize the sum of their local memory-time products (i.e., the sum of the A's of existing jobs in the machine). If we had unlimited far memory, this would be sufficient. However, given that we have a limited amount of far memory that is shared by several machines we also need to ensure efficient allocation

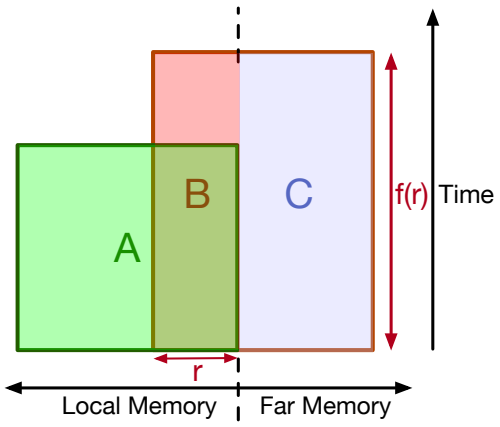

Figure 4.5: How a job can reduce its local memory consumption by using far memory. A is the original memory-time product when no far memory is used. B+C is the new memory-time product, where B is the local portion, and C is the far portion of the product. r is the local memory ratio of the job.

of far memory. Unfortunately, optimizing memory allocations over all local and far memory is at least APX-Hard [128] and thus not feasible in our setting.

To resolve this, CFM optimizes each node independently by choosing local memory ratios for jobs such that they maximize the ratio between the savings in local memory-time products and the increase in far memory usage. This avoids global optimization while still using local and far memory efficiently. We do this by solving the following optimization problem:

$$\begin{aligned}
&\underset{r_i : i=1,\ldots,N}{\text{maximize}} && \frac{A-B}{C} \\
&\text{subject to} && \sum_{i=1}^{N} mem_i \cdot r_i = local_mem \\
&A-B && = \sum_{i=1}^{N} mem_i \cdot (1-p_i) \cdot f_i(1) - mem_i \cdot (1-p_i) \cdot r_i \cdot f_i(r_i) \\
&C && = \sum_{i=1}^{N} mem_i \cdot (1-p_i) \cdot (1-r_i) \cdot f_i(r_i)
\end{aligned}$$

where p_i is a ratio between 0 and 1 that represents the progress of this job according to its profile, and $(1-p_i) \cdot f_i(r_i)$ is the remaining run time for job i using local memory ratio r_i. Therefore $A-B$ is the total local memory-time product saving, C is the total far memory-time product, and the equality constraint ensures that local memory is fully utilized.

Each node tracks the progress of its own jobs (updating each p_i when the local memory ratios change) and solves the optimization problem when `rebalance` is executed. The dimension of this optimization is the number of jobs running on the node, and it converges in a small number of iterations using conventional optimization tools; *i*.e., we found that SciPy [121] was fast enough for our scheduler and our simulator.

The memory-time policy does not prescribe how to determine whether or not a job fits on a machine; *i*.e., it is not involved in job admission. For simplicity, we use the same `fit` function as the variable policy. We leave more advanced admission control schemes (*e.g.*, [54]) for future work.

Although CFM does not primarily target latency-sensitive production workloads, it is flexible enough to exempt these from using far memory. We can do so by specifying a minimum admission memory ratio of 100% for the job in the `fit` function, and by giving the job a profile where $f(1) = \text{runtime}$, and $f(\text{anything else}) = \infty$. Note that this way, other jobs can still trade local memory for far in order to fit the far memory-exempt job. We leave for future work analysis and evaluation of these scenarios; in this chapter we assume every workload can be slowed down.

4.4.3 Scheduler Implementation

Our scheduler is comprised of a central scheduler and a per-node daemon. We implemented both in 1500 lines of Python. The central scheduler implements the design outlined in §4.4.2, and uses gRPC to communicate with all daemons. When the scheduler dispatches jobs to a daemon, the daemon creates a cgroup with a memory limit for it— the memory limit is defined by the memory policy being used. Often, the memory limit is smaller than the amount of memory the job needs, which triggers far memory usage. The scheduler can be configured to not use far memory at all; in this case, the scheduler propagates this configuration to the daemons so they will not use far memory.

4.5 Evaluation

Our evaluation of CFM focuses on three main questions:

1. How does CFM perform in a real testbed? (§4.5.2)
2. What are the benefits of far memory and when should one use far memory instead of local memory? (§4.5.3)
3. How do CFM's individual components contribute to its overall performance? (§4.5.4)

4.5.1 Experimental Setup

We evaluate CFM on a small testbed rack and in simulation at rack scale using a cluster simulator.

Testbed rack. Our testbed consists of 14 machines; we use 9 as compute nodes, 1 as the scheduler, and up to 4 as memory servers. Each machine has an 8-core Intel Xeon E5-1680 v2 CPU, 32 GB of memory and a 40 Gbps Mellanox ConnectX-3 NIC. We use one hyperthread on each core and disable TurboBoost and CPU frequency scaling in order to reduce variability. Each machine runs Ubuntu 16.04 with Linux kernel version 4.11. Each job runs in its own cgroup; memory and core allocations are decided by the CFM scheduler.

Cluster simulator. We implemented our own cluster simulator in 2200 lines of Python. It implements the same `fit` and `rebalance` functions as our testbed implementation. Our simulator takes as input a degradation profile for each job produced by using Fastswap (§4.4.1), and uses these to determine how long each job takes to complete, given the memory resources allocated to it. If the cluster scheduler changes a job's local memory allocation during its execution, our simulator adjusts the job's remaining runtime accordingly. Since our profiles are generated when applications are executed individually, our simulation results represent a best case performance analysis of using far memory. We evaluate the accuracy of our simulator in §4.5.3. Our simulated cluster consists of 1 rack of 40 machines, each with 48 cores and 192 GB of memory.

Evaluated Systems

We evaluate four systems. First, we evaluate a baseline cluster that uses only local memory, denoted by **NOFAR**. To understand the benefits of additional local memory, we use **NOFAR (+X%)** to denote the configuration in which each machine in the baseline cluster has been augmented with X% additional local memory. Second, we evaluate **Infiniswap** [57], a system that enables applications to leverage remote memory on other servers (rather that in a dedicated memory server), via RDMA. Infiniswap requires writing to disk to correctly function in a cluster. However, we found that writing to disk severely degrades its performance, so we include a modified version where we disabled writing to disk. Infiniswap does not specify how to schedule jobs, so we only evaluate it in single-job experiments. Third, we implemented a Fastswap **DRAM** backend for comparison and to understand how swapping operations are impacted by RDMA performance.

Finally, we evaluate CFM using the memory-time policy unless specified otherwise. We use **FAR (+X%)** to denote the configuration in which the cluster runs CFM and has been provisioned with far memory such that the cluster's total memory is X% more than in NOFAR. When evaluating any FAR configuration, we use one fewer server per rack than in NOFAR to ensure that there is available space in the rack for a memory server to support far memory.[3] In addition, when far memory is in use, CFM dedicates at least one core per server to running the reclaimer. We found that one core was sufficient for the reclaimer on our 8-core testbed machines; based on its utilization (39.5% on average), we estimate that our larger simulated 48-core machines require 3 reclaimer cores and thus we give three fewer cores per simulated server for CFM.

Jobs and Workloads

In the experiments that follow, a *job* can be any of the applications described in Table 4.1. We focus on applications that can benefit from cluster throughput improvements such as analytics applications. We now describe each application in more detail. linpack is a linear algebra performance benchmark, and we use an Intel provided binary which we limit to use 4 CPUs [11]. quicksort uses the C++ standard library to sort 8GB of integers. kmeans uses sklearn to classify 15M samples [13]. tensorflow-inception does inference on an inception reference implementation

[3] Because the CFM configurations use one fewer server, FAR (+0%) has one server's worth of far memory, to yield the same total rack memory as NOFAR.

Benchmark	Memory (GB)	# cpus
linpack	1.56	4
quicksort	8.05	1
kmeans	4.73	1
tensorflow-inception	2.07	2
memcached	12.00	2
spark-pagerank	4.29	3

Table 4.1: Applications that comprise our workloads.

for benchmarking [14]. memcached uses memaslap to SET 30M keys and then query 100M keys using the ETC distribution (*i*.e., 5% SET, 95% GET), while memcached is pinned to another CPU [16, 57]; memcached could be used as a parameter server in this context. spark-pagerank uses a dataset of 685K nodes and 7.6M edges [79].

A *workload* is a list of 6000 mixed jobs with uniformly random arrivals. Every workload has at least one instance of each job from Table 4.1. We characterize workloads using two properties: **m2c** and **packability**.[4] m2c captures how a workload's demand for memory relative to compute compares to that of the underlying cluster. For a workload W with N jobs and a cluster C with C_{cpu} cores and C_{mem} GB of memory, we define $m2c$ as:

$$m2c(W,C) = \frac{\sum_{j=1}^{N} mem_j \cdot duration_j}{\sum_{j=1}^{N} cpu_j \cdot duration_j} \cdot \frac{C_{cpu}}{C_{mem}}$$

For example, a workload that consists of jobs that require the full memory but only half the cores of any machine in the cluster has an m2c of 2. To produce workloads with a given range of m2c values, we randomly generate many workloads by varying the ratio of each job in the workload, and then select those that have an m2c within the given range.

Packability captures how easily a workload's jobs can be scheduled in a cluster without using far memory. We define the packability metric for a workload and cluster as the makespan achieved when all of a cluster's resources (memory and cores) are pooled into one large server (eliminating any resource fragmentation),[5] divided by the makespan achieved in the default NoFar configuration. Thus, a packability of close to one indicates that a workload suffers little from resource fragmentation; as fragmentation increases, packability decreases.[6]

4.5.2 Testbed Performance

We use our testbed to evaluate how far memory behaves in real executions using our far memory-aware scheduler (§4.4) and Fastswap (§4.3). In this subsection, Far (+0%) is an 8-node rack with

[4]Both of these properties depend on the infrastructure on which the workload is run. Since we use a particular rack configuration as a baseline, we refer to these two measures without specifically mentioning the infrastructure.

[5]This pooling approach mimics the upper bound described in [54].

[6]Note that packability is defined with respect to a certain cluster scheduling algorithm; we assume jobs are scheduled as described in §4.4.2.

m2c	NOFAR	FAR (+0%)	FAR (+11%)	FAR (+33%)
1.0	1.00	1.05	1.04	1.07
1.2	1.00	1.12	1.08	1.10
1.4	1.00	1.07	1.12	1.11
1.6	1.00	1.15	1.21	1.28

Table 4.2: Makespan *improvement* in testbed normalized to NOFAR (*i*.e., 9 node cluster without far memory).

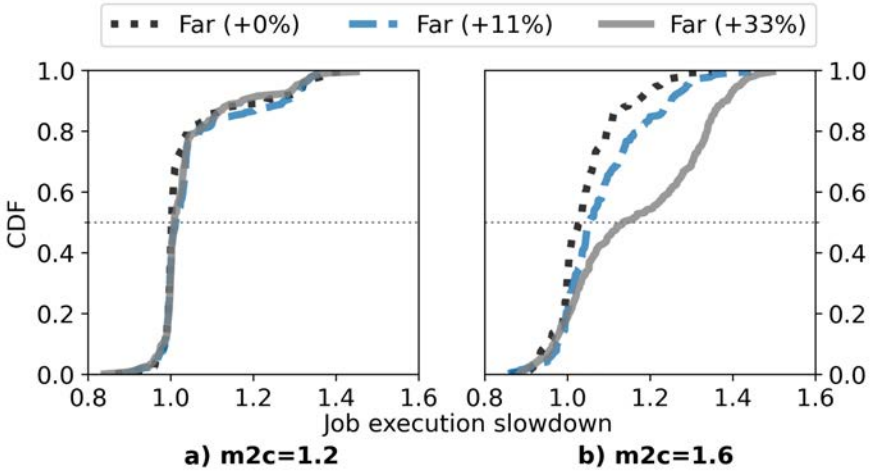

Figure 4.6: Job execution time slowdown when using far memory, relative to execution time in NOFAR, in our testbed. Left shows workloads with $m2c = 1.2$ and right shows workloads with $m2c = 1.6$.

32 GB of far memory, and NOFAR is a *9-node* rack without far memory. We further explore the performance of FAR by adding 32 GB and 96 GB of memory to the far memory server, yielding the FAR (+11%) and FAR (+33%) configurations. We evaluate workloads that have m2c between 1.0 and 1.6, with a granularity of 0.2; in each case, we picked a single workload with an m2c close to the prescribed value.

Table 4.2 shows the makespan improvement over NOFAR. We make three key observations from these results. First, for memory-constrained workloads, using far memory reduces makespan. Most notably, with the same amount of total memory, FAR (+0%) outperforms NOFAR in throughput by 5-15%. Second, far memory helps more when workloads have a higher m2c. In the extreme case, FAR (+33%) outperforms NOFAR by 28%. Third, additional far memory does not always lead to better performance. The reasons are two-fold: (1) Additional memory can help only when memory is the constraining resource. Therefore, when workloads have a low m2c, a small amount of far memory can sufficiently mitigate contention over memory, so additional far memory provides minimal benefits. (2) As we observe from FAR (+11%) and FAR (+33%) for an m2c of 1.4, adding more far memory can slightly degrade performance. We believe this is due to our overeager

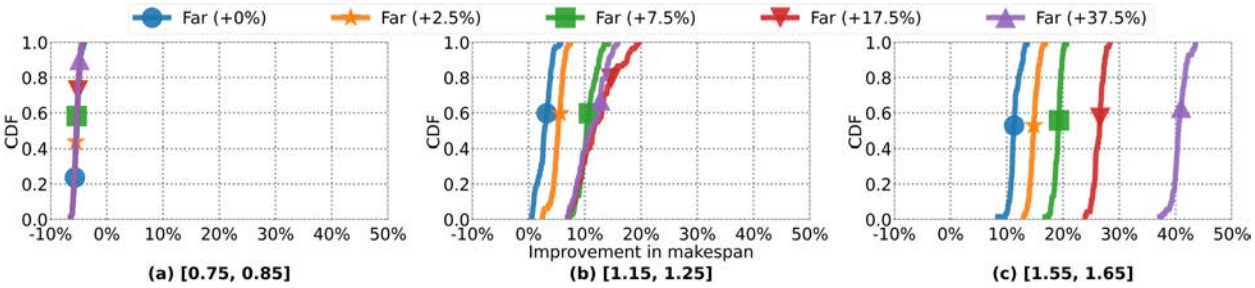

Figure 4.7: The percent improvement in workload makespan, relative to the NOFAR configuration, for workloads with three different ratios of memory to compute (m2c).

admission policy; *i*.e., memory-time uses the variable policy for admission control (see §4.4.2). In other words, it is possible that a job is admitted using a large fraction of far memory, while a better decision would have been to wait and admit the job later using less far memory, resulting in lighter slowdown. We leave the design of a better admission control policy as future work.

While CFM improves cluster throughput, it does increase the execution time of individual jobs. Figure 4.6 shows the CDF of execution time slowdown when using far memory relative to the same job's execution time in NOFAR, in our testbed. When the workloads are lightly memory constrained (*i*.e., $m2c = 1.2$), using far memory slows down jobs by 0.2%–1.1% at the median, and by 35%–37% at the 99$^{\text{th}}$ percentile. As the workloads become more constrained by available memory, our scheduler can use more far memory at the expense of job execution time. As such, when $m2c = 1.6$, using far memory slows down jobs by 2%–13.4% at the median, and 28%–45% at the 99$^{\text{th}}$ percentile.

4.5.3 Rack-scale Evaluation

We use our simulator to evaluate the benefits of far memory for a full rack of 40 machines.

Simulation Validation

To validate that our simulator accurately emulates the behavior of our testbed experiment, we simulated the executions presented in Table 4.2. We found that our simulated makespans ranged from 9% less than to 3% more than the actual makespans measured in the testbed.

Benefits of Far Memory

We quantify the benefits of far memory in a rack by simulating many workloads with different amounts of far memory. Each workload has 6000 mixed jobs from Table 4.1, and lasts for an average of 107 minutes of simulated time. For this experiment, we consider three different ranges of m2c values: $[0.75, 0.85]$, $[1.15, 1.25]$ and $[1.55, 1.65]$; each range includes 260 different workloads and each makespan is the average of 15 trials. Every workload is simulated without far memory

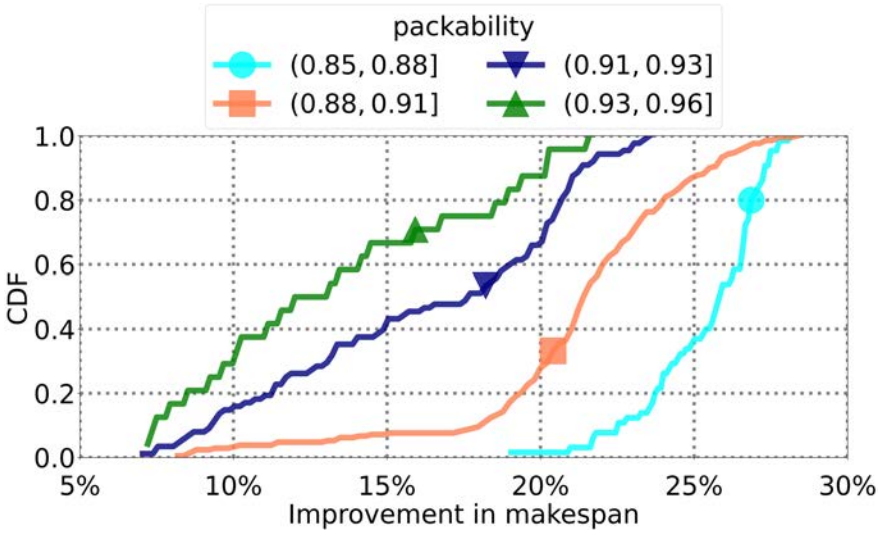

Figure 4.8: The impact of packability on far memory's ability to improve makespan with FAR (+17.5%) for workloads with $\mathbf{1.15 < m2c \leq 1.65}$.

(*i*.e., NOFAR) and with different amounts of added far memory (*i*.e., FAR (+X%)); we compute the percent improvement in makespan, relative to NOFAR, for each far memory configuration.

In Figure 4.7, each CDF shows the distribution of workload makespan improvements for a given m2c range and amount of far memory. Figure 4.7a demonstrates that for workloads that are on average CPU-constrained rather than memory-constrained (m2c values between 0.75 and 0.85), adding far memory imposes a small performance penalty of 6% at the median and 7% at the 99$^{\text{th}}$ percentile. This is because with far memory the rack has one fewer server, and if far memory is used, each server dedicates 3 cores to handle memory reclamation leaving fewer cores for jobs.

For workloads that are on average slightly or significantly memory-constrained, adding far memory provides benefits for all workloads and amounts of far memory. Even for the configuration with the smallest amount of far memory (*i*.e., 192 GB, or FAR (+0%)), which replaces a compute server with a memory server with an equivalent amount of far memory, makespan improves at the median by 3% and 11%, as Figure 4.7b and 4.7c show. Even though this configuration has *the same total amount of memory and many fewer cores* per rack than NOFAR, it achieves lower makespans because the presence of a sharable far memory allows jobs to be packed onto machines that have available cores but not memory, thereby enabling resources to be used more efficiently.

Adding additional far memory continues to provide further benefits. For m2c values in $[1.15, 1.25]$, improvements relative to NOFAR plateau at a median of 12% with FAR (+17.5%); for m2c values in $[1.55, 1.65]$ improvements continue until our largest far memory point FAR (+37.5%) with a median improvement of 47%. We expect improvements to plateau because our memory policy prevents jobs from being admitted if doing so would require shrinking any job beyond its minimum ratio. Further, when a previously memory-constrained node has enough far memory, memory stops being the constraining resource, CPUs become the limiting resource, and additional far memory

provides no benefit.

While we explicitly controlled the m2c ratio in our workloads, packability (as defined in §4.5.1) arises from the difficulty of packing jobs from each workload in the rack. Figure 4.8 shows improvement in makespan for FAR (+17.5%), relative to NOFAR, for a wide range of m2c values ($[1.15, 1.65]$). It illustrates the effect that packability has on throughput improvement when far memory is available. As packability decreases, workloads are harder to pack in NOFAR, so far memory is able to provide more benefit; the median improvement at $(0.93, 0.96]$ is 12.5% while the median improvement at $(0.85, 0.88]$ is 25.8%. Thus we observe that both m2c and packability can significantly impact the makespan improvements a workload achieves from far memory.

Adding Far Memory vs. Local Memory

We now evaluate system performance when adding far memory compared to adding memory locally to each machine, *i*.e., NOFAR (+X%). We aim to answer the question "If I purchase X GB of memory, should I add $\frac{X}{N}$ GB to each machine or X GB to a shared memory server?" We again consider different ranges of m2c values. For each range and memory configuration, Figure 4.9 plots the median percent improvement in makespan, relative to NOFAR.

When we add memory locally to each server, we can only do so in a few discrete amounts dictated by the current memory configuration and standard DIMM sizes (§4.1.1). For our simulated rack where each machine has 192 GB of memory, we consider two ways of initially provisioning each machine: (1) with 12×16 GB DIMMs, and (2) with 24×8 GB DIMMs. The smallest feasible upgrade for case 1 involves purchasing 48 GB per server, for an additional rack memory of 1.92 TB and a total rack memory of 9.6 TB. The smallest feasible upgrade for case 2 requires purchasing 12×16 GB=192 GB and *discarding* 12×8 GB=96 GB per machine, resulting in an additional 3.84 TB of memory in the rack and a total purchased memory of 15.36 TB (including the discarded memory). We simulate these two upgrade options and illustrate them in Figure 4.9 with the stars at x=9.6 TB (NOFAR (+25%)) and x=15.36 TB (NOFAR (+50%)), respectively.

We see that overall it is better to add memory locally on every machine, rather than add the same total amount of memory as far memory, in terms of the makespan. For example, when $m2c > 1.15$ and 9.6 TB of memory are given to the rack, NOFAR (+25%) has a makespan that is lower by 2.5% on average across the different m2c ratios, relative to FAR (+25%) (see $x = 9.6$ TB in Figure 4.9). Similarly, when we add 3.84 TB to the rack, doing so locally results in a makespan that is lower by 9.7% on average, relative to FAR (+50%), shown by the right-most set of stars vs. the right-most dots. However, with far memory we can add memory at *finer granularity*. Thus, if we do not want to invest the money required to add memory to all machines in a rack, we can reap much of the performance benefits by adding memory to a far memory server instead at a fraction of the cost. For example, for an m2c ratio in $[1.75, 1.79]$, we could achieve a median makespan improvement of 36% by adding 48 GB locally to each machine, or we could achieve 58% of that improvement by adding 30% as much far memory with FAR (+7.5%).

These results also have implications for how racks should be constructed in the future. Figure 4.9 demonstrates that for $m2c > 1.15$, the smallest far memory configuration FAR (+0%) achieves an average of 9.3% improved makespan relative to NOFAR (+0%). This suggests that if

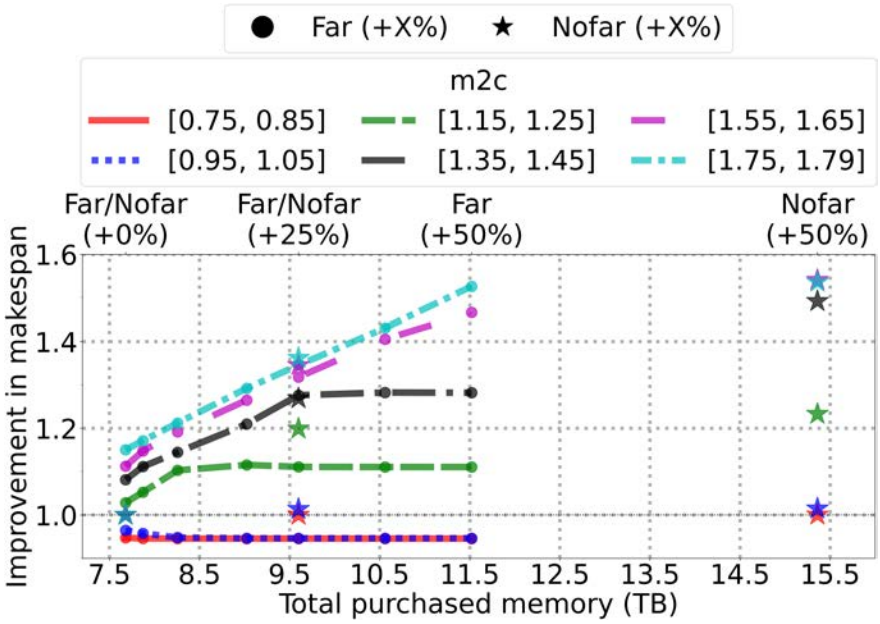

Figure 4.9: Makespan improvement as we add memory to the cluster in different ways, for workloads with different m2c values. Dots show addition of far memory, while stars are addition of local memory, for a given amount of purchased memory.

your workloads are even slightly memory-constrained when you build your rack, you will achieve better performance by equipping it with 39 servers and 1 memory server with an equivalent amount of memory, than by equipping it with 40 servers, despite the loss of one server's worth of cores. Furthermore, this design allows you to easily add more memory as workloads change in the future.

4.5.4 Microbenchmarks

We now use microbenchmarks to study CFM's performance and how it is impacted by Fastswap's design elements. The benchmarks in this subsection were executed in our testbed setup, unless noted otherwise.

Page Fetch Rate

In this benchmark, we measure how quickly Fastswap can fetch pages over the network. Our benchmark triggers page fetches as quickly as possible by performing memory reads that are strided by the size of a page (4 KB) such that each memory read causes a major page fault. To isolate the performance of fetches, we prevent evictions from occurring during the experiment. We run multiple instances of the benchmark process and pin each to its own core.

We evaluate Fastswap as well as a variant of Fastswap in which all page fetches (including those for the faulting page) raise interrupts on completions (Fastswap-interrupt-only). We also evaluate

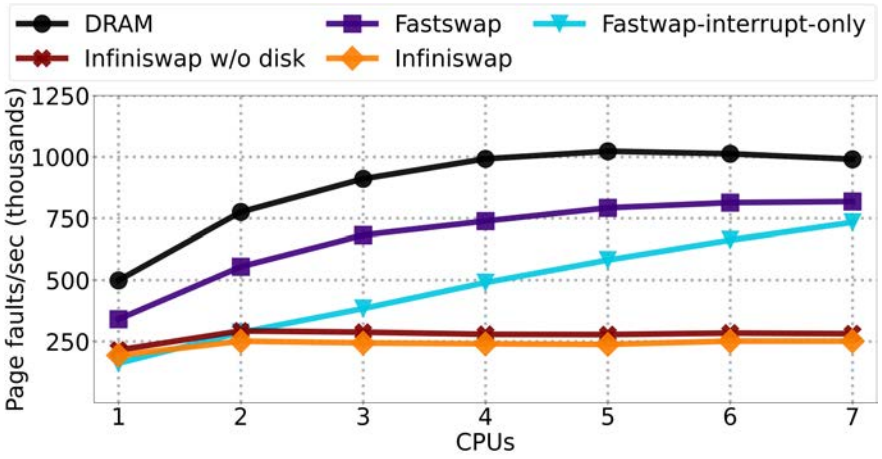

Figure 4.10: Page fetch rates supported by different swapping backends when eviction is disabled (prefetch set size is 8 pages).

Infiniswap, both with and without asynchronous writes to disk enabled, and a DRAM backend, which provides an upper bound on achievable performance using Linux's swapping mechanisms.

As shown in Figure 4.10, the DRAM backend's fetch rate scales sublinearly and achieves a peak page fault rate of 1.02M pages/sec with 5 cores. Fastswap can achieve up to 80% of this, peaking at 818,000 pages/sec with 7 cores. Even though DRAM can copy a page in <1us [110] compared to 3.9μs[7] for an RDMA read of 4KB, Fastswap achieves 80% of the fetch rate of DRAM because of its hybrid polling-interrupt mechanism to fetch pages. DRAM has no asynchronous mechanism: it copies all pages synchronously whether for prefetching or reading the faulting page. Meanwhile, Fastswap only waits for the faulting page, and handles prefetched pages via interrupts. Fastswap-interrupt-only demonstrates the benefits of polling for the fetched page; when all pages generate interrupts, it takes even longer to finish fetching the faulting page, leading to lower fetch throughput.

Infiniswap achieves significantly lower page fault rates, peaking at 320k pages per second with 6 cores (39% of Fastswap), even without writes to disk. Unfortunately these rates are insufficient to support our experiments. For our testbed experiments described in §4.5.2 we observed peak page access rates of 431k pages per second with $m2c$=1.6; therefore, Infiniswap would not be able to support them.

Cgroup Memory Bandwidth

In this experiment, we measure the memory bandwidth that cgroups using Fastswap provide to applications. We use STREAM, a well-known industry standard benchmark [94]. This benchmark performs operations over large regions of memory, triggering both fetches and evictions. We configure the benchmark to use 4 GB of memory. As Figure 4.11 shows, we set the percentage of local

[7] Average 4KB RDMA read latency measured in our testbed.

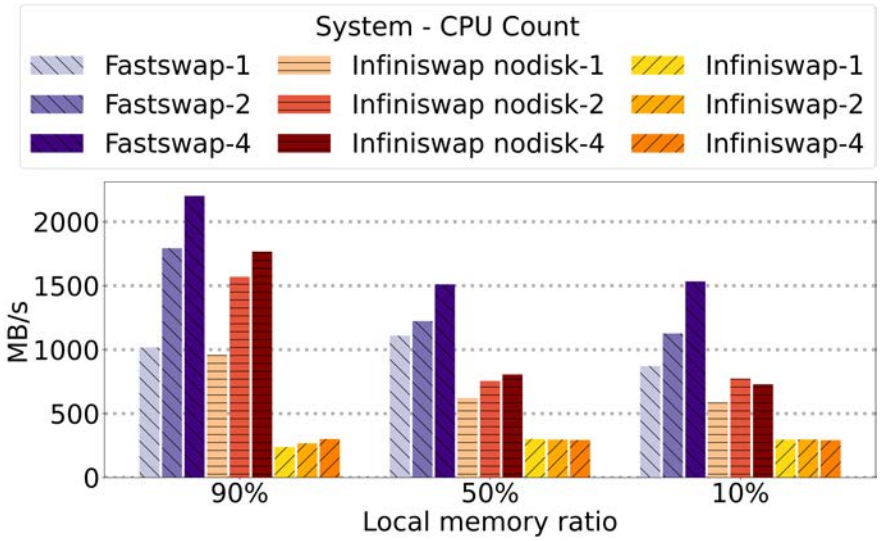

Figure 4.11: Memory bandwidth achieved by the STREAM benchmark for Fastswap and Infiniswap with different core counts and local memory ratios.

memory to 90%, 50% and 10%. We report the Triad component of the benchmark and measure the average bandwidth instead of the peak bandwidth. Fastswap provides 6–78% higher bandwidth than Infiniswap with 1 CPU and 25–110% higher bandwidth with 4 CPUs. At 100% local memory, 1 CPU achieves 13,867 MB/s and 4 CPUs achieve 32,253 MB/s, demonstrating that local memory can achieve 12.5× or 14.6× as much bandwidth as far memory, respectively. This highlights the importance of continuing research efforts on lowering swapping overheads to further improve its performance.

Memory Reclaimer

We evaluate the efficacy of the Fastswap reclaimer (§4.3.3) by measuring the percent of kernel time spent performing reclamation both *with* and *without* the reclaimer. We run each application with 50% local memory and measure the time spent doing reclamation. Table 4.3 shows that without the reclaimer, 61.8–84.9% of kernel time is spent performing direct reclaim. However, enabling the reclaimer reduces this significantly so that applications spend up to 85.3% less kernel time performing direct reclamation. The reason why kernel time reduction is not 100% is that the reclaimer cannot offload all memory reclaim when large memory allocations occur, *i*.e., larger than $\alpha = 8MB$ (see §4.3.3).

Memory Rebalance Policies

We use simulation to compare the three memory `rebalance` policies we introduced before: uniform, variable, and memory-time (§4.4.2). We simulate the execution of 1000 workloads following our setup for §4.5.3 with m2c values between 1.4 to 1.8 in a cluster that mimics our testbed cluster.

Workloads	% of kernel time	Kernel time reduction
linpack	61.8%	37.7%
quicksort	72.9%	30.8%
kmeans	74.3%	25.5%
tensorflow-inception	76.7%	85.3%
spark	84.9%	35.6%

Table 4.3: Fraction of kernel time spent on direct memory reclamation without the Fastswap reclaimer and the percentage reduction when the Fastswap reclaimer is used.

We use high m2c workloads to highlight the differences between memory `rebalance` policies, because this is where far memory provides the most benefit. We set the amount of far memory to 128 GB. Overall, memory-time always performs slightly better than variable, and variable always performs significantly better than uniform. The median makespan improvement of memory-time over uniform is 12.4%, and the median improvement of memory-time over variable is 2%.

We believe memory-time performs only slightly better than variable for two reasons. First, memory-time uses the variable policy for job admission; therefore, memory-time can improve performance relative to variable only by choosing better shrink ratios. Second, memory-time's improvements over variable arise from non-linearity in job degradation profiles, but most of the jobs in this experiment have close to linear degradation profiles. This is because at these high m2c values, most jobs are memory-intensive jobs such as quicksort and memcached, which have close to linear degradation profiles, rather than jobs like spark or kmeans whose profiles are less linear (see Figure 4.4). We believe that a greater diversity of job degradation profiles amongst the memory-intensive applications would yield larger makespan improvements for memory-time relative to variable.

4.6 Related work

Hardware resource disaggregation. The idea of *disaggregating* hardware resources in datacenters has gained popularity in recent years. As a result, recent work has considered how to adapt various components of datacenters to support disaggregation, proposing new hardware designs [12,15,29,44,73,85], operating systems [86,111], memory abstractions [1,118], and network stacks [34], and studying the requirements imposed on underlying networks [52]. CFM's scheduling policies and faster swapping mechanisms are complimentary to these efforts.

Far memory access. Several previous systems have used paging over a network to leverage remote memory [28,31,46,49,60,85]. More recent efforts such as HPBD [84] and Infiniswap [57] leverage RDMA to implement swapping over the network with lower latency. Though Fastswap also implements swapping over RDMA, it overcomes several challenges that limit the latency and throughput of swapping of these existing systems (§4.2.2). Another recent approach implements "far memory" by compressing cold pages and storing them locally in DRAM [77]. With this ap-

proach, the authors were able to store about 20% of their data compressed in DRAM. However, this approach has a limited ability to address the ever-increasing demands for memory. Fabric Attached Memory [74] proposes to use far memory without paging; however, to the best of our knowledge no publicly available implementation of hardware exists yet.

Cluster scheduling. Many existing cluster schedulers such as Decima [90], Tetris [54], and others [53, 55, 63] have considered how to pack jobs onto compute clusters in order to maximize for efficient use of cluster resources such as memory, CPU, disk, and network. Cluster managers such as Borg [120], Omega [109], YARN [119], and Mesos [59] schedule jobs across machines at a large-scale (e.g., thousands of machines), while also addressing issues such as failures and heterogeneous hardware. However, none of these approaches specify how to schedule jobs when their memory can be split across local and shared remote memory; we expect that Fastswap's policies could be incorporated into many of these schedulers.

4.7 Conclusion

This chapter studies the confluence of two trends: the increasing memory requirements of cluster workloads, and the emergence of memory disaggregation. We focus on two main questions: (1) can we develop fast swapping techniques and scheduling algorithms that make far memory feasible, and (2) can we characterize some scenarios where the use of far memory leads to reduced makespans for memory-intensive workloads. Our results suggest that the answer to both questions is "yes".

Bibliography

[1] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote regions: A simple abstraction for remote memory. In *USENIX Annual Technical Conference*, USENIX ATC '18, pages 775–787, 2018.

[2] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote memory in the age of fast networks. In *ACM Symposium on Cloud Computing*, SoCC'17, pages 121–127, 2017.

[3] Marcos K Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. Designing far memory data structures: Think outside the box. In *Workshop on Hot Topics in Operating Systems*, HotOS'19, pages 120–126, 2019.

[4] 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 *European Conference on Computer Systems*, EUROSYS'17, pages 1–16, 2020.

[5] Emmanuel Amaro, Zhihong Luo, Amy Ousterhout, Arvind Krishnamurthy, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Remote memory calls. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, pages 38–44, 2020.

[6] Amazon Web Service. Aws nitro system. https://aws.amazon.com/ec2/nitro/, 2021.

[7] Anonymous. Frontswap, Accessed 2019/09/7. https://www.kernel.org/doc/html/latest/vm/frontswap.html.

[8] Anonymous. Recommended intel xeon sp memory configurations, Accessed 2019/1/1. http://bladesmadesimple.com/wp-content/uploads/2019/06/Intel_Xeon_SP_Memory_Recommendations_v4.pdf.

[9] Anonymous. Cgroups v2, Accessed 2019/11/3. https://www.kernel.org/doc/Documentation/cgroup-v2.txt.

[10] Anonymous. The machine: A new kind of computer. https://www.hpl.hp.com/research/systems-research/themachine/, Accessed 2019/9/24.

[11] Anonymous. Intel optimized linpack benchmark for linux, Accessed 2020/3/5. https://software.intel.com/en-us/mkl-linux-developer-guide-intel-optimized-linpack-benchmark-for-linux.

[12] Anonymous. Intel rack scale design (intel rsd), Accessed 2020/3/5. https://www.intel.ca/content/www/ca/en/architecture-and-technology/rack-scale-design-overview.html.

[13] Anonymous. Machine learning in python, Accessed 2020/3/5. https://scikit-learn.org/stable/.

[14] Anonymous. Tensorflow benchmarks, Accessed 2020/3/5. https://github.com/tensorflow/benchmarks/.

[15] Krste Asanovic. Firebox: A hardware building block for 2020 warehouse-scale computers. In *Keynote presentation at the USENIX Conference on File and Storage Technologies*, FAST'14, 2014.

[16] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, SIGMETRICS'12, pages 53–64, 2012.

[17] Lewis Baker. C++ coroutines: Understanding operator co_await, Accessed 2021/12/08. https://lewissbaker.github.io/2017/11/17/understanding-operator-co-await.

[18] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobbler, Michael Wei, and John D Davis. Corfu: A shared log design for flash clusters. In *Symposium on Networked Systems Design and Implementation*, NSDI'14, pages 1–14, 2012.

[19] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, 2017.

[20] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. *The Datacenter as a Computer: Designing Warehouse-scale Machines*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2018.

[21] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. Ix: A protected dataplane operating system for high throughput and low latency. In *Symposium on Operating Systems Design and Implementation*, OSDI'15, pages 49–65, 2014.

[22] Maciej Besta and Torsten Hoefler. Active access: A mechanism for high-performance distributed data-centric computations. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 155–164, 2015.

[23] Ankit Bhardwaj, Chinmay Kulkarni, and Ryan Stutsman. Adaptive placement for in-memory storage functions. In *USENIX Annual Technical Conference*, ATC'20, pages 127–141, 2020.

[24] Daniel Bittman, Robert Soulé, Ethan L Miller, Vishal Shrivastav, Pankaj Mehra, Matthew Boisvert, Avi Silberschatz, and Peter Alvaro. Don't Let RPCs Constrain Your API. In *HotNets*, 2021.

[25] Tomash Brechko and Bob Jenkins. C library implementing cuckoo hash, Accessed 2021/12/08. https://github.com/kroki/Cuckoo-hash.

[26] Matthew Burke, Sowmya Dharanipragada, Shannon Joyner, Adriana Szekeres, Jacob Nelson, Irene Zhang, and Dan RK Ports. Prism: Rethinking the rdma interface for distributed systems. In *ACM Symposium on Operating Systems Principles*, SOSP'21, pages 228–242, 2021.

[27] Antonio Carzaniga, Gian Pietro Picco, and Giovanni Vigna. Is code still moving around? looking back at a decade of code mobility. In *29th International Conference on Software Engineering (ICSE'07 Companion)*, pages 9–20. IEEE, 2007.

[28] Haogang Chen, Yingwei Luo, Xiaolin Wang, Binbin Zhang, Yifeng Sun, and Zhenlin Wang. A transparent remote paging model for virtual machines. In *International Workshop on Virtualization Technology*, 2008.

[29] I-Hsin Chung, Bulent Abali, and Paul Crumley. Towards a composable computer system. In *International Conference on High Performance Computing in Asia-Pacific Region*, HPC Asia 2018, pages 137–147, 2018.

[30] Dan Colglazier, Joseph Jakubowski, and Jamal Ayoubi. Intel xeon scalable family balanced memory configurations, Accessed 2019/11/1. https://lenovopress.com/lp0742.pdf.

[31] Douglas E. Comer and James Griffioen. A new design for distributed systems: The remote memory model, March 1990. Technical Report 90-977. Purdue University, Department of Computer Science.

[32] CCIX Consortium et al. Cache coherent interconnect for accelerators (ccix), Accessed 2019/9/24. http://www.ccixconsortium.com.

[33] GenZ Consortium et al. Genz consortium, Accessed 2019/9/24. https://www.genzconsortium.org.

[34] Paolo Costa, Hitesh Ballani, Kaveh Razavi, and Ian Kash. R2c2: A network stack for rack-scale computers. In *ACM Special Interest Group on Data Communications*, SIGCOMM'15, pages 551–564, 2015.

[35] Tianyi Cui, Wei Zhang, Kaiyuan Zhang, and Arvind Krishnamurthy. Offloading load balancers onto smartnics. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 56–62, 2021.

[36] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCabooter, Marc De Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, and Amin Vahdat. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *Symposium on Networked Systems Design and Implementation*, NSDI'18, pages 373–387, 2018.

[37] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. When idling is ideal: Optimizing tail-latency for heavy-tailed datacenter workloads with perséphone. In *ACM Symposium on Operating Systems Principles*, SOSP'21, pages 621–637, 2021.

[38] Salvatore Di Girolamo, Andreas Kurth, Alexandru Calotoiu, Thomas Benz, Timo Schneider, Jakub Beránek, Luca Benini, and Torsten Hoefler. A RISC-V in-network accelerator for flexible high-performance low-power packet processing. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 958–971. IEEE, 2021.

[39] Jaeyoung Do, Sudipta Sengupta, and Steven Swanson. Programmable solid-state storage in future cloud datacenters. *Communications of the ACM*, 62(6):54–62, 2019.

[40] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. Farm: Fast remote memory. In *Symposium on Networked Systems Design and Implementation*, NSDI'14, pages 401–414, 2014.

[41] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *SOSP*, 2015.

[42] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing dram footprint with nvm in facebook. In *European Conference on Computer Systems*, EUROSYS'18, 2018.

[43] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. NICA: An infrastructure for inline acceleration of network applications. In *ATC*, 2019.

[44] Paolo Faraboschi, Kimberly Keeton, Tim Marsland, and Dejan Milojicic. Beyond processor-centric operating systems. In *Workshop on Hot Topics in Operating Systems*, HotOS'15, 2015.

[45] Viacheslav Fedorov, Jinchun Kim, Mian Qin, Paul V. Gratz, and A. L. Narasimha Reddy. Speculative paging for future nvm storage. In *International Symposium on Memory Systems*, MEMSYS'17, pages 399–410, 2017.

[46] Edward W. Felten and John Zahorjan. Issues in the implementation of a remote memory paging system, 1991. Technical Report. University of Washington, Department of Computer Science and Engineering.

[47] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking: Smartnics in the public cloud. In *Symposium on Networked Systems Design and Implementation*, NSDI'18, pages 51–66, 2018.

[48] Werner Fischer. Optimize memory performance of intel xeon scalable systems, Accessed 2019/11/1. https://www.thomas-krenn.com/en/wiki/Optimize_memory_performance_of_Intel_Xeon_Scalable_systems.

[49] Michail D. Flouris and Evangelos P. Markatos. The network ramdisk: Using remote memory on heterogeneous nows. *Cluster Computing*, 2(4):281–293, 1999.

[50] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding code mobility. *IEEE Transactions on software engineering*, 24(5):342–361, 1998.

[51] Shay Gal-On and Markus Levy. Exploring coremark a benchmark maximizing simplicity and efficacy. *The Embedded Microprocessor Benchmark Consortium*, 2012.

[52] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *Symposium on Operating Systems Design and Implementation*, OSDI'16, pages 249–264, 2016.

[53] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. Firmament: Fast, centralized cluster scheduling at scale. In *Symposium on Operating Systems Design and Implementation*, OSDI'16, pages 99–115, 2016.

[54] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In *ACM Special Interest Group on Data Communications*, SIGCOMM'14, pages 455–466, 2014.

[55] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *Symposium on Operating Systems Design and Implementation*, OSDI'16, pages 81–97, 2016.

[56] Stewart Grant, Anil Yelam, Maxwell Bland, and Alex C Snoeren. Smartnic performance isolation with fairnic: Programmable networking for the cloud. In *ACM Special Interest Group on Data Communications*, SIGCOMM'20, pages 681–693, 2020.

[57] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient memory disaggregation with infiniswap. In *Symposium on Networked Systems Design and Implementation*, NSDI'17, pages 649–667, 2017.

[58] Md E. Haque, Yong hun Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, and Kathryn S. McKinley. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'15, pages 161–175, 2015.

[59] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Symposium on Networked Systems Design and Implementation*, NSDI'11, pages 295–308, 2011.

[60] Michael R. Hines, Mark Lewandowski, and Kartik Gopalan. Anemone: Adaptive network memory engine, 2005. Technical Report TR-050128. Florida State University, Department of Computer Science.

[61] Torsten Hoefler, Salvatore Di Girolamo, Konstantin Taranov, Ryan E Grant, and Ron Brightwell. spin: High-performance streaming processing in the network. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16, 2017.

[62] Wilson C Hsieh, M Frans Kaashoek, and William E Weihl. Dynamic computation migration in DSM systems. In *Supercomputing*, 1996.

[63] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *ACM Symposium on Operating Systems Principles*, SOSP'09, pages 261–276, 2009.

[64] Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin Levandoski, and Gor Nishanov. Exploiting coroutines to attack the ̈killer nanoseconds. *Proceedings of the VLDB Endowment*, 11(11):1702–1714, 2018.

[65] Norman P Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. A domain-specific supercomputer for training deep neural networks. *Communications of the ACM*, 63(7):67–78, 2020.

[66] ISO/IEC JTC1/SC22/WG21. Standard for programming language c++, Accessed 2021/12/08. https://timsong-cpp.github.io/cppwp/n4861/dcl.fct.def.coroutine.

[67] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter rpcs can be general and fast. In *Symposium on Networked Systems Design and Implementation*, NSDI'19, pages 1–16, 2019.

[68] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using rdma efficiently for key-value services. In *ACM Special Interest Group on Data Communications*, SIGCOMM'14, pages 295–306, 2014.

[69] Anuj Kalia, Michael Kaminsky, and David G Andersen. Design guidelines for high performance rdma systems. In *USENIX Annual Technical Conference*, ATC'16, pages 437–450, 2016.

[70] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance rdma systems. In *USENIX Annual Technical Conference*, USENIX ATC '16, pages 437–450, 2016.

[71] Anuj Kalia, Michael Kaminsky, and David G Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs. In *Symposium on Operating Systems Design and Implementation*, OSDI'16, pages 185–201, 2016.

[72] Uksong Kang, Hak-Soo Yu, Churoo Park, Hongzhong Zheng, John Halbert, Kuljit Bains, S. Jang, and Joo Sun Choi. Co-architecting controllers and dram to enhance dram process scaling. In *The memory forum*, 2014.

[73] K. Katrinis, D. Syrivelis, D. Pnevmatikatos, G. Zervas, D. Theodoropoulos, I. Koutsopoulos, K. Hasharoni, D. Raho, C. Pinto, F. Espina, S. Lopez-Buedo, Q. Chen, M. Nemirovsky, D. Roca, H. Klos, and T. Berends. Rack-scale disaggregated cloud data centers: The dredbox project vision. In *Design, Automation and Test in Europe Conference and Exhibition*, DATE'16, pages 690–695, 2016.

[74] Kimberly Keeton. Memory-driven computing. In *Keynote presentation at the USENIX Conference on File and Storage Technologies*, FAST'17, 2017.

[75] Linux kernel. No_hz: Reducing scheduling-clock ticks, Accessed 2021/12/08. https://www.kernel.org/doc/Documentation/timers/NO_HZ.txt.

[76] Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman. Splinter: Bare-metal extensions for multi-tenant low-latency storage. In *Symposium on Operating Systems Design and Implementation*, OSDI'18, pages 627–643, 2018.

[77] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. Software-defined far memory in warehouse-scale computers. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'19, pages 317–330, 2019.

[78] Seok-Hee Lee. Technology scaling challenges and opportunities of memory devices. In *IEEE International Electron Devices Meeting*, 2016.

[79] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[80] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *ACM Symposium on Operating Systems Principles*, SOSP'17, pages 137–152, 2017.

[81] Jing Li, Kunal Agrawal, Sameh Elnikety, Yuxiong He, I-Ting Angelina Lee, Chenyang Lu, and Kathryn S. McKinley. Work stealing for interactive services to meet target latency. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP'16, 2016.

[82] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *Symposium on Operating Systems Design and Implementation*, OSDI'14, pages 583–598, 2014.

[83] Sheng Li, Hyeontaek Lim, Victor W Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G Andersen, O Seongil, Sukhan Lee, and Pradeep Dubey. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *International Symposium on Computer Architecture*, ISCA'15, pages 476–488, 2015.

[84] Shuang Liang, Ranjit Noronha, and Dhabaleswar K. Panda. Swapping to remote memory over InfiniBand: An approach using a high performance network block device. In *IEEE International Conference on Cluster Computing*, 2005.

[85] 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 *International Symposium on Computer Architecture*, ISCA'09, pages 267–278, 2009.

[86] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. System-level implications of disaggregated memory. In *IEEE Symposium on High-Performance Computer Architecture*, HPCA'12, 2012.

[87] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartnics using ipipe. In *ACM Special Interest Group on Data Communications*, SIGCOMM'19, 2019.

[88] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: an rdma-enabled distributed persistent memory file system. In *USENIX Annual Technical Conference*, ATC'17, pages 773–785, 2017.

[89] Yixin Luo, Sriram Govindan, Bikash Sharma, Mark Santaniello, Justin Meza, Aman Kansal, Jie Liu, Badriddine Khessib, Kushagra Vaid, and Onur Mutlu. Characterizing application memory error vulnerability to optimize datacenter cost via heterogeneous-reliability memory. In *International Conference on Dependable Systems and Networks*, DSN'14, pages 467–478, 2014.

[90] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *ACM Special Interest Group on Data Communications*, SIGCOMM'19, pages 270–288, 2019.

[91] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *European Conference on Computer Systems*, EUROSYS'12, 2012.

[92] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: A microkernel approach to host networking. In *ACM Symposium on Operating Systems Principles*, SOSP'19, pages 399–413, 2019.

[93] David Mazières. My tutorial and take on c++20 coroutines, Accessed 2021/12/08. https://www.scs.stanford.edu/~dm/blog/c++-coroutines.html.

[94] John D McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Technical Committee on Computer Architecture Newsletter*, 2(19–25), 1995.

[95] Mellanox. RDMA Aware Networks Programming User Manual. https://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf.

[96] Microsoft. Azure support for Linux RDMA. https://azure.microsoft.com/en-us/updates/azure-support-for-linux-rdma/, 2015.

[97] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. Gimbal: enabling multi-tenant storage disaggregation on smartnic jbofs. In *ACM Special Interest Group on Data Communications*, SIGCOMM'21, pages 106–122, 2021.

[98] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *USENIX Annual Technical Conference*, ATC'13, 2013.

[99] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *USENIX Annual Technical Conference*, USENIX ATC'13, pages 103–114, 2013.

[100] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily R. Blem, Hassan M. G. Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. In *ACM Special Interest Group on Data Communications*, SIGCOMM'15, 2015.

[101] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. AccelTCP: Accelerating network applications with stateful TCPoffloading. In *Symposium on Networked Systems Design and Implementation*, NSDI'20, 2020.

[102] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high cpu efficiency for latency-sensitive datacenter workloads. In *Symposium on Networked Systems Design and Implementation*, NSDI'19, pages 361–378, 2019.

[103] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: A programming system for NIC-accelerated network applications. In *Symposium on Operating Systems Design and Implementation*, OSDI'18, 2018.

[104] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 325–341, 2017.

[105] Yiming Qiu, Jiarong Xing, Kuo-Feng Hsu, Qiao Kang, Ming Liu, Srinivas Narayana, and Ang Chen. Automated smartnic offloading insights for network functions. In *SOSP*, 2021.

[106] CPP Reference. Coroutines (c++20), Accessed 2021/11/26. https://en.cppreference.com/w/cpp/language/coroutines.

[107] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *ACM Symposium on Cloud Computing*, SoCC'12, 2012.

[108] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. Aifm: High-performance, application-integrated far memory. In *Symposium on Operating Systems Design and Implementation*, OSDI'20, pages 315–332, 2020.

[109] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: Flexible, scalable schedulers for large compute clusters. In *European Conference on Computer Systems*, EUROSYS'13, pages 351–364, 2013.

[110] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. Rowclone: Fast and energy-efficient in-dram bulk data copy and initialization. In *IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 185–197, 2013.

[111] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. Legoos: A disseminated, distributed os for hardware resource disaggregation. In *Symposium on Operating Systems Design and Implementation*, OSDI'18, pages 69–87, 2018.

[112] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. Fast and concurrent rdf queries with rdma-based distributed graph exploration. In *Symposium on Operating Systems Design and Implementation*, OSDI'16, 2016.

[113] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. Strom: smart remote memory. In *European Conference on Computer Systems*, EUROSYS'20, pages 1–16, 2020.

[114] Gagandeep Singh, Lorenzo Chelini, Stefano Corda, Ahsan Javed Awan, Sander Stuijk, Roel Jordans, Henk Corporaal, and Albert-Jan Boonstra. Near-memory computing: Past, present, and future. *Microprocess. Microsystems*, 71, 2019.

[115] Mellanox Technologies. Nvidia mellanox bluefield-2 smartnic, Accessed 2021/09/12. https://www.mellanox.com/files/doc-2020/pb-bluefield-2-smart-nic-eth.pdf.

[116] Mellanox Technologies. Libibverbs programmer's manual, Accessed 2021/09/21. https://manpages.debian.org/testing/libibverbs-dev/ibv_wr_start.3.en.html.

[117] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the next generation. In *European Conference on Computer Systems*, EUROSYS'20, pages 1–14, 2020.

[118] Shin-Yeh Tsai and Yiying Zhang. Lite kernel rdma support for datacenter applications. In *ACM Symposium on Operating Systems Principles*, SOSP'17, pages 306–324, 2017.

[119] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *ACM Symposium on Cloud Computing*, SOCC'13, 2013.

[120] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *European Conference on Computer Systems*, EUROSYS'15, 2015.

[121] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng,

Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0–Fundamental Algorithms for Scientific Computing in Python. *arXiv e-prints*, July 2019.

[122] Carl A. Waldspurger, Trausti Saemundson, Irfan Ahmad, and Nohhyun Park. Cache modeling and optimization using miniature simulations. In *USENIX Annual Technical Conference*, USENIX ATC '17, pages 487–498, 2017.

[123] Xingda Wei, Rong Chen, and Haibo Chen. Fast rdma-based ordered key-value store using remote learned cache. In *Symposium on Operating Systems Design and Implementation*, OSDI'20, pages 117–135, 2020.

[124] Xingda Wei, Rong Chen, and Haibo Chen. Fast rdma-based ordered key-value store using remote learned cache. In *Symposium on Operating Systems Design and Implementation*, OSDI'20, pages 117–135, 2020.

[125] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing rdma-enabled distributed transactions: Hybrid is better! In *Symposium on Operating Systems Design and Implementation*, OSDI'18, 2018.

[126] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 87–104, 2015.

[127] Kyle Wiggers. Intel partners with google to deploy mount evans asic-based ipu, Accessed 2021/09/12. https://venturebeat.com/2021/10/27/intel-partners-with-google-to-deploy-mount-evans-asic-based-gpu/.

[128] Gerhard J. Woeginger. There is no asymptotic ptas for two-dimensional vector packing. *Information Processing Letters*, 64(6):293–297, 1997.

[129] Jilong Xue, Youshan Miao, Cheng Chen, Ming Wu, Lintao Zhang, and Lidong Zhou. Fast distributed deep learning over rdma. In *European Conference on Computer Systems*, EUROSYS'19, 2019.

[130] Jie You, Jingfeng Wu, Xin Jin, and Mosharaf Chowdhury. Ship compute or ship data? why not both? In *Symposium on Networked Systems Design and Implementation*, NSDI'21, pages 633–651, 2021.

[131] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. In *ACM Special Interest Group on Data Communications*, SIGCOMM'15, 2015.