UNIVERSITY OF CALIFORNIA SAN DIEGO

Distributing and Disaggregating Hardware Resources in Data Centers

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

in

Computer Science

by

Yizhou Shan

Committee in charge:

Professor Yiying Zhang, Chair
Professor George C. Papen
Professor Alex C. Snoeren
Professor Stefan Savage
Professor Geoffrey M. Voelker

2022

The dissertation of Yizhou Shan is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2022

DEDICATION

Dedicated to my family for their love and support.

EPIGRAPH

Don’t adventures ever have an end? I suppose not.

Someone else always has to carry on the story.

– The Fellowship of the Ring

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

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

First of all, I want to thank my advisor Professor Yiying Zhang for supporting and believing in me during this journey. In the beginning of my PhD, I had a hard time navigating research and life. I'm very thankful for Yiying's encouragement, trust, and patience during that period. I'm also very fortunate to work with Yiying on various disaggregation related projects, most of which are somewhat big and grand vision projects. This type of project is not easy to get through, but definitely a rewarding experience once done. Yiying's research taste, critical thinking ability, insights, and attention to detail always amazes me. Most importantly, she made me realize what type of a system researcher I want to be and also guided me to become one.

I'm thankful for the researchers and interns during my 2018 and 2019 VMware Research internships. I thank Stanko Novakovic and Marcos Aguilera for their mentoring and insights. I also want to thank Sanidhya Kashyap for showing me how knowledgeable and experienced a seasoned system PhD student could be. His dedication and humbleness encouraged me to become a better researcher. I also thank Yanfang Le, Rohan Kadekodi, Soujanya Ponnapalli, Arjun Singhvi, Alin Nicolescu, Amogh Akshintala, Ted Yin, Emmanuel Amaro, and many others for that epic internship experience. I'm also very grateful to work with Ziqiao Zhou, Andrew Baumann, Marcus Peinado, Weidong Cui, and Xinyang Ge during my 2021 Microsoft Research internship, they showed me how to do system security research.

I thank Shin-Yeh Tsai, the RDMA guy in our lab, for providing us with a fast and easy to use network stack, also for his suggestions during the start of my PhD. I thank Yilun Chen, Yutong Huang, Sumukh H. Ravindra for their contribution to the LegoOS project. I thank Ke Liu for helping out the experiments and plotting during several deadline closings. I thank Arvind Krishnamurthy's insights and Ryan Kosta's OVS testing in the SuperNIC project. I thank Will Lin for his contribution in the SuperNIC project, this project will not be possible without his

insightful questions, hard work, commitment, and dedication. I thank Zhiyuan Guo and Xuhao Luo for their contribution in the Clio project. I'm very grateful to Alex Forencich and Moein Khazraee for helping me on debugging and bringing up FPGA boards. The Clio and SuperNIC projects literally would not be possible without their help.

I thank my committee members, Professor Geoffrey M. Voelker, Professor Alex C. Snoeren, Professor Stefan Savage, and Professor George C. Papen for serving as a committee member. I would like to thank Liwei Guo, Heejin Park, and Hongyu Miao for the technical discussion back in Purdue. I thank Lixiang Ao, Zachary Blanco, Stewart Grant, Anil Yelam and many others for our discussions in UCSD.

I also want to thank my friends, this journey would not be possible without their love and support. I want to thank Jingwen Zhang, Xiaoyi Huang, Xiaosi Zhu, Heejin Park, Jianyue Zhang, Haolan Liu, Meihan Li, Zesen Zhang, Hanwen Yao, Yi Xu, Jian Yang, Guo Xiang, and many others for making my life in US colorful and meaningful. I wish them nothing but the best.

Finally, I am immensely grateful to my family. They encouraged me to pursue my dream and provide their unconditional love through the whole process. There are no words to express my gratitude to my parents, Wenliang Shan and Jinyan Zhang. I dedicate this dissertation to them.

Chapter 2, in full, is a reprint of Yizhou Shan, Shin-Yeh Tsai, Yiying Zhang, "Distributed Shared Persistent Memory", *SoCC, 2017*. The dissertation author was the primary investigator and author of this paper.

Chapter 3, in full, is a reprint of Yizhou Shan, Yutong Huang, Yilun Chen, Yiying Zhang, "LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation", *OSDI, 2018*. The dissertation author was the primary investigator and author of this paper.

Chapter 4, in full, is a reprint of Yizhou Shan, Zhiyuan Guo (co-first authors), Xuhao Luo, Yutong Huang, Yiying Zhang, "Clio: A Hardware-Software Co-Designed Disaggregated Memory

System", *ASPLOS, 2022*. The dissertation author was the primary investigator and author of this paper.

Chapter 5, in part, has been submitted for publication of the material as it may appear in Yizhou Shan, Will Lin, Ryan Kosta, Arvind Krishnamurthy, Yiying Zhang, "Disaggregating and Consolidating Network Functionalities with SuperNIC", *arXiv, 2022*. The dissertation author was the primary investigator and author of this paper.

VITA

2014	B. S. in Computer Engineering, Beihang University, Beijing
2014-2016	Research Assistant, University of Chinese Academy of Sciences, Institution of Computing Technology, Beijing
2022	Ph. D. in Computer Science, University of California San Diego

PUBLICATIONS

Yizhou Shan, Will Lin, Ryan Kosta, Arvind Krishnamurthy, Yiying Zhang, "Disaggregating and Consolidating Network Functionalities with SuperNIC", *arXiv, 2022*.

Yizhou Shan, Zhiyuan Guo (co-first authors), Xuhao Luo, Yutong Huang, Yiying Zhang, "Clio: A Hardware-Software Co-Designed Disaggregated Memory System", *ASPLOS, 2022*

Shin-Yeh Tsai, Yizhou Shan, Yiying Zhang, "Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores", *ATC, 2020*

Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Liran Liss, Michael Wei, Dan Tsafrir, Marcos Aguilera, "Storm: a fast transactional dataplane for remote data structures", *SYSTOR, 2019*, **Best Paper Award**

Yizhou Shan, Yutong Huang, Yilun Chen, Yiying Zhang, "LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation", *OSDI, 2018*, **Best Paper Award**

Yizhou Shan, Shin-Yeh Tsai, Yiying Zhang, "Distributed Shared Persistent Memory", *SoCC, 2017*

ABSTRACT OF THE DISSERTATION

Distributing and Disaggregating Hardware Resources in Data Centers

by

Yizhou Shan

Doctor of Philosophy in Computer Science

University of California San Diego, 2022

Professor Yiying Zhang, Chair

Hardware resource disaggregation is a solution that decomposes general-purpose monolithic servers into segregated, network-attached resource pools, each of which can be built, managed, and scaled independently. Despite its management, cost, and fault-tolerance benefits, hardware resource disaggregation is a drastic departure from the traditional computing paradigm and it calls for a top-down redesign on system software, hardware, and data center networks.

This dissertation shows that it is possible to overcome the challenges of building and deploying hardware resource disaggregation solutions in real data centers, delivering its promises on better manageability, scalability, and cost.

We first explored logical resource disaggregation for emerging persistent memory technologies. Logical resource disaggregation *logically* breaks the server boundary by building an indirection layer on top of monolithic servers to collectively expose a logical resource pool abstraction. However, we fail to overcome the inherent problems of monolithic servers. We then explored hardware resource disaggregation to overcome these limitations by physically separating hardware resources into network-attached pools. We emulated disaggregated devices using monolithic servers and built the first operating system designed for managing disaggregated resources. It provides backward compatible interfaces while delivering good performance. However, emulation incurs non-trivial overhead and has limited parallelism in serving highly-concurrent requests. To avoid such overhead, we then built the first publicly known hardware-based disaggregated memory device, which co-designs networking transport, virtual memory, and hardware. We soon realized that while an increasing amount of effort goes into disaggregating compute, memory, and storage, the network has been completely left out. The final piece of this dissertation proposes the concept of network disaggregation, which decouples network functionalities from endpoints and then consolidates them into a centralized network resource pool. We built a new hardware-based networking device along with a distributed runtime system to realize such a network resource pool. Together, these four pieces outline a practical path to enable hardware resource disaggregation solutions in real data centers, especially how one can navigate the complex trade-offs among performance, cost, and manageability.

Chapter 1

Introduction

Today, data centers that host or rent huge computing resources run large-scale applications that infiltrate billions of people's daily life. Data centers see an unprecedentedly rapid growth in workload diversity spanning social media, finance, smart health, IoT, and cloud computing. Applications that run in data centers keep changing, with ever increasing velocity, volume, and variety. At the same time, computing hardware used in data centers is also changing quickly. With the slowdown of Moore's Law and the diminishing of Dennard scaling, specialized domain-specific computing devices and accelerators such as Google TPU and VPU [149, 263], AWS Nitro [27], FPGA [258], GPU, and programmable switches [190] have made their ways into modern data centers. These domain-specific accelerators offer higher computing efficiency while operating at a lower cost than traditional general-purpose processors. As a result, the data center hardware infrastructure and resource management systems are under constant changes, not only because the demand from applications change frequently, but also due to the changes required to host new hardware accelerators.

Unfortunately, innovations in the data center are hindered by the traditional monolithic server deployment model. For many years, the unit of deployment, operation, and failure in data centers has been a monolithic server, one that contains all the hardware resources required to

run user programs. This long-standing server-centric architecture has several key limitations. First, with a server being the physical boundary of resource allocation, it is difficult to fully utilize all resources in a datacenter [282]. Second, it has poor hardware elasticity since it is difficult to add, move, remove, or reconfigure hardware devices after a server is deployed. Third, it has a coarse failure domain. If one of the devices is faulty, usually the whole server becomes unavailable. Fourth, it has poor support for heterogeneity. The root cause of these problems is that the monolithic server model tightly couples hardware devices with each other. As a result, making new hardware devices work with existing servers is a painful and lengthy process [257]. In all, the monolithic server model makes data center resource management inefficient and difficult.

The server-centric architecture is a bad fit for the fast-changing data center software and hardware needs. Traditional distributed systems enable applications to utilize resources beyond what a single server can offer. Distributed data processing systems [337], distributed shared memory [224, 189], distributed storage systems [58, 84, 116] have been widely deployed in the real world. Those solutions *logically* break the server boundary by collectively expose a logical resource pool abstraction using physically distributed resources. We call this model *logical resource disaggregation*. However, they cannot overcome all the issues of using monolithic servers, since fundamentally the hardware unit of operation, a server, still bundles different types of resources together. To fully support the growing heterogeneity in hardware and fast-changing demand in software and to provide elasticity and flexibility at the hardware level, a better way is to *physically* break the monolithic servers.

Hardware resource disaggregation is a solution that breaks full-blown, general-purpose monolithic servers into segregated, network-attached hardware resource pools, each of which can be built, managed, and scaled independently. The disaggregated approach largely increases the management flexibility of a data center. Applications can freely use resources from any hardware component, which makes resource management efficient and easy, thereby improving data center utilization. Different types of hardware resources can scale and fail independently. It is also easy

to move, remove, or reconfigure hardware devices. In addition, adding new hardware is as simple as directly attaching it to the network. Finally, hardware resource disaggregation shrinks the failure-sharing domain thereby enables a finer-grained failure isolation scheme.

Despite its management, cost, and failure-handling benefits, hardware resource disaggregation is a completely different computing paradigm from the traditional monolithic server model. With such a drastic departure, it introduces many new challenges and calls for a top-down redesign on system software, hardware devices, and data center networks.

A hardware resource disaggregation of processor, memory, and storage means that when managing one of them, there will be no or limited local accesses to the other two. Resources that used to be accessible via intra-server interconnect are now disaggregated across data center network. However, commodity operating system (OS) assumes local accesses to all resources. Therefore, it is not clear how an OS can run on top of a disaggregated architecture. To make it worse, the communication latency increases by several orders of magnitude when going from intra-server interconnect to even the fastest data center network such as RDMA (i.e., increasing from one or two hundreds of nanoseconds to several microseconds). It is not clear whether we can deliver reasonable performance when deploying OSs and applications over a much slower network interconnect. Finally, breaking monolithic servers into multiple independent network-attached hardware devices demands high network bandwidth and larger connectivity, as the network needs a lot more ports to connect to the increased number of devices, while preserving or even increasing network speed. In all, there are many open questions on when and how disaggregation should be deployed, as well as questions on what the trade-offs among performance, cost, and manageability are when building systems for disaggregation.

This dissertation seeks to address the challenges of building and deploying hardware resource disaggregation in real data centers. We demonstrated the feasibility of resource disaggregation, presented several critical techniques for improving performance, increasing scalability, and lowering costs. We also confirmed its advantages in better resource packing, failure isolation,

cost, and resource elasticity.

We first explored logical resource disaggregation using monolithic servers. We are among the first to build a distributed system for the emerging persistent memory (PM), enabling a wider adoption for it in data centers [286]. However, the inherent limitations of monolithic servers still exist. We then explored hardware resource disaggregation to overcome these limitations, by physically separating hardware resources into network-attached resource pools. We built LegoOS, the first operating system designed for managing disaggregated hardware resources. It provides a binary-compatible interface to existing software while delivering good performance [283]. Our solution in LegoOS is achieved by emulating disaggregated devices using servers, which has non-trivial overhead. To address this, we built real disaggregated devices using Field Programmable Gate Arrays (FPGAs). We tackled a type of resource that is probably the hardest to disaggregate, memory and built the first publicly known hardware-based disaggregated memory device called Clio [131]. Clio co-designs the networking stack and virtual memory subsystems, both tailored for resource disaggregation. We soon found that it is difficult to customize the network task for various heterogeneous hardware devices. More importantly, we realized that network, the fourth major computing resource in data centers, can also be disaggregated. We then propose the concept of network disaggregation, which decouples network tasks from endpoints and consolidates them into a centralized network resource pool [284]. Our network resource pool consists of a distributed control plane with efficient, fair, and safe resource sharing. It also has SuperNIC, a new hardware-based programmable network device that efficiently consolidates multi-tenant network functionalities from various endpoints.

While all projects included in this dissertation can work individually, when combined, they collectively outline a principled path to managing resources in disaggregated data centers. Specifically, LegoOS serves as the overall management (*i.e.*, an OS) of a disaggregated data center that is connected by a set of SuperNICs; each endpoint in the data center could be a compute node as described as the "pComponent" in the LegoOS work, a memory node built with the Clio

hardware board, or a storage node as described as the "sComponent" in the LegoOS work.

This dissertation advances the state-of-art in hardware resource disaggregation, transforming it from a vague research idea into one that is tangible, practical, deployable, and quantitatively shown to be beneficial. We propose principled guidelines for building both disaggregated hardware devices, software systems and navigate the complex design trade-offs among manageability, performance and cost.

Below, we give a brief overview of the four projects in this dissertation.

Chapter 2: Distributed Shared Persistent Memory. Persistent memory (PM) provides byte-addressability, persistence, high density, and DRAM-like performance. Even though it has the potential to greatly benefit large-scale applications, it was unclear how to best utilize it in data centers. The first part of the dissertation focuses on enabling PM in a distributed, large-scale data center environment. We propose Distributed Shared Persistent Memory, a framework that exposes a logical, virtual disaggregated PM resource pool abstraction using a set of physically distributed PM attached to monolithic servers. This framework unifies distributed shared memory and distributed storage system into one layer. This system not only outlines a path for wider PM adoption in data centers but also showcases the performance improvements over similar systems.

Chapter 3: LegoOS, A Disseminated, Distributed OS for Hardware Resource Disaggregation. While exploring the logical, virtual resource disaggregation (as the first part of this dissertation), we realized that the inherent limitations of monolithic servers still persist. In the second part of this dissertation, we take a radical departure to enabling physical hardware resource disaggregation in data center. The key question we seek to answer is *how to manage these physically disaggregated resources and run existing applications on top of them, while improving performance per dollar.* We propose a new OS model called *splitkernel* to manage disaggregated resources. Splitkernel disseminates traditional OS functionalities into loosely-coupled monitors, each of which runs on and manages a hardware device. LegoOS has performance comparable to monolithic Linux servers, while largely improving resource packing and reducing failure rate

over monolithic servers. LegoOS is only 1.3× to 1.7× slower with 25% of application working set available as DRAM cache at processor components.

Chapter 4: Clio, A Hardware-Software Co-Designed Disaggregated Memory System. The third piece of this dissertation tackles a type of resource that is probably the hardest to disaggregate, memory. All existing memory disaggregation solutions have taken one of two approaches: building/emulating memory nodes using regular servers or building them using raw memory devices with no processing power. Both fail to balance cost, scalability, and management problems. We seek a sweet spot in the middle by proposing a hardware-based memory disaggregation solution that co-design OS functionalities, hardware architecture, and the network system. Our system scales much better and has orders of magnitude lower tail latency than RDMA.

Chapter 5: Disaggregating and Consolidating Network Functionalities with SuperNIC. While increasing amounts of effort go into disaggregating compute, memory, and storage, the fourth major resource, network, has been completely left out. The final piece of this dissertation, for the first time, proposes the concept of network disaggregation and builds a real disaggregated network system. The core of our proposal is the concept of a rack-scale disaggregated network resource pool, which consists of a set of hardware devices that can execute network tasks and collectively provide Network-as-a-Service. In this work, we built SuperNIC which is a new hardware-based programmable network device that efficiently consolidates multi-tenant network functionalities from various endpoints. Our system guarantees an efficient, safe, and fair consolidation, with little performance penalty.

Chapter 2

Distributed Shared Persistent Memory

2.1 Introduction

Next-generation non-volatile memories (*NVMs*), such as 3DXpoint [141], phase change memory (*PCM*), spin-transfer torque magnetic memories (*STTMs*), and the memristor will provide byte addressability, persistence, high density, and DRAM-like performance [302]. These developments are poised to radically alter the landscape of memory and storage technologies and have already inspired a host of research projects [39, 72, 73, 93, 217, 320, 331, 346, 200]. However, most previous research on NVMs has focused on using them in a single machine environment. Even though NVMs have the potential to greatly improve the performance and reliability of large-scale applications, it is still unclear how to best utilize them in distributed, datacenter environments.

This paper takes a significant step towards the goal of using NVMs in distributed datacenter environments. We propose *Distributed Shared Persistent Memory (DSPM)*, a framework that provides a global, shared, and persistent memory space using a pool of machines with NVMs attached at the main memory bus. Applications can perform native memory load and store instructions to access both local and remote data in this global memory space and can at the same

time make their data persistent and reliable. DSPM can benefit both single-node persistent-data applications that want to scale out efficiently and shared-memory applications that want to add durability to their data.

Unlike traditional systems with separate memory and storage layers [104, 105, 287, 288], we propose to use just one layer that incorporates both distributed memory and distributed storage in DSPM. DSPM's one-layer approach eliminates the performance overhead of data marshaling and unmarshaling, and the space overhead of storing data twice. With this one-layer approach, DSPM can potentially provide the low-latency performance, vast persistent memory space, data reliability, and high availability that many modern datacenter applications demand.

Building a DSPM system presents its unique challenges. Adding "Persistence" to Distributed Shared Memory (DSM) is not as simple as just making in-memory data durable. Apart from data durability, DSPM needs to provide two key features that DSM does not have: persistent naming and data reliability. In addition to accessing data in PM via native memory loads and stores, applications should be able to easily name, close, and re-open their in-memory data structures. User data should also be reliably stored in NVM and sustain various types of failures; they need to be consistent both within a node and across distributed nodes after crashes. To make it more challenging, DSPM has to deliver these guarantees without sacrificing application performance in order to preserve the low-latency performance of NVMs.

We built *Hotpot*, a DSPM system in the Linux kernel. Hotpot offers low-latency, direct memory access, data persistence, reliability, and high availability to datacenter applications. It exposes a global virtual memory address space to each user application and provides a new persistent naming mechanism that is both easy-to-use and efficient. Internally, Hotpot organizes and manages data in a flexible way and uses a set of adaptive resource management techniques to improve performance and scalability.

Hotpot builds on two main ideas to efficiently provide data reliability with distributed shared memory access. Our first idea is to integrate distributed memory caching and data

replication by imposing *morphable* states on persistent memory (*PM*) pages.

In DSM systems, when an application on a node accesses shared data in remote memory *on demand*, DSM caches these data copies in its local memory for fast accesses and later evicts them when reclaiming local memory space. Like DSM, Hotpot caches application-accessed data in local PM and ensures the coherence of multiple cached copies on different nodes. But Hotpot also uses these cached data as *persistent data replicas* and ensures their reliability and crash consistency.

On the other hand, unlike distributed storage systems, which *creates* extra data replicas to meet user-specified reliability requirements, Hotpot makes use of data copies that *already exist* in the system when they were fetched to a local node due to application memory accesses.

In essence, every local copy of data serves two simultaneous purposes. First, applications can access it locally without any network delay. Second, by placing the fetched copy in PM, it can be treated as a persistent replica for data reliability.

This seemingly-straightforward integration is not simple. Maintaining wrong or outdated versions of data can result in inconsistent data. To make it worse, these inconsistent data will be persistent in PM. We carefully designed a set of protocols to deliver data reliability and crash consistency guarantees while integrating memory caching and data replication.

Our second idea is to exploit application behaviors and intentions in the DSPM setting. Unlike traditional memory-based applications, persistent-data-based applications, DSPM's targeted type of application, have well-defined data *commit points* where they specify what data they want to make persistent. When a process in such an application makes data persistent, it usually implies that the data can be *visible* outside the process (*e.g.*, to other processes or other nodes). Hotpot utilizes these data commit points to also push updates to cached copies on distributed nodes to avoid maintaining coherence on every PM write. Doing so greatly improves the performance of Hotpot, while still ensuring correct memory sharing and data reliability.

To demonstrate the benefits of Hotpot, we ported the MongoDB [218] NoSQL database

to Hotpot and built a distributed graph engine based on PowerGraph [119] on Hotpot. Our MongoDB evaluation results show that Hotpot outperforms a PM-based replication system [346] by up to 3.1×, a recent PM-based distributed file systems [200] by up to 3.0×, and a DRAM-based file system by up to 53×. Hotpot outperforms PowerGraph by 2.3× to 5×, a recent DSM system [224] by 1.3× to 3.2×. Moreover, Hotpot delivers stronger data reliability and availability guarantees than these alternative systems.

Overall, this paper makes the following key contributions:

- We are the first to introduce the Distributed Shared Persistent Memory (DSPM) model and among the first to build distributed PM-based systems. The DSPM model provides direct and shared memory accesses to a distributed set of PMs and is an easy and efficient way for datacenter applications to use PM.

- We propose a one-layer approach to build DSPM by integrating memory coherence and data replication. The one-layer approach avoids the performance cost of two or more indirection layers.

- We designed two distributed data commit protocols with different consistency levels and corresponding recovery protocols to ensure data durability, reliability, and availability.

- We built the first DSPM system, Hotpot, in the Linux kernel, and two traditional kernel-level DSM systems (as comparison to Hotpot). Hotpot and the two DSM systems are both open sourced.

- We demonstrated Hotpot's performance benefits and ease of use with two real datacenter applications and extensive microbenchmark evaluation. We compared Hotpot with five existing file systems and distributed memory systems, and two in-house DSM systems.

The rest of the paper is organized as follows. Section 2 presents and analyzes several recent datacenter trends that motivated our design of DSPM. We discuss the benefits and challenges

of DSPM in Section 3. Section 4 presents the architecture and abstraction of Hotpot. We then discuss Hotpot's data management in Section 5. We present our protocols and mechanisms to ensure data durability, consistency, reliability, and availability in Section 6. Section 7 briefly discusses the network layer we built underlying Hotpot, and Section 8 presents detailed evaluation of Hotpot. We cover related work in Section 9 and conclude in Section 10.

Hotpot is available at `https://github.com/WukLab/Hotpot`.

2.2 Motivation

DSPM is motivated by three datacenter trends: emerging hardware PM technologies, modern data-intensive applications' data sharing, persistence, and reliability needs, and the availability of fast datacenter network.

2.2.1 Persistent Memory and PM Apps

Next-generation non-volatile memories (*NVMs*), such as 3DXpoint [141], phase change memory (*PCM*), spin-transfer torque magnetic memories (*STTMs*), and the memristor will provide byte addressability, persistence, and latency that is within an order of magnitude of DRAM [136, 179, 180, 183, 260, 302, 334, 200]. These developments are poised to radically alter the landscape of memory and storage technologies.

NVMs can attach directly to the main memory bus to form Persistent Memory, or PM. If applications want to exploit all the low latency and byte-addressability benefits of PM, they should directly access it via memory load and store instructions without any software overheads [72, 320, 346, 248, 211, 167] (we call this model *durable in-memory computation*), rather than accessing it via a file system [73, 91, 93, 331, 240, 200].

Unfortunately, most previous durable in-memory systems were designed for the single-node environment. With modern datacenter applications' computation scale, we have to be able

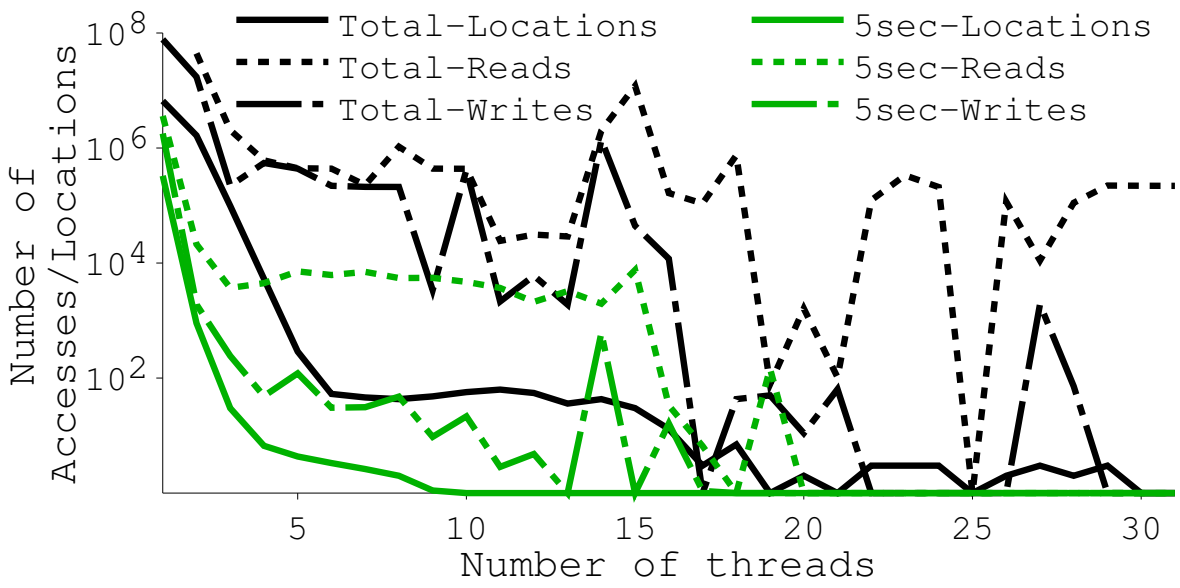


Figure 2.1: PowerGraph Sharing Analysis. Results of running PageRank [178] on a Twitter graph [175]. Black lines represent total amount of sharing. Green lines represent sharing within five seconds.

to scale out these single-node PM systems.

2.2.2 Shared Memory Applications

Modern data-intensive applications increasingly need to access and share vast amounts of data fast. We use PIN [202] to collect memory access traces of two popular data-intensive applications, TensorFlow [12] and PowerGraph [119]. Figures 2.1 and 2.2 show the total number of reads and writes performed to the same memory location by *N* threads and the amount of these shared locations. There are a significant amount of shared read accesses in these applications, especially across a small set of threads. We further divided the memory traces into smaller time windows and found that there is still a significant amount of sharing, indicating that many shared accesses occur at similar times.

Distributed Shared Memory (*DSM*) takes the shared memory concept a step further

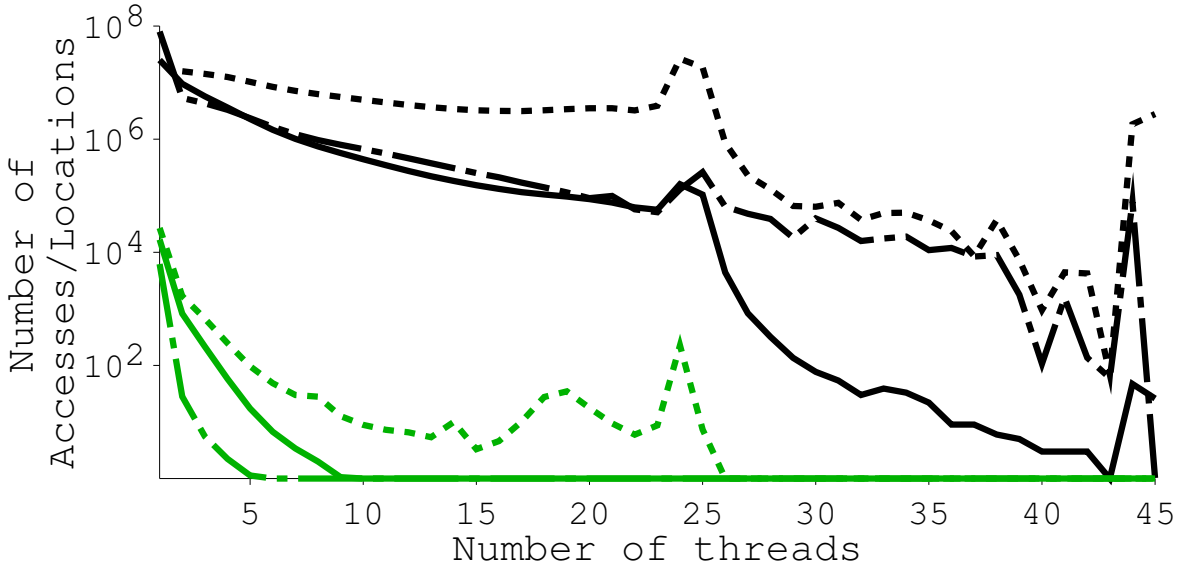


Figure 2.2: TensorFlow Sharing Analysis. Results of running a hand-writing recognition workloads provided by TensorFlow. Black lines represent total amount of sharing. Green lines represent sharing within five seconds.

by organizing a pool of machines into a globally shared memory space. Researchers and system builders have developed a host of software and hardware DSM systems in the past few decades [48, 52, 53, 86, 108, 118, 169, 197, 162, 160, 262, 348, 299, 300, 349, 277]. Recently, there is a new interest in DSM [224] to support modern data-intensive applications.

However, although DSM scales out shared-memory applications, there has been no persistent-memory support for DSM. DSM systems all had to checkpoint to disks [298, 267, 227]. Memory persistence can allow these applications to checkpoint fast and recover fast [223].

2.2.3 Fast Network and RDMA

Datacenter network performance has improved significantly over the past decades. InfiniBand (*IB*) NICs and switches support high bandwidth ranging from 40 to 100 Gbps. Remote Direct Memory Access (*RDMA*) technologies that provide low-latency remote memory accesses

have become more mature for datacenter uses in recent years [153, 91, 152, 129]. These network technology advances make remote-memory-based systems [224, 126, 253, 64, 51, 338] more attractive than decades ago.

2.2.4 Lack of Distributed PM Support

Many large-scale datacenter applications require fast access to vast amounts of persistent data and could benefit from PM's performance, durability, and capacity benefits. For PMs to be successful in datacenter environments, they have to support these applications. However, neither traditional distributed storage systems or DSM systems are designed for PM. Traditional distributed storage systems [14, 58, 84, 116, 173, 254] target slower, block-based storage devices. Using them on PMs will result in excessive software and network overheads that outstrip PM's low latency performance [346]. DSM systems were designed for fast, byte-addressable memory, but lack the support for data durability and reliability.

Octopus [200] is a recent RDMA-enabled distributed file system built for PM. Octopus and our previous work Mojim [346] are the only distributed PM-based systems that we are aware of. Octopus was developed in parallel with Hotpot and has a similar goal as Hotpot: to manage and expose distributed PM to datacenter applications. However, Octopus uses a file system abstraction and is built in the user level. These designs add significant performance overhead to native PM accesses (Section 2.8.2). Moreover, Octopus does not provide any data reliability or high availability, both of which are key requirements in datacenter environments.

2.3 Distributed Shared Persistent Memory

The datacenter application and hardware trends described in Section 2.2 clearly point to one promising direction of using PM in datacenter environments — as distributed, shared, persistent memory (DSPM). A DSPM system manages a distributed set of PM-equipped machines

and provides the abstraction of a global virtual address space and a data persistence interface to applications. This section gives a brief discussion on the DSPM model.

2.3.1 DSPM Benefits and Usage Scenarios

DSPM offers low-latency, shared access to vast amount of durable data in distributed PM, and the reliability and high availability of these data. Application developers can build in-memory data structures with the global virtual address space and decide how to name their data and when to make data persistent.

Applications that fit DSPM well have two properties: accessing data with memory instructions and making data durable explicitly. We call the time when an application makes its data persistent a *commit point*. There are several types of datacenter applications that meet the above two descriptions and can benefit from running on DSPM.

First, applications that are built for single-node PM can be easily ported to DSPM and scale out to distributed environments. These applications store persistent data as in-memory data structures and already express their commit points explicitly. Similarly, storage applications that use memory-mapped files also fit DSPM well, since they operate on in-memory data and explicitly make them persistent at well-defined commit points (*i.e.*, *msync*). Finally, DSPM fits shared-memory or DSM-based applications that desire to incorporate durability. These applications do not yet have durable data commit points, but we expect that when developers want to make their applications durable, they should have the knowledge of when and what data they want make durable.

2.3.2 DSPM Challenges

Building a DSPM system presents several new challenges.

First, *what type of abstraction should DSPM offer to support both direct memory accesses*

and data persistence (Section 2.4)? To perform native memory accesses, application processes should use virtual memory addresses. But virtual memory addresses are not a good way to *name* persistent data. DSPM needs a naming mechanism that applications can easily use to retrieve their in-memory data after reboot or crashes (Section 2.4.2). Allowing direct memory accesses to DSPM also brings another new problem: pointers need to be both persistent in PM and consistent across machines (Section 2.4.3).

Second, *how to efficiently organize data in DSPM to deliver good application performance (Section 2.5)?* To make DSPM's interface easy to use and transparent, DSPM should manage the physical PM space for applications and handle PM allocation. DSPM needs a flexible and efficient data management mechanism to deliver good performance to different types of applications.

Finally, *DSPM needs to ensure both distributed cache coherence and data reliability at the same time* (Section 2.6). The former requirement ensures the coherence of multiple cached copies at different machines under concurrent accesses and is usually enforced in a distributed memory layer. The latter provides data reliability and availability when crashes happen and is implemented in distributed storage systems or distributed databases. DSPM needs to incorporate both these two different requirements in one layer in a correct and efficient way.

2.4 Hotpot Architecture and Abstraction

We built *Hotpot*, a kernel-level DSPM system that provides applications with direct memory load/store access to both local and remote PM and a mechanism to make in-PM data durable, consistent, and reliable. Hotpot is easy to use, delivers low-latency performance, and provides flexible choices of data consistency, reliability, and availability levels. This section presents the overall architecture of Hotpot and its abstraction to applications.

We built most of Hotpot as a loadable kernel module in Linux 3.11.0 with only a few small changes to the original kernel. Hotpot has around 19K lines of code, out of which 6.4K

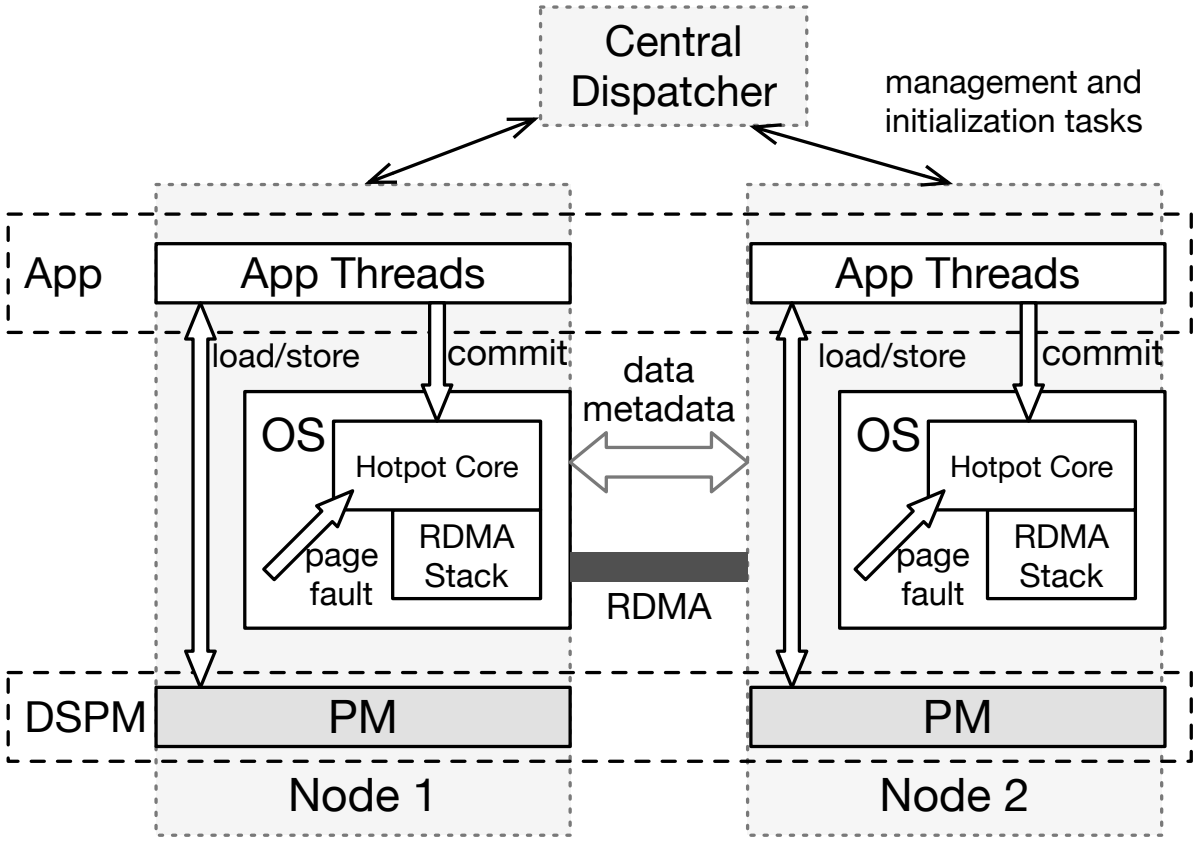


Figure 2.3: Hotpot Architecture.

lines are for a customized network stack (Section 2.7).

Hotpot sits in the kernel space and manages PMs in a set of distributed nodes, or *Hotpot nodes*. Hotpot provides applications with an easy-to-use, memory-based abstraction that encapsulates both memory and persistent data access in a transparent way. Figure 2.3 presents Hotpot's architecture. Hotpot uses a *Central Dispatcher (CD)* to manage node membership and initialization tasks (*e.g.*, create a dataset). All data and metadata communication after a dataset has been created takes place between Hotpot nodes and does not involve the CD.

2.4.1 Application Execution and Data Access Abstraction

Most data-intensive applications are multithreaded and distribute their data processing work across threads [218, 119]. Thus, Hotpot adopts a thread-based model to run applications on a set of Hotpot nodes. Hotpot uses application threads as the unit of deployment and lets applications decide what operations and what data accesses they want to include in each thread. Applications specify what threads to run on each Hotpot node and Hotpot runs an application by starting all its threads together on all Hotpot nodes. We give users full flexibility in choosing their initial thread and workload distributions. However, such user-chosen distributions may not be optimal, especially as workloads change over time. To remedy this situation, Hotpot provides a mechanism to adaptively move data closer to computation based on workload behavior, as will be discussed in Section 2.5.5.

Hotpot provides a global virtual memory address space to each application. Application threads running on a node can perform native memory load and store instructions using global virtual memory addresses to access both local and remote PM. The applications do not know where their data physically is or whether a memory access is local or remote. Internally, a virtual memory address can map to a local physical page if the page exists locally or generate a page fault which will be fulfilled by Hotpot by fetching a remote page (more in Section 2.5.3). Figure 2.4 presents an example of Hotpot's global virtual address space. Unlike an I/O-based interface, Hotpot's native memory interface can best exploit PMs' low-latency, DRAM-like performance, and byte addressability.

On top of the memory load/store interfaces, Hotpot provides a mechanism for applications to name their data, APIs to make their data persistent, and helper functions for distributed thread synchronization. Table 2.1 lists Hotpot APIs. We also illustrate Hotpot's programming model with a simple program in Figure 2.5. We will explain Hotpot's data commit semantics in Section 2.6.

Table 2.1: Apart from these APIs, Hotpot also supports direct memory loads and stores.

API	Explanation	Backward
open (*close*)	open or create (close) a DSPM dataset	same as current
mmap (*munmap*)	map (unmap) a DSPM region in a dataset to application address space	same as current
commit	commit a set of data and make *N* persistent replicas	similar to msync
acquire	acquire single writer permission	
thread-barrier	helper function to synchronize threads on different nodes	similar to pthread barrier

2.4.2 Persistent Naming

To be able to store persistent data and to allow applications to re-open them after closing or failures, Hotpot needs to provide a naming mechanism that can sustain power recycles and crashes.

Many modern data-intensive applications such as in-memory databases [218] and graphs [120, 119] work with only one or a few big datasets that include all of an application's data and then manage their own fine-grained data structures within these datasets. Thus, instead of traditional hierarchical file naming, we adopt a flat naming mechanism in Hotpot to reduce metadata management overhead.

Specifically, Hotpot applications assign names by *datasets* and can use these names to open the datasets. A dataset is similar to the traditional file concept, but Hotpot places all datasets directly under a mounted Hotpot partition without any directories or hierarchies. Since under Hotpot's targeted application usage, there will only be a few big datasets, dataset lookup and metadata management with Hotpot's flat namespace are easy and efficient. We use a simple (persistent) hash table internally to lookup datasets.

The *open* and *mmap* APIs in Table 2.1 let applications create or open a dataset with a name and map it into the application's virtual memory address space. Afterwards, all data access is through native memory instructions.

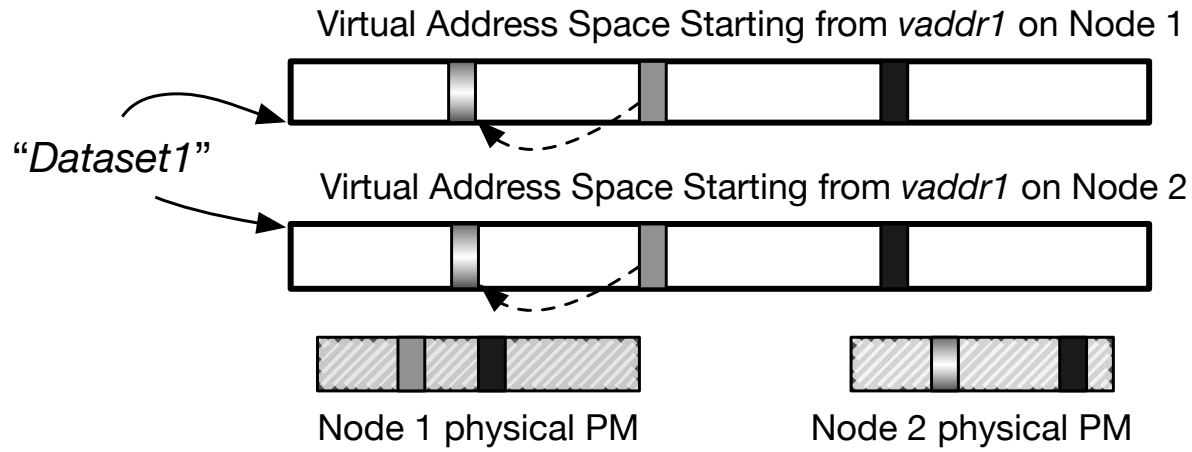


Figure 2.4: Hotpot Addressing. Hotpot maps "*Dataset1*" to Node 1 and Node 2's virtual address space using the same base virtual addresses. The physical address mapping on each node is different. The grey blocks in the middle are pointers that point to the blocks on the left.

2.4.3 Consistent and Persistent Pointers

Hotpot applications can use DSPM as memory and store arbitrary data structures in it. One resulting challenge is the management of pointers in DSPM. To make it easy to build persistent applications with memory semantics, Hotpot ensures that pointers in DSPM have the same value (*i.e.*, virtual addresses of the data that they point to) both across nodes and across crashes. Application threads on different Hotpot nodes can use pointers directly without pointer marshaling or unmarshaling, even after power failure. We call such pointers *globally-consistent and persistent pointers*. Similar to NV-Heaps [72], we restrict DSPM pointers to only point to data within the same dataset. Our targeted type of applications which build their internal data structures within big datasets already meet this requirement.

To support globally-consistent and persistent pointers, Hotpot guarantees that the same virtual memory address is used as the starting address of a dataset across nodes and across re-opens of the dataset. With the same base virtual address of a dataset and virtual addresses within a dataset being consecutive, all pointers across Hotpot nodes will have the same value.

```
/* Open a dataset in Hotpot DSPM space */
int fd = open("/mnt/hotpot/dataset", O_CREAT|O_RDWR);

/* Obtain virtual address of dataset with traditional mmap() */
void *base= mmap(0,40960,PROT_WRITE,MAP_PRIVATE,fd,0);

/* Size of the application log */
int *log_size = base;
/* The application log */
int *log = base + sizeof(int);

/* Append an entry to the end of the log */
int new_data = 24;
log[*log_size] = new_data;
*log_size += 1;

/* Prepare memory region metadata for commit */
struct commit_area_struct {void *address; int length;};
struct commit_area_struct areas[2];
areas[0].address = log_size;
areas[0].length = sizeof(int);
areas[1].address = &log[*log_size];
areas[1].length = sizeof(int);

/* Commit the two data areas, each with two replicas */
commit(areas, 2);
```

Figure 2.5: Sample code using Hotpot. Code snippet that implements a simple log append operation with Hotpot.

We developed a new mechanism to guarantee that the same base virtual address is used across nodes and crashes. When an application opens a dataset for the first time, Hotpot uses a consensus protocol to discover the current available virtual address ranges on all nodes and select one for the dataset. Nodes that have not opened the dataset will reserve this virtual address range for possible future opening of the dataset. Since the total amount of virtual addresses for DSPM is bound to the total size of DSPM datasets, Hotpot can always find available virtual address ranges on 64-bit platforms. Hotpot records the virtual address range persistently and forces applications to use the same virtual address the next time it starts. To ensure that recorded

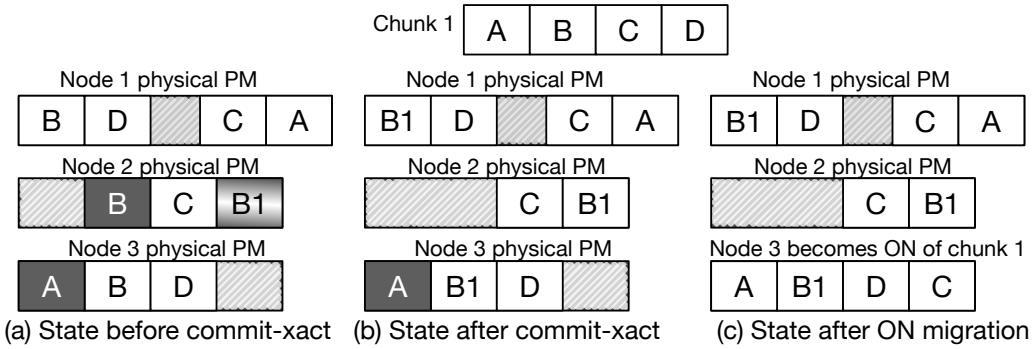


Figure 2.6: Data State Change Example. White, black, and striped blocks represent committed, redundant, and dirty states. Before commit, Node 2 and Node 3 both have cached copies of data page *B*. Node 2 has written to *B* and created a dirty page, *B*1. During commit, Node 2 pushes the content *B*1 to its ON, Node 1. Node 1 updates its committed copy to *B*1 and also sends this update to Node 3. Figure (c) shows the state after migrating the ON of chunk 1 from Node 1 to Node 3. After migration, Node 3 has all the pages of the chunk and all of them are in committed states.

persistent virtual address ranges are always available when opening datasets, we change the kernel loader and virtual memory address allocator (*i.e.*, *brk* implementation) to exclude all recorded address ranges.

2.5 Data Management and Access

This section presents how Hotpot manages user data in DSPM. We postpone the discussion of data durability and reliability to Section 2.6.

2.5.1 PM Page Morphable States

One of Hotpot's design philosophies is to use one layer for both memory and storage and to integrate distributed memory caching and data replication. To achieve this goal, we propose to impose *morphable* states on PM pages, where the same PM page in Hotpot can be used both as a local memory cached copy to improve performance and as a redundant data page to improve data

reliability and availability.

We differentiate three states of a PM page: active and dirty, active and clean, and inactive and clean, and we call these three states *dirty*, *committed*, and *redundant* respectively. A page being clean means that it has not been updated since the last commit point; committing a dirty page moves it to the clean state. A page being active means that it is currently being accessed by an application, while an redundant page is a page which the application process has not mapped or accessed. Several Hotpot tasks can change page states, including page read, page write, data commit, data replication, page migration, and page eviction. We will discuss how page states change throughout the rest of this section. Figure 2.6 illustrates two operations that cause Hotpot data state changes.

2.5.2 Data Organization

Hotpot aims to support large-scale, data-intensive applications on a fairly large number of nodes. Thus, it is important to minimize Hotpot's performance and scalability bottlenecks. In order to enable flexible load balancing and resource management, Hotpot splits the virtual address range of each dataset into *chunks* of a configurable size (*e.g.*, 4 MB). PM pages in a chunk do not need to be physically consecutive and not all pages in a chunk need to exist on a node.

Each chunk in Hotpot is owned by an *owner node (ON)*, similar to the "home" node in home-based DSM systems [349]. An ON maintains all the data and metadata of the chunk it owns. Other nodes, called *data node* or *DN*, always fetch data from the ON when they initially access the data. A single Hotpot node can simultaneously be the ON for some data chunks and the DN for other chunks. When the application creates a dataset, Hotpot CD performs an initial assignment of ONs to chunks of the dataset.

Two properties separate Hotpot ONs from traditional home nodes. First, Hotpot ON is responsible for the reliability and crash consistency of the pages it owns, besides serving read data and ensure the coherence of cached copies. Second, Hotpot does not fix which node owns a chunk

and the location of ON adapts to application workload behavior dynamically. Such flexibility is important for load balancing and application performance (see Section 2.5.5).

2.5.3 Data Reads and Writes

Hotpot minimizes software overhead to improve application performance. It is invoked only when a page fault occurs or when applications execute data persistence operations (see Section 2.6 for details of data persistence operations).

When a page fault happens because of read, it means that there is no valid local page. Hotpot first checks if there is any local redundant page. If so, it will move this page to the committed state and establish a page table entry (PTE) for it. Otherwise, there is no available local data and Hotpot will fetch it from the remote ON. Hotpot writes the received data to a newly-allocated local physical PM page. Afterwards, applications will use memory instructions to access this local page directly.

Writing to a committed page also causes a page fault in Hotpot. This is because a committed page can contribute towards user-specified degree of replication as one data replica, and Hotpot needs to protect this committed version from being modified. Thus, Hotpot write protects all committed pages. When these pages are written to (and generating a write page fault), Hotpot creates a local Copy-On-Write (COW) page and marks the new page as dirty while leaving the original page in committed state. Hotpot does not write protect this COW page, since it is already in the dirty state.

Following Hotpot's design philosophy to exploit hints from our targeted data-intensive applications, we avoid propagating updates to cached copies at other nodes on each write and only do so at each application commit point. Thus, all writes in Hotpot is local and only writing to a committed page will generate a page fault.

Not updating remote cached copies on each write also has the benefit of reducing write amplification in PM. In general, other software mechanisms and policies such as wear-aware PM

allocation and reclamation and hardware techniques like Start-Gap [261] can further reduce PM wear. We do not focus on PM wear in this paper and leave such optimizations for future work.

2.5.4 PM Page Allocation and Eviction

Each Hotpot node manages its own physical PM space and performs PM page allocation and eviction. Since physical pages do not need to be consecutive, we use a simple and efficient allocation mechanism by maintaining a free page list and allocating one page at a time.

Hotpot uses an approximate-LRU replacement algorithm that is similar to Linux's page replacement mechanism. Different from Linux, Hotpot distinguishes pages of different states. Hotpot never evicts a dirty page and always tries to evict redundant pages before evicting committed pages. We choose to first evict redundant pages, because these are the pages that have not been accessed by applications and less likely to be accessed in the future than committed pages.

Since both redundant and committed pages can serve as a redundant copy for data reliability, Hotpot cannot simply throw them away during eviction. The evicting node of a page will contact its ON, which will check the current degree of replication of the candidate pages and prioritize the eviction of pages that already have enough replicas. For pages that will drop below the user-defined replication degree after the eviction, the ON will make a new redundant page at another node.

2.5.5 Chunk ON Migration

An ON serves both page read and data commit requests that belong to the chunks it owns. Thus, the location of ON is important to Hotpot's performance. Ideally, the node that performs the most reads and commits of data in a chunk should be its ON to avoid network communication.

By default, Hotpot initially spreads out a dataset's chunks to all Hotpot nodes in a

round robin fashion (other static placement policies can easily replace round robin). Static placement alone cannot achieve optimal run-time performance. Hotpot remedies this limitation by performing online chunk migration, where one ON and one DN of a chunk can switch their identities and become the new DN and new ON of the chunk.

Hotpot utilizes application behavior in recent history to decide how to migrate ONs. Each ON records the number of page read requests and the amount of committing data it receives in the most recent time window.

ONs make their migration decisions with a simple greedy algorithm based on the combination of two criteria: maximizing the *benefit* while not exceeding a configurable *cost* of migration. The benefit is the potential reduction in network traffic during remote data reads and commits. The node that performs most data communication to the ON in recent history is likely to benefit the most from being the new ON, since after migration these operations will become local. We model the cost of migration by the amount of data needed to copy to a node so that it has all the chunk data to become ON.

Once Hotpot has made a decision, it performs the actual chunk migration using a similar method as process and VM migration [239, 87, 70] by temporary stopping commits to the chunk under migration and resume them at the new ON after migration.

2.6 Data Durability, Consistency, and Reliability

Being distributed shared memory and distributed storage at the same time, DSPM should ensure both correct shared memory accesses to PM and the persistence and reliability of in-PM data. Hotpot provides three guarantees: coherence among cached copies of in-PM data, recovery from various types of failures into a consistent state, and user data reliability and availability under concurrent failures. Although each of these three properties have been explored before, as far as we know, Hotpot is the first system that integrates all of them in one layer. Hotpot also has

the unique requirement of low software overhead to retain the performance benefit of PM.

- *Cache coherence.* In Hotpot, application processes on different nodes cache remote data in their local PM for fast accesses. Hotpot provides two consistency levels across cached copies: ***R1.a***, multiple readers and single writer (*MRSW*) and ***R1.b***, multiple readers and multiple writers (*MRMW*). MRMW allows multiple nodes to concurrently write and commit their local cached copies. With MRMW, there can be multiple versions of dirty data in the system (but still one committed version), while MRSW guarantees only one dirty version at any time. An application can use different modes for different datasets, but only one mode with the same dataset. This design allows flexibility at the dataset granularity while guaranteeing correctness.

- *Crash consistency.* Data storage applications usually have well-defined *consistent* states and need to move from one consistent state to another atomically. When a crash happens, user data should be recovered to a consistent state (*i.e.*, *crash consistency*). Hotpot guarantees crash consistency both within a single node (***R2.a***) and across distributed nodes (***R2.b***). Note that crash consistency is different and orthogonal to cache coherence in **R1.a** and **R1.b**.

- *Reliability and availability.* To ensure that user persistent data can sustain $N-1$ concurrent node failures, where N is a user defined value, Hotpot guarantees that ***R3***, once data has been committed, there are always N copies of clean, committed data.

This section first discusses how Hotpot ensures crash consistency within a single node, then presents the MRMW and MRSW modes and their atomic commit protocols, and ends with the discussion of Hotpot's recovery mechanisms under different crash scenarios.

2.6.1 Single-Node Persistence and Consistency

Before ensuring user data's global reliability and consistency in DSPM, Hotpot first needs to make sure that data on a single node can properly sustain power crashes (**R2.a**) [248]. Hotpot makes data persistent with the standard Intel persistent memory instructions [168], *i.e.*, `clflush`, `mfence` (note that we do not include the deprecated `pcommit` instruction [145]).

After a node crashes, if its PM is still accessible, Hotpot will use the PM content to recover; otherwise, Hotpot will use other nodes to reconstruct data on a new node (Section 2.6.4). For the former case, Hotpot needs to guarantee that user data in DSPM is in a consistent state after crash. Hotpot also needs to ensure that its own metadata is persistent and is consistent with user data.

Hotpot maintains metadata on a local node to find user data and record their morphable states (*i.e.*, committed, dirty, or redundant). Since these metadata are only used within a single node, Hotpot does not need to replicate them on other nodes. Hotpot makes these metadata persistent at known locations in PM — a pre-allocated beginning area of PM. Hotpot also uses metadata to record online state of the system (*e.g.*, ON maintains a list of active DNs that have a cached copy of data). These metadata can be reconstructed by re-examining system states after recovery. Thus, Hotpot does not make these metadata persistent.

Similar to traditional file systems and databases, it is important to enforce *ordering* of metadata and data persistence in order to recover to a consistent state. For single-node non-commit operations (we defer the discussion of commit operations to Sections 2.6.2 and 2.6.3), Hotpot uses a simple shadow-paging mechanism to ensure that the consistency of metadata and data. Specifically, we associate each physical memory page with a metadata slot and use a single 8-byte index value to locate both the physical page and its metadata. When an application performs a memory store to a committed page, Hotpot allocates a new physical page, writes the new data to it, and writes the new metadata (*e.g.*, the state of the new page) to the metadata slot associated with this physical page. After making all the above data and metadata persistent, Hotpot changes

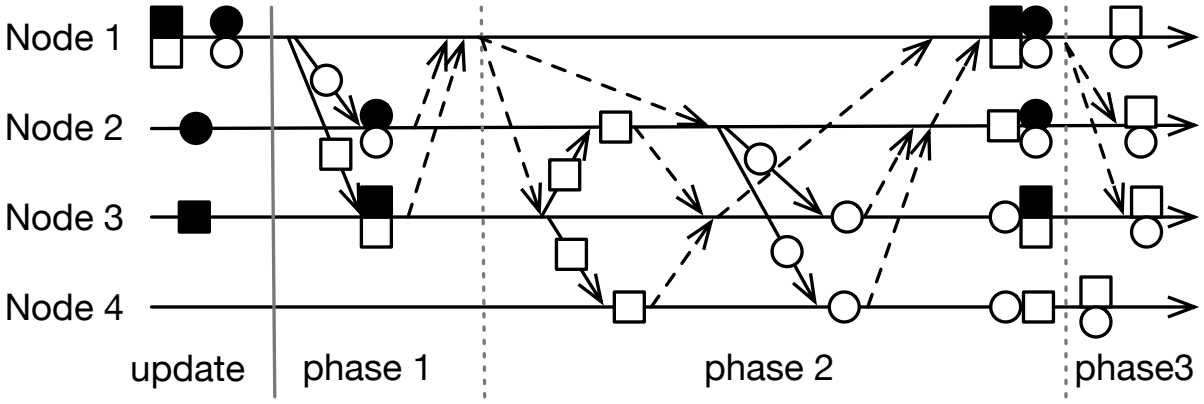


Figure 2.7: MRMW Commit Example. Solid arrows represent data communication. Dashed arrows represent metadata communication. Node 1 (CN) commit data to ONs at Node 2 and 3 with replication degree four. Black shapes represent old committed states before the update and white shapes represent new states.

the index from pointing to the old committed page to pointing to the new dirty page. Since most architectures support atomic 8-byte writes, this operation atomically moves the system to a new consistent state with both the new data and the new metadata.

2.6.2 MRMW Mode

Hotpot supports two levels of concurrent shared-PM accesses and uses different protocols to commit data. The MRMW mode allows multiple concurrent versions of dirty, uncommitted data to support great parallelism. MRMW meets **R1.b**, **R2.b**, and **R3**.

MRMW uses a distributed atomic commit protocol at each commit point to make local updates globally visible, persistent, and replicated. Since MRMW supports concurrent commit operations and each commit operation can involve multiple remote ONs, Hotpot needs to ensure that all the ONs reach consensus on the commit operation they serve. We designed a three-phase commit protocol for the MRMW mode based on traditional two-phase commit protocols [275, 124, 177] but differs in that Hotpot needs to ensure cache coherence, crash consistency, and data replication all in one protocol. Figure 2.7 illustrates an example of MRMW.

Commit phase 1. When a node receives a *commit* call (we call this node *CN*), it checks if data specified in the *commit* call is dirty and commits only the dirty pages. CN persistently records the addresses of these dirty pages for recovery reasons (Section 2.6.4). CN also assigns a unique ID (*CID*) for this *commit* request and persistently records the CID and its state of starting phase 1.

Afterwards, CN sends the committing data to its ONs to prepare these ONs for the commit. Each ON accepts the commit request if it has not accepted other commit request to the same pages, and it stores the committing data in a *persistent redo log* in PM. The ON also persistently records the CID and its state (*i.e.*, completed phase 1) persistently. The ON will block future commit requests to these data until the whole commit process finishes. The CN can proceed to phase 2 only when all ONs return successfully.

Commit phase 2. In commit phase 2, Hotpot makes the committing data persistent, coherent, and replicated. This is the phase that Hotpot differs most from traditional distributed commit protocols.

CN sends a command to all the involving ONs to indicate the beginning of phase 2. Each ON then performs two tasks in one multicast operation (Section 2.7): updating DNs' cached copies of the committing data and making extra replicas. ON looks up its metadata to find what DNs have a cached copy. If these DNs alone cannot meet the replication degree, ON will choose new DNs that do not have a copy of the data and send the data to them.

When a DN receives the committing data from an ON, it checks the state of its local data pages. If a local page is in the committed state or the redundant state, the DN will directly overwrite the local page with the received data. In doing so, the DN's cached PM data is updated. If the local page is dirty or if there is no corresponding local page, the DN allocates a new physical page and writes the new data to this page. The new physical page will be in the redundant state and will not affect the DN's dirty data. In this way, all DNs that receive updated data from the ON will have a clean, committed copy, either in the committed or the redundant state.

After all DNs have replied to the ON indicating that there are now N copies of the committing data, the ON commits data locally by checkpointing (copying) data from the redo log to their home locations. Unlike traditional databases and file systems that lazily checkpoint logged data, Hotpot checkpoints all committing data in this phase so that it can make the updated version of the data visible to applications immediately, a requirement of shared-memory cache coherence. During checkpointing, the ON will block both local and remote reads to the committing data to prevent applications from reading intermediate, inconsistent data.

After the CN receives successful replies from all the ONs, it deletes its old local data and moves to the new, committed version. At this point, the whole system has a coherent view of the new data and has at least N copies of it.

Commit phase 3. In the last phase, the CN informs all ONs that the *commit* operation has succeeded. The ONs then delete their redo logs.

Committing to a single ON and to local ON. When only one remote ON is involved in a *commit* operation, there is no need to coordinate multiple ONs and Hotpot performs the above commit protocol in a single phase.

The CN can also be the ON of committing data. In this case, the CN performs the *commit* operation locally. Since all local dirty pages are the COW of old committed pages, CN already has an undo and a redo copy of the committing data and does not need to create any other redo log as in remote ON's phase 1.

2.6.3 MRSW Mode

The MRSW mode allows only one writer to a PM page at a time to trade parallelism for stronger consistency. MRSW meets **R1.a**, **R2.b**, and **R3**.

Traditional MRSW protocols in DSM systems are usually invoked at every memory store (*e.g.*, to update cached read copies, to revoke current writer's write permission). Unlike DSM systems, DSPM applications store and manage persistent data; they do not need to ensure

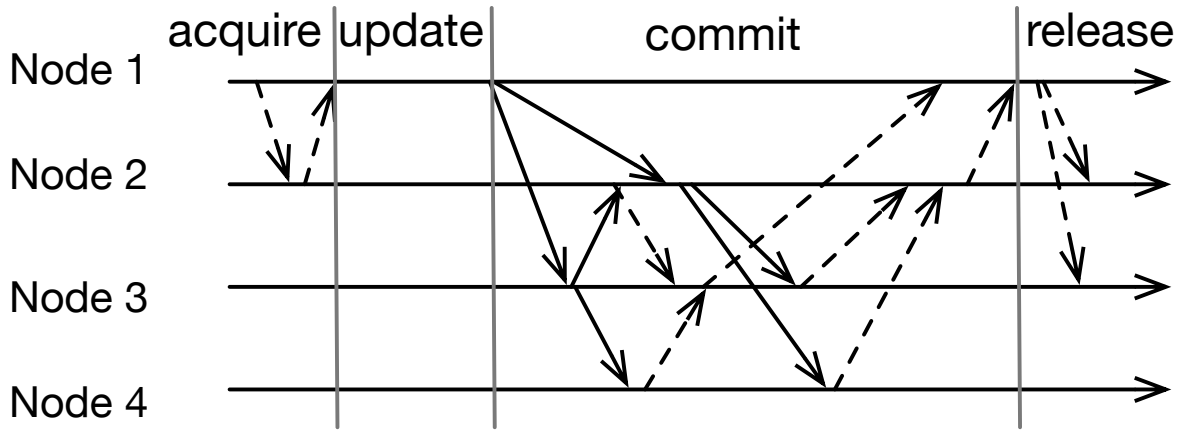


Figure 2.8: MRSW Example. Node 1 (CN) first acquires write permission from Node 2 (MN) before writing data. It then commits the new data to ONs at Node 2 and 3 with replication degree four and finally releases the write permission to MN.

coherence on every memory store, since they have well-defined points of when they want to start updating data and when they want to commit. To avoid the cost of invoking coherence events on each memory store while ensuring only one writer at a time, Hotpot uses an *acquire* API for applications to indicate the data areas they want to update. Afterwards, applications can update any date that they have acquired and use the *commit* call to both commit updates and release corresponding data areas. Figure 2.8 shows an example of MRSW.

Acquire write permission. Hotpot uses a master node (*MN*) to maintain the active writer of each page. An MN can be one of the Hotpot node, the CD, or a dedicated node. When a node receives the *acquire* call, it sends the virtual addresses of the data specified in the call to the MN. If the MN finds that at least one of these addresses are currently being written to, it will reject the *acquire* request and let the requesting node retry later.

Commit and release data. MRSW's commit protocol is simpler and more efficient than MRMW's, since there is no concurrent commit operations to the same data in MRSW (concurrent commit to different data pages is still allowed). MRSW combines phase 1 and phase 2 of the MRMW commit protocol into a single phase where the CN sends committing data to all ONs and

Table 2.2: Crash and Recovery Scenarios. Columns represent crashing node, if PM is accessible after crash, time of crash, and actions taken at recovery. NC represents non-commit time.

	Node	PM	Time	Action
	any	Y	any	resume normal operation after reboot
	CD	N	any	reconstruct using mirrored copy
	ON	N	NC	promote an existing DN to ON
	DN	N	NC	reconstruct data to meet replication degree
MRMW Commit	CN/ON	N	p1	undo commit, ONs delete redo logs
	CN	N	p2	redo commit, ONs send new data to DNs
	ON	N	p2	redo commit, CN sends new data to new ON
	DN	N	p2	continue commit, ON sends data to new DN
	CN	N	p3	complete commit, ONs delete redo logs
	ON/DN	N	p3	complete commit, new chunk reconstructed using committed data
MRSW	CN	N	commit	undo commit, ONs send old data to DNs
	ON	N	commit	CN redo commit from scratch
	CN	N	release	complete commit, release data
	ON/DN	N	release	complete commit, new chunk reconstructed using committed data

all ONs commit data on their own. Each ON individually handles commit in the same way as in the MRMW mode, except that it does not need to coordinate with any other ONs or the CN. ON directly proceeds to propagating data to DNs after it has written its own redo log.

At the end of the commit process, the CN informs the ONs to delete their redo logs (same as MRMW commit phase 3) and the MN to release the data pages.

2.6.4 Crash Recovery

Hotpot can safely recover from different crash scenarios without losing applications' data. Hotpot detects node failures by request timeout and by periodically sending heartbeat messages from the CD to all Hotpot nodes. We now explain Hotpot's crash recovery mechanism in the following four crash scenarios. Table 2.2 summarizes various crash scenarios and Hotpot's recovery mechanisms.

Recovering CD and MN. CD maintains node membership and dataset name mappings. Hotpot currently uses one CD but can be easily extended to include a hot stand-by CD (*e.g.*, using

Mojim [346]).

MN tracks which node has acquired write access to a page under the MRSW mode. Hotpot does not make this information persistent and simply reconstructs it by contacting all other nodes during recovery.

Non-commit time crashes. Recovering from node crashes during non-commit time is fairly straightforward. If the PM in the crashed node is accessible after the crash (we call it *with-PM failure*), Hotpot directly restarts the node and lets applications access data in PM. As described in Section 2.6.1, Hotpot ensures crash consistency of a single node. Thus, Hotpot can always recover to a consistent state when PM survives a crash. Hotpot can sustain arbitrary number of with-PM failures concurrently.

When a crash results in corrupted or inaccessible PM (we call it *no-PM failure*), Hotpot will reconstruct the lost data using redundant copies. Hotpot can sustain $N-1$ concurrent no-PM failures, where N is the user-defined degree of replication.

If a DN chunk is lost, the ON of this chunk will check what data pages in the chunk have dropped below user-defined replication degree and replicating them on the new node that replaces the failed node. There is no need to reconstruct the rest of the DN data; Hotpot simply lets the new node access them on demand.

When an ON chunk is lost, it is critical to reconstruct it quickly, since an ON serves both remote data read and commit operations. Instead of reconstructing a failed ON chunk from scratch, Hotpot promotes an existing DN chunk to an ON chunk and creates a new DN chunk. The new ON will fetch locally-missing committed data from other nodes and reconstruct ON metadata for the chunk. Our evaluation results show that it takes at most 2.3 seconds to promote a 1GB DN chunks to ON.

Crash during commit. If a with-PM failure happens during a *commit* call, Hotpot will just continue its commit process after restart. When a no-PM failure happens during commit, Hotpot takes different actions to recover depending on when the failure happens.

For MRMW commit, if no-PM failure happens before all the ONs have created the persistent redo logs (*i.e.*, before starting phase 2), Hotpot will undo the commit and revert to the old committed state by deleting the redo logs at ONs. If a no-PM failure happens after all ONs have written the committing data to their persistent redo logs (*i.e.*, after commit phase 1), Hotpot will redo the commit by replaying redo logs.

For MRSW, since we combine MRMW's phase 1 and phase 2 into one commit phase, we will not be able to tell whether or not an ON has pushed the committing data to DNs when this ON experience a no-PM failure. In this case, Hotpot will let CN redo the commit from scratch. Even if the crashed ON has pushed updates to some DNs, the system is still correct after CN redo the commit; it will just have more redundant copies. When the CN fails during MRSW commit, Hotpot will undo the commit by letting all ONs delete their redo logs and send old data to DNs to overwrite DNs' updated data.

During commit, Hotpot only supports either CN no-PM failure or ON no-PM failure. We choose not to support concurrent CN and ON no-PM failures during commit, because doing so largely simplifies Hotpot's commit protocol and improves its performance. Hotpot's commit process is fast (under 250 μs with up to 16 nodes, see Section 2.8.4). Thus, the chance of CN and ON both fail and lose their PM during commit is very small. Hotpot always supports DN no-PM failures during commit regardless of whether there are concurrent CN or ON failure.

2.7 Network Layer

The networking delay in DSPM systems is crucial to their overall performance. We implement Hotpot's network communication using RDMA. RDMA provides low-latency, high-bandwidth direct remote memory accesses with low CPU utilization. Hotpot's network layer is based on LITE [315], an efficient RDMA software stack we built in the Linux kernel on top of the RDMA native APIs, *Verbs* [209].

Most of Hotpot's network communication is in the form of RPC. We implemented a customized RPC-like interface in our RDMA layer based on the two-sided RDMA send and receive semantics. We further built a multicast RPC interface where one node can send a request to multiple nodes in parallel and let them each perform their processing functions and reply with the return values to the sending node. Similar to the findings from recent works [153], two-sided RDMA works better and is more flexible for these RPC-like interfaces than one-sided RDMA.

To increase network bandwidth, our RDMA layer enables multiple connections between each pair of nodes. It uses only one busy polling thread per node to poll a shared ring buffer for all connections, which delivers low-latency performance while keeping CPU utilization low. Our customized RDMA layer achieves an average latency of 7.9μs to perform a Hotpot remote page read. In comparison, IPoIB, a standard IP layer on top of Verbs, requires 77μs for a round trip with the same size.

2.8 Applications and Evaluation

This section presents the performance evaluation of two applications and a set of microbenchmarks. We ran all experiments on a cluster of 17 machines, each with two Intel Xeon CPU E5-2620 2.40GHz processors, 128 GB DRAM, and one 40 Gbps Mellanox ConnectX-3 InfiniBand network adapter; a Mellanox 40 Gbps InfiniBand switch connects all of the machines. All machines run the CentOS 7.1 distribution and the 3.11.1 Linux kernel.

The focus of our evaluation is to understand the performance of DSPM's distributed memory model, its commit protocols, and its data persistence cost. As there is no real PM in production yet, we use DRAM as stand-in for PM. A previous study [336] shows that even though PM and DRAM can have some performance difference, the difference is small and has much lower impact on application performance than the cost of flushing data from CPU caches to PM, which we have included in Hotpot and can measure accurately.

2.8.1 Systems in Comparison

We compare Hotpot with one in-memory file system, two PM-based file systems, one replicated PM-based system, and three distributed shared memory systems. Below we briefly describe these systems in comparison.

Single-Node File Systems. Tmpfs is a Linux file system that stores all data in main memory and does not perform any I/Os to storage devices. PMFS [93] is a file system designed for PM. The key difference between PMFS and a conventional file system is that its implementation of *mmap* maps the physical PM pages directly into the applications' address spaces rather than moving them back and forth between the file store and the buffer cache. PMFS ensures data persistence using `sfence` and `clflush` instructions.

Distributed PM-Based Systems Octopus [200] is a user-level RDMA-based distributed file system designed for PM. Octopus provides a set of customized file APIs including read and write, but does not support memory-mapped I/Os or provide data reliability and availability.

Mojim [346] is our previous work that uses a primary-backup model to replicate PM data over a customized IB layer. Similar to Hotpot, PMFS, and Octopus, Mojim maps PM pages directly into application virtual memory address spaces. Mojim supports application reads and writes on the primary node but only reads on backup nodes.

Distributed Shared Memory Systems. We implemented two kernel-level DSM systems, *DSM-Xact* and *DSM-NoXact*, on top of the same network stack as Hotpot's. Both of them support multiple readers and single writer (MRSW) and use a home node for each memory page to serve remote read and to store which nodes are the current readers and writer of the page, similar to HLRC [189, 349]. We open source both these DSM systems together with Hotpot.

DSM-Xact guarantees release consistency using a transaction interface that is similar to Hotpot's MRSW mode. Applications first call a transaction begin API to specify the data that they want to write. Transaction begin only succeeds if no other writer is writing to any of the transaction data. After beginning a transaction, applications can read and write to any transaction

Table 2.3: YCSB Workload Properties. The percentage of operations in each YCSB workload. R&U stands for Read and Update.

Workload	Read	Update	Scan	Insert	R&U
A	50%	50%	-	-	-
B	95%	5%	-	-	-
C	100%	-	-	-	-
D	95%	-	-	5%	-
E	-	-	95%	5%	-
F	50%	-	-	-	50%

data and use a transaction commit call to end a transaction. When committing a transaction, DSM-Xact writes all updated transaction data to the home node, invalidates the read caches on all other nodes, and releases the write permission.

DSM-NoXact supports write (memory stores) without transactions and does not require applications to declare which data they want to write in advance. On each write (memory store), DSM-NoXact revokes the write permission from the current writer, writes the current dirty data to the home node, and grants the write permission to the new writer. Compared to DSM-Xact, DSM-NoXact supports stronger consistency, requires less programmer efforts, but incurs higher performance overhead because of its more frequent writer invalidation.

Apart from the two DSM systems that we built, we also compare Hotpot with Grappa [224], a recent DSM system that supports modern data-parallel applications. Different from traditional DSM systems and our DSM systems, Grappa moves computation to data instead of fetching data to where computation is.

2.8.2 In-Memory NoSQL Database

MongoDB [218] is a popular distributed NoSQL database that supports several different storage engines including its own storage engine that is based on memory-mapped files (called MMAPv1). Applications like MongoDB can largely benefit from having a fast means to store and access persistent data. We ported MongoDB v2.7.0 to Hotpot by modifying its storage engine

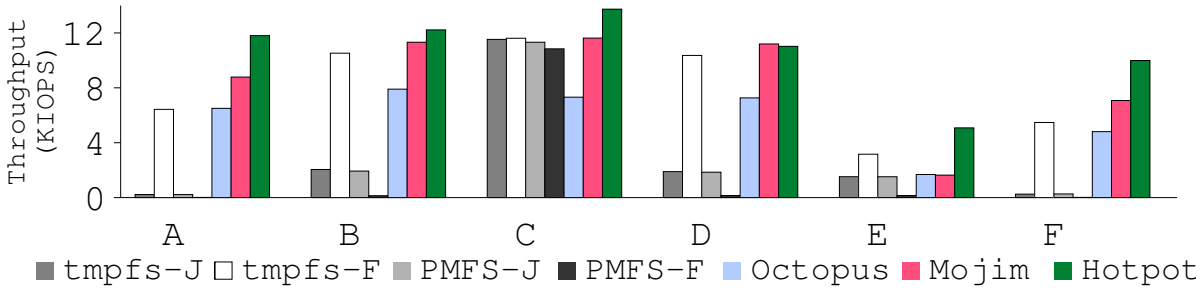


Figure 2.9: YCSB Workloads Throughput.

to keep track of all writes to the memory-mapped data file. We then group the written memory regions belonging to the same client request into a Hotpot *commit* call. In total, porting MongoDB to Hotpot requires modifying 120 lines of code.

To use the ported MongoDB, administrators can simply configure several machines to share a DSPM space under Hotpot and run ported MongoDB on each machine. Applications on top of the ported MongoDB can issue requests to any machine, since all machines access the same DSPM space. In our experiments, we ran the ported MongoDB on three Hotpot nodes and set data replication degree to three.

We compare this ported MongoDB with the default MongoDB running on tmpfs, PMFS, and Octopus, and a ported MongoDB to Mojim on three nodes connected with IB. Because Octopus does not memory-mapped operations and MongoDB's storage engine is based on memory-mapped files, MongoDB cannot directly run on Octopus. We run MongoDB on top of FUSE [2], a full-fledged user-level file system, which in turn runs on Octopus.

For tmpfs and PMFS, we use two consistency models (called MongoDB write concerns): the JOURNALED write concern and the FSYNC_SAFE write concern. With the JOURNALED write concern, MongoDB logs data in a journal file and checkpoints the data in a lazy fashion. MongoDB blocks a client call until the updated data is written to the journal file. With FSYNC_SAFE, MongoDB does not perform journaling. Instead, it flushes all the dirty pages to

the data file after each write operation and blocks the client call until this operation completes. We run Octopus and Mojim with the FSYNC_SAFE write concern. Octopus, tmpfs, and PMFS provide no replication, while Mojim and Hotpot use their own replication mechanisms to make three replicas of all data (Mojim uses one node as the primary node and the other two nodes as backup nodes).

YCSB [74] is a key-value store benchmark that imitates web applications' data access models. Figure 2.3 summarizes the number of different operations in the YCSB workloads. Each workload performs 10,000 operations on a database with 100,000 1 KB records. Figure 2.9 presents the throughput of MongoDB on tmpfs, PMFS, Octopus, Mojim, and Hotpot using YCSB workloads.

For all workloads, Hotpot outperforms tmpfs, PMFS, Octopus, and Mojim for both the JOURNALED and the FSYNC_SAFE write concerns. The performance improvement is especially high for write-heavy workloads. PMFS performs worst mainly because of its inefficient process of making data persistent with default MongoDB. The default MongoDB `fsync`s the whole data file after each write under FSYNC_SAFE, and PMFS flushes all cache lines of the file to PM by performing one `clflush` at a time. Hotpot and Mojim only commit dirty data, largely improving MongoDB performance over PMFS. Compared to tmpfs and PMFS under JOURNALED, Hotpot and Mojim use their own mechanisms to ensure data reliability and avoid the performance cost of journaling. Moreover, Hotpot and Mojim make three persistent replica for all data, while PMFS makes only one. Tmpfs is slower than Hotpot even though tmpfs does not make any data persistent, because MongoDB's slower replication mechanism on IPoIB. Hotpot's network layer is significantly better than IPoIB [315].

Octopus performs worse than Hotpot and Mojim because it incurs significant overhead of additional *indirection layers*: each memory operation within the memory-mapped file goes through the FUSE file system and then through Octopus. Hotpot and Mojim both support native memory instructions and incurs no indirection overhead. Finally, even though Mojim's replication

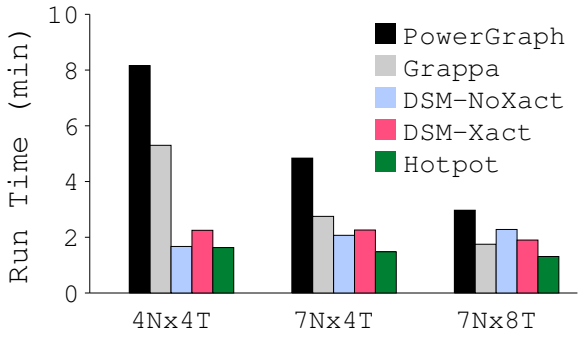


Figure 2.10: Pagerank Total Run Time. N stands for total number of nodes, T stands for number of threads running on a node.

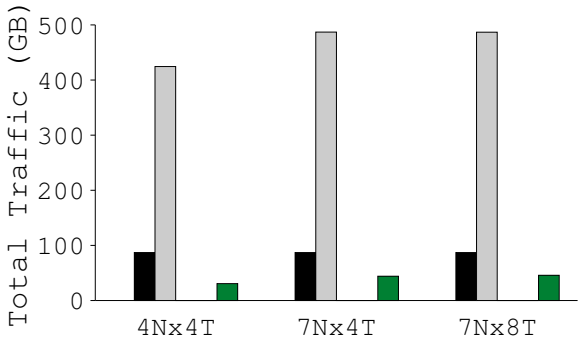


Figure 2.11: Pagerank Total Network Traffic.

protocol is simpler and faster than Hotpot's, Hotpot outperforms Mojim because Mojim only supports write on one node while Hotpot supports write on all nodes.

2.8.3 Distributed (Persistent) Graph

Graph processing is an increasingly important type of applications in modern datacenters [119, 120, 176, 198, 199, 203]. Most graph systems require large memory to run big graphs. Running graph algorithms on PM not only enables them to exploit the big memory space the high-density PM provides, but can also enable graph algorithms to stop and resume in the middle

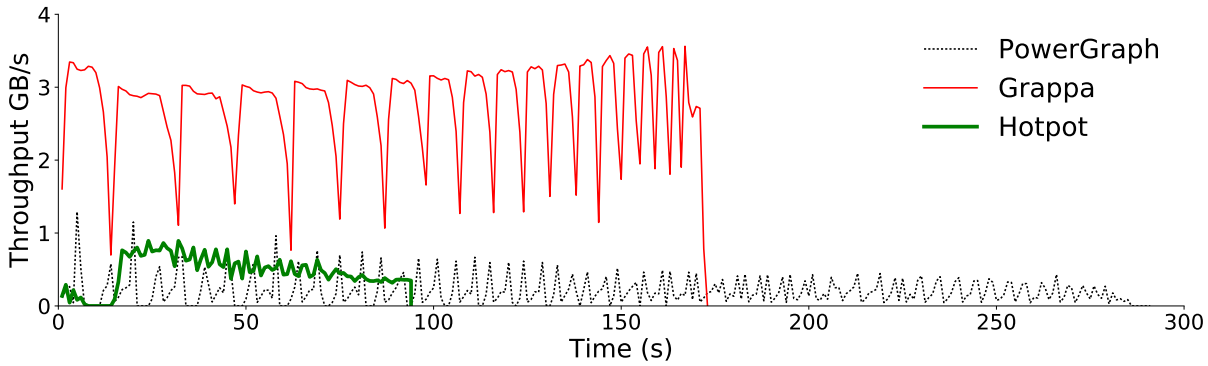


Figure 2.12: Pagerank Network Traffic Over Time.

of a long run.

We implemented a distributed graph processing engine on top of Hotpot based on the PowerGraph design [119]. It stores graphs with vertex-centric representation in DSPM with random order of vertices and distributes graph processing load to multiple threads across all Hotpot nodes. Each thread performs graph algorithms on a set of vertices in three steps: gather, apply, and scatter, with the optimization of delta caching [119]. After each step, we perform a global synchronization with *thread-barrier* and only start the next step when all threads have finished the last step. At the scatter step, the graph engine uses Hotpot's MRSW *commit* to make local changes of the scatter values visible to all nodes in the system. We implemented the Hotpot graph engine with only around 700 lines of code. Similarly, we implemented two distributed graph engines on top of DSM-Xact and DSM-NoXact; these engines differ from Hotpot's graph engine only in the way they perform data write and commit.

We compare Hotpot's graph engine with DSM-Xact, DSM-NoXact, PowerGraph, and Grappa [224] with two real datasets, Twitter (41 M vertices, 1 B directed edges) [175] and LiveJournal (4 M vertices, 34.7 M undirected edges) [186]. For space reason, we only present the results of the Twitter graph, but the results of LiveJournal are similar. Figure 2.10 shows the total run time of the PageRank [178] algorithm with Hotpot, DSM-Xact, DSM-NoXact, PowerGraph,

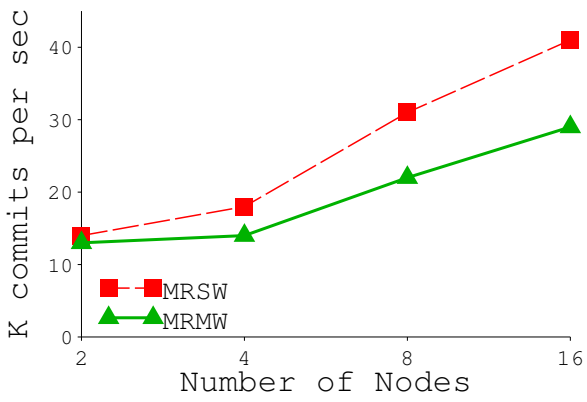


Figure 2.13: Hotpot Scalability. Commit throughput with 2 to 16 nodes.

and Grappa under three system settings: four nodes each running four graph threads, seven nodes each running four threads, and seven nodes each running eight threads.

Hotpot outperforms PowerGraph by 2.3× to 5× and Grappa by 1.3× to 3.2×. In addition, Hotpot makes all intermediate results of graph persistent for fast restart. A major reason why Hotpot outperforms PowerGraph and Grappa even when Hotpot requires data persistence and replication is Hotpot's network stack. Compare to the IPoIB used in PowerGraph and Grappa's own network stack, Hotpot's RDMA stack is more efficient.

Our implementation of DSM-Xact and DSM-NoXact use the same network stack as Hotpot, but Hotpot still outperforms DSM-NoXactḊSM-NoXact ensures cache coherence on every write and thus incurs much higher performance overhead than Hotpot and DSM-Xact.

To further understand the performance differences, we traced the network traffic of these three systems. Figure 2.11 plots the total amount of traffic and Figure 2.12 plots a detailed trace of network activity of the 7Nx4T setting. Hotpot sends less total traffic and achieves higher bandwidth than PowerGraph and Grappa.

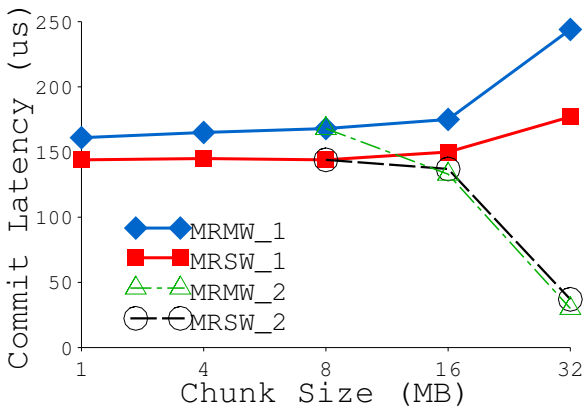


Figure 2.14: Chunk Size. For 16 MB and 32 MB cases, 1 represents ON being remote and 2 represents CN being ON.

2.8.4 Micro-Benchmark Results

We now present our microbenchmark results that evaluate the effect of different system settings and parameters. Since Hotpot reads have a constant latency (around 7.9μs) and Hotpot writes do not go through network, Hotpot's performance is largely affected by its data commit process. Because of space reasons, we focus our microbenchmark experiments on commit.

Scalability. Figure 2.13 shows the total commit throughput of Hotpot on 2 to 16 nodes with a workload that lets all nodes concurrently commit 32 random 4 KB areas with replication degree 1. Overall, both MRMW and MRSW commit scale. As expected, MRMW *commit* is more costly than MRSW.

Replication degree and committing size. We next evaluate the effect of replication degree and the total amount of data in a *commit* call. As expected, with higher replication degree and with more committing data, *commit* takes longer for both MRMW and MRSW. Because of space reasons, we do not include figures for these experiments.

Chunk size. We use a controlled microbenchmark to showcase the effect of chunk size (Figure 2.14). Each run has one node in a cluster of four nodes committing 32 1 KB areas that span a 32 MB region evenly with replication degree 1. Since Hotpot distributes chunks in Round

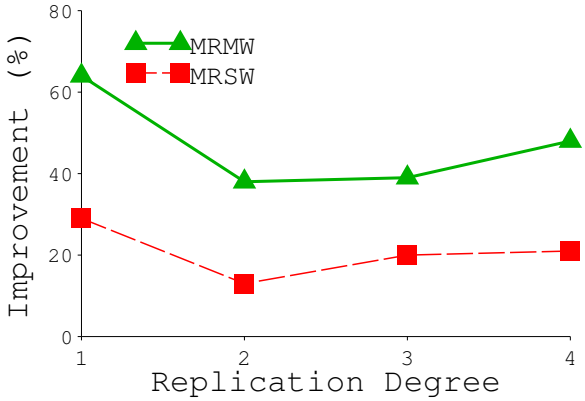


Figure 2.15: ON Migration. The improvement of average commit latency with ON migration over no migration.

Robin, when chunk size is below 8 MB, the 32 MB region will be distributed equally to all four nodes. The *commit* performance stays similar with 1, 4, and 8 MB chunk size, since *commit* will always use all four nodes as ONs. When chunk size is 16 MB (or 32 MB), only two (or one) nodes are ON. We observe two different behaviors: when the CN happens to also be the ON of the chunk that contains the committing data, the *commit* performance is better than when chunk size is below 8 MB, since half (or all) commit happens locally at the CN. But when the CN is not ON, all commit traffic goes to only two (or one) remote nodes, resulting in worse performance than when chunk size is small. This result suggest that smaller chunk size has better load balancing.

ON migration. From the previous experiments, we find that the *commit* performance depends heavily on the location of ON and the initial Hotpot ON assignment may not be optimal. We now evaluate how effective Hotpot's ON migration technique is in improving *commit* performance (Figure 2.15). We ran a workload with Zipf distribution to model temporal locality in datacenter applications [37, 55] on four nodes with replication degree 1 to 4. Each node issues 100,000 *commit* calls to commit two locations generated by Zipf. With ON migration, the *commit* performance improves by 13% to 29% under MRSW and 38% to 64% under MRMW. ON migration improves performance because the node that performs most *commit* on a chunk

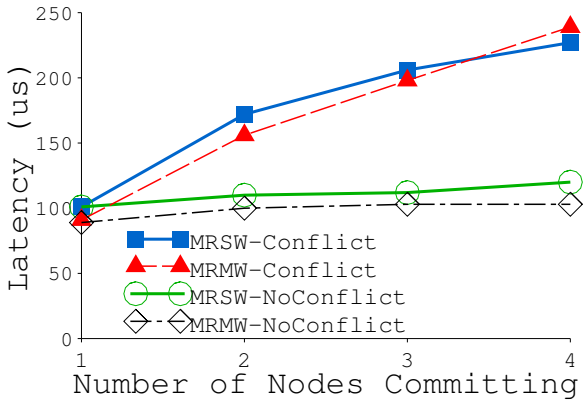


Figure 2.16: Commit Conflict. Average commit latency with and without conflict.

becomes its ON after migration. The improvement is most significant with replication degree one, because when CN is ON and replication degree is one, there is no need to perform any network communication. MRMW's improvement is higher than MRSW, because MRMW can benefit more from committing data locally — the MRMW commit process that involves remote ONs is more costly than that of MRSW.

Effect of conflict commits. Figure 2.16 compares the *commit* performance of when 1 to 4 nodes in a four node cluster concurrently commit data in two scenarios: all CNs commit the same set of data (32 sequential 1 KB areas) at the same time which results in commit conflict, and CNs use different set of data without any conflict. Commit conflict causes degraded performance, and the degradation is worse with more conflicting nodes. However, conflict is rare in reality, since *commit* is fast. Conflict only happens when different nodes commit the same data page at exactly the same time. In fact, we had to manually synchronize all nodes at every *commit* call using *thread-barrier* to create conflict.

2.9 Related Work

There have been a host of distributed shared memory systems and distributed storage systems [14, 58, 84, 116, 173, 254, 309, 67, 118, 172, 341, 348, 299, 300, 349, 277] over the past few decades. While some of Hotpot's coherence protocols may resemble existing DSM systems, none of them manages persistent data. There are also many single-node PM systems [249, 167, 168, 73, 91, 93, 331, 240, 211, 72, 320], but they do not support distributed environments.

Octopus [200] is a user-level RDMA-based distributed PM file system developed in parallel with Hotpot. Octopus manages file system metadata and data efficiently in a pool of PM-equipped machines. Octopus provides a set of customized file APIs including read and write but not any memory-mapped interfaces. Octopus does not provide data reliability and high availability either. Hotpot's abstraction is memory based rather than file based, and it offers data reliability, availability, and different consistency levels.

Grappa [224] is a DSM system that supports modern data-parallel applications. Instead of fetching remote memory to a local cached copy, Grappa executes functions at the remote side. Hotpot is a DSPM system and lets applications store persistent data. It fetches remote data for both fast local access and data replication.

FaRM [152, 91] is an RDMA-based distributed system on battery-backed DRAM. RAMCloud is a low-latency distributed key-value store system that keeps a single copy of all data in DRAM [236] and replicates data on massive slower storages for fast recovery. The major difference between Hotpot and FaRM or RAMCloud is that FaRM and RAMCloud both adds a software indirection layer for key-value stores which can cause significant latency overhead over native load/store operations and obscures much of the performance of the underlying PM. Hotpot uses a memory-like abstraction and directly stores persistent data in PM. Hotpot also performs data persistence and replication differently and uses a different network layer based on two-sided

RDMA.

Crail [139] is an RDMA-based high-performance multi-tiered distributed storage system that integrates with the Apache Spark ecosystem [337]. Crail mainly consists of a file system that manages tiered storage resources (*e.g.*, DRAM, flash, disk) with flexible allocation policies across tiers. Hotpot is a pure PM-based system that exposes a memory-like interface.

PerDis [288] and Larchant [104, 287] use a distributed file system below a DSM layer. Unlike these systems, Hotpot is a single-layer system that provides shared memory access, data persistence, and reliability.

Our own previous work, Mojim [346], provides an efficient mechanism to replicate PM over IB using a primary-backup protocol. Hotpot is a DSPM system that provides a shared-memory abstraction and integrates cache coherence and data replication.

2.10 Conclusion

We presented Hotpot, a kernel-level DSPM system that provides applications with a shared persistent memory abstraction. Our evaluation results show that it is easy to port existing applications to Hotpot and the resulting systems significantly outperform existing solutions.

2.11 Acknowledgments

Chapter 2, in full, is a reprint of Yizhou Shan, Shin-Yeh Tsai, Yiying Zhang, "Distributed Shared Persistent Memory", *SoCC, 2017*. The dissertation author was the primary investigator and author of this paper.

Chapter 3

LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation

3.1 Introduction

For many years, the unit of deployment, operation, and failure in datacenters has been a *monolithic server*, one that contains all the hardware resources that are needed to run a user program (typically a processor, some main memory, and a disk or an SSD). This monolithic architecture is meeting its limitations in the face of several issues and recent trends in datacenters.

First, datacenters face a difficult bin-packing problem of fitting applications to physical machines. Since a process can only use processor and memory in the same machine, it is hard to achieve full memory and CPU resource utilization [44, 85, 205]. Second, after packaging hardware devices in a server, it is difficult to add, remove, or change hardware components in datacenters [98]. Moreover, when a hardware component like a memory controller fails, the entire server is unusable. Finally, modern datacenters host increasingly heterogeneous hardware [291, 257, 149, 15]. However, designing new hardware that can fit into monolithic servers and deploying them in datacenters is a painful and cost-ineffective process that often

limits the speed of new hardware adoption.

We believe that datacenters should break monolithic servers and organize hardware devices like CPU, DRAM, and disks as *independent, failure-isolated, network-attached components*, each having its own controller to manage its hardware. This *hardware resource disaggregation* architecture is enabled by recent advances in network technologies [274, 115, 206, 237, 144, 56] and the trend towards increasing processing power in hardware controller [281, 20, 54]. Hardware resource disaggregation greatly improves resource utilization, elasticity, heterogeneity, and failure isolation, since each hardware component can operate or fail on its own and its resource allocation is independent from other components. With these benefits, this new architecture has already attracted early attention from academia and industry [6, 133, 36, 191, 230, 156].

Hardware resource disaggregation completely shifts the paradigm of computing and presents a key challenge to system builders: *How to manage and virtualize the distributed, disaggregated hardware components?*

Unfortunately, existing kernel designs cannot address the new challenges hardware resource disaggregation brings, such as network communication overhead across disaggregated hardware components, fault tolerance of hardware components, and the resource management of distributed components. Monolithic kernels, microkernels [95], and exokernels [96] run one OS on a monolithic machine, and the OS assumes local accesses to shared main memory, storage devices, network interfaces, and other hardware resources in the machine. After disaggregating hardware resources, it may be viable to run the OS at a processor and remotely manage all other hardware components. However, remote management requires significant amount of network traffic, and when processors fail, other components are unusable. Multi-kernel OSes [47, 229, 329, 62] run a kernel at each processor (or core) in a monolithic computer and these per-processor kernels communicate with each other through message passing. Multi-kernels still assume local accesses to hardware resources in a monolithic machine and their message passing is over local buses instead of a general network. While existing OSes could be retrofitted

to support hardware resource disaggregation, such retrofitting will be invasive to the central subsystems of an OS, such as memory and I/O management.

We propose *splitkernel*, a new OS architecture for hardware resource disaggregation (Figure 3.2c). The basic idea is simple: *When hardware is disaggregated, the OS should be also*. A splitkernel breaks traditional operating system functionalities into loosely-coupled *monitors*, each running at and managing a hardware component. Monitors in a splitkernel can be heterogeneous and can be added, removed, and restarted dynamically without affecting the rest of the system. Each splitkernel monitor operates locally for its own functionality and only communicates with other monitors when there is a need to access resources there. There are only two global tasks in a splitkernel: orchestrating resource allocation across components and handling component failure.

We choose not to support coherence across different components in a splitkernel. A splitkernel can use any general network to connect its hardware components. All monitors in a splitkernel communicate with each other via *network messaging* only. With our targeted scale, explicit message passing is much more efficient in network bandwidth consumption than the alternative of implicitly maintaining cross-component coherence.

Following the splitkernel model, we built LegoOS, the *first* OS designed for hardware resource disaggregation. LegoOS is a distributed OS that appears to applications as a set of virtual servers (called *vNodes*). A vNode can run on multiple processor, memory, and storage components and one component can host resources for multiple vNodes. LegoOS cleanly separates OS functionalities into three types of *monitors*, process monitor, memory monitor, and storage monitor. LegoOS monitors share no or minimal states and use a customized RDMA-based network stack to communicate with each other.

The biggest challenge and our focus in building LegoOS is the separation of processor and memory and their management. Modern processors and OSes assume all hardware memory units including main memory, page tables, and TLB are local. Simply moving all memory hardware

and memory management software to across the network will not work.

Based on application properties and hardware trends, we propose a hardware plus software solution that cleanly separates processor and memory functionalities, while meeting application performance requirements. LegoOS moves all memory hardware units to the disaggregated memory components and organizes all levels of processor caches as virtual caches that are accessed using virtual memory addresses. To improve performance, LegoOS uses a small amount (*e.g.*, 4 GB) of DRAM organized as a virtual cache below current last-level cache.

LegoOS process monitor manages application processes and the extended DRAM-cache. Memory monitor manages all virtual and physical memory space allocation and address mappings. LegoOS uses a novel two-level distributed virtual memory space management mechanism, which ensures efficient foreground memory accesses and balances load and space utilization at allocation time. Finally, LegoOS uses a space- and performance-efficient memory replication scheme to handle memory failure.

We implemented LegoOS on the x86-64 architecture. LegoOS is fully backward compatible with Linux ABIs by supporting common Linux system call APIs. To evaluate LegoOS, we emulate disaggregated hardware components using commodity servers. We evaluated LegoOS with microbenchmarks, the PARSEC benchmarks [49], and two unmodified datacenter applications, Phoenix [264] and TensorFlow [12]. Our evaluation results show that compared to monolithic Linux servers that can hold all the working sets of these applications, LegoOS is only 1.3× to 1.7× slower with 25% of application working set available as DRAM cache at processor components. Compared to monolithic Linux servers whose main memory size is the same as LegoOS' DRAM cache size and which use local SSD/DRAM swapping or network swapping, LegoOS' performance is 0.8× to 3.2×. At the same time, LegoOS largely improves resource packing and reduces system mean time to failure.

Overall, this work makes the following contributions:

- We propose the concept of splitkernel, a new OS architecture that fits the hardware resource

disaggregation architecture.

- We built LegoOS, the first OS that runs on and manages a disaggregated hardware cluster.
- We propose a new hardware architecture to cleanly separate processor and memory hardware functionalities, while preserving most of the performance of monolithic server architecture.

LegoOS is publicly available at `LegoOS.io`.

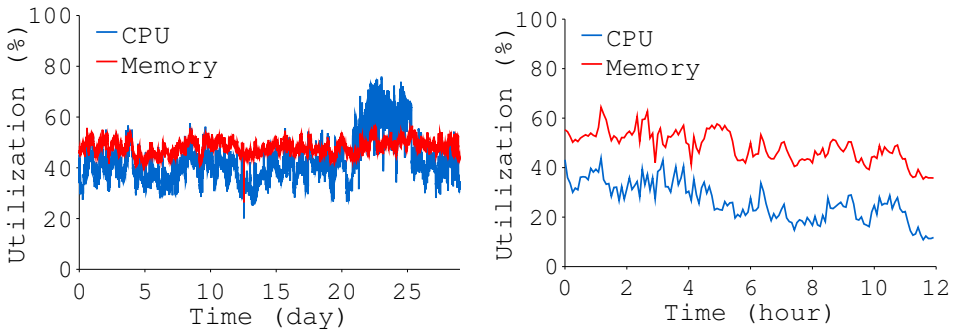
3.2 Disaggregate Hardware Resource

This section motivates the hardware resource disaggregation architecture and discusses the challenges in managing disaggregated hardware.

3.2.1 Limitations of Monolithic Servers

A monolithic server has been the unit of deployment and operation in datacenters for decades. This long-standing *server-centric* architecture has several key limitations.

Inefficient resource utilization. With a server being the physical boundary of resource allocation, it is difficult to fully utilize all resources in a datacenter [44, 85, 205]. We analyzed two production cluster traces: a 29-day Google one [122] and a 12-hour Alibaba one [21]. Figure 3.1 plots the aggregated CPU and memory utilization in the two clusters. For both clusters, only around half of the CPU and memory are utilized. Interestingly, a significant amount of jobs are being evicted at the same time in these traces (*e.g.*, evicting low-priority jobs to make room for high-priority ones [317]). One of the main reasons for resource under-utilization in these production clusters is the constraint that CPU and memory for a job have to be allocated from the same physical machine.



(a) Google Cluster.

(b) Alibaba Cluster.

Figure 3.1: Data center resource utilization.

Poor hardware elasticity. It is difficult to add, move, remove, or reconfigure hardware components after they have been installed in a monolithic server [98]. Because of this rigidity, datacenter owners have to plan out server configurations in advance. However, with today's speed of change in application requirements, such plans have to be adjusted frequently, and when changes happen, it often comes with waste in existing server hardware.

Coarse failure domain. The failure unit of monolithic servers is coarse. When a hardware component within a server fails, the whole server is often unusable and applications running on it can all crash. Previous analysis [278] found that motherboard, memory, CPU, power supply failures account for 50% to 82% of hardware failures in a server. Unfortunately, monolithic servers cannot continue to operate when any of these devices fail.

Bad support for heterogeneity. Driven by application needs, new hardware technologies are finding their ways into modern datacenters [291]. Datacenters no longer host only commodity servers with CPU, DRAM, and hard disks. They include non-traditional and specialized hardware like GPGPU [123, 24], TPU [149], DPU [15], FPGA [257, 25], non-volatile memory [141], and NVMe-based SSDs [307]. The monolithic server model tightly couples hardware devices with each other and with a motherboard. As a result, making new hardware devices work with existing servers is a painful and lengthy process [257]. Mover, datacenters often need to purchase new servers to host certain hardware. Other parts of the new servers can go underutilized and old

servers need to retire to make room for new ones.

3.2.2 Hardware Resource Disaggregation

The server-centric architecture is a bad fit for the fast-changing datacenter hardware, software, and cost needs. There is an emerging interest in utilizing resources beyond a local machine [111], such as distributed memory [90, 224, 17, 233] and network swapping [126]. These solutions improve resource utilization over traditional systems. However, they cannot solve all the issues of monolithic servers (*e.g.*, the last three issues in §3.2.1), since their hardware model is still a monolithic one. To fully support the growing heterogeneity in hardware and to provide elasticity and flexibility at the hardware level, we should *break the monolithic server model.*

We envision a *hardware resource disaggregation* architecture where hardware resources in traditional servers are disseminated into network-attached *hardware components*. Each component has a controller and a network interface, can operate on its own, and is an *independent, failure-isolated* entity.

The disaggregated approach largely increases the flexibility of a datacenter. Applications can freely use resources from any hardware component, which makes resource allocation easy and efficient. Different types of hardware resources can *scale independently*. It is easy to add, remove, or reconfigure components. New types of hardware components can easily be deployed in a datacenter — by simply enabling the hardware to talk to the network and adding a new network link to connect it. Finally, hardware resource disaggregation enables fine-grain failure isolation, since one component failure will not affect the rest of a cluster.

Three hardware trends are making resource disaggregation feasible in datacenters. First, network speed has grown by more than an order of magnitude and has become more scalable in the past decade with new technologies like Remote Direct Memory Access (*RDMA*) [209] and new topologies and switches [36, 77, 76], enabling fast accesses of hardware components

that are disaggregated across the network. InfiniBand will soon reach 200Gbps and sub-600 nanosecond speed [206], being only 2× to 4× slower than main memory bus in bandwidth. With main memory bus facing a bandwidth wall [269], future network bandwidth (at line rate) is even projected to exceed local DRAM bandwidth [310].

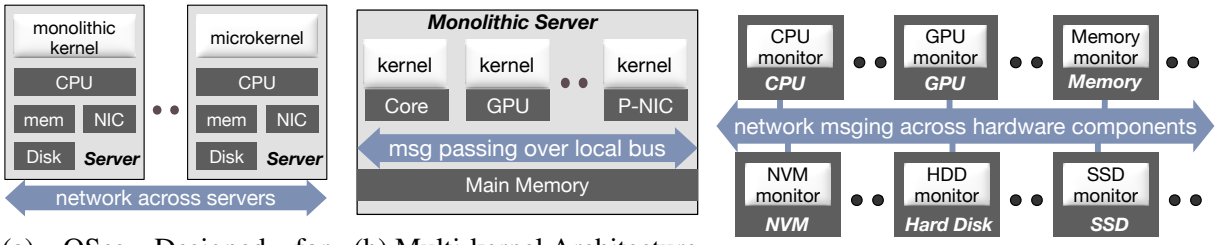
Second, network interfaces are moving closer to hardware components, with technologies like Intel OmniPath [142], RDMA [209], and NVMe over Fabrics [214, 69]. As a result, hardware devices will be able to access network directly without the need to attach any processors.

Finally, hardware devices are incorporating more processing power [20, 54, 207, 208, 225, 19], allowing application and OS logics to be offloaded to hardware [281, 158]. On-device processing power will enable system software to manage disaggregated hardware components locally.

With these hardware trends and the limitations of monolithic servers, we believe that future datacenters will be able to largely benefit from hardware resource disaggregation. In fact, there have already been several initial hardware proposals in resource disaggregation [6], including disaggregated memory [191, 232, 230], disaggregated flash [164, 166], Intel Rack-Scale System [143], HP "The Machine" [133, 100], IBM Composable System [68], and Berkeley Firebox [36].

3.2.3 OSes for Resource Disaggregation

Despite various benefits hardware resource disaggregation promises, it is still unclear how to manage or utilize disaggregated hardware in a datacenter. Unfortunately, existing OSes and distributed systems cannot work well with this new architecture. Single-node OSes like Linux view a server as the unit of management and assume all hardware components are local (Figure 3.2a). A potential approach is to run these OSes on processors and access memory, storage, and other hardware resources remotely. Recent disaggregated systems like soNUMA [232] take this approach. However, this approach incurs high network latency and bandwidth consumption



(a) OSes Designed for Monolithic Servers.

(b) Multi-kernel Architecture. P-NIC: programmable NIC.

(c) Splitkernel Architecture.

Figure 3.2: Operating System Architecture.

with remote device management, misses the opportunity of exploiting device-local computation power, and makes processors the single point of failure.

Multi-kernel solutions [47, 339, 229, 329, 62] (Figure 3.2b) view different cores, processors, or programmable devices within a server separately by running a kernel on each core/device and using message passing to communicate across kernels. These kernels still run in a single server and all access some common hardware resources in the server like memory and the network interface. Moreover, they do not manage distributed resources or handle failures in a disaggregated cluster.

There have been various distributed OS proposals, most of which date decades back [304, 243, 41]. Most of these distributed OSes manage a set of monolithic servers instead of hardware components.

Hardware resource disaggregation is fundamentally different from the traditional monolithic server model. A complete disaggregation of processor, memory, and storage means that when managing one of them, there will be no local accesses to the other two. For example, processors will have no local memory or storage to store user or kernel data. An OS also needs to manage distributed hardware resource and handle hardware component failure. We summarize the following key challenges in building an OS for resource disaggregation, some of which have previously been identified [100].

- How to deliver good performance when application execution involves the access of

network-partitioned disaggregated hardware and current network is still slower than local buses?

- How to locally manage individual hardware components with limited hardware resources?
- How to manage distributed hardware resources?
- How to handle a component failure without affecting other components or running applications?
- What abstraction should be exposed to users and how to support existing datacenter applications?

Instead of retrofitting existing OSes to confront these challenges, we take the approach of designing a new OS architecture from the ground up for hardware resource disaggregation.

3.3 The Splitkernel OS Architecture

We propose *splitkernel*, a new OS architecture for resource disaggregation. Figure 3.2c illustrates splitkernel's overall architecture. The splitkernel disseminates an OS into pieces of different functionalities, each running at and managing a hardware component. All components communicate by message passing over a common network, and splitkernel globally manages resources and component failures. Splitkernel is a general OS architecture we propose for hardware resource disaggregation. There can be many types of implementation of splitkernel. Further, we make no assumption on the specific hardware or network type in a disaggregated cluster a splitkernel runs on. Below, we describe four key concepts of the splitkernel architecture.

Split OS functionalities. Splitkernel breaks traditional OS functionalities into *monitors*. Each monitor manages a hardware component, virtualizes and protects its physical resources. Monitors in a splitkernel are loosely-coupled and they communicate with other monitors to access

remote resources. For each monitor to operate on its own with minimal dependence on other monitors, we use a stateless design by sharing no or minimal *states*, or metadata, across monitors.

Run monitors at hardware components. We expect each non-processor hardware component in a disaggregated cluster to have a controller that can run a monitor. A hardware controller can be a low-power general-purpose core, an ASIC, or an FPGA. Each monitor in a splitkernel can use its own implementation to manage the hardware component it runs on. This design makes it easy to integrate heterogeneous hardware in datacenters — to deploy a new hardware device, its developers only need to build the device, implement a monitor to manage it, and attach the device to the network. Similarly, it is easy to reconfigure, restart, and remove hardware components.

Message passing across non-coherent components. Unlike other proposals of disaggregated systems [133] that rely on coherent interconnects [115, 56, 237], a splitkernel runs on general-purpose network layer like Ethernet and neither underlying hardware nor the splitkernel provides cache coherence across components. We made this design choice mainly because maintaining coherence for our targeted cluster scale would cause high network bandwidth consumption. Instead, all communication across components in a splitkernel is through *network messaging*. A splitkernel still retains the coherence guarantee that hardware already provides within a component (*e.g.*, cache coherence across cores in a CPU), and applications running on top of a splitkernel can use message passing to implement their desired level of coherence for their data across components.

Global resource management and failure handling. One hardware component can host resources for multiple applications and its failure can affect all these applications. In addition to managing individual components, the splitkernel also needs to globally manage resources and failure. To minimize performance and scalability bottleneck, the splitkernel only involves global resource management occasionally for coarse-grained decisions, while individual monitors make their own fine-grained decisions. The splitkernel handles component failure by adding redundancy for recovery.

3.4 LegoOS Design

Based on the splitkernel architecture, we built *LegoOS*, the first OS designed for hardware resource disaggregation. LegoOS is a research prototype that demonstrates the feasibility of the splitkernel design, but it is not the only way to build a splitkernel. LegoOS' design targets three types of hardware components: processor, memory, and storage, and we call them *pComponent, mComponent*, and *sComponent*.

This section first introduces the abstraction LegoOS exposes to users and then describes the hardware architecture of components LegoOS runs on. Next, we explain the design of LegoOS' process, memory, and storage monitors. Finally, we discuss LegoOS' global resource management and failure handling mechanisms.

Overall, LegoOS achieves the following design goals:

- Clean separation of process, memory, and storage functionalities.
- Monitors run at hardware components and fit device constraints.
- Comparable performance to monolithic Linux servers.
- Efficient resource management and memory failure handling, both in space and in performance.
- Easy-to-use, backward compatible user interface.
- Supports common Linux system call interfaces.

3.4.1 Abstraction and Usage Model

LegoOS exposes a distributed set of *virtual nodes*, or *vNode*, to users. From users' point of view, a vNode is like a virtual machine. Multiple users can run in a vNode and each user can run multiple processes. Each vNode has a unique ID, a unique virtual IP address, and its own

storage mount point. LegoOS protects and isolates the resources given to each vNode from others. Internally, one vNode can run on multiple pComponents, multiple mComponents, and multiple sComponents. At the same time, each hardware component can host resources for more than one vNode. The internal execution status is transparent to LegoOS users; they do not know which physical components their applications run on.

With splitkernel's design principle of components not being coherent, LegoOS does not support writable shared memory across processors. LegoOS assumes that threads within the same process access shared memory and threads belonging to different processes do not share writable memory, and LegoOS makes scheduling decision based on this assumption (§3.4.3). Applications that use shared writable memory across processes (*e.g.*, with MAP_SHARED) will need to be adapted to use message passing across processes. We made this decision because writable shared memory across processes is rare (we have not seen a single instance in the datacenter applications we studied), and supporting it makes both hardware and software more complex (in fact, we have implemented this support but later decided not to include it because of its complexity).

One of the initial decisions we made when building LegoOS is to support the Linux system call interface and unmodified Linux ABI, because doing so can greatly ease the adoption of LegoOS. Distributed applications that run on Linux can seamlessly run on a LegoOS cluster by running on a set of vNodes.

3.4.2 Hardware Architecture

LegoOS pComponent, mComponent, and sComponent are independent devices, each having their own hardware controller and network interface (for pComponent, the hardware controller is the processor itself). Our current hardware model uses CPU in pComponent, DRAM in mComponent, and SSD or HDD in sComponent. We leave exploring other hardware devices for future work.

To demonstrate the feasibility of hardware resource disaggregation, we propose a pCom-

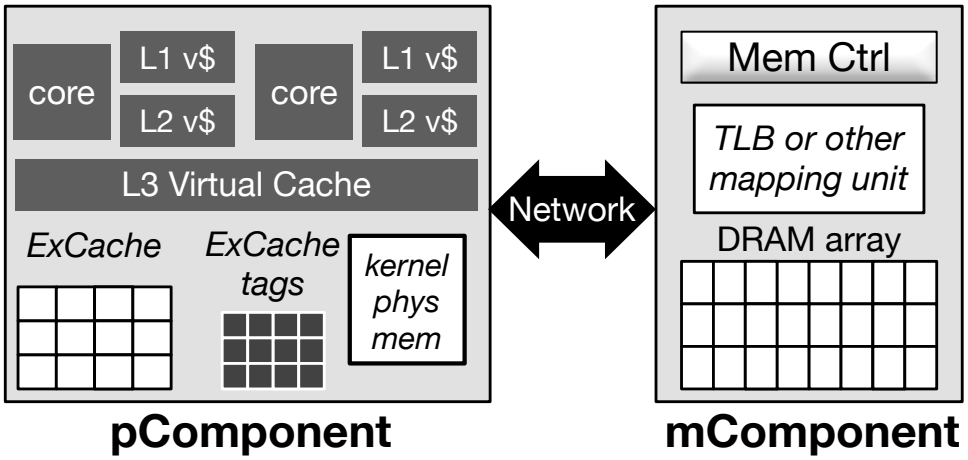


Figure 3.3: LegoOS pComponent and mComponent Architecture.

ponent and an mComponent architecture designed within today's network, processor, and memory performance and hardware constraints (Figure 3.3).

Separating process and memory functionalities. LegoOS moves all hardware memory functionalities to mComponents (e.g., page tables, TLBs) and leaves *only* caches at the pComponent side. With a clean separation of process and memory hardware units, the allocation and management of memory can be completely transparent to pComponents. Each mComponent can choose its own memory allocation technique and virtual to physical memory address mappings (*e.g.*, segmentation).

Processor virtual caches. After moving all memory functionalities to mComponents, pComponents will only see virtual addresses and have to use virtual memory addresses to access its caches. Because of this, LegoOS organizes all levels of pComponent caches as *virtual caches* [121, 327], *i.e.*, virtually-indexed and virtually-tagged caches.

A virtual cache has two potential problems, commonly known as synonyms and homonyms [296]. Synonyms happens when a physical address maps to multiple virtual addresses (and thus multiple virtual cache lines) as a result of memory sharing across processes, and the update of one virtual cache line will not reflect to other lines that share the data. Since

LegoOS does not allow writable inter-process memory sharing, it will not have the synonym problem. The homonym problem happens when two address spaces use the same virtual address for their own different data. Similar to previous solutions [46], we solve homonyms by storing an address space ID (ASID) with each cache line, and differentiate a virtual address in different address spaces using ASIDs.

Separating memory for performance and for capacity. Previous studies [111, 126] and our own show that today's network speed cannot meet application performance requirements if all memory accesses are across the network. Fortunately, many modern datacenter applications exhibit strong memory access temporal locality. For example, we found 90% of memory accesses in PowerGraph [119] go to just 0.06% of total memory and 95% go to 3.1% of memory (22% and 36% for TensorFlow [12] respectively, 5.1% and 6.6% for Phoenix [264]).

With good memory-access locality, we propose to leave a small amount of memory (*e.g.*, 4 GB) at each pComponent and move most memory across the network (*e.g.*, few TBs per mComponent). pComponents' local memory can be regular DRAM or the on-die HBM [147, 220], and mComponents use DRAM or NVM.

Different from previous proposals [191], we propose to organize pComponents' DRAM/HBM as cache rather than main memory for a clean separation of process and memory functionalities. We place this cache under the current processor Last-Level Cache (LLC) and call it an extended cache, or *ExCache*. ExCache serves as another layer in the memory hierarchy between LLC and memory across the network. With this design, ExCache can serve hot memory accesses fast, while mComponents can provide the capacity applications desire.

ExCache is a virtual, inclusive cache, and we use a combination of hardware and software to manage ExCache. Each ExCache line has a (virtual-address) tag and two access permission bits (one for read/write and one for valid). These bits are set by software when a line is inserted to ExCache and checked by hardware at access time. For best hit performance, the hit path of ExCache is handled purely by hardware — the hardware cache controller maps a virtual address

to an ExCache set, fetches and compares tags in the set, and on a hit, fetches the hit ExCache line. Handling misses of ExCache is more complex than with traditional CPU caches, and thus we use LegoOS to handle the miss path of ExCache (see §3.4.3).

Finally, we use a small amount of DRAM/HBM at pComponent for LegoOS' own kernel data usages, accessed directly with physical memory addresses and managed by LegoOS. LegoOS ensures that all its own data fits in this space to avoid going to mComponents.

With our design, pComponents do not need any address mappings: LegoOS accesses all pComponent-side DRAM/HBM using physical memory addresses and does simple calculations to locate the ExCache set for a memory access. Another benefit of not handling address mapping at pComponents and moving TLBs to mComponents is that pComponents do not need to access TLB or suffer from TLB misses, potentially making pComponent cache accesses faster [159].

3.4.3 Process Management

The LegoOS *process monitor* runs in the kernel space of a pComponent and manages the pComponent's CPU cores and ExCache. pComponents run user programs in the user space.

Process Management and Scheduling

At every pComponent, LegoOS uses a simple local thread scheduling model that targets datacenter applications (we will discuss global scheduling in § 3.4.6). LegoOS dedicates a small amount of cores for kernel background threads (currently two to four) and uses the rest of the cores for application threads. When a new process starts, LegoOS uses a global policy to choose a pComponent for it (§ 3.4.6). Afterwards, LegoOS schedules new threads the process spawns on the same pComponent by choosing the cores that host fewest threads. After assigning a thread to a core, we let it run to the end with no scheduling or kernel preemption under common scenarios. For example, we do not use any network interrupts and let threads busy wait on the completion of outstanding network requests, since a network request in LegoOS is fast (*e.g.*,

fetching an ExCache line from an mComponent takes around 6.5 μs). LegoOS improves the overall processor utilization in a disaggregated cluster, since it can freely schedule processes on any pComponents without considering memory allocation. Thus, we do not push for perfect core utilization when scheduling individual threads and instead aim to minimize scheduling and context switch performance overheads. Only when a pComponent has to schedule more threads than its cores will LegoOS start preempting threads on a core.

ExCache Management

LegoOS process monitor configures and manages ExCache. During the pComponent's boot time, LegoOS configures the set associativity of ExCache and its cache replacement policy. While ExCache hit is handled completely in hardware, LegoOS handles misses in software. When an ExCache miss happens, the process monitor fetches the corresponding line from an mComponent and inserts it to ExCache. If the ExCache set is full, the process monitor first evicts a line in the set. It throws away the evicted line if it is clean and writes it back to an mComponent if it is dirty. LegoOS currently supports two eviction policies: FIFO and LRU. For each ExCache set, LegoOS maintains a FIFO queue (or an approximate LRU list) and chooses ExCache lines to evict based on the corresponding policy (see §3.5.3 for details).

Supporting Linux Syscall Interface

One of our early decisions is to support Linux ABIs for backward compatibility and easy adoption of LegoOS. A challenge in supporting the Linux system call interface is that many Linux syscalls are associated with *states*, information about different Linux subsystems that is stored with each process and can be accessed by user programs across syscalls. For example, Linux records the states of a running process' open files, socket connections, and several other entities, and it associates these states with file descriptors (*fd*s) that are exposed to users. In contrast, LegoOS aims at the clean separation of OS functionalities. With LegoOS' stateless

design principle, each component only stores information about its own resource and each request across components contains all the information that the destination component needs to handle the request. To solve this discrepancy between the Linux syscall interface and LegoOS' design, we add a layer on top of LegoOS' core process monitor at each pComponent to store Linux states and translate these states and the Linux syscall interface to LegoOS' internal interface.

3.4.4 Memory Management

We use mComponents for three types of data: anonymous memory (*i.e.*, heaps, stacks), memory-mapped files, and storage buffer caches. The LegoOS *memory monitor* manages both the virtual and physical memory address spaces, their allocation, deallocation, and memory address mappings. It also performs the actual memory read and write. No user processes run on mComponents and they run completely in the kernel mode (same is true for sComponents).

LegoOS lets a process address space span multiple mComponents to achieve efficient memory space utilization and high parallelism. Each application process uses one or more mComponents to host its data and a *home mComponent*, an mComponent that initially loads the process, accepts and oversees all system calls related to virtual memory space management (*e.g.*, `brk`, *mmap*, `munmap`, and `mremap`). LegoOS uses a global memory resource manager (*GMM*) to assign a home mComponent to each new process at its creation time. A home mComponent can also host process data.

Memory Space Management

Virtual memory space management. We propose a two-level approach to manage distributed virtual memory spaces, where the home mComponent of a process makes coarse-grained, high-level virtual memory allocation decisions and other mComponents perform fine-grained virtual memory allocation. This approach minimizes network communication during both normal memory accesses and virtual memory operations, while ensuring good load balancing and

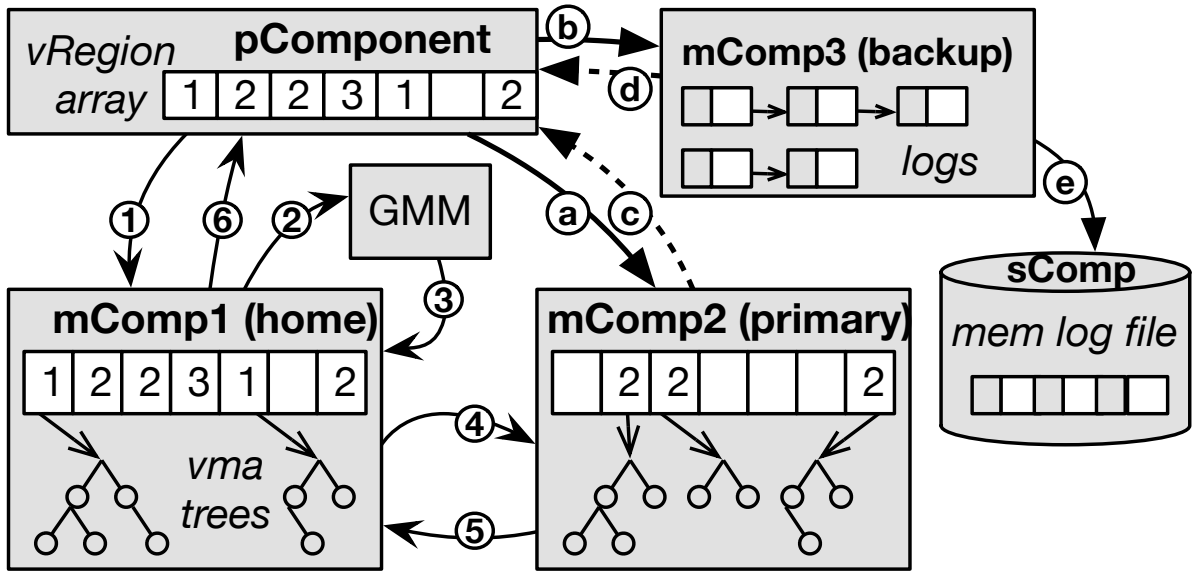


Figure 3.4: Distributed Memory Management.

memory utilization. Figure 3.4 demonstrates the data structures used.

At the higher level, we split each virtual memory address space into coarse-grained, fix-sized *virtual regions*, or *vRegions* (*e.g.*, of 1 GB). Each vRegion that contains allocated virtual memory addresses (an active vRegion) is *owned* by an mComponent. The owner of a vRegion handles all memory accesses and virtual memory requests within the vRegion.

The lower level stores user process virtual memory area (*vma*) information, such as virtual address ranges and permissions, in *vma trees*. The owner of an active vRegion stores a vma tree for the vRegion, with each node in the tree being one vma. A user-perceived virtual memory range can split across multiple mComponents, but only one mComponent owns a vRegion.

vRegion owners perform the actual virtual memory allocation and vma tree set up. A home mComponent can also be the owner of vRegions, but the home mComponent does not maintain any information about memory that belongs to vRegions owned by other mComponents. It only keeps the information of which mComponent owns a vRegion (in a *vRegion array*) and how much free virtual memory space is left in each vRegion. These metadata can be easily

reconstructed if a home mComponent fails.

When an application process wants to allocate a virtual memory space, the pComponent forwards the allocation request to its home mComponent (① in Figure 3.4). The home mComponent uses its stored information of available virtual memory space in vRegions to find one or more vRegions that best fit the requested amount of virtual memory space. If no active vRegion can fit the allocation request, the home mComponent makes a new vRegion active and contacts the GMM (② and ③) to find a candidate mComponent to own the new vRegion. GMM makes this decision based on available physical memory space and access load on different mComponents (§ 3.4.6). If the candidate mComponent is not the home mComponent, the home mComponent next forwards the request to that mComponent (④), which then performs local virtual memory area allocation and sets up the proper vma tree. Afterwards, the pComponent directly sends memory access requests to the owner of the vRegion where the memory access falls into (*e.g.*, ⓐ and ⓒ in Figure 3.4).

LegoOS' mechanism of distributed virtual memory management is efficient and it cleanly separates memory operations from pComponents. pComponents hand over all memory-related system call requests to mComponents and only cache a copy of the vRegion array for fast memory accesses. To fill a cache miss or to flush a dirty cache line, a pComponent looks up the cached vRegion array to find its owner mComponent and sends the request to it.

Physical memory space management. Each mComponent manages the physical memory allocation for data that falls into the vRegion that it owns. Each mComponent can choose their own way of physical memory allocation and own mechanism of virtual-to-physical memory address mapping.

Optimization on Memory Accesses

With our strawman memory management design, all ExCache misses will go to mComponents. We soon found that a large performance overhead in running real applications is caused by

filling empty ExCache, *i.e.*, *cold misses*. To reduce the performance overhead of cold misses, we propose a technique to avoid accessing mComponent on first memory accesses.

The basic idea is simple: since the initial content of anonymous memory (non-file-backed memory) is zero, LegoOS can directly allocate a cache line with empty content in ExCache for the first access to anonymous memory instead of going to mComponent (we call such cache lines *p-local lines*). When an application creates a new anonymous memory region, the process monitor records its address range and permission. The application's first access to this region will be an ExCache miss and it will trap to LegoOS. LegoOS process monitor then allocates an ExCache line, fills it with zeros, and sets its R/W bit according to the recorded memory region's permission. Before this p-local line is evicted, it only lives in the ExCache. No mComponents are aware of it or will allocate physical memory or a virtual-to-physical memory mapping for it. When a p-local cache line becomes dirty and needs to be flushed, the process monitor sends it to its owner mComponent, which then allocates physical memory space and establishes a virtual-to-physical memory mapping. Essentially, LegoOS *delays physical memory allocation until write time*. Notice that it is safe to only maintain p-local lines at a pComponent ExCache without any other pComponents knowing them, since pComponents in LegoOS do not share any memory and other pComponents will not access this data.

3.4.5 Storage Management

LegoOS supports a hierarchical file interface that is backward compatible with POSIX through its vNode abstraction. Users can store their directories and files under their vNodes' mount points and perform normal read, write, and other accesses to them.

LegoOS implements core storage functionalities at sComponents. To cleanly separate storage functionalities, LegoOS uses a stateless storage server design, where each I/O request to the storage server contains all the information needed to fulfill this request, *e.g.*, full path name, absolute file offset, similar to the server design in NFS v2 [276].

While LegoOS supports a hierarchical file use interface, internally, LegoOS storage monitor treats (full) directory and file paths just as unique names of a file and place all files of a vNode under one internal directory at the sComponent. To locate a file, LegoOS storage monitor maintains a simple hash table with the full paths of files (and directories) as keys. From our observation, most datacenter applications only have a few hundred files or less. Thus, a simple hash table for a whole vNode is sufficient to achieve good lookup performance. Using a non-hierarchical file system implementation largely reduces the complexity of LegoOS' file system, making it possible for a storage monitor to fit in storage devices controllers that have limited processing power [281].

LegoOS places the storage buffer cache at mComponents rather than at sComponents, because sComponents can only host a limited amount of internal memory. LegoOS memory monitor manages the storage buffer cache by simply performing insertion, lookup, and deletion of buffer cache entries. For simplicity and to avoid coherence traffic, we currently place the buffer cache of one file under one mComponent. When receiving a file read system call, the LegoOS process monitor first uses its extended Linux state layer to look up the full path name, then passes it with the requested offset and size to the mComponent that holds the file's buffer cache. This mComponent will look up the buffer cache and returns the data to pComponent on a hit. On a miss, mComponent will forward the request to the sComponent that stores the file, which will fetch the data from storage device and return it to the mComponent. The mComponent will then insert it into the buffer cache and returns it to the pComponent. Write and fsync requests work in a similar fashion.

3.4.6 Global Resource Management

LegoOS uses a two-level resource management mechanism. At the higher level, LegoOS uses three global resource managers for process, memory, and storage resources, *GPM, GMM*, and *GSM*. These global managers perform coarse-grained global resource allocation and load

balancing, and they can run on one normal Linux machine. Global managers only maintain approximate resource usage and load information. They update their information either when they make allocation decisions or by periodically asking monitors in the cluster. At the lower level, each monitor can employ its own policies and mechanisms to manage its local resources.

For example, process monitors allocate new threads locally and only ask GPM when they need to create a new process. GPM chooses the pComponent that has the least amount of threads based on its maintained approximate information. Memory monitors allocate virtual and physical memory space on their own. Only home mComponent asks GMM when it needs to allocate a new vRegion. GMM maintains approximate physical memory space usages and memory access load by periodically asking mComponents and chooses the memory with least load among all the ones that have at least vRegion size of free physical memory.

LegoOS decouples the allocation of different resources and can freely allocate each type of resource from a pool of components. Doing so largely improves resource packing compared to a monolithic server cluster that packs all type of resources a job requires within one physical machine. Also note that LegoOS allocates hardware resources only *on demand*, *i.e.*, when applications actually create threads or access physical memory. This on-demand allocation strategy further improves LegoOS' resource packing efficiency and allows more aggressive over-subscription in a cluster.

3.4.7 Reliability and Failure Handling

After disaggregation, pComponents, mComponents, and sComponents can all fail independently. Our goal is to build a reliable disaggregated cluster that has the same or lower application failure rate than a monolithic cluster. As a first (and important) step towards achieving this goal, we focus on providing memory reliability by handling mComponent failure in the current version of LegoOS because of three observations. First, when distributing an application's memory to multiple mComponents, the probability of memory failure increases and not handling

mComponent failure will cause applications to fail more often on a disaggregated cluster than on monolithic servers. Second, since most modern datacenter applications already provide reliability to their distributed storage data and the current version of LegoOS does not split a file across sComponent, we leave providing storage reliability to applications. Finally, since LegoOS does not split a process across pComponents, the chance of a running application process being affected by the failure of a pComponent is similar to one affected by the failure of a processor in a monolithic server. Thus, we currently do not deal with pComponent failure and leave it for future work.

A naive approach to handle memory failure is to perform a full replication of memory content over two or more mComponents. This method would require at least $2\times$ memory space, making the monetary and energy cost of providing reliability prohibitively high (the same reason why RAMCloud [236] does not replicate in memory). Instead, we propose a space- and performance-efficient approach to provide in-memory data reliability in a best-effort way. Further, since losing in-memory data will not affect user persistent data, we propose to provide memory reliability in a best-effort manner.

We use one primary mComponent, one secondary mComponent, and a backup file in sComponent for each vma. A mComponent can serve as the primary for some vma and the secondary for others. The primary stores all memory data and metadata. LegoOS maintains a small append-only log at the secondary mComponent and also replicates the vma tree there. When pComponent flushes a dirty ExCache line, LegoOS sends the data to both primary and secondary in parallel (step ⓐ and ⓑ in Figure 3.4) and waits for both to reply (ⓒ and ⓓ). In the background, the secondary mComponent flushes the backup log to a sComponent, which writes it to an append-only file.

If the flushing of a backup log to sComponent is slow and the log is full, we will skip replicating application memory. If the primary fails during this time, LegoOS simply reports an error to application. Otherwise when a primary mComponent fails, we can recover memory

content by replaying the backup logs on sComponent and in the secondary mComponent. When a secondary mComponent fails, we do not reconstruct anything and start replicating to a new backup log on another mComponent.

3.5 LegoOS Implementation

We implemented LegoOS in C on the x86-64 architecture. LegoOS can run on commodity, off-the-shelf machines and support most commonly-used Linux system call APIs. Apart from being a proof-of-concept of the splitkernel OS architecture, our current LegoOS implementation can also be used on existing datacenter servers to reduce the energy cost, with the help of techniques like Zombieland [230]. Currently, LegoOS has 206K SLOC, with 56K SLOC for drivers. LegoOS supports 113 syscalls, 15 pseudo-files, and 10 vectored syscall opcodes. Similar to the findings in [312], we found that implementing these Linux interfaces are sufficient to run many unmodified datacenter applications.

3.5.1 Hardware Emulation

Since there is no real resource disaggregation hardware, we emulate disaggregated hardware components using commodity servers by limiting their internal hardware usages. For example, to emulate controllers for mComponents and sComponents, we limit the usable cores of a server to two. To emulate pComponents, we limit the amount of usable main memory of a server and configure it as LegoOS software-managed ExCache.

3.5.2 Network Stack

We implemented three network stacks in LegoOS. The first is a customized RDMA-based RPC framework we implemented based on LITE [315] on top of the Mellanox mlx4 InfiniBand driver we ported from Linux. Our RDMA RPC implementation registers physical memory

addresses with RDMA NICs and thus eliminates the need for NICs to cache physical-to-virtual memory address mappings [315]. The resulting smaller NIC SRAM can largely reduce the monetary cost of NICs, further saving the total cost of a LegoOS cluster. All LegoOS internal communications use this RPC framework. For best latency, we use one dedicated polling thread at RPC server side to keep polling incoming requests. Other thread(s) (which we call worker threads) execute the actual RPC functions. For each pair of components, we use one physically consecutive memory region at a component to serve as the receive buffer for RPC requests. The RPC client component uses RDMA write with immediate value to directly write into the memory region and the polling thread polls for the immediate value to get the metadata information about the RPC request (*e.g.*, where the request is written to in the memory region). Immediately after getting an incoming request, the polling thread passes it along to a work queue and continues to poll for the next incoming request. Each worker thread checks if the work queue is not empty and if so, gets an RPC request to process. Once it finishes the RPC function, it sends the return value back to the RPC client with an RDMA write to a memory address at the RPC client. The RPC client allocates this memory address for the return value before sending the RPC request and piggy-backs the memory address with the RPC request.

The second network stack is our own implementation of the socket interface directly on RDMA. The final stack is a traditional socket TCP/IP stack we adapted from lwip [94] on our ported e1000 Ethernet driver. Applications can choose between these two socket implementations and use virtual IPs for their socket communication.

3.5.3 Processor Monitor

We reserve a contiguous physical memory region during kernel boot time and use fixed ranges of memory in this region as ExCache, tags and metadata for these caches, and kernel physical memory. We organize ExCache into virtually indexed sets with a configurable set associativity. Since x86 (and most other architectures) uses hardware-managed TLB and walks

page table directly after TLB misses, we have to use paging and the only chance we can trap to OS is at page fault time. We thus use paged memory to emulate ExCache, with each ExCache line being a 4 KB page. A smaller ExCache line size would improve the performance of fetching lines from mComponents but increase the size of ExCache tag array and the overhead of tag comparison.

An ExCache miss causes a page fault and traps to LegoOS. To minimize the overhead of context switches, we use the application thread that faults on a ExCache miss to perform ExCache replacement. Specifically, this thread will identify the set to insert the missing page using its virtual memory address, evict a page in this set if it is full, and if needed, flush a dirty page to mComponent (via a LegoOS RPC call to the owner mComponent of the vRegion this page is in). To minimize the network round trip needed to complete a ExCache miss, we piggy-back the request of dirty page flush and new page fetching in one RPC call when the mComponent to be flushed to and the mComponent to fetch the missing page are the same.

LegoOS maintains an approximate LRU list for each ExCache set and uses a background thread to sweep all entries in ExCache and adjust LRU lists. LegoOS supports two ExCache replacement policies: FIFO and LRU. For FIFO replacement, we simply maintain a FIFO queue for each ExCache set and insert a corresponding entry to the tail of the FIFO queue when an ExCache page is inserted into the set. Eviction victim is chosen as the head of the FIFO queue. For LRU, we use one background thread to sweep all sets of ExCache to adjust their LRU lists. For both policies, we use a per-set lock and lock the FIFO queue (or the LRU list) when making changes to them.

3.5.4 Memory Monitor

We use regular machines to emulate mComponents by limiting usable cores to a small number (2 to 5 depending on configuration). We dedicate one core to busy poll network requests and the rest for performing memory functionalities. The LegoOS memory monitor performs

all its functionalities as handlers of RPC requests from pComponents. The memory monitor handles most of these functionalities locally and sends another RPC request to a sComponent for storage-related functionalities (*e.g.*, when a buffer cache miss happens). LegoOS stores application data, application memory address mappings, vma trees, and vRegion arrays all in the main memory of the emulating machine.

The memory monitor loads an application executable from sComponents to the mComponent, handles application virtual memory address allocation requests, allocates physical memory at the mComponent, and reads/writes data to the mComponent. Our current implementation of memory monitor is purely in software, and we use hash tables to implement the virtual-to-physical address mappings. While we envision future mComponents to implement memory monitors in hardware and to have specialized hardware parts to store address mappings, our current software implementation can still be useful for users that want to build software-managed mComponents.

3.5.5 Storage Monitor

Since storage is not the focus of the current version of LegoOS, we chose a simple implementation of building storage monitor on top of the Linux *vfs* layer as a loadable Linux kernel module. LegoOS creates a normal file over vfs as the mount partition for each vNode and issues vfs file operations to perform LegoOS storage I/Os. Doing so is sufficient to evaluate LegoOS, while largely saving our implementation efforts on storage device drivers and layered storage protocols. We leave exploring other options of building LegoOS storage monitor to future work.

3.5.6 Experience and Discussion

We started our implementation of LegoOS from scratch to have a clean design and implementation that can fit the splitkernel model and to evaluate the efforts needed in building

different monitors. However, with the vast amount and the complexity of drivers, we decided to port Linux drivers instead of writing our own. We then spent our engineering efforts on an "as needed" base and took shortcuts by porting some of the Linux code. For example, we re-used common algorithms and data structures in Linux to easily port Linux drivers. We believe that being able to support largely unmodified Linux drivers will assist the adoption of LegoOS.

When we started building LegoOS, we had a clear goal of sticking to the principle of "clean separation of functionalities". However, we later found several places where performance could be improved if this principle is relaxed. For example, for the optimization in §3.4.4 to work correctly, pComponent needs to store the address range and permission for anonymous virtual memory regions — memory-related information that otherwise only mComponents need to know. Another example is the implementation of `mremap`. With LegoOS' principle of mComponents handling all memory address allocations, memory monitors will allocate new virtual memory address ranges for `mremap` requests. However, when data in the `mremap` region is in ExCache, LegoOS needs to move it to another set if the new virtual address does not fall into the current set. If mComponents are ExCache-aware, they can choose the new virtual memory address to fall into the same set as the current one. Our strategy is to relax the clean-separation principle only by giving "hints", and only for frequently-accessed, performance-critical operations.

3.6 Evaluation

This section presents the performance evaluation of LegoOS using micro- and macro-benchmarks and two unmodified real applications. We also quantitatively analyze the failure rate of LegoOS. We ran all experiments on a cluster of 10 machines, each with two Intel Xeon CPU E5-2620 2.40GHz processors, 128 GB DRAM, and one 40 Gbps Mellanox ConnectX-3 InfiniBand network adapter; a Mellanox 40 Gbps InfiniBand switch connects all of the machines. The Linux version we used for comparison is v4.9.47.

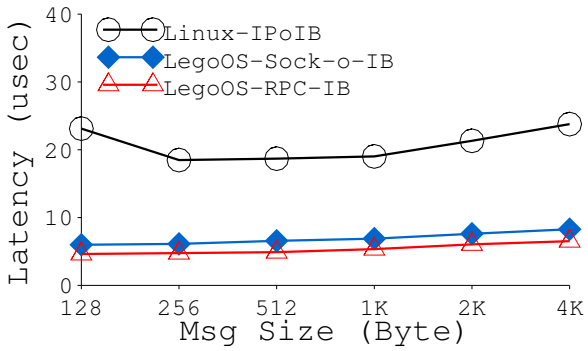


Figure 3.5: Network Latency.

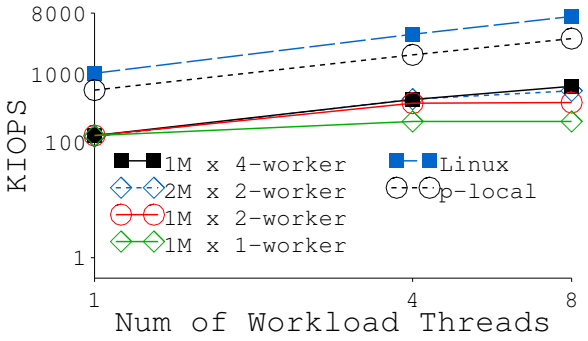


Figure 3.6: Memory Throughput.

3.6.1 Micro- and Macro-benchmark Results

Network performance. Network communication is at the core of LegoOS' performance. Thus, we evaluate LegoOS' network performance first before evaluating LegoOS as a whole. Figure 3.5 plots the average latency of sending messages with socket-over-InfiniBand (Linux-IPoIB) in Linux, LegoOS' implementation of socket on top of InfiniBand (LegoOS-Sock-o-IB), and LegoOS' implementation of RPC over InfiniBand (LegoOS-RPC-IB). LegoOS uses LegoOS-RPC-IB for all its internal network communication across components and uses LegoOS-Sock-o-IB for all application-initiated socket network requests. Both LegoOS' networking stacks significantly outperform Linux's.

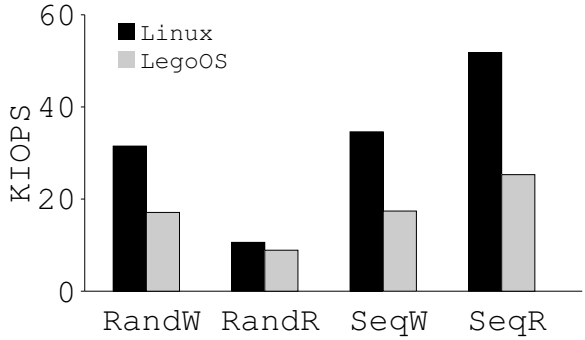


Figure 3.7: Storage Throughput.

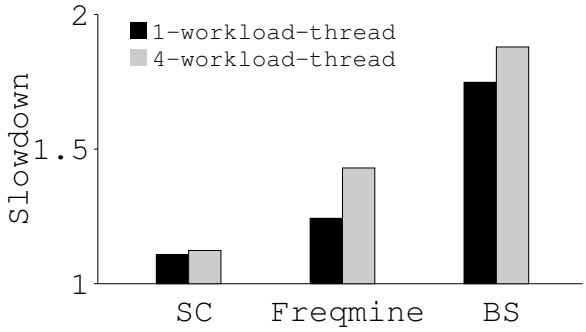


Figure 3.8: PARSEC Results. SC: StreamClsuter. BS: BlackScholes.

Memory performance. Next, we measure the performance of mComponent using a multi-threaded user-level micro-benchmark. In this micro-benchmark, each thread performs one million sequential 4 KB memory loads in a heap. We use a huge, empty ExCache (32 GB) to run this test, so that each memory access can generate an ExCache (cold) miss and go to the mComponent.

Figure 3.6 compares LegoOS' mComponent performance with Linux's single-node memory performance using this workload. We vary the number of per-mComponent worker threads from 1 to 8 with one and two mComponents (only showing representative configurations in Figure 3.6). In general, using more worker threads per mComponent and using more mComponents both improve throughput when an application has high parallelism, but the improvement

largely diminishes after the total number of worker threads reaches four. We also evaluated the optimization technique in § 3.4.4 (*p-local* in Figure 3.6). As expected, bypassing mComponent accesses with p-local lines significantly improves memory access performance. The difference between p-local and Linux demonstrates the overhead of trapping to LegoOS kernel and setting up ExCache.

Storage performance. To measure the performance of LegoOS' storage system, we ran a single-thread micro-benchmark that performs sequential and random 4 KB read/write to a 25 GB file on a Samsung PM1725s NVMe SSD (the total amount of data accessed is 1 GB). For write workloads, we issue an *fsync* after each *write* to test the performance of writing all the way to the SSD.

Figure 3.7 presents the throughput of this workload on LegoOS and on single-node Linux. For LegoOS, we use one mComponent to store the buffer cache of this file and initialize the buffer cache to empty so that file I/Os can go to the sComponent (Linux also uses an empty buffer cache). Our results show that Linux's performance is determined by the SSD's read/write bandwidth. LegoOS' random read performance is close to Linux, since network cost is relatively low compared to the SSD's random read performance. With better SSD sequential read performance, network cost has a higher impact. LegoOS' write-and-fsync performance is worse than Linux because LegoOS requires one RTT between pComponent and mComponent to perform write and two RTTs (pComponent to mComponent, mComponent to sComponent) for fsync.

PARSEC results. We evaluated LegoOS with a set of workloads from the PARSEC benchmark suite [49], including BlackScholes, Freqmine, and StreamCluster. These workloads are a good representative of compute-intensive datacenter applications, ranging from machine-learning algorithms to streaming processing ones. Figure 3.8 presents the slowdown of LegoOS over single-node Linux with enough memory for the entire application working sets. LegoOS uses one pComponent with 128 MB ExCache, one mComponent with one worker thread, and

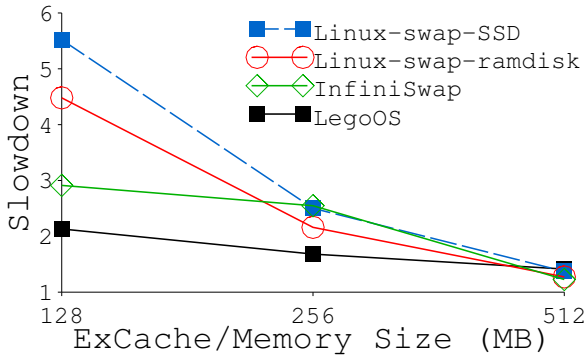
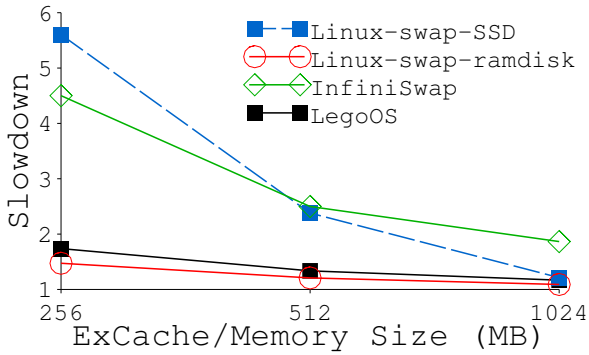


Figure 3.9: TensorFlow Performance.

one sComponent for all the PARSEC tests. For each workload, we tested one and four workload threads. StreamCluster, a streaming workload, performs the best because of its batching memory access pattern (each batch is around 110 MB). BlackScholes and Freqmine perform worse because of their larger working sets (630 MB to 785 MB). LegoOS performs worse with higher workload threads, because the single worker thread at the mComponent becomes the bottleneck to achieving higher throughput.



Phoenix Performance.

Figure 3.10: Phoenix Performance.

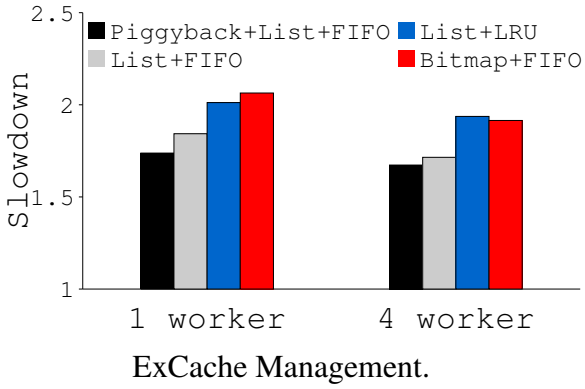


Figure 3.11: ExCache Management.

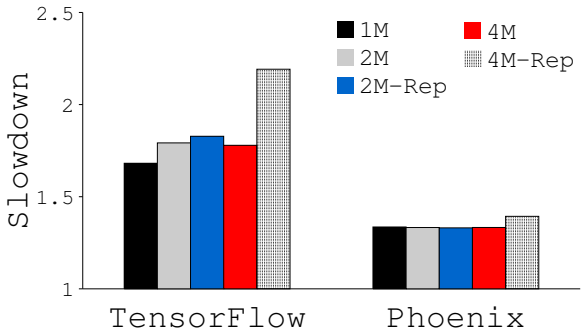


Figure 3.12: Memory Config.

3.6.2 Application Performance

We evaluated LegoOS' performance with two real, unmodified applications, TensorFlow [12] and Phoenix [264], a single-node multi-threaded implementation of MapReduce [83]. TensorFlow's experiments use the Cifar-10 dataset [7] and Phoenix's use a Wikipedia dataset [9]. Unless otherwise stated, the base configuration used for all TensorFlow experiments is 256 MB 64-way ExCache, one pComponent, one mComponent, and one sComponent. The base configuration for Phoenix is the same as TensorFlow's with the exception that the base ExCache size is 512 MB. The total amount of virtual memory addresses touched in TensorFlow is 4.4 GB (1.75 GB for Phoenix). The total working sets of the TensorFlow and Phoenix execution are

0.9 GB and 1.7 GB. Our default ExCache sizes are set as roughly 25% of total working sets. We ran both applications with four threads.

Impact of ExCache size on application performance. Figures 3.9 and 3.10 plot the TensorFlow and Phoenix run time comparison across LegoOS, a remote swapping system (InfiniSwap [126]), a Linux server with a swap file in a local high-end NVMe SSD, and a Linux server with a swap file in local ramdisk. All values are calculated as a slowdown to running the applications on a Linux server that have enough local resources (main memory, CPU cores, and SSD). For systems other than LegoOS, we change the main memory size to the same size of ExCache in LegoOS, with rest of the memory on swap file. With around 25% working set, LegoOS only has a slowdown of 1.68× and 1.34× for TensorFlow and Phoenix compared to a monolithic Linux server that can fit all working sets in its main memory.

LegoOS' performance is significantly better than swapping to SSD and to remote memory largely because of our efficiently-implemented network stack, simplified code path compared with Linux paging subsystem, and the optimization technique proposed in §3.4.4. Surprisingly, it is similar or even better than swapping to local memory, even when LegoOS' memory accesses are across network. This is mainly because ramdisk goes through buffer cache and incurs memory copies between the buffer cache and the in-memory swap file.

LegoOS' performance results are not easy to achieve and we went through rounds of design and implementation refinement. Our network stack and RPC optimizations yield a total improvement of up to 50%. For example, we made all RPC server (mComponent's) replies *unsignaled* to save mComponent' processing time and to increase its request handling throughput. Another optimization we did is to piggy-back dirty cache line flush and cache miss fill into one RPC. The optimization of the first anonymous memory access (§3.4.4) improves LegoOS' performance further by up to 5%.

ExCache management. Apart from its size, how an ExCache is managed can also largely affect application performance. We first evaluated factors that could affect ExCache hit rate and

found that higher associativity improves hit rate but the effect diminishes when going beyond 512-way. We then focused on evaluating the miss cost of ExCache, since the miss path is handled by LegoOS in our design. We compare the two eviction policies LegoOS supports (FIFO and LRU), two implementations of finding an empty line in an ExCache set (linearly scan a free bitmap and fetching the head of a free list), and one network optimization (piggyback flushing a dirty line with fetching the missing line).

Figure 3.11 presents these comparisons with one and four mComponent worker threads. All tests run the Cifar-10 workload on TensorFlow with 256 MB 64-way ExCache, one mComponent, and one sComponent. Using bitmaps for this ExCache configuration is always worse than using free lists because of the cost to linearly scan a whole bitmap, and bitmaps perform even worse with higher associativity. Surprisingly, FIFO performs better than LRU in our tests, even when LRU utilizes access locality pattern. We attributed LRU's worse performance to the lock contention it incurs; the kernel background thread sweeping the ExCache locks an LRU list when adjusting the position of an entry in it, while ExCache miss handler thread also needs to lock the LRU list to grab its head. Finally, the piggyback optimization works well and the combination of FIFO, free list, and piggyback yields the best performance.

Number of mComponents and replication. Finally, we study the effect of the number of mComponents and memory replication. Figure 3.12 plots the performance slowdown as the number of mComponents increases from one to four. Surprisingly, using more mComponents lowers application performance by up to 6%. This performance drop is due to the effect of ExCache piggyback optimization. When there is only one mComponent, flushes and misses are all between the pComponent and this mComponent, thus enabling piggyback on every flush. However, when there are multiple mComponents, LegoOS can only perform piggyback when flushes and misses are to the same mComponent.

We also evaluated LegoOS' memory replication performance in Figure 3.12. Replication has a performance overhead of 2% to 23% (there is a constant 1 MB space overhead to store the

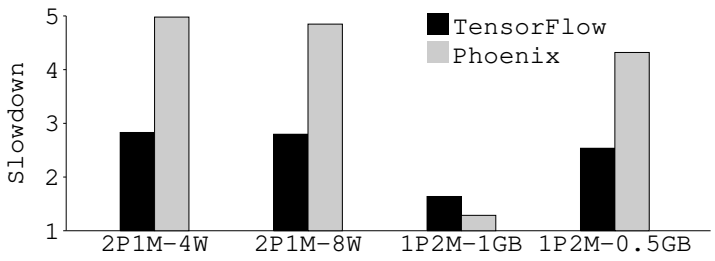


Figure 3.13: Multiple Applications.

backup log). LegoOS uses the same application thread to send the replica data to the backup mComponent and then to the primary mComponent, resulting in the performance lost.

Running multiple applications together. All our experiments so far run only one application at a time. Now we evaluate how multiple applications perform when running them together on a LegoOS cluster. We use a simple scenario of running one TensorFlow instance and one Phoenix instance together in two settings: 1) two pComponents each running one instance, both accessing one mComponent(2P1M), and 2) one pComponent running two instances and accessing two mComponents (1P2M). Both settings use one sComponent. Figure 3.13 presents the runtime slowdown results. We also vary the number of mComponent worker threads for the 2P1M setting (4 and 8 workers) and the amount of ExCache for the 1P2M setting (1 GB and 0.5 GB). With 2P1M, both applications suffer from a performance drop because their memory access requests saturate the single mComponent. Using more worker threads at the mComponent improves the performance slightly. For 1P2M, application performance largely depends on ExCache size, similar to our findings with single-application experiments.

Table 3.1: Mean Time To Failure Analysis. MTTF numbers of devices (columns 2 to 7) are obtained from [278] and MTTF values of monolithic server and LegoOS are calculated using the per-device MTTF numbers.

	Processor	Disk	Memory	NIC	Power	Other	Monolithic	LegoOS
MTTF (year)	204.3	33.1	289.9	538.8	100.5	27.4	5.8	6.8 - 8.7

3.6.3 Failure Analysis

Finally, we provide a qualitative analysis on the failure rate of a LegoOS cluster compared to a monolithic server cluster. Table 3.1 summarizes our analysis. To measure the failure rate of a cluster, we use the metric Mean Time To (hardware) Failure (MTTF), the mean time to the failure of a server in a monolithic cluster or a component in a LegoOS cluster. Since the only real per-device failure statistics we can find are the mean time to hardware replacement in a cluster [278], the MTTF we refer to in this study indicates the mean time to the type of hardware failures that require replacement. Unlike traditional MTTF analysis, we are not able to include transient failures.

To calculate MTTF of a monolithic server, we first obtain the replacement frequency of different hardware devices in a server (CPU, memory, disk, NIC, motherboard, case, power supply, fan, CPU heat sink, and other cables and connections) from the real world (the COM1 and COM2 clusters in [278]). For LegoOS, we envision every component to have a NIC and a power supply, and in addition, a pComponent to have CPU, fan, and heat sink, an mComponent to have memory, and an sComponent to have a disk. We further assume both a monolithic server and a LegoOS component to fail when any hardware devices in them fails and the devices in them fail independently. Thus, the MTTF can be calculated using the harmonic mean (*HM*) of the MTTF of each device.

$$MTTF = \frac{HM_{i=0}^{n}(MTTF_i)}{n} \tag{3.1}$$

where n includes all devices in a machine/component.

Further, when calculating MTTF of LegoOS, we estimate the amount of components needed in LegoOS to run the same applications as a monolithic cluster. Our estimated worst case for LegoOS is to use the same amount of hardware devices (*i.e.*, assuming same resource utilization as monolithic cluster). LegoOS' best case is to achieve full resource utilization and thus requiring only about half of CPU and memory resources (since average CPU and memory

resource utilization in monolithic server clusters is around 50% [122, 21]).

With better resource utilization and simplified hardware components (*e.g.*, no motherboard), LegoOS improves MTTF by 17% to 49% compared to an equivalent monolithic server cluster.

3.7 Related Work

Hardware Resource Disaggregation. There have been a few hardware disaggregation proposals from academia and industry, including Firebox [36], HP "The Machine" [133, 100], dRedBox [156], and IBM Composable System [68]. Among them, dRedBox and IBM Composable System package hardware resources in one big case and connect them with buses like PCIe. The Machine's scale is a rack and it connects SoCs with NVMs with a specialized coherent network. FireBox is an early-stage project and is likely to use high-radix switches to connect customized devices. The disaggregated cluster we envision to run LegoOS on is one that hosts hundreds to thousands of non-coherent, heterogeneous hardware devices, connected with a commodity network.

Memory Disaggregation and Remote memory. Lim *et al.* first proposed the concept of hardware disaggregated memory with two models of disaggregated memory: using it as network swap device and transparently accessing it through memory instructions [191, 192]. Their hardware models still use a monolithic server at the local side. LegoOS' hardware model separates processor and memory completely. Another set of recent projects utilize remote memory without changing monolithic servers [90, 224, 16, 126, 233, 285]. For example, InfiniSwap [126] transparently swaps local memory to remote memory via RDMA. These remote memory systems help improve the memory resource packing in a cluster. However, as discussed in §3.2, unlike LegoOS, these solutions cannot solve other limitations of monolithic servers like the lack of hardware heterogeneity and elasticity.

Storage Disaggregation. Cloud vendors usually provision storage at different physical machines [344, 26, 318]. Remote access to hard disks is a common practice, because their high latency and low throughput can easily hide network overhead [181, 212, 328, 185]. While disaggregating high-performance flash is a more challenging task [164, 97]. Systems such as ReFlex [166], Decibel [222], and PolarFS [59], tightly integrate network and storage layers to minimize software overhead in the face of fast hardware. Although storage disaggregation is not our main focus now, we believe those techniques can be realized in future LegoOS easily.

Multi-Kernel and Multi-Instance OSes. Multi-kernel OSes like Barrelfish [47, 339], Helios [229], Hive [62], and fos [329] run a small kernel on each core or programmable device in a monolithic server, and they use message passing to communicate across their internal kernels. Similarly, multi-instance OSes like Popcorn [42] and Pisces [244] run multiple Linux kernel instances on different cores in a machine. Different from these OSes, LegoOS runs on and manages a distributed set of hardware devices; it manages distributed hardware resources using a two-level approach and handles device failures (currently only mComponent). In addition, LegoOS differs from these OSes in how it splits OS functionalities, where it executes the split kernels, and how it performs message passing across components. Different from multi-kernels' message passing mechanisms which are performed over buses or using shared memory in a server, LegoOS' message passing is performed using a customized RDMA-based RPC stack over InfiniBand or RoCE network. Like LegoOS, fos [329] separates OS functionalities and run them on different processor cores that share main memory. Helios [229] runs *satellite kernels* on heterogeneous cores and programmable NICs that are not cache-coherent. We took a step further by disseminating OS functionalities to run on individual, network-attached hardware devices. Moreover, LegoOS is the first OS that separates memory and process management and runs virtual memory system completely at network-attached memory devices.

Distributed OSes. There have been several distributed OSes built in late 80s and early 90s [303, 304, 243, 41, 65, 265, 45, 35]. Many of them aim to appear as a single machine to

users and focus on improving inter-node IPCs. Among them, the most closely related one is Amoeba [303, 304]. It organizes a cluster into a shared process pool and disaggregated specialized servers. Unlike Amoeba, LegoOS further separates memory from processors and different hardware components are loosely coupled and can be heterogeneous instead of as a homogeneous pool. There are also few emerging proposals to build distributed OSes in datacenters [148, 279], *e.g.*, to reduce the performance overhead of middleware. LegoOS achieves the same benefits of minimal middleware layers by only having LegoOS as the system management software for a disaggregated cluster and using the lightweight vNode mechanism.

3.8 Discussion and Conclusion

We presented LegoOS, the first OS designed for hardware resource disaggregation. LegoOS demonstrated the feasibility of resource disaggregation and its advantages in better resource packing, failure isolation, and elasticity, all without changing Linux ABIs. LegoOS and resource disaggregation in general can help the adoption of new hardware and thus encourage more hardware and system software innovations.

LegoOS is a research prototype and has a lot of room for improvement. For example, we found that the amount of parallel threads an mComponent can use to process memory requests largely affect application throughput. Thus, future developers of real mComponents can consider use large amount of cheap cores or FPGA to implement memory monitors in hardware.

We also performed an initial investigation in load balancing and found that memory allocation policies across mComponents can largely affect application performance. However, since we do not support memory data migration yet, the benefit of our load-balancing mechanism is small. We leave memory migration for future work. In general, large-scale resource management of a disaggregated cluster is an interesting and important topic, but is outside of the scope of this paper.

3.9 Acknowledgments

Chapter 3, in full, is a reprint of Yizhou Shan, Yutong Huang, Yilun Chen, Yiying Zhang, "LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation", *OSDI, 2018*. The dissertation author was the primary investigator and author of this paper.

Chapter 4

Clio: A Hardware-Software Co-Designed Disaggregated Memory

4.1 Introduction

Modern datacenter applications like graph computing, data analytics, and deep learning have an increasing demand for access to large amounts of memory [23]. Unfortunately, servers are facing *memory capacity walls* because of pin, space, and power limitations [134, 146, 332]. Going forward, it is imperative for datacenters to seek solutions that can go beyond what a (local) machine can offer, *i.e.*, using remote memory. At the same time, datacenters are seeing the needs from management and resource utilization perspectives to *disaggregate* resources [308, 322, 66]—separating hardware resources into different network-attached pools that can be scaled and managed independently. These real needs have pushed the idea of memory disaggregation (*MemDisagg* for short): organizing computation and memory resources as two separate network-attached pools, one with compute nodes (*CN*s) and one with memory nodes (*MN*s).

So far, MemDisagg researches have all taken one of two approaches: building/emulating MNs using regular servers [273, 127, 23, 283, 231] or using raw memory devices with no

processing power [314, 191, 192, 133]. The fundamental issues of server-based approaches such as RDMA-based systems are the monetary and energy cost of a host server and the inherent performance and scalability limitations caused by the way NICs interact with the host server's virtual memory system. Raw-device-based solutions have low costs. However, they introduce performance, security, and management problems because when MNs have no processing power, all the data and control planes have to be handled at CNs [314].

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

We built *Clio*[1], a hardware-based disaggregated memory system. Clio includes a CN-side user-space library called *CLib* and a new hardware-based MN device called *CBoard*. Multiple application processes running on different CNs can allocate memory from the same CBoard, with each process having its own *remote virtual memory address space*. Furthermore, one remote virtual memory address space can span multiple CBoards. Applications can perform byte-granularity remote memory read/write and use Clio's synchronization primitives for synchronizing concurrent accesses to shared remote memory .

A key research question in designing Clio is ***how to use limited hardware resources to achieve 100 Gbps, microsecond-level average and tail latency for TBs of memory and thousands of concurrent clients?*** These goals are important and unique for MemDisagg. A good MemDisagg solution should reduce the total CapEx and OpEx costs compared to traditional non-disaggregated systems and thus cannot afford to use large amounts of hardware resources at MNs. Meanwhile, remote memory accesses should have high throughput and low average and tail latency, because even after caching data at CN-local memory, there can still be fairly frequent

[1]Clio is the daughter of Mnemosyne, the Greek goddess of memory.

accesses to MNs and the overall application performance can be impacted if they are slow [112]. Finally, unlike traditional single-server memory, a disaggregated MN should allow many CNs to store large amounts of data so that we only need a few of them to reduce costs and connection points in a cluster. How to achieve each of the above cost, performance, and scalability goals *individually* is relatively well understood. However, achieving all these seemingly conflicting goals *simultaneously* is hard and previously unexplored.

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

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

a client process and involve per-process/client metadata (*e.g.*, one page table per process, one connection per client, etc.). Overcoming these challenges require the re-design of traditional memory and network systems.

Our first approach is to separate the metadata/control plane and the data plane, with the former running as software on a low-power ARM-based SoC at MN and the latter in hardware at MN. Metadata operations like memory allocation usually need more memory but are rarer (thus not as performance critical) compared to data operations. A low-power SoC's computation speed and its local DRAM are sufficient for metadata operations. On the other hand, data operations (*i.e.*, all memory accesses) should be fast and are best handled purely in hardware. Even though the separation of data and control plane is a common technique that has been applied in many areas [125, 170, 252], a separation of memory system control and data planes has not been explored before and is not easy, as we will show in this paper.

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

With the above two approaches, the hardware can be largely simplified and thus cheaper, faster, and more scalable. However, we found that ***complete state elimination at MNs is neither feasible nor ideal***. To ensure correctness, the MN has to maintain some state (*e.g.*, to deal with non-idempotent operations). To ensure good data-plane performance, not every operation that

involves state should be moved to the low-power SoC or to CNs. Thus, our approach is to eliminate as much state as we can without affecting performance or correctness and to carefully design the remaining state so that it causes small and bounded space and performance overhead.

For example, we perform paging-based virtual-to-physical memory address mapping and access permission checking at the MN hardware pipeline, as these operations are needed for every data access. Page table is a kind of state that could potentially cause performance and scalability issues but has to be accessed in the data path. We propose a new overflow-free, hash-based page table design where 1) all page table lookups have bounded and low latency (at most one DRAM access time in our implementation), and 2) the total size of all page table entries does not grow with the number of client processes. As a result, even though we cannot eliminate page table from the MN hardware, we can still meet our cost, performance, or scalability requirements.

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

We prototyped CBoard with a small set of Xilinx ZCU106 MPSoC FPGA boards [333] and built three applications using Clio: a FaaS-style image compression utility, a radix-tree index, and a key-value store. We compared Clio with native RDMA, two RDMA-based disaggregated/remote memory systems [314, 154], a software emulation of hardware-based disaggregated memory [283], and a software-based SmartNIC [210]. Clio scales much better and has orders of magnitude lower tail latency than RDMA, while achieving similar throughput and median latency as RDMA (even with the slower FPGA frequency in our prototype). Clio has 1.1× to 3.4× energy

saving compared to CPU-based and SmartNIC-based disaggregated memory systems and is 2.7× faster than SmartNIC solutions.

Clio is publicly available at `https://github.com/WukLab/Clio`.

4.2 Goals and Related Works

Resource disaggregation separates different types of resources into different pools, each of which can be independently managed and scaled. Applications can allocate resources from any node in a resource pool, resulting in tight resource packing. Because of these benefits, many datacenters have adopted the idea of disaggregation, often at the storage layer [99, 66, 322, 29, 28, 22, 306]. With the success of disaggregated storage, researchers in academia and industry have also sought ways to disaggregate memory (and persistent memory) [191, 36, 143, 192, 283, 285, 242, 314, 273, 23, 127, 324, 230]. Different from storage disaggregation, MemDisagg needs to achieve at least an order of magnitude higher performance and it should offer a byte-addressable interface. Thus, MemDisagg poses new challenges and requires new designs. This section discusses the requirements of MemDisagg and why existing solutions cannot fully meet them.

4.2.1 MemDisagg Design Goals

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

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

R2: Supporting a huge number of concurrent clients. To ensure tight and efficient

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

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

R4: Low tail latency. Maintaining a low tail latency is important in meeting service-level objectives (SLOs) in data centers. Long tails like RDMA's 16.8 *ms* remote memory access can be detrimental to applications that are short running (*e.g.*, serverless computing workloads) or have large fan-outs or big DAGs (because they need to wait for the slowest step to finish) [82].

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

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

R7: Flexible. With the fast development of datacenter applications, hardware, and network, a sustainable MemDisagg solution should be flexible and extendable, for example, to support high-level APIs like pointer chasing [273, 18], to offload some application logic to memory devices [273, 290], or to incorporate different network transports [219, 132, 33] and

congestion control algorithms [174, 294, 190].

4.2.2 Server-Based Disaggregated Memory

MemDisagg research so far has mainly taken a server-based approach by using regular servers as MNs [127, 23, 324, 283, 273, 231, 89], usually on top of RDMA. The common limitation of these systems is their reliance on a host server and the resulting CPU energy costs, both of which violate **R6**.

RDMA is what most server-based MemDisagg solutions are based on, with some using RDMA for swapping memory between CNs and MNs [127, 23, 324] and some using RDMA for explicitly accessing MNs [273, 231, 89]. Although RDMA has low average latency and high throughput, it has a set of scalability and tail-latency problems.

A process (P_M) running at an MN needs to allocate memory in its virtual memory address space and *register* the allocated memory (called a memory region, or MR) with the RDMA NIC (RNIC). The host OS and MMU set up and manage the page table that maps P_M's virtual addresses (*VA*s) to physical memory addresses (*PA*s). To avoid always accessing host memory for address mapping, RNICs cache page table entries (PTEs), but when more PTEs are accessed than what this cache can hold, RDMA performance degrades significantly (Figure 4.5 and [89, 315]). Similarly, RNICs cache MR metadata and incur degraded performance when the cache is full. Thus, RDMA has serious performance issues with either large memory (PTEs) or many disjoint memory regions (MRs), violating **R1**. Moreover, RDMA uses a slow way to support on-demand allocation: the RNIC interrupts the host OS for handling page faults. From our experiments, a faulting RDMA access is 14100× slower than a no-fault access (violating **R4**).

To mitigate the above performance and scalability issues, most RDMA-based systems today [89, 315] preallocate a big MR with huge pages and pin it in physical memory. This results in inefficient memory space utilization and violates **R1**. Even with this approach, there can still be a scalability issue (**R2**), as RDMA needs to create at least one MR for each protection domain

(*i.e.*, each client).

In addition to problems caused by RDMA's memory system design, reliable RDMA, the mode used by most MemDisagg solutions, suffers from a connection queue pair (QP) scalability issue, also violating **R2**. Finally, today's RNICs violate **R7** because of their rigid one-sided RDMA interface and the close-sourced, hardware-based transport implementation. Solutions like 1RMA [294] and IRN [216] mitigate the above issues by either onloading part of the transport back to software or proposing a new hardware design.

LegoOS [283], our own previous work, is a distributed operating system designed for resource disaggregation. Its MN includes a virtual memory system that maps VAs of application processes running at CNs to MN PAs. Clio's MN performs the same type of address translation. However, LegoOS emulates MN devices using regular servers and we built its virtual memory system in software, which has a stark difference from a hardware-based virtual memory system. For example, LegoOS uses a thread pool that handles incoming memory requests by looking up a hash table for address translation and permission checking. This software approach is the major performance bottleneck in LegoOS (§4.7), violating **R3**. Moreover, LegoOS uses RDMA for its network communication hence inheriting its limitations.

4.2.3 Physical Disaggregated Memory

One way to build MemDisagg without a host server is to treat it as raw, physical memory, a model we call *PDM*. The PDM model has been adopted by a set of coherent interconnect proposals [114, 78], HPE's Memory-Driven Computing project [133, 101, 319, 135]. A recent disaggregated hashing system [352] and our own recent work on disaggregated key-value systems [314] also adopt the PDM model and emulate remote memory with regular servers. To prevent applications from accessing raw physical memory, these solutions add an indirection layer at CNs in hardware [114, 78] or software [314, 352] to map client process VAs or keys to MN PAs.

There are several common problems with all the PDM solutions. First, because MNs in PDM are raw memory, CNs need multiple network round trips to access an MN for complex operations like pointer chasing and concurrent operations that need synchronization [314], violating **R3** and **R7**. Second, PDM requires the client side to manage disaggregated memory. For example, CNs need to coordinate with each other or use a global server [314] to perform tasks like memory allocation. Non-MN-side processing is much harder, performs worse compared to memory-side management (violating **R3**), and could even result in higher overall costs because of the high computation added at CNs (violating **R6**). Third, exposing physical memory makes it hard to provide security guarantees (**R5**), as MNs have to authenticate that every access is to a legit physical memory address belonging to the application. Finally, all existing PDM solutions require physical memory pinning at MNs, causing memory wastes and violating **R1**.

In addition to the above problems, none of the coherent interconnects or HPE's Memory-Driven Computing have been fully built. When they do, they will require new hardware at all endpoints and new switches. Moreover, the interconnects automatically make caches at different endpoints coherent, which could cause performance overhead that is not always necessary (violating **R3**).

Besides the above PDM works, there are also proposals to include some processing power in between the disaggregated memory layer and the computation layer. soNUMA [234] is a hardware-based solution that scales out NUMA nodes by extending each NUMA node with a hardware unit that services remote memory accesses. Unlike Clio which physically separates MNs from CNs across generic data-center networks, soNUMA still bundles memory and CPU cores, and it is a single-server solution. Thus, soNUMA works only on a limited scale (violating **R2**) and is not flexible (violating **R7**). MIND [184], a concurrent work with Clio, proposes to use a programmable switch for managing coherence directories and memory address mappings between compute nodes and memory nodes. Unlike Clio which adds processing power to every MN, MIND's single programmable switch has limited hardware resources and could be the

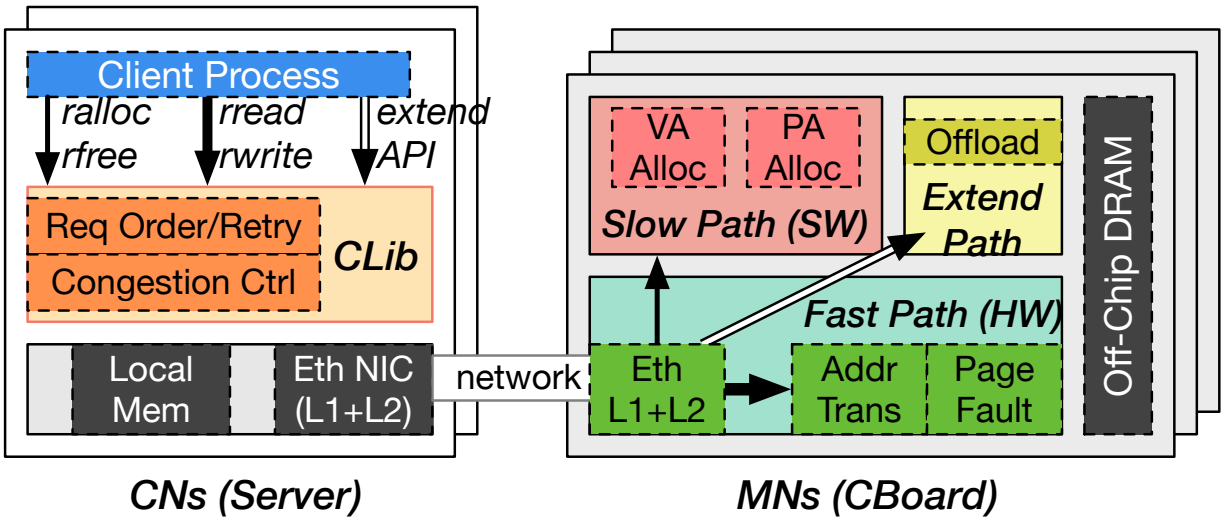


Figure 4.1: Clio Architecture.

bottleneck for both performance and scalability.

4.3 Clio Overview

Clio co-designs software with hardware, CNs with MNs, and network stack with virtual memory system, so that at the MN, the entire data path is handled in hardware with high throughput, low (tail) latency, and minimal hardware resources. This section gives an overview of Clio's interface and architecture (Figure 4.1).

4.3.1 Clio Interface

Similar to recent MemDisagg proposals [273, 30], our current implementation adopts a non-transparent interface where applications (running at CNs) allocate and access disaggregated memory via explicit API calls. Doing so gives users opportunities to perform application-specific performance optimizations. By design, Clio's APIs can also be called by a runtime like the AIFM runtime [273] or by the kernel/hardware at CN like LegoOS' pComponent [283] to support a

transparent interface and allow the use of unmodified user applications. We leave such extension to future work.

Apart from the regular (local) virtual memory address space, each process has a separate ***R**emote virtual memory **A**ddress **S**pace* (*RAS* for short). Each application process has a unique global *PID* across all CNs which is assigned by Clio when the application starts. Overall, programming in RAS is similar to traditional multi-threaded programming except that memory read and write are explicit and that processes running on different CNs can share memory in the same RAS. Figure 4.2 illustrates the usage of Clio with a simple example.

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

Intra-thread request ordering. Within a thread, synchronous APIs follow strict ordering. An application thread that calls a synchronous API blocks until it gets the result. Asynchronous APIs are non-blocking. A calling thread proceeds after calling an asynchronous API and later calls `rpoll` to get the result. Asynchronous APIs follow a release order. Specifically, asynchronous APIs may be executed out of order as long as 1) all asynchronous operations before a `rrelease` complete before the `rrelease` returns, and 2) `rrelease` operations are strictly ordered. On top of this release order, we guarantee that there is no concurrent asynchronous operations with dependencies (Write-After-Read, Read-After-Write, Write-After-Write) and target the same page. The resulting memory consistency level is the same as architecture like ARMv8 [34]. In addition, we also ensure consistency between metadata and data operations, by ensuring that potentially conflicting operations execute synchronously in the program order. For example, if there is an ongoing `rfree` request to a VA, no read or write to it can start until the `rfree` finishes. Finally,

```
/* Alloc one remote page. Define a remote lock */
#define PAGE_SIZE (1<<22)
void *remote_addr = ralloc(PAGE_SIZE);
ras_lock lock;

/* Acquire lock to enter critical section.
   Do two AYSNC writes then poll completion. */
void thread1(void *) {
 rlock(lock);
 e[0]=rwrite(remote_addr, local_wbuf1,len, ASYNC);
 e[1]=rwrite(remote_addr+len, local_wbuf2,len, ASYNC);
 runlock(lock);
 rpoll(e, 2);
}

/* Synchronously read from remote */
void thread2(void *) {
 rlock(lock);
 rread(remote_addr, local_rbuf, len, SYNC);
 runlock(lock);
}
```

Figure 4.2: Sample code using Clio.

failed or unresponsive requests are transparently retried, and they follow the same ordering guarantees.

Thread synchronization and data coherence. Threads and processes can share data even when they are not on the same CN. Similar to traditional concurrent programming, Clio threads can use synchronization primitives to build critical sections (*e.g.*, with `rlock`) and other semantics (*e.g.*, flushing all requests with `rfence`).

An application can choose to cache data read from `rread` at the CN (*e.g.*, by maintaining `local_rbuf` in the code example). Different processes sharing data in a RAS can have their own cached copies at different CNs. Similar to [283], Clio does not make these cached copies coherent automatically and lets applications choose their own coherence protocols. We made this deliberate decision because automatic cache coherence on every read/write would incur high

performance overhead with commodity Ethernet infrastructure and application semantics could reduce this overhead.

4.3.2 Clio Architecture

In Clio (Figure 4.1), CNs are regular servers each equipped with a regular Ethernet NIC and connected to a top-of-rack (ToR) switch. MNs are our customized devices directly connected to a ToR switch. Applications run at CNs on top of our user-space library called *CLib*. It is in charge of request ordering, request retry, congestion, and incast control.

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

4.4 Clio Design

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

4.4.1 Design Challenges and Principles

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

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

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

Challenge 3: The hardware pipeline should be deterministic and smooth, *i.e.*, it uses a bounded, known number of cycles to process a data unit, and for each cycle, the pipeline can take in one new data unit (from the network). The former would ensure low tail latency, while the latter would guarantee a throughput that could match network line rate. Another subtle benefit of a deterministic pipeline is that we can know the maximum time a data unit stays at MN, which could help bound the size of certain buffers (*e.g.*, §4.4.5). However, many traditional hardware solutions are not designed to be deterministic or smooth, and we cannot directly adapt their approaches. For example, traditional CPU pipelines could have stalls because of data hazards and

have non-deterministic latency to handle memory instructions.

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

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

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

Principle 3: Shifting functionalities and state to CNs. While hardware resources are scarce at MNs, CNs have sufficient memory and processing power, and it is faster to develop functionalities in CN software. A viable solution is to shift state and functionalities from MNs to CNs. The key question here is how much and what to shift. Our strategy is to shift functionalities to CNs only if doing so 1) could largely reduce hardware resource consumption at MNs, 2) does not slow down common-case foreground data operations, 3) does not sacrifice security guarantees, and 4) adds bounded memory space and CPU cycle overheads to CNs. As a tradeoff, the shift may result in certain uncommon operations (*e.g.*, handling a failed request) being slower.

Principle 4: Making off-chip data structures efficient and scalable. Principles 1 to 3

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

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

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

4.4.2 Scalable, Fast Address Translation

Similar to traditional virtual memory systems, we use fix-size pages as address allocation and translation unit, while data accesses are in the granularity of byte. Despite the similarity in the goal of address translation, the radix-tree-style, per-address space page table design used by all current architectures [295] does not fit MemDisagg for two reasons. First, each request from the network could be from a different client process. If each process has its own page table, MN would need to cache and look up many page table roots, causing additional overhead. Second, a multi-level page table design requires multiple DRAM accesses when there is a translation lookaside buffer (TLB) miss [335]. TLB misses will be much more common in a MemDisagg environment, since with more applications sharing an MN, the total working set size is much

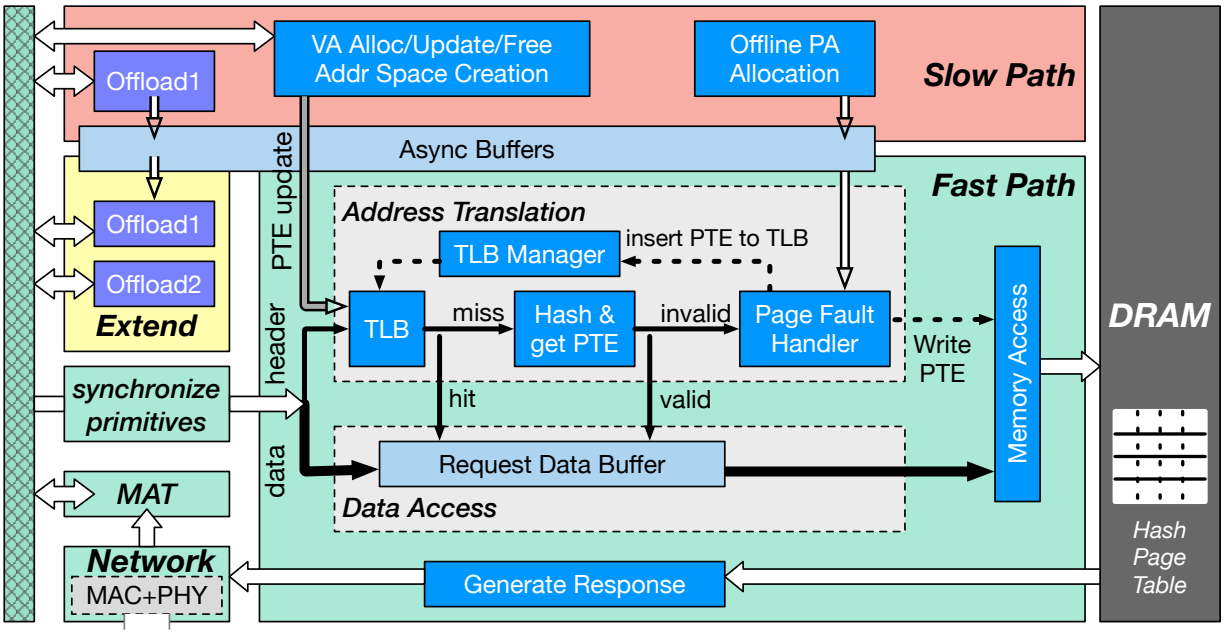


Figure 4.3: CBoard Design. Green, yellow, and red areas are anticipated to be built with ASIC, FPGA, and low-power cores.

bigger than that in a single-server setting, while the TLB size in an MN will be similar or even smaller than a single server's TLB (for cost concerns). To make matters worse, each DRAM access is more costly for systems like RDMA NIC which has to cross the PCIe bus to access the page table in main memory [313, 226].

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

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

A well-known problem with hash-based page table design is hash collisions that could overflow a bucket. Existing hash-based page table designs rely on collision chaining [43] or open addressing [335] to handle overflows, both require multiple DRAM accesses or even costly software intervention. In order to bound address translation to at most one DRAM access, we use a novel technique to avoid hash overflows at *VA allocation time*.

VA allocation (Principle 2). The slow path software handles `ralloc` requests and allocates VA. The software allocator maintains a per-process VA allocation tree that records allocated VA ranges and permissions, similar to the Linux vma tree [161]. To allocate size k of VAs, it first finds an available address range of size k in the tree. It then calculates the hash values of the virtual pages in this address range and checks if inserting them to the page table would cause any hash overflow. If so, it does another search for available VAs. These steps repeat until it finds a valid VA range that does not cause hash overflow.

Our design trades potential retry overhead at allocation time (at the slow path) for better run-time performance and simpler hardware design (at the fast path). This overhead is manageable because 1) each retry takes only a few microseconds with our implementation (§4.5), 2) we employ huge pages, which means fewer pages need to be allocated, 3) we choose a hash function that has very low collision rate [330], and 4) we set the page table to have extra slots (2× by default) which absorbs most overflows. We find no conflicts when memory is below half utilized and has only up to 60 retries when memory is close to full (Figure 4.13).

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

consistency of inflight operations.

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

4.4.3 Low-Tail-Latency Page Fault Handling

A key reason to disaggregate memory is to consolidate memory usages on less DRAM so that memory utilization is higher and the total monetary cost is lower (**R1**). Thus, remote memory space is desired to run close to full capacity, and we allow memory over-commitment at an MN, necessitating page fault handling. Meanwhile, applications like JVM-based ones allocate a large heap memory space at the startup time and then slowly use it to allocate smaller objects. Similarly, many existing far-memory systems [314, 273, 89] allocate a big chunk of remote memory and then use different parts of it for smaller objects to avoid frequently triggering the slow remote allocation operation. In these cases, it is desirable for a MemDisagg system to delay the allocation of physical memory to when the memory is actually used (*i.e.*, *on-demand* allocation) or to "reshape" memory [293] during runtime, necessitating page fault handling.

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

We propose to *handle page faults in hardware and with bounded latency*—a *constant*

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

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

A recent work [182] also handles page faults in hardware. Its focus is on the complex interaction with kernel and storage devices, and it is a simulation-only work. Clio uses a different design for handling page faults in hardware with the goal of low tail latency, and we built it in FPGA.

Putting the virtual memory system together. We illustrate how CBoard's virtual memory system works using a simple example of allocating some memory and writing to it. The first step (`ralloc`) is handled by the slow path, which allocates a VA range by finding an available set of slots in the hash page table. The slow path forwards the new PTEs to the fast path, which inserts them to the page table. At this point, the PTEs are invalid. This VA range is returned to

the client. When the client performs the first write, the request goes to the fast path. There will be a TLB miss, followed by a fetch of the PTE. Since the PTE is invalid, the page fault handler will be triggered, which fetches a free PA from the async buffer and establishes the valid PTE. It will then execute the write, update the page table, and insert the PTE to TLB.

4.4.4 Asymmetric Network Tailored for MemDisagg

With large amounts of research and development efforts, today's data-center network systems are highly optimized in their performance. Our goal of Clio's network system is unique and fits MemDisagg's requirements—minimizing the network stack's hardware resource consumption at MNs and achieving great scalability while maintaining similar performance as today's fast network. Traditional software-based reliable transports like Linux TCP incurs high performance overhead. Today's hardware-based reliable transports like RDMA are fast, but they require a fair amount of on-chip memory to maintain state, *e.g.*, per-connection sequence numbers, congestion state [33], and bitmaps [216, 201], not meeting our low-cost goal.

Our insight is that different from general-purpose network communication where each endpoint can be both the sender (requester) and the receiver (responder) that exchange general-purpose messages, MNs only respond to requests sent by CNs (except for memory migration from one MN to another MN (§4.4.7), in which case we use another simple protocol to achieve the similar goal). Moreover, these requests are all memory-related operations that have their specific properties. With these insights, we design a new network system with two main ideas. Our first idea is to maintain transport logic, state, and data buffers only at CNs, essentially making MNs "transportless" (**Principle 3**). Our second idea is to relax the reliability of the transport and instead enforce ordering and loss recovery at the memory request level, so that MNs' hardware pipeline can process data units as soon as they arrive (**Principle 5**).

With these ideas, we implemented a transport in CLib at CNs. CLib bypasses the kernel to directly issue raw Ethernet requests to an Ethernet NIC. CNs use regular, commodity Ethernet

NICs and regular Ethernet switches to connect to MNs. MNs include only standard Ethernet physical, link, and network layers and a slim layer for handling corner-case requests (§4.4.5). We now describe our detailed design.

Removing connections with request-response semantics. Connections (*i.e.*, QPs) are a major scalability issue with RDMA. Similar to recent works [219, 294], we make our network system connection-less using request-response pairs. Applications running at CNs directly initiate Clio APIs to an MN without any connections. CLib assigns a unique request ID to each request. The MN attaches the same request ID when sending the response back. CLib uses responses as ACKs and matches a response with an outstanding request using the request ID. Neither CNs nor MNs send ACKs.

Lifting reliability to the memory request level. Instead of triggering a retransmission protocol for every lost/corrupted packet at the transport layer, CLib retries the entire memory request if any packet is lost or corrupted in the sending or the receiving direction. On the receiving path, MN's network stack only checks a packet's integrity at the link layer. If a packet is corrupted, the MN immediately sends a NACK to the sender CN. CLib retries a memory request if one of three situations happens: a NACK is received, the response from MN is corrupted, or no response is received within a `TIMEOUT` period. In addition to lifting retransmission from transport to the request level, we also lift ordering to the memory request level and allow out-of-order packet delivery (see details in §4.4.5).

CN-managed congestion and incast control. Our goal of controlling congestion in the network and handling incast that can happen both at a CN and an MN is to minimize state at MN. To this end, we build the entire congestion and incast control at the CN in the CLib. To control congestion, CLib adopts a simple delay-based, reactive policy that uses end-to-end RTT delay as the congestion signal, similar to recent sender-managed, delay-based mechanisms [215, 174, 294]. Each CN maintains one congestion window, *cwnd*, per MN that controls the maximum number of outstanding requests that can be made to the MN from this CN. We adjust *cwnd* based on

measured delay using a standard Additive Increase Multiplicative Decrease (AIMD) algorithm.

To handle incast to a CN, we exploit the fact that the CN knows the sizes of expected responses for the requests that it sends out and that responses are the major incoming traffic to it. Each CLib maintains one incast window, *iwnd*, which controls the maximum bytes of expected responses. CLib sends a request only when both *cwnd* and *iwnd* have room.

Handling incast to an MN is more challenging, as we cannot throttle incoming traffic at the MN side or would otherwise maintain state at MNs. To have CNs handle incast to MNs, we draw inspiration from Swift [174] by allowing *cwnd* to fall below one packet when long delay is observed at a CN. For example, a *cwnd* of 0.1 means that the CN can only send a packet within 10 RTTs. Essentially, this situation happens when the network between a CN and an MN is really congested, and the only way is to slow the sending speed.

4.4.5 Request Ordering and Data Consistency

As explained in §4.3.1, Clio supports both synchronous and asynchronous remote memory APIs, with the former following a sequential, one-at-a-time order in a thread and the latter following a release order in a thread. Furthermore, Clio provides synchronization primitives for inter-thread consistency. We now discuss how Clio achieves these correctness guarantees by presenting our mechanisms for handling intra-request intra-thread ordering, inter-request intra-thread ordering, inter-thread consistency, and retries. At the end, we will provide the rationales behind our design.

One difficulty in designing the request ordering and consistency mechanisms is our relaxed network ordering guarantees, which we adopt to minimize the hardware resource consumption for the network layer at MNs (§4.4.4). On an asynchronous network, it is generally hard to guarantee any type of request ordering when there can be multiple outstanding requests (either multiple threads accessing shared memory or a single thread issuing multiple asynchronous APIs). It is even harder for Clio because we aim to make MN stateless as much as possible. Our

general approaches are 1) using CNs to ensure that no two concurrently outstanding requests are dependent on each other, and 2) using MNs to ensure that every user request is only executed once even in the event of retries.

Allowing intra-request packet re-ordering (T1). A request or a response in Clio can contain multiple link-layer packets. Enforcing packet ordering above the link layer normally requires maintaining state (*e.g.*, packet sequence ID) at both the sender and the receiver. To avoid maintaining such state at MNs, our approach is to deal with packet reordering only at CNs in CLib (**Principle 3**). Specifically, CLib splits a request that is bigger than link-layer maximum transmission unit (MTU) into several link-layer packets and attaches a Clio header to each packet, which includes sender-receiver addresses, a request ID, and request type. This enables the MN to treat each packet independently (**Principle 5**). It executes packets as soon as they arrive, even if they are not in the sending order. This out-of-order data placement semantic is in line with RDMA specification [216]. Note that only write requests will be bigger than MTU, and the order of data writing within a write request does not affect correctness as long as proper *inter-request* ordering is followed. When a CN receives multiple link-layer packets belonging to the same request response, CLib reassembles them before delivering them to the application.

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

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

Specifically, CLib keeps track of all inflight requests and matches every new request's

virtual page number (VPN) to the inflight ones'. If a WAR, RAW, or WAW dependency is detected, CLib blocks the new request until the conflicting request finishes. When CLib sees a `rrelease` operation, it waits until all inflight requests return or time out. We currently track dependencies at the page granularity mainly to reduce tracking complexity and metadata overhead. The downside is that false dependencies could happen (*e.g.*, two accesses to the same page but different addresses). False dependencies could be reduced by dynamically adapting the tracking granularity if application access patterns are tracked—we leave this improvement for future work.

Inter-thread/process consistency (T3). Multi-threaded or multi-process concurrent programming on Clio could use the synchronization primitives Clio provides to ensure data consistency (§4.3.1). We implemented all synchronization primitives like `rlock` and `rfence` at MN, because they need to work across threads and processes that possibly reside on different CNs. Before a request enters either the fast or the slow paths, MN checks if it is a synchronization primitive. For primitives like `rlock` that internally is implemented using atomic operations like `TAS`, MN blocks future atomic operations until the current one completes. For `rfence`, MN blocks all future requests until all inflight ones complete. Synchronization primitives are one of the only two cases where MN needs to maintain state. As these operations are infrequent and each of these operations executes in bounded time, the hardware resources for maintaining their state are minimal and bounded.

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

First, CLib attaches a new request ID to each retry, essentially making it a new request with its own matching response. Together with CLib's ordering enforcement, it ensures that there is only one outstanding request (or a retry) at any time. Second, we maintain a small buffer at

MN to record the request IDs of recently executed writes and atomic APIs and the results of the atomic APIs. A retry attaches its own request ID and the ID of the failed request. If MN finds a match of the latter in the buffer, it will not execute the request. For atomic APIs, it sends the cached result as the response. We set this buffer's size to be $3 \times \texttt{TIMEOUT} \times \textit{bandwidth}$, which is 30 KB in our setting. It is one of the only two types of state MN maintains and does not affect the scalability of MN, since its size is statically associated with the link bandwidth and the `TIMEOUT` value. With this size, the MN can "remember" an operation long enough for two retries from the CN. Only when both retries and the original request all fail, the MN will fail to properly handle a future retry. This case is extremely rare [219], and we report the error to the application, similar to [154, 294].

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

4.4.6 Extension and Offloading Support

To avoid network round trips when working with complex data structures and/or performing data-intensive operations, we extend the core MN to support application computation offloading in the extend path. Users can write and deploy application offloads both in FPGA and in software (run in the ARM). To ease the development of offloads, Clio offers the same virtual memory interface as the one to applications running at CNs. Each offload has its own PID and virtual memory address space, and they use the same virtual memory APIs (§4.3.1) to access on-board memory. It could also share data with processes running at CNs in the same way that two CN processes share memory. Finally, an offload's data and control paths could be split to FPGA and ARM and use the same async-buffer mechanism for communication between them. These unique designs made developing computation offloads easier and closer to traditional multi-threaded software programming.

4.4.7 Distributed MNs

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

The main difference between LegoOS and Clio's distributed memory system is that in Clio, each MN can be over-committed (*i.e.*, allocating more virtual memory than its physical memory size), and when an MN is under memory pressure, it migrates data to another MN that is less pressured (coordinated by the global controller). The traditional way of providing

memory over-commitment is through memory swapping, which could be potentially implemented by swapping memory between MNs. However, swapping would cause performance impact on the data path and add complexity to the hardware implementation. Instead of swapping, we *proactively* migrate a rarely accessed memory region to another MN when an MN is under memory pressure (its free physical memory space is below a threshold). During migration, we pause all client requests to the region being migrated. With our 10 Gbps experimental board, migrating a 1 GB region takes 1.3 second. Migration happens rarely and, unlike swapping, happens in the background. Thus, it has little disturbance to foreground application performance.

4.5 Clio Implementation

Apart from challenges discussed in §4.4, our implementation of Clio also needs to overcome several practical challenges, for example, how can different hardware components most efficiently work together in CBoard, how to minimize software overhead in CLib. This section describes how we implemented CBoard and CLib, focusing on the new techniques we designed to overcome these challenges. Currently, Clio consists of 24.6K SLOC (excluding computation offloads and third-party IPs). They include 5.6K SLOC in SpinalHDL [297] and 2K in C HLS for FPGA hardware, and 17K in C for CLib and ARM software. We use vendor-supplied interconnect and DDR IPs, and an open-source MAC and PHY network stack [109].

CBoard Prototyping. We prototyped CBoard with a low-cost ($2495 retail price) Xilinx MPSoC board [333] and build the hardware fast path (which is anticipated to be built in ASIC) with FPGA. All Clio's FPGA modules run at 250 MHz clock frequency and 512-bit data width. They all achieve an *Initiation Interval* (*II*) of one (II is the number of clock cycles between the start time of consecutive loop iterations, and it decides the maximum achievable bandwidth). Achieving II of one is not easy and requires careful pipeline design in all the modules. With II one, our data path can achieve a maximum of 128 Gbps throughput even with just the slower

FPGA clock frequency and would be higher with real ASIC implementation.

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

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

To mitigate the problem of slow accesses to on-board DRAM from ARM, we maintain shadow copies of metadata at ARM's local DRAM. For example, we store a *shadow* version of the page table in ARM's local memory, so that the control path can read page table content faster. When the control path needs to perform a virtual memory space allocation, it reads the shadow page table to test if an address would cause an overflow (§4.4.2). We keep the shadow page table in sync with the real page table by updating both tables when adding, removing, or updating the page table entries.

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

tasks to other worker threads for task handling and post responses to a TX ring buffer.

CBoard's network stack builds on top of standard, vendor-supplied Ethernet physical and link-layer IPs, with just an additional thin checksum-verify and ack-generation layer on top. This layer uses much fewer resources compared to a normal RDMA-like stack (§4.7.3). We use lossless Ethernet with Priority Flow Control (PFC) for less packet loss and retransmission. Since PFC has issues like head-of-line blocking [350, 190, 113, 216], we rely on our congestion and incast control to avoid triggering PFC as much as possible.

Finally, to assist Clio users in building their applications, we implemented a simple software simulator of CBoard which works with CLib for developers to test their code without the need to run an actual CBoard.

CLib Implementation. Even though we optimize the performance of CBoard, the end-to-end application performance can still be hugely impacted if the host software component (CLib) is not as fast. Thus, our CLib implementation aims to provide low-latency performance by adopting several ideas (e.g., data inlining, doorbell batching) from recent low-latency I/O solutions [151, 154, 32, 315, 150, 241, 340]. We implemented CLib in the user space. It has three parts: a user-facing request ordering layer that performs dependency check and ordering of address-conflicting requests, a transport layer that performs congestion/incast control and request-level retransmission, and a low-level device driver layer that interacts with the NIC (similar to DPDK [88] but simpler). CLib bypasses kernel and directly issues raw Ethernet requests to the NIC with zero memory copy. For synchronous APIs, we let the requesting thread poll the NIC for receiving the response right after each request. For asynchronous APIs, the application thread proceeds with other computations after issuing the request and only busy polls when the program calls `rpoll`.

4.6 Building Applications on Clio

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

Image compression. We build a simple image compression/decompression utility that runs purely at CN. Each client of the utility (*e.g.*, a Facebook user) has its own collection of photos, stored in two arrays at MNs, one for compressed and one for original, both allocated with `ralloc`. Because clients' photos need to be protected from each other, we use one process per client to run the utility. The utility simply reads a photo from MN using `rread`, compresses/decompresses it, and writes it back to the other array using `rwrite`. Note that we use compression and decompression as an example of image processing. These operations could potentially be offloaded to MNs. However, in reality, there can be many other types of image processing that are more complex and are hard and costly to implement in hardware, necessitating software processing at CNs. We implemented this utility with 1K C code in 3 developer days.

Radix tree. To demonstrate how to build a data structure on Clio using Clio's extended API, we built a radix tree with linked lists and pointers. Data-structure-level systems like AIFM [273] could follow this example to make simple changes in their libraries to run on Clio. We first built an extended pointer-chasing functionality in FPGA at the MN which follows pointers in a linked list and performs a value comparison at each traversed list node. It returns either the node value when there is a match or null when the next pointer becomes null. We then expose this functionality to CNs as an extended API. The software running at CN allocates a big contiguous remote memory space using `ralloc` and uses this space to store radix tree nodes. Nodes in each layer are linked to a list. To search a radix tree, the CN software goes through each layer of the tree and calls the pointer chasing API until a match is found. We implemented the radix tree with 300 C code at CN and 150 SpinalHDL code at CBoard in less than one developer day.

Key-value store. We built *Clio-KV*, a key-value store that supports concurrent create/update/read/delete key-value entries with atomic write and read committed consistency. Clio-KV runs at an MN as a computation offloading module. Users can access it through a key-value interface from multiple CNs. The Clio-KV module has its own virtual memory address space and uses Clio virtual memory APIs to access it. Clio-KV uses a chained hash table in its virtual memory space for managing the metadata of key-value pairs, and it stores the actual key values at separate locations in the space. Each hash bucket has a chain of slots. Each slot contains the virtual addresses of seven key-value pairs. It also stores a fingerprint for each key-value pair.

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

To perform a read, Clio-KV locates the hash bucket (with the key's hash value) and fetches one slot in the bucket chain at a time using `rread`. It then compares the fingerprint of the key to the seven entries in the slot. If there is no match, it fetches the next slot in the bucket. Otherwise, with a matched entry, it reads the key-value pair using the address stored in that entry with an `rread`. It then compares the full key and returns the value if it is a match. Otherwise, it keeps searching the bucket.

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

We implemented Clio-KV with 772 SpinalHDL code in 6 developer days. To evaluate Clio's virtual memory API overhead at CBoard, we also implemented a key-value store with the

same design as Clio-KV but with raw physical memory interface. This physical-memory-based implementation takes more time to develop and only yields 4%–12% latency improvement and 1%–5% throughput improvement over Clio-KV.

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

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

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

is improved by avoiding data serialization/deserialization. We implemented Clio-DF with 202 lines of SpinalHDL code and 170 lines of C interface code in 7 developer days.

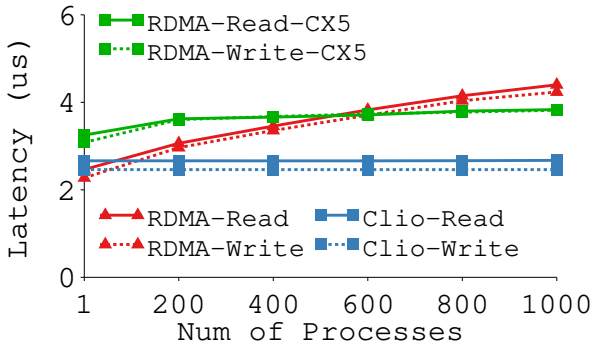


Figure 4.4: Process (Connection) Scalability.

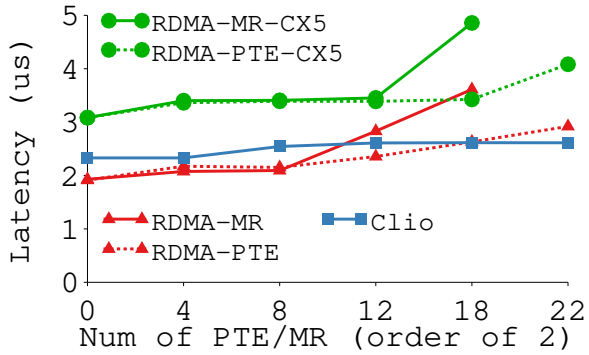


Figure 4.5: PTE and MR Scalability. RDMA fails beyond 2^{18} MRs.

4.7 Evaluation

Our evaluation reveals the scalability, throughput, median and tail latency, energy and resource consumption of Clio. We compare Clio's end-to-end performance with industry-grade NICs (ASIC) and well-tuned RDMA-based software systems. All Clio's results are FPGA-based, which would be improved with ASIC implementation.

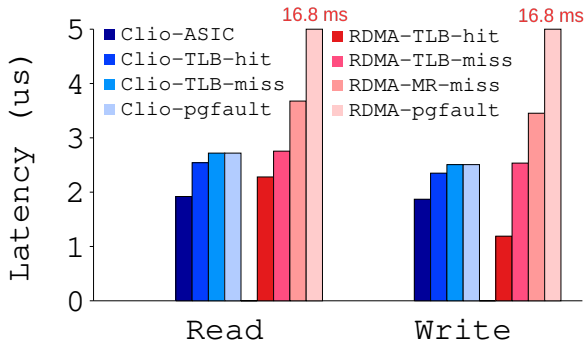


Figure 4.6: Comparison of TLB Miss and page fault. Clio-ASIC are projected values of TLB hit.

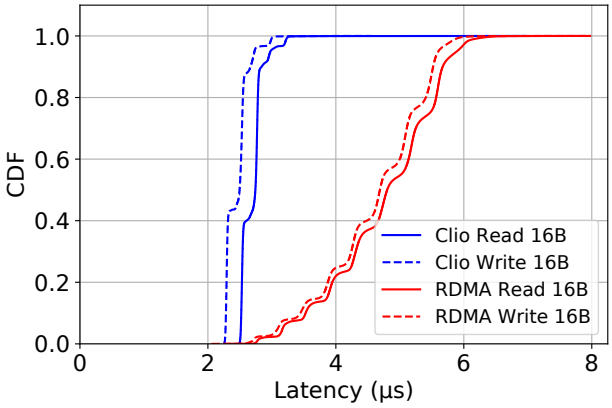


Figure 4.7: Latency CDF.

Environment. We evaluated Clio in our local cluster of four CNs and four MNs (Xilinx ZCU106 boards), all connected to an Nvidia 40 Gbps VPI switch. Each CN is a Dell PowerEdge R740 server equipped with a Xeon Gold 5128 CPU and a 40 Gbps Nvidia ConnectX-3 NIC, with two of them also having an Nvidia BlueField SmartNIC [210]. We also include results from CloudLab [71] with the Nvidia ConnectX-5 NIC.

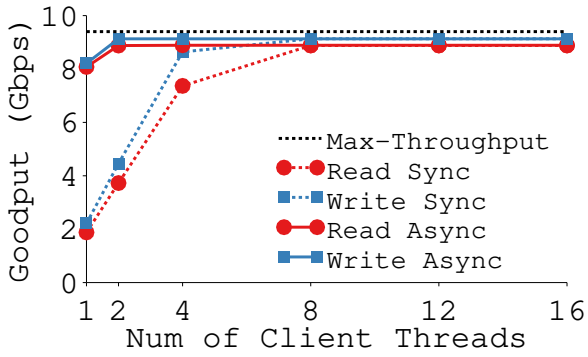


Figure 4.8: End-to-End Goodput. 1 KB requests.

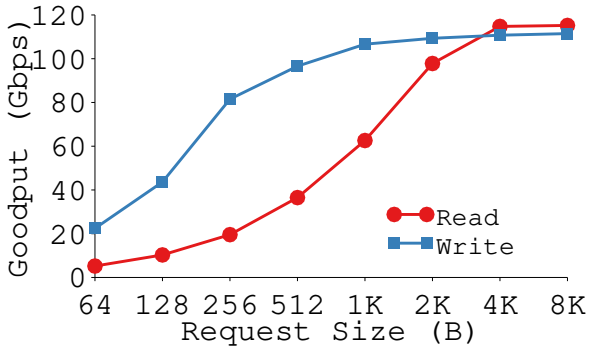


Figure 4.9: On-board Goodput. FPGA test module generates requests at maximum speed.

4.7.1 Basic Microbenchmark Performance

Scalability. We first compare the scalability of Clio and RDMA. Figure 4.4 measures the latency of Clio and RDMA as the number of client processes increases. For RDMA, each process uses its own QP. Since Clio is connectionless, it scales perfectly with the number of processes. RDMA scales poorly with its QP, and the problem persists with newer generations of RNIC, which is also confirmed by our previous works [313, 235].

Figure 4.5 evaluates the scalability with respect to PTEs and memory regions. For the memory region test, we register multiple MRs using the same physical memory for RDMA. For

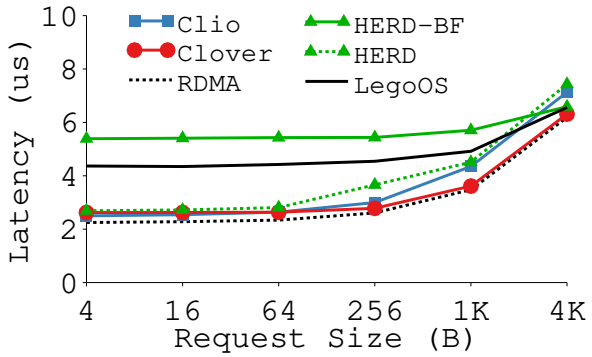


Figure 4.10: Read Latency. HERD-BF: HERD running on BlueField.

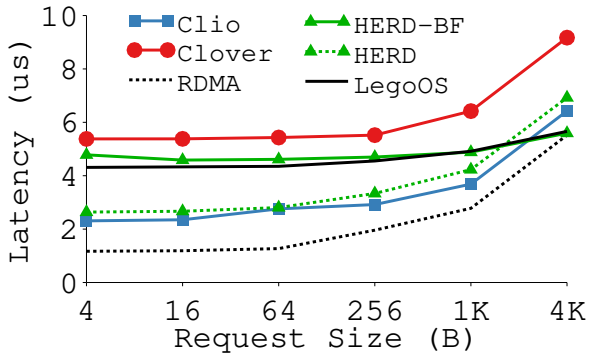


Figure 4.11: Write Latency. Clover requires $\geq$ 2 RTTs for write.

Clio, we map a large range of VAs (up to 4 TB) to a small physical memory space, as our testbed only has 2 GB physical memory. However, the number of PTEs and the amount of processing needed are the same for CBoard as if it had a real 4 TB physical memory. Thus, this workload stress tests CBoard's scalability. RDMA's performance starts to degrade when there are more than 2^8 (local cluster) or 2^{12} (CloudLab), and the scalability wrt MR is worse than wrt PTE. In fact, RDMA fails to run beyond 2^{18} MRs. In contrast, Clio scales well and never fails (at least up to 4 TB memory). It has two levels of latency that are both stable: a lower latency below 2^4 for TLB hit and a higher latency above 2^4 for TLB miss (which always involves one DRAM access). A CBoard could use a larger TLB if optimal performance is desired.

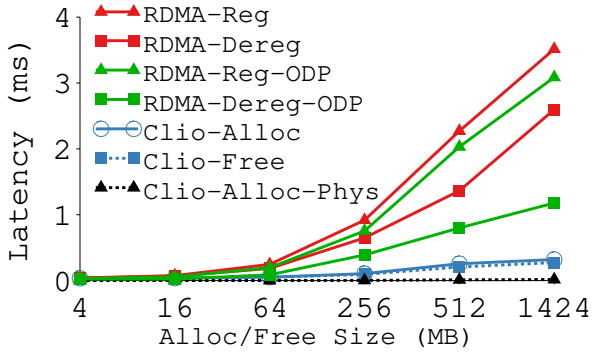


Figure 4.12: Alloc/Free Latency. ODP means On-Demand-Paging mode

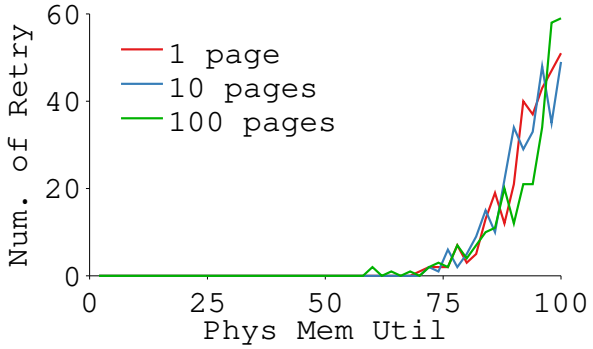


Figure 4.13: Alloc Retry Rate.

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

Latency variation. Figure 4.6 plots the latency of reading/writing 16 B data when the operation results in a TLB hit, a TLB miss, a first-access page fault, and MR miss (for RDMA only, when the MR metadata is not in RNIC). RDMA's performance degrades significantly with misses. Its page fault handling is extremely slow (16.8 *ms*). We confirm the same effect on CloudLab with the newer ConnectX-5 NICs. Clio only incurs a small TLB miss cost and **no additional cost of page fault handling**.

We also include a projection of Clio's latency if it was to be implemented using a real

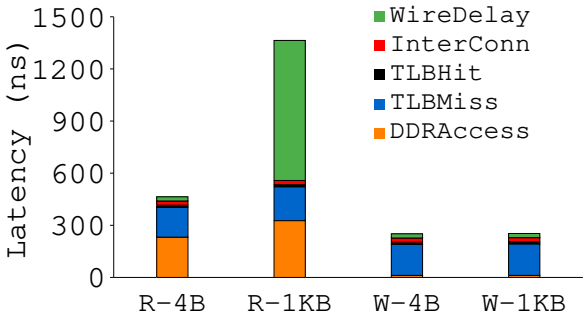


Figure 4.14: Latency Breakdown. Breakdown of time spent at CBoard.

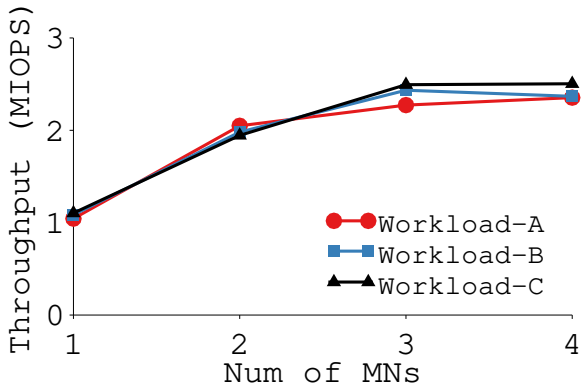


Figure 4.15: Clio-KV Scalability against MNs.

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

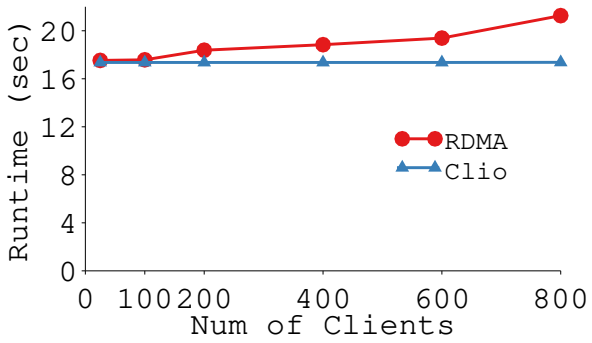


Figure 4.16: Image Compression.

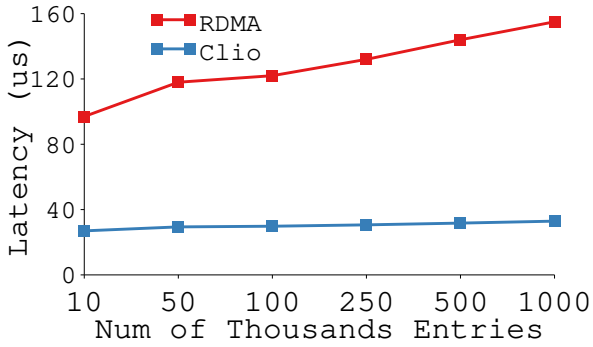


Figure 4.17: Radix Tree Search Latency.

Figure 4.7 plots the request latency CDF of continuously running read/write 16 B data while not triggering page faults. Even without page faults, Clio has much less latency variation and a much shorter tail than RDMA.

Read/write throughput. We measure Clio's throughput by varying the number of concurrent client threads (Figure 4.8). Clio's default asynchronous APIs quickly reach the line rate of our testbed (9.4 Gbps maximum throughput). Its synchronous APIs could also reach line rate fairly quickly.

Figure 4.9 measures the maximum throughput of Clio's FPGA implementation without the bottleneck of the board's 10 Gbps port, by generating traffic on board. Both read and write can

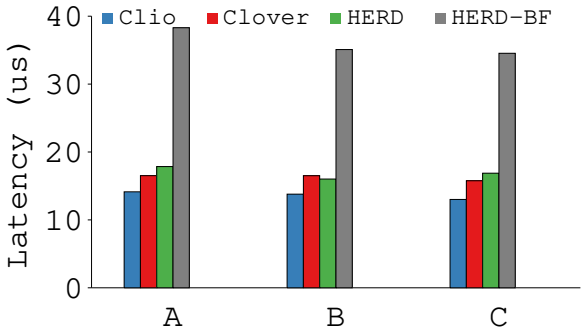


Figure 4.18: Key-Value Store YCSB Latency.

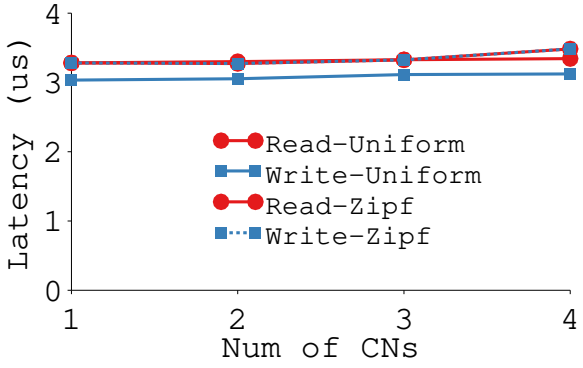


Figure 4.19: Clio-MV Object Read/Write Latency.

reach more than 110 Gbps when request size is large. Read throughput is lower than write when request size is smaller. We found the throughput bottleneck to be the third-party non-pipelined DMA IP (which could potentially be improved).

Comparison with other systems. We compare Clio with native one-sided RDMA, Clover [314], HERD [154], and LegoOS [283]. We ran HERD on both CPU and BlueField (HERD-BF). Clover is a passive disaggregated persistent memory system which we adapted as a passive disaggregated memory (PDM) system. HERD is an RDMA-based system that supports a key-value interface with an RPC-like architecture. LegoOS builds its virtual memory system in software at MN.

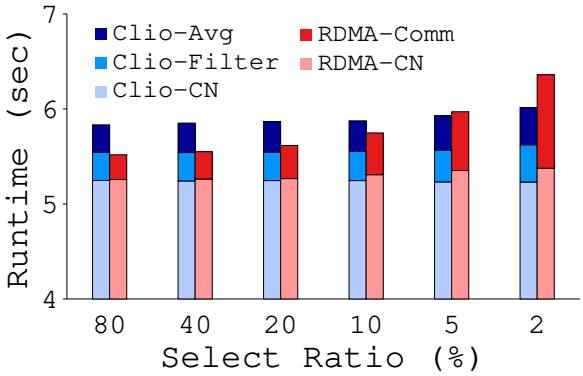


Figure 4.20: Select-Aggregate-Shuffle. Y axis starts at 4 sec. CN represents computation done at CN.

Table 4.1: Clio FPGA Utilization.

System/Module	Logic (LUT)	Memory (BRAM)
StRoM-RoCEv2	39%	76%
Tonic-SACK	48%	40%
Clio (Total)	31%	31%
VirtMem	5.5%	3%
NetStack	2.3%	1.7%
Go-Back-N	5.8%	2.6%

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

Allocation performance. Figure 4.12 shows Clio's VA and PA allocation and RDMA's

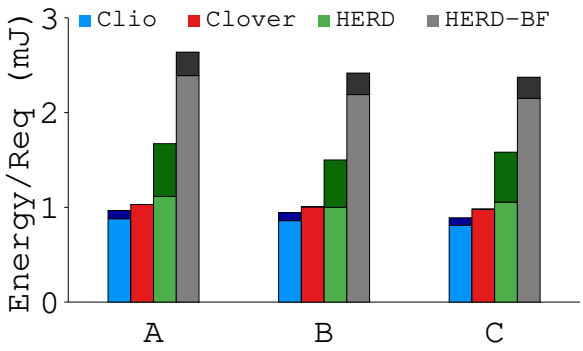


Figure 4.21: Energy Comparison. Darker/lighter shades represent energy spent at MNs and CNs.

MR registration performance. Clio's PA allocation takes less than 20 μs, and the VA allocation is much faster than RDMA MR registration, although both get slower with larger allocation/registration size. Figure 4.13 shows the number of retries at allocation time with three allocation sizes as the physical memory fills up. There is no retry when memory is below half utilized. Even when memory is close to full, there are at most 60 retries per allocation request, with roughly 0.5 ms per retry. This confirms that our design of avoiding hash overflows at allocation time is practical.

Close look at CBoard components. To further understand Clio's performance, we profile different parts of Clio's processing for read and write of 4 B to 1 KB. CLib adds a very small overhead (250 ns in total), thanks to our efficient threading model and network stack implementation. Figure 4.14 shows the latency breakdown at CBoard. Time to fetch data from DRAM (DDRAccess) and to transfer it over the wire (WireDelay) are the main contributor to read latency, especially with large read size. Both could be largely improved in a real CBoard with better memory controller and higher frequency. TLB miss (which takes one DRAM read) is the other main part of the latencies.

4.7.2 Application Performance

Image Compression. We run a workload where each client compresses and decompresses 1000 256*256-pixel images with increasing number of concurrently running clients. Figure 4.16 shows the total runtime per client. We compare Clio with RDMA, with both performing computation at the CN side and the RDMA using one-sided operations instead of Clio APIs to read/write images in remote memory. Clio's performance stays the same as the number of clients increase. RDMA's performance does not scale because it requires each client to register a different MR to have protected memory accesses. With more MRs, RDMA runs into the case where the RNIC cannot hold all the MR metadata and many accesses would involve a slow read to host main memory.

Radix Tree. Figure 4.17 shows the latency of searching a key in pre-populated radix trees when varying the tree size. We again compare with RDMA which uses one-sided read operations to perform the tree traversal task. RDMA's performance is worse than Clio, because it requires multiple RTTs to traverse the tree, while Clio only needs one RTT for each pointer chasing (each tree level). In addition, RDMA also scales worse than Clio.

Key-value store. Figure 4.18 evaluates Clio-KV using the YCSB benchmark [10] and compares it to Clover, HERD, and HERD-BF. We run two CNs and 8 threads per CN. We use 100K key-value entries and run 100K operations per test, with YCSB's default key-value size of 1 KB. The accesses to keys follow the Zipf distribution ($\theta = 0.99$). We use three YCSB workloads with different *get-set* ratios: 100% *get* (workload C), 5% *set* (B), and 50% *set* (A). Clio-KV performs the best. HERD running on BlueField performs the worst, mainly because BlueField's slower crossing between its NIC chip and ARM chip.

Figures 4.15 shows the throughput of Clio-KV when varying the number of MNs. Similar to our Clio scalability results, Clio-KV can reach a CN's maximum throughput and can handle concurrent get/set requests even under contention. These results are similar to or better than previous FPGA-based and RDMA-based key-value stores that are fine-tuned for just key-value

workloads (Table 3 in [187]), while we got our results without any performance tuning.

Multi-version data store. We evaluate Clio-MV by varying the number of CNs that concurrently access data objects (of 16 B) on an MN using workloads of 50% read (of different versions) and 50% write under uniform and Zipf distribution of objects (Figure 4.19). Clio-MV's read and write have the same performance, and reading any version has the same performance, since we use an array-based version design.

Data analytics. We run a simple workload which first `select` rows in a table whose field-A matches a value (*e.g.*, gender is female) and calculate `avg` of field-B (*e.g.*, final score) of all the rows. Finally, it calculates the histogram of the selected rows (*e.g.*, score distribution), which can be presented to the user together with the avg value. Clio executes the first two steps at MN offloads and the final step at CN, while RDMA always reads rows to CN and then does each operation. Figure 4.20 plots the total run time as the select ratio decreases (fewer rows selected). When the select ratio is low, Clio transfers much less data than RDMA, resulting in its better performance.

4.7.3 CapEx, Energy, and FPGA Utilization

We estimate the cost of server and CBoard using market prices of different hardware units. When using 1 TB DRAM, a server-based MN costs 1.1-1.5× and consumes 1.9-2.7× power compared to CBoard. These numbers become 1.4-2.5× and 5.1-8.6× with OptaneDimm [251], which we expect to be the more likely remote memory media in future systems.

We measure the total energy used for running YCSB workloads by collecting the total CPU (or FPGA) cycles and the Watt of a CPU core [11], ARM processor [250], and FPGA (measured). We omit the energy used by DRAM and NICs in all the calculations. Clover, a system that centers its design around low cost, has slightly higher energy than Clio. Even though there is no processing at MNs for Clover, its CNs use more cycles to process and manage memory. HERD consumes 1.6× to 3× more energy than Clio, mainly because of its CPU overhead at MNs.

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

Figure 5.1 compares the FPGA utilization among Clio, StRoM's RoCEv2 [290], and Tonic's selective ack stack [33]. Both StRoM and Tonic include only a network stack but they consume more resources than Clio. Within Clio, the virtual memory (VirtMem) and the network stack (NetStack) consume a small fraction of the total resources, with the rest being vendor IPs (PHY, MAC, DDR4, and interconnect). Overall, our efficient hardware implementation leaves most FPGA resources available for application offloads.

4.8 Discussion and Conclusion

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

Security and performance isolation. Clio's protection domain is a user process, which is the same as the traditional single-server process-address-space-based protection. The difference is that Clio performs permission checks at MNs: it restricts a process' access to only its (remote) memory address space and does this check based on the global PID. Thus, the safety of Clio relies on PIDs to be authentic (*e.g.*, by letting a trusted CN OS or trusted CN hardware attach process IDs to each Clio request). There have been researches on attacking RDMA systems by forging requests [270] and on adding security features to RDMA [294, 305]. How these and other existing security works relate and could be extended in a memory disaggregation setting is an open problem, and we leave this for future work.

There are also designs in our current implementation that could be improved to provide

more protection against side-channel and DoS attacks. For example, currently, the TLB is shared across application processes, and there is no network bandwidth limit for an individual connection. Adding more isolation to these components would potentially increase the cost of CBoard or reduce its performance. We leave exploring such tradeoffs to future work.

Failure handling. Although memory systems are usually assumed to be volatile, there are still situations that require proper failure handling (*e.g.*, for high availability or to use memory for storing data). As there can be many ways to build memory services on Clio and many such services are already or would benefit from handling failure on their own, we choose not to have any built-in failure handling mechanism in Clio. Instead, Clio should offer primitives like replicated writes for users to build their own services. We leave adding such API extensions to Clio as future work.

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

4.9 Acknowledgments

Chapter 4, in full, is a reprint of Yizhou Shan, Zhiyuan Guo (co-first authors), Xuhao Luo, Yutong Huang, Yiying Zhang, "Clio: A Hardware-Software Co-Designed Disaggregated Memory System", *ASPLOS, 2022*. The dissertation author was the primary investigator and author of this paper.

Chapter 5

Disaggregating and Consolidating Network Functionalities with SuperNIC

5.1 Introduction

Hardware resource disaggregation is a solution that decomposes full-blown, general-purpose servers into segregated, network-attached hardware resource pools, each of which can be built, managed, and scaled independently. With disaggregation, different resources can be allocated from any device in their corresponding pools, exposing vast amounts of resources to applications and at the same time improving resource utilization. Disaggregation also allows data-center providers to independently deploy, manage, and scale different types of resources. Because of these benefits, disaggregation has gained significant traction from both academia [283, 36, 314, 230, 31, 273, 324, 57, 127, 23] and industry [133, 143, 60, 194, 322].

While increasing amounts of effort go into disaggregating compute [283, 128], memory (or persistent memory) [283, 133, 191, 16, 314, 324, 127, 23, 285], and storage [59, 322, 50, 165, 213], the fourth major resource, *network*, has been completely left out. At first glance, "network" cannot be disaggregated from either a traditional monolithic server or a disaggregated device

(in this paper collectively called *endpoints*), as they both need to be attached to the network. However, we observe that even though endpoints need basic connectivity, it is not necessary to run *network-related tasks* at the endpoints. These network tasks, or *NT*s, include the transport layer and all high-level layers such as network virtualization, packet filtering and encryption, and application-specific functions.

This paper, for the first time, proposes the concept of *network disaggregation* and builds a real disaggregated network system to segregate NTs from endpoints.

At the core of our network-disaggregation proposal is the concept of a rack-scale disaggregated *network resource pool*, which consists of a set of hardware devices that can execute NTs and collectively provide "network" as a service (Figure 5.1), similar to how today's disaggregated storage pool provides data storage service to compute nodes. Endpoints can offload (*i.e.*, disaggregate) part or all of their NTs to the network resource pool. After NTs are disaggregated, we further propose to *consolidate* them by aggregating a rack's endpoint NTs onto a small set of network devices.

We foresee two architectures of the network resource pool within a rack. The first architecture inserts a network pool between endpoints and the ToR switch by attaching a small set of endpoints to one network device, which is then connected to the ToR switch (Figure 5.1 (a)). The second architecture attaches the pool of network devices to the ToR switch, which then connects to all the endpoints (Figure 5.1 (b)).

Network disaggregation and consolidation have several key benefits. (1) Disaggregating NTs into a separate pool allows data center providers to build and manage network functionalities only at one place instead of at each endpoint. This is especially helpful for heterogeneous disaggregated clusters where a full network stack would otherwise need to be developed and customized for each type of endpoint. (2) Disaggregating NTs into a separate pool allows the *independent scaling* of hardware resources used for network functionalities without the need to change endpoints. (3) Each endpoint can use more network resources than what can traditionally

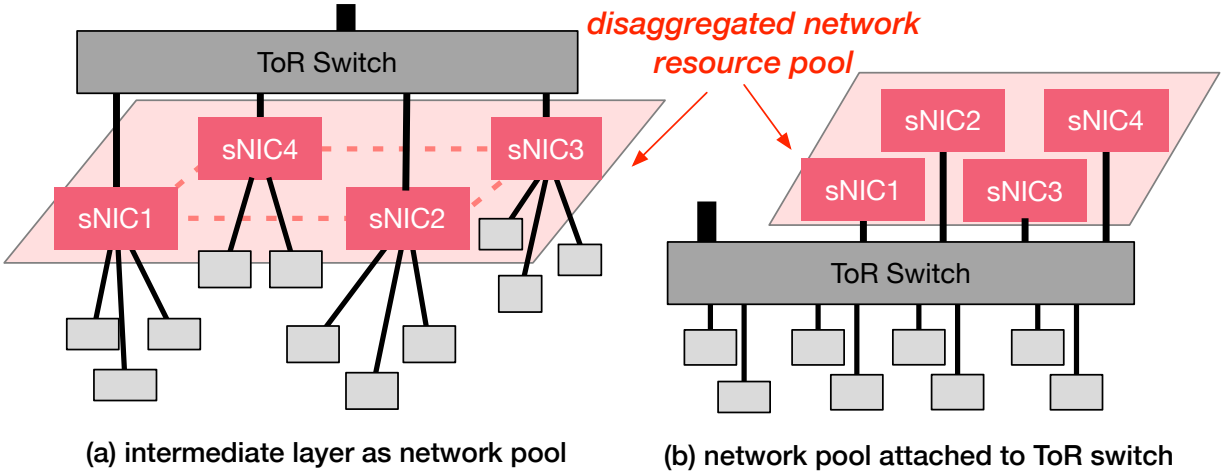


Figure 5.1: Overall Architectures of SuperNIC. Two ways of connecting sNICs to form a disaggregated network resource pool. In (a), dashed lines represent links that are optional.

fit in a single NIC. (4) With NT consolidation, the total number of network devices can be reduced, allowing a rack to host more endpoints. (5) The network pool only needs to provision hardware resources for the peak *aggregated* bandwidth in a rack instead of each endpoint provisioning for its own peak, reducing the overall CapEx cost.

Before these benefits can be exploited in a real data center, network disaggregation needs to first meet several goals, which no existing solutions fully support (see §5.2.2).

First, each disaggregated network device should meet endpoints' original performance goals even when handling a much larger (aggregated) load than what each endpoint traditionally handles. The aggregated load will also likely require many different NTs, ranging from transports to application-specific functionalities. Moreover, after aggregating traffic, there are likely more load spikes (each coming from a different endpoint) that the device needs to handle.

Second, using a disaggregated network pool should reduce the total cost of a cluster. This means that each disaggregated network device should provision the right amount of hardware resources (CapEx) and use as little of them as needed at run time (OpEx). At the same time, the

remaining part of a rack (*e.g.*, endpoints, ToR switch, cables) needs to be carefully designed to be low cost.

Third, as we are consolidating NTs from multiple endpoints, in a multi-tenant environment, there would be more entities that need to be isolated. We should ensure that they fairly and safely share various hardware resources in a disaggregated network pool.

Finally, network devices in a pool need to work together so that lightly loaded devices can handle traffic for other devices that are overloaded. This load balancing would allow each device to provision less hardware resources as long as the entire pool can handle the peak aggregated load of the rack.

Meeting these requirements together is not easy as they imply that the disaggregated network devices need to use minimal and properly isolated hardware resources to handle large loads with high variation, while achieving application performance as if there is no disaggregation.

To tackle these challenges and to demonstrate the feasibility of network disaggregation, we built ***SuperNIC*** (or *sNIC* for short), a new hardware-based programmable network device designed for network disaggregation. An sNIC device consists of an ASIC for fixed systems logic, FPGA for running and reconfiguring NTs, and software cores for executing the control plane. We further built a distributed sNIC platform that serves as a disaggregated network pool. Users can deploy a single NT written for FPGA or a directed acyclic graph (DAG) execution plan of NTs to the pool.

To tightly **consolidate** NTs within an sNIC, we support three types of resource sharing: (1) splitting an sNIC's hardware resources across different NTs (*space sharing*), (2) allowing multiple applications to use the same NT at different times (*time sharing*), and (3) configuring the same hardware resources to run different NTs at different times (*time sharing with context switching*). For space sharing, we partition the FPGA space into *region*s, with each hosting one or more NTs. Each region could be individually *reconfigured* (via FPGA partial reconfiguration, or *PR*) for starting new NTs or to context switch NTs. Different from traditional software systems,

hardware context switching with PR is orders of magnitude slower, which could potentially impact application performance significantly. To solve this unique challenge, we propose a set of policies and mechanisms to reduce the need to perform PR or to move it off the performance-critical path, *e.g.*, by keeping de-scheduled NTs around like a traditional victim cache, by not over-reacting to load spikes, and by utilizing other sNICs when one sNIC is overloaded.

To achieve high **performance** under large, varying load with minimal cost, we automatically scale (auto-scale) an NT by adding/removing instances of it and sending different flows in an application across these instances. We further launch different NTs belonging to the same application in parallel and send forked packets to them in parallel for faster processing. To achieve low scheduling latency and improve scalability, we group NTs that are likely to be executed in a sequence into a chain. Our scheduler reserves credits for the entire chain as much as possible so that packets execute the chain as a whole without involving the scheduler in between.

To provide **fairness**, we adopt a fine-grained approach that treats each internal hardware resource separately, *e.g.*, ingress/egress bandwidth, internal bandwidth of each shared NT, payload buffer space, and on-board memory, as doing so allows a higher degree of consolidation. We adopt Dominant Resource Fairness (DRF) [117] for this multi-dimensional resource sharing. Instead of user-supplied, static per-resource demands as in traditional DRF systems, we monitor the actual load demands at run time and use them as the target in the DRF algorithm. Furthermore, we propose to use ingress bandwidth throttling to control the allocation of other types of resources. We also build a simple virtual memory system to **isolate and protect** accesses to on-board memory.

Finally, for **distributed sNICs**, we automatically scale out NTs beyond a single sNIC when load increases and support different mechanisms for balancing loads across sNICs depending on the network pool architectures. For example, with the switch-attached pool architecture, we use the ToR switch to balance all traffic across sNICs. With the intermediate pool architecture, we further support a peer-to-peer, sNIC-initiated load migration when one sNIC is overloaded.

We prototype sNIC with FPGA using two 100 Gbps, multi-port HiTech Global HTG-9200 boards [3]. We build three types of NTs to run on sNIC: reliable transport, traditional network functions, and application-specific tasks, and port two end-to-end use cases to sNIC. The first use case is a key-value store we built on top of real disaggregated memory devices [347]. We explore using sNICs for traditional NTs like the transport layer and customized NTs like key-value data replication and caching. The second use case is a Virtual Private Cloud application we built on top of regular servers by connecting sNICs at both the sender and the receiver side. We disaggregate NTs like encapsulation, firewall, and encryption to the sNICs. We evaluate sNIC and the ported applications with micro- and macro-benchmarks and compare sNIC with no network disaggregation and disaggregation using alternative solutions such as multi-host NICs and a recent multi-tenant SmartNIC [193]. Overall, sNIC achieves 52% to 56% CapEx and OpEx cost savings with only 4% performance overhead compared to a traditional non-disaggregated per-endpoint SmartNIC scenario. Furthermore, the customized key-value store caching and replication functionalities on sNIC improves throughput by 1.31× to 3.88× and latency by 1.21× to 1.37× when compared to today's remote memory systems with no sNIC.

5.2 Motivation and Related Works

5.2.1 Benefits of Network Disaggregation

As discussed in §5.1, disaggregating network functionalities into a separate pool has several key benefits for data centers, some of which are especially acute for future disaggregated, heterogeneous data centers [283, 228, 196].

Flexible management and low development cost. Modern data centers are deploying an increasing variety of network tasks to endpoints, usually in different forms (*e.g.*, software running on a CPU, fixed-function tasks built as ASIC in a NIC, software and programmable hardware deployed in a SmartNIC). It requires significant efforts to build and deploy them to

different network devices on regular servers and to different types of disaggregated hardware devices. After deployment, configuring, monitoring, and managing them on all the endpoints is also hard. In contrast, developing, deploying, and managing network tasks in a disaggregated network pool with homogeneous devices is easy and flexible.

Independent scaling. It is easy to increase/decrease network hardware resources in a disaggregated network pool by adding/removing network devices in the pool. Without disaggregation, changing network resources would involve changing endpoints (*e.g.*, upgrading or adding NICs).

Access to large network resources. With disaggregation, an endpoint can potentially use the entire network pool's resources, far beyond what a single NIC or server can offer. This is especially useful when there are occasional huge traffic spikes or peak usages of many NTs. Without network disaggregation, the endpoint would need to install a large NIC/SmartNIC that is bloated with features and resources not commonly used [107, 61].

Beside the above benefits, a major benefit of network disaggregation is cost savings. A consolidated network pool only needs to collectively provision for the peak aggregated traffic and the maximum total NTs used by the whole rack at any single time. In contrast, today's non-disaggregated network systems require each endpoint to provision for its individual peak traffic and maximum NTs. To understand how significant this cost is in the real world, we analyze a set of traces from both traditional server-based production data centers and disaggregated clusters.

Server-based data center traffic analysis. To understand network behavior in server-based data centers, we analyze two sets of traces: a Facebook trace that consists of Web, Cache, and Hadoop workloads [271], and an Alibaba trace that hosts latency-critical and batch jobs together [130].

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

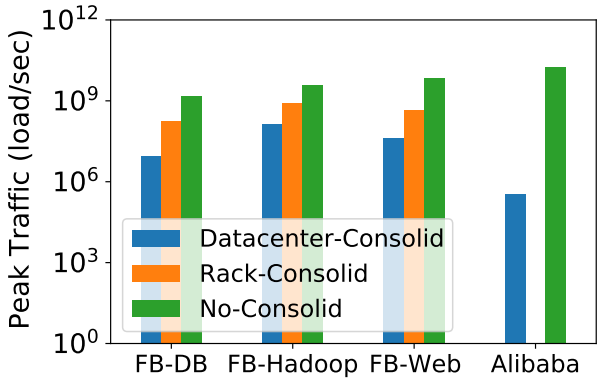


Figure 5.2: Consolidation Analysis of Facebook and Alibaba Traces. Load represent relative amount and have different units for FB and Alibaba.

and across the entire data center (peak of sum). These calculations model the cases of no disaggregation, disaggregation and consolidation at the rack level and at the data-center level. Figure 5.2 shows this result for the two data centers. For both of them, a rack-level consolidation consumes one magnitude fewer resources than no consolidation.

We then analyze the load spikes in these traces by comparing different endhosts' spikes and analyzing whether they spike at similar or different times, which will imply how much chance there is for efficient consolidation. Specifically, we count how much time in the entire 1-day trace X number of endhosts spike together. Figure 5.3 shows that 55% of the time only one or two servers spike together, and only 14% of the time four or more servers spike together. This result shows that servers mostly spike at different times, confirming the large potential of consolidation benefits.

Disaggregated cluster traffic analysis. Resource disaggregation introduces new types of network traffic that used to be within a server, *e.g.*, a CPU device accesses data in a remote memory device. If not handled properly, such traffic could add a huge burden to the data-center network [40]. To understand this type of traffic, we analyzed a set of disaggregated-memory network traces collected by Gao et al. using five endhosts [111]. Figure 5.4 plots the CDF and

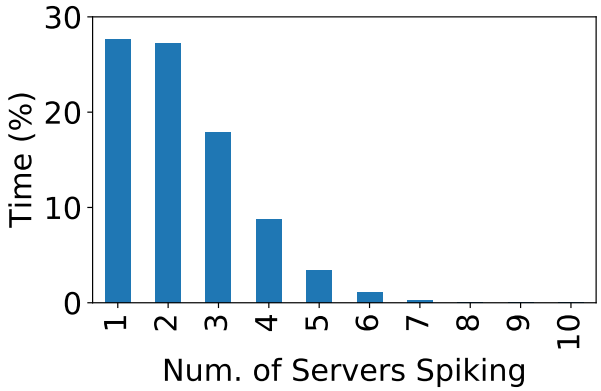


Figure 5.3: Load Spike Variation across Endhosts in FB.

the timeline of network traffic from four workloads. These workloads all exhibit fluctuating loads, with some having clear patterns of high and low load periods. We further perform a similar analysis using sum of peaks vs. peak of aggregated traffic as our server-based trace analysis. Consolidating just five endhosts already results in 1.1× to 2.4× savings with these traces.

5.2.2 Limitations of Alternative Solutions

The above analysis makes a case for disaggregating and consolidating network tasks from individual servers and devices. A question that follows is *where* to host these NTs and whether existing solutions could achieve the goals of network disaggregation and consolidation.

The first possibility is to host them at a **programmable ToR switch**. Programmable switches allow for configurable data planes, but they typically support only a small amount of computation at high line rates. SmartNICs, on the other hand, handle more stateful and complex computations but at lower rates. Transport protocol processing and encrypted communications are examples of complex network tasks better supported by a SmartNIC than a programmable switch. Moreover, existing programmable switches lack proper multi-tenancy and consolidation support [326]. As a consequence, most data center designs require the use of SmartNICs even in

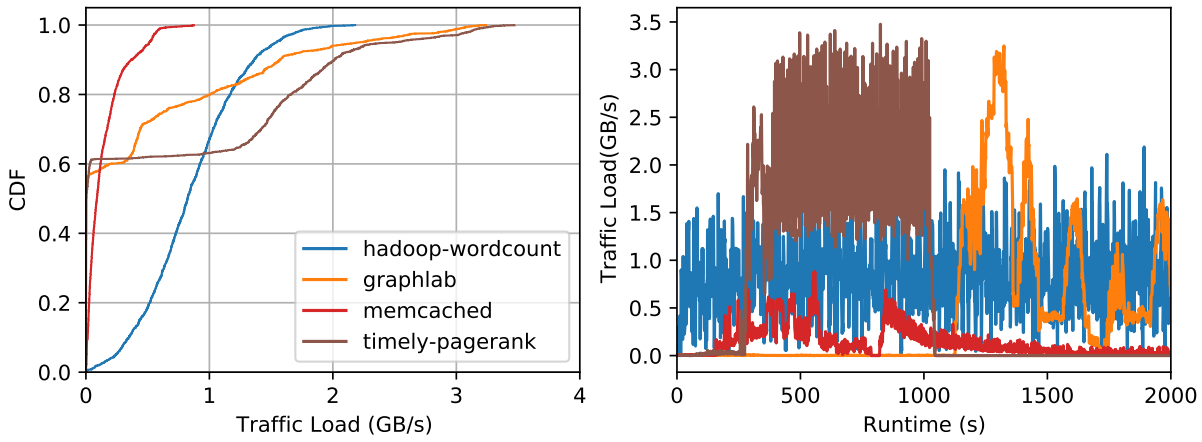


Figure 5.4: Network traffic for accessing disaggregated memory.

the presence of programmable switches, and our proposal simply disaggregates SmartNIC-like capabilities into a consolidated tier.

Another possibility is upcoming multi-host SmartNICs (e.g., Mellanox BlueField3) that are enhancements of today's **multi-host NICs** [1, 140]. These NICs connect to multiple servers via PCIe connections and provide general-purpose programmable cores. Our work identifies three key extensions to such devices. (1) Our approach enables a greater degree of aggregation as we enable coordinated management of a distributed pool of network devices. (2) Moreover, in the case of these multi-host SmartNICs, NTs that cannot be mapped to the NIC's fixed functions have to be offloaded as software. In contrast, sNIC allows the acceleration of NTs in hardware, enabling higher performance while tackling issues related to runtime reconfigurability of hardware. (3) Our approach provides adaptive mechanisms for adjusting to workloads and providing fair allocation of resources across applications. It is also worth noting that (1) and (3) can by themselves be used in conjunction with commercial multi-host SmartNICs to achieve a different software-based instantiation of our vision.

Middleboxes are a traditional way of running network functions inside the network either through hardware black-box devices that cannot be changed after deployment [289, 280, 323]

or through server-based Network Function Virtualization (NFV) that enables flexible software-based middleboxes [204, 245, 157, 301, 345], but at the cost of lower performance [246, 138]. Our deployment setting differs from traditional datacenter middleboxes: we target ”nearby” disaggregation, as in the endhost or SmartNIC tasks are disaggregated to a nearby entity typically located on the same rack. Consequently, our mechanisms are co-designed to take advantage of this locality (e.g., we use simple hardware mechanisms for flow control between the end-host and the disaggregated networking unit). Further, we target network functionality that is expected either at the endhost itself or at the edge of the network, such as endhost transport protocols, applying network virtualization, enhancing security, which all require nearby disaggregation and are also not typically targeted by middleboxes. We do note that our dynamic resource allocation mechanisms are inspired by related NFV techniques, but we apply them in the context of reconfigurable hardware devices.

Finally, there are emerging **interconnections designed for disaggregated devices** such as Gen-Z [115] and CXL [79]. These solutions mainly target the coherence problem where the same data is cached at different disaggregated devices. The transparent coherence these systems provide requires new hardware units at every device, in addition to a centralized manager. sNIC supports the disaggregation and consolidation of all types of network tasks and does not require specialized units at endpoints.

5.3 SuperNIC Overview

This section gives a high-level overview of the overall architecture of the sNIC platform and how to use it.

Overall Architectures. We support two ways of attaching an sNIC pool in a rack (Figure 5.1). In the first architecture, the sNIC pool is an intermediate layer between endpoints (servers or devices) and the ToR switch. Each sNIC uses one port to connect to the ToR switch.

Optionally, all the sNICs can be directly connected to each other, *e.g.*, with a ring topology. All remaining ports in the sNIC connect endpoints. We expect each of these endpoint-connecting links to have high bandwidth (*e.g.*, 100 Gbps) and the uplink to the switch to have the same or slightly higher bandwidth (*e.g.*, 100 Gbps or 200 Gbps). The second architecture attaches sNICs to the ToR switch, and endpoints directly attach to the ToR switch. In this architecture, the ToR switch re-directs incoming or outgoing traffic to one or more sNICs. Note that for both architectures, the actual implementation could either package the network pool with the ToR switch to form a new "switch box" or be separated out as an pluggable pool.

Requirements for endpoints and the last hop. For basic connectivity, an endpoint needs to have the equivalence of physical and link layers. For reliable transmission, the link layer needs to provide basic reliability functionality if the reliable transport is offloaded to sNIC. This is because packets could still be corrupted or dropped during the point-to-point transmission between an endpoint and its connected sNIC/switch (the last hop). Thus, the endpoint's link layer should be able to detect corrupted or dropped packets. It will either correct the corruption or treat it as a lost packet and retransmit it. The link layer also requires a simple flow control to slow down packet sending when the sNIC pool is overloaded or the application's fair share is exceeded.

Any interconnect fabric that meets the above requirements can be used as the last-hop link. PCIe is one such example, as it supports reliable data transfer and flow control. Our sNIC prototype uses Ethernet as it is more flexible. We use Priority Flow Control (PFC) for the one-hop flow control and add simple retransmission support. Unlike a traditional reliable link layer, our *point-to-point* reliable link layer is lightweight, as it only needs to maintain one logical flow and a small retransmission buffer for the small Bandwidth-Delay Product (BDP) of the last hop (64 KB in our prototype).

Using SuperNIC. To use the sNIC platform, users first write and deploy NTs. They specify which sNIC (sender side or receiver side) to deploy an NT. Users also specify whether an NT needs to access the packet payload and whether it needs to use on-board memory. For the

latter, we provide a virtual memory interface that gives each NT its own virtual address space. Optionally, users can specify which applications share the same NT(s). Currently, our FPGA prototype only supports NTs written on FPGA (deployed as netlists). Future implementation could extend sNICs to support p4 programs running on RMT pipelines [325] and generic software programs running on a processor.

After all the NTs that a user desires have been deployed, the user specifies one or multiple user-written or compiler-generated [188, 301] DAGs of the execution order of deployed NTs. Users could also add more DAGs at run time. Compared to existing works which let users specify an NF DAG when deploying NFs [245, 102, 188], we allow more flexible usages and sharing of deployed NTs. The sNIC stores user-specified DAGs in its memory and assigns a unique identifier (UID) to each DAG. At run time, each packet carries a UID, which sNIC uses to fetch the DAG.

5.4 SuperNIC Board Design

Traditional server SmartNICs have plenty of hardware resources when hosting network functions for applications running on the local server [107, 61]. In contrast, sNIC is anticipated to often be fully occupied or even over-committed, as it needs to host NTs from more tenants with limited hardware resources to save costs. Thus, a key and unique challenge in designing sNICs is space- and performance-efficient consolidation in a multi-tenant environment. Moreover, sNIC faces a more dynamic environment where not only the load of an application but also applications themselves could change from time to time. Thus, unlike traditional SmartNICs that focus on packet processing and packet scheduling, sNIC also needs to schedule NTs efficiently. This section first goes over the high-level architecture of sNIC, then discusses our mechanisms for efficient packet and NT scheduling, followed by the discussion of our scheduling and fairness policies, and ends with a description of sNIC's virtual memory system.

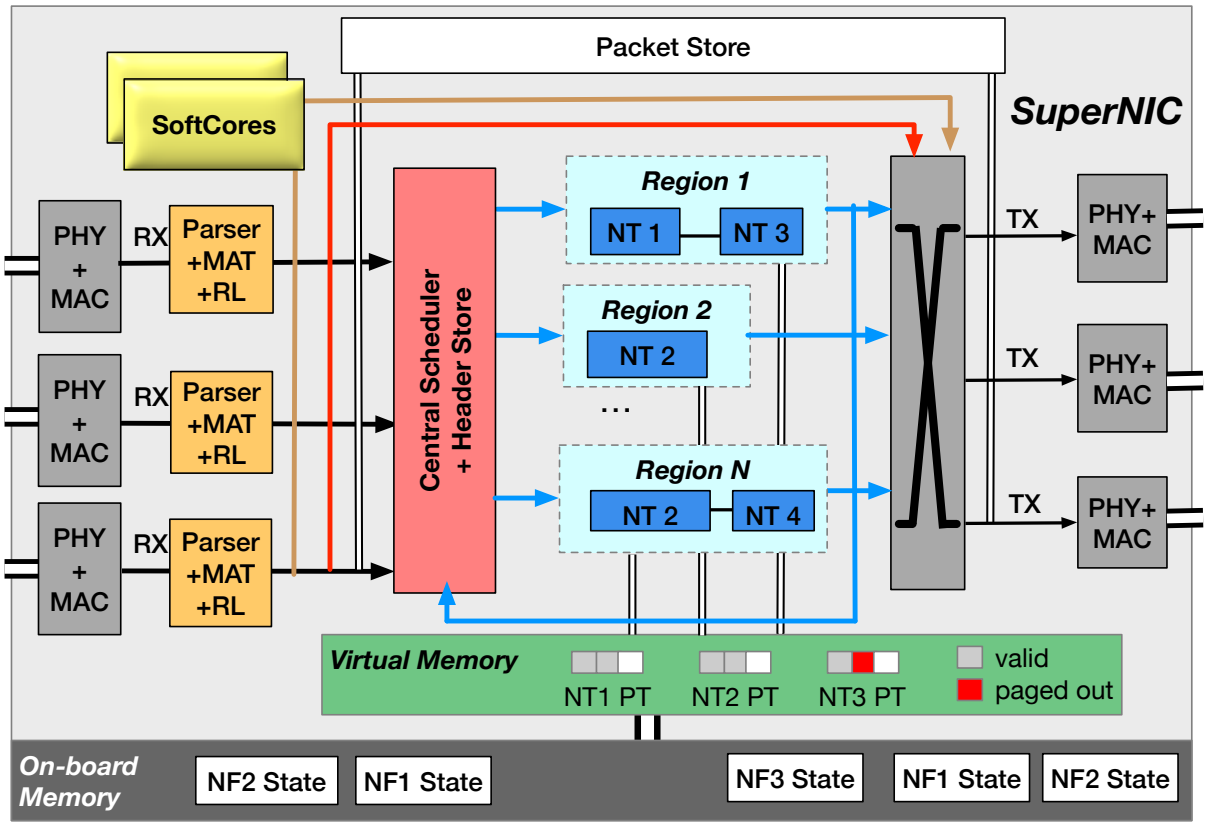


Figure 5.5: sNIC On-Board Design. RL: Rate Limiter. PT: Page Table

5.4.1 Board Architecture and Packet Flow

We design the sNIC board to simultaneously achieve several critical goals: **G1)** parsing/deparsing and scheduling packets at line rate; **G2)** high-throughput, low-latency execution of NT DAGs; **G3)** safe and fair sharing of all on-board resources; **G4)** quick adaptation to traffic load and workload changes; **G5)** good scalability to handle many concurrent workloads and NTs; **G6)** flexible configuration and adjustment of control-plane policies; and **G7)** efficient usage of on-board hardware resources. Figure 5.5 illustrates the high-level architecture of the sNIC board.

sNIC's data plane consists of reconfigurable hardware (*e.g.*, FPGA) for running user NTs (blue parts in Figure 5.5) and a small amount of non-reconfigurable hardware (ASIC) for non-user functionalities, similar to the "shell" or "OS" concept [13, 25, 163, 171]. We choose

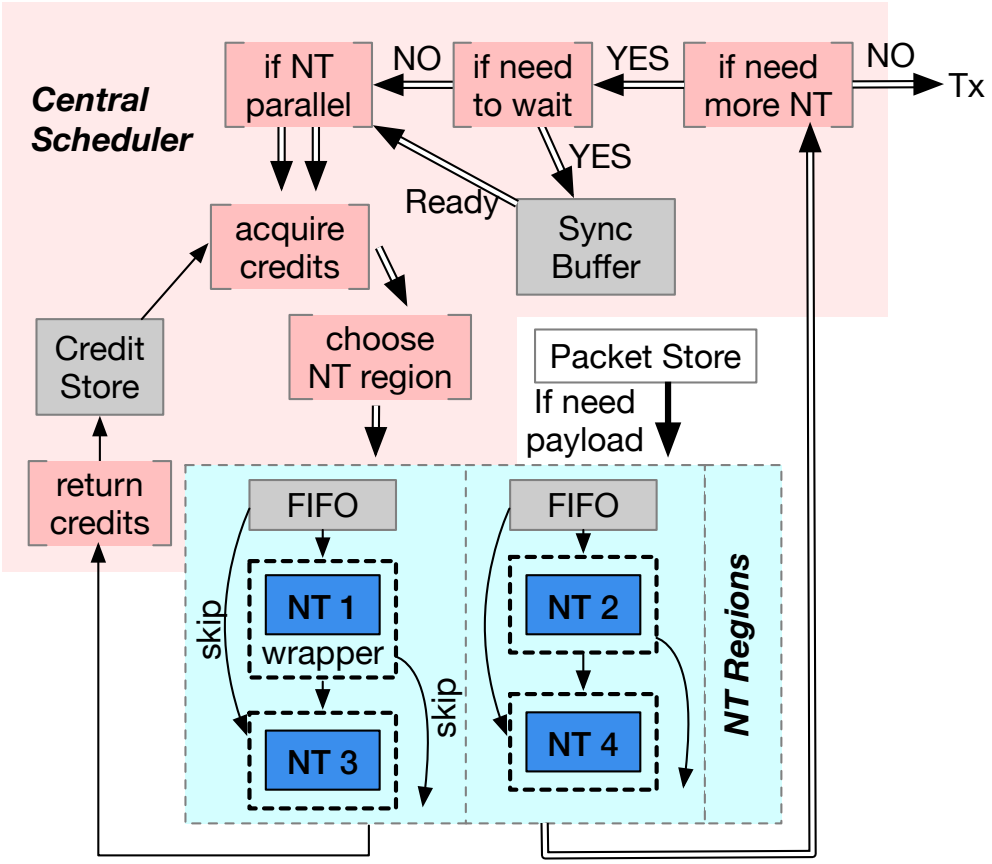


Figure 5.6: sNIC Packet Scheduler and NT Region Design. Double arrows, single arrows, and thick arrows represent packet headers, credits, and packet payload.

a hardware-based data-plane design because NTs like transports demand high-speed, parallel processing, and a fully reconfigurable hardware allows the maximum flexibility in NT hardware designs. Many of our design ideas can potentially be applied to other types of hardware and software NT implementations, such as PISA pipelines and ARM cores.

We divide the NT area into *region*s, each of which could be individually reprogrammed to run different NTs. Different NT regions can be executed in parallel.

The control plane runs as software on a small set of general-purpose cores (SoftCores for short) (*e.g.*, a small ARM-based SoC). To achieve the performance that the data plane requires and the flexibility that the control plane needs, we cleanly separate these two planes. The data plane handles all packet processing on ASIC and FPGA (**G1**). The control plane is responsible

for setting up policies and scheduling NTs and is handled by the SoftCores (**G6**). In our prototype, we built everything on FPGA.

When a packet arrives at an RX port, it goes through a standard physical and reliable link layer. Then our parser parses the packet's header and uses a Match-and-Action Table (MAT) to decide where to route the packet next. The parser also performs rate limiting for multi-tenancy fairness (§5.4.4). The parser creates a packet descriptor for each packet and attaches it to its header. The descriptor contains fields for storing metadata, such as an NT DAG UID and the address of the payload in the packet store. The SoftCores determine and install rules in the MAT, which include three cases for routing packets to the next step. First, if a packet specifies no NT information or is supposed to be handled by another sNIC (§5.5), the sNIC will only perform simple switching functionality and send it to the corresponding TX port (red line). Second, if a packet specifies the operation type `CTRL`, it will be routed to the SoftCores (orange line). These packets are for control tasks like adding or removing NTs, adding NT DAGs (§5.4.3), and control messages sent from other sNICs (§5.5).

Finally, all the remaining packets need to be processed on the sNIC, which is the common case. Their payloads are sent to the *packet store*, and their headers go to a central scheduler (black arrows). The scheduler determines when and which NT chain(s) will serve a packet and sends the packet to the corresponding region(s) for execution (blue arrows). If an NT needs the payload for processing, the payload is fetched from the packet store and sent to the NT. During the execution, an NT could access the on-board memory through a virtual memory interface, in addition to accessing on-chip memory. After an NT chain finishes, if there are more NTs to be executed, the packet is sent back to the scheduler to begin another round of scheduling and execution. When all NTs are done, the packet is sent to the corresponding TX port.

5.4.2 Packet Scheduling Mechanism

We now discuss the design of sNIC's packet scheduling mechanism. Figure 5.6 illustrates the overall flow of sNIC's packet scheduling and execution.

NT-chain-based FPGA architecture and scheduling. As sNIC enables more types of endpoints and workloads to offload their network tasks, the number of NTs and their execution flows will become even more complex, which could impact both the complexity of board design and the performance of packet scheduling. Our idea to confront these challenges is to execute as many NTs as possible in one go, by chaining NTs together. We put chained NTs (called an *NT chain*) in one NT region (*e.g.*, NT1→NT3 and NT2→NT4 in Figure 5.5). Instead of connecting each NT to a central scheduler (as what PANIC [193] does), we connect each region to the scheduler. Doing so allows us to use a much smaller crossbar between the NT regions and the central scheduler, thereby reducing hardware complexity and area cost (**G7**).

Furthermore, we leverage NT chains to reduce the scheduling overhead and improve the scalability of the central scheduler. Our idea is to *reserve* credits for an *entire* NT chain as much as possible and then execute the chain as a whole; only when that is not possible, we fall back to a mechanism that may involve the scheduler in the middle of a chain. Doing so reduces the need for a packet to go through the scheduler after every NT, thereby improving both the packet's processing latency and the central scheduler's scalability (**G5**).

On top of the fixed chain design, we propose an optimization to enable efficient NT time sharing across multiple users and to accommodate cases where some packets of an application only access a part of a chain (**G4**, **G6**). Our idea is to support the *skipping* of arbitrary NT(s) in a chain. For example, a user can access NT1 and NT4 by first skipping NT3 in Region-1 and then skipping NT2 in Region-2 in Figure 5.6.

Scheduling packets with NT-level and instance-level parallelism. At an sNIC, different regions run in parallel. We exploit two types of parallelism by controlling what NTs to put in parallel regions. The first type concurrently executes *different packets* at multiple instances of

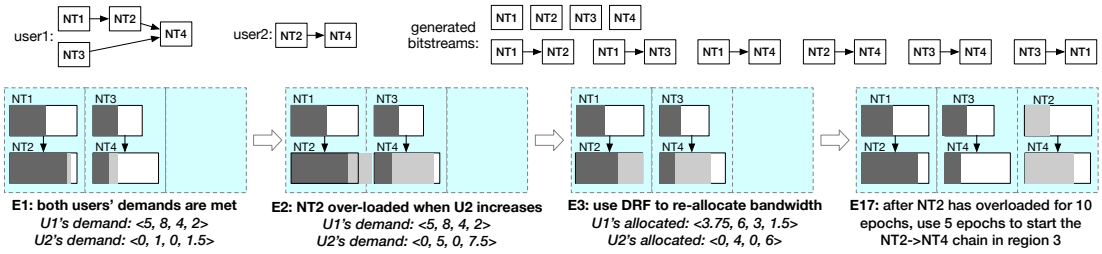


Figure 5.7: An Example of NT chaining and scheduling. Top: user1 and user2's NT DAGs and sNIC's generated bitstreams for them. Bottom: timeline of NT bandwidth allocation change. Dark grey and light grey represent user1 and user2's load. The launched chains are NT1→NT2 and NT3→NT4, with NT2 and NT4 being shared by the two users. The maximum throughput of NT1, NT2, and NT4 are 10 units each, and NT3's is 7 units. NT2 is the dominant resource for user1, and NT4 is the dominant for user2.

the *same NT chain* (what we call *instance-level parallelism*). We automatically create more/less instances of an NT chain based on load and send different packets in a round-robin way to the parallel instances. We will discuss our NT autoscaling policy in 5.4.4.

The second type concurrently executes the *same packet* at multiple *different NTs* (what we call *NT-level parallelism*). We infer what NTs can run in parallel in an NT DAG (*e.g.*, in Figure 5.7, NT1 and NT2 can run in parallel with NT3 for user1). We expect a fair amount of opportunities to explore NT-level parallelism, as previous work found that 53.8% NF pairs can run in parallel [301]. To execute a packet at several NTs concurrently, the scheduler makes copies of the packet header and sends them to these NTs concurrently. To obey the order of NTs that users specify, we maintain a *synchronization buffer* to store packet headers after they return from an NT's execution and before they could go to the next stage of NTs (Figure 5.6).

5.4.3 NT (De-)Launching Mechanism

sNIC's SoftCore handles NT deployment, launch, and scheduling tasks, as a part of the control path. A new challenge specific to sNIC comes from the need to do more frequent NT reconfigurations than traditional programmable network devices. To enable more consolidation,

we allow multiple NTs to *time share* an FPGA space, and we auto-scale NTs. Both these cases involve the slow FPGA PR process. We propose a set of new mechanisms and policies to reduce or hide the PR costs. We now discuss the mechanisms and defer the policy discussions to §5.4.4.

NT deployment. Users deploy NTs to the sNIC platform ahead of time as FPGA netlists (which can be thought of as Intermediate Representations in software). When receiving newly deployed NT netlists for an application, we first generate a set of FPGA bitstreams (which can be thought of as executable binaries in software). We enumerate all possible combinations of NTs under user-specified NT DAG ordering requirements when generating bitstreams. This is because bitstream generation is a slow process that usually takes a few hours or even longer. Generating more bitstreams at deployment time gives the sNIC more flexibility to choose different NT combinations at the run time. Figure 5.7 shows an example of generated bitstreams based on two DAGs of two users. We store pre-generated bitstreams in the sNIC's on-board memory; each bitstream is small, normally less than 5 MB.

When generating bitstreams, we attach a small sNIC wrapper to each NT (Figure 5.6). This wrapper is essential: it enables skipping an NT in a chain (§5.4.2), monitors the runtime load of the NT (§5.4.4), ensures signal integrity during PR, and provides a set of virtual interfaces for NTs to access other board resources like on-board memory (§5.4.5).

NT chain launching. We start a new NT chain when an application is deployed (pre-launch), when the chain is first accessed by a packet in an application (on-demand), or when we scale out an existing NT chain. For the first and third cases, we start the new NT only when there is a free region (see §5.4.4 for detail). For the on-demand launching case, when all regions are full, we still need to launch the new chain to be able to serve the application. In this case, we need to de-schedule a current NT chain to launch the new chain (see §5.4.4 for how we pick the region).

The sNIC SoftCore handles this context switching with a *stop-and-launch* process. Specifically, the SoftCore sends a signal to the current NTs to let them "stop". These NTs then store

their states in on-board memory to prepare for the stop. At the same time, the SoftCore informs the scheduler to stop accepting new packets. The scheduler will buffer packets received after this point. After the above *stop step*s finish, the SoftCore reads the new bitstream from the on-board memory via DMA and starts the FPGA PR process (*launch*). This process is the slowest step, as the maximum reported achievable PR throughput is around 800 MB/s [171], or about 5 *ms* for our default region size. Afterwards, the newly launched chain can start serving packets, and it will first serve previously buffered packets, if any.

To reduce the overhead caused by NT reconfiguration, we use a technique similar to the traditional victim cache design. We keep a de-scheduled NT chain in a region around for a while unless the region is needed to launch a new chain. If the de-scheduled NT chain is accessed again during this time, we can directly use it in that region, reducing the need to do PR at that time.

5.4.4 Packet and NT Scheduling Policy

We now discuss our packet and NT scheduling policies. Figure 5.7 shows an example of how an sNIC with three regions evolves as load changes.

Overall NT scheduling strategy. Our overall strategy is to avoid FPGA PR as much as possible and treat NT context switching (*i.e.*, replacing a current NT chain with a new one through FPGA PR) as a last resort, since context switching prevents the old NT from running altogether and could result in thrashing in the worst case.

For on-demand NT launching, we first check if the NT is the same as any existing NT on the sNIC. If so, and if the existing NT still has available bandwidth, we time share the existing NT with the new application. In this case, new traffic can be served immediately. Otherwise, we check if there is a free region. If so, we launch the NT at this region, and new traffic can be served after FPGA PR finishes. Otherwise, we reach out to the distributed sNIC platform and check if any other sNIC has the same NT with available bandwidth or a free region. If so, we route traffic to that sNIC (to be discussed in §5.5). Otherwise, we run the NT at the endpoint if users provide

the alternative way of executing it there. If all of the above fail, we resort to context switching by picking the region that is least loaded and using stop-and-launch to start the NT.

We also try to hide PR latency behind the performance-critical path as much as possible. Specifically, when a new application is deployed, we check if any of its NTs is missing on an sNIC. If there are any and if there are free regions on the sNIC for these NTs, we *pre-launch* them at the free regions, instead of launching them *on-demand* when the first packet accesses the NTs, as the latter would require waiting for the slow PR and slow down the first few packets.

NT auto-scaling. To adapt to load changes, sNIC automatically scales out/down instances of the same NT (instance-level parallelism) (**G2, G4**). Specifically, we use our per-NT monitoring module to identify bottleneck NTs and the load that they are expected to handle. If there are free regions, we add more instances of these NTs by performing PR on the free regions. When the load to an NT reduces to what can be handled with one instance less, we stop one of its instances and migrate the traffic of the stopped instance to other running instances. Since PR is slow, we should scale out/down an NT only if there is a persistent load increase/decrease instead of just occasional load spikes. To do so, we only scale out/down an NT if the past `MONITOR_PERIOD` time has overloaded/underloaded the NT. `MONITOR_PERIOD` should be at least longer than the PR latency to avoid thrashing. Since our measured PR latency is 5 *ms*, we set `MONITOR_PERIOD` to be 5 *ms* by default. After experimenting other length, we find this length to work the best with most real-world traffic [271, 38].

Scheduling with fairness. As we target a multi-tenant environment, sNIC needs to fairly allocate its resources to different applications (**G3**). Different from existing fairness solutions, we treat every NT as a separate type of resource, in addition to ingress bandwidth, egress bandwidth, packet store, and on-board memory space. This is because we support the time sharing of an NT, and different NTs can be shared by different sets of users. Our fairness policy follows Dominant Resource Fairness (DRF) [117], where we identify the *dominant* resource type for each application and seek a fair share for each application's dominant type. We also support weighted

DRF [117, 247] for users with different priorities.

Instead of a user-supplied static resource demand vector used in traditional DRF systems, we use *dynamically monitored resource demands* as the target in the DRF algorithm. Specifically, at each *epoch*, we use the ingress parser, egress de-parser, the central scheduler, and our virtual memory system to monitor the actual load demand before requests are dispatched to a particular type of resource. For example, for each user, the central scheduler measures the rate of packets that should be sent next to an NT before assigning credits; *i.e.*, even if there is no credit for the NT, we still capture the intended load it should handle. Based on the measured load at every type of resource for an application, we determine the dominant type of resource and use DRF to allocate resources after considering all applications' measured load vectors. At the end of every epoch, our DRF algorithm outputs a new vector of resource allocation for each application, which the next epoch will use. Compared to static resource demand vectors, our run-time monitoring and dynamic resource vectors can promptly adapt to load changes to maximize resource utilization.

Another novelty is in how we achieve the assigned allocation. Instead of throttling an application's packets at each NT and every type of resource to match the DRF allocation, we only control the application's ingress bandwidth allocation. Our observation is that since each NT's throughput for an application, its packet buffer space consumption, and egress bandwidth are all proportional to its ingress bandwidth, we could effectively control these allocations through the ingress bandwidth allocation. Doing so avoids the complexity of throttling management at every type of resource. Moreover, throttling traffic early on at the ingress ports helps reduce the load going to the central scheduler and the amount of payload going to the packet store. Our current implementation controls ingress bandwidth through rate limiting. Future work could also use other mechanisms like Weighted Fair Queuing. The only resource that is not proportional to ingress bandwidth is on-board memory space. We control it through our virtual memory system (§5.4.5).

Finally, the length of an epoch, `EPOCH_LEN`, is a configurable parameter. At every epoch,

we need to run the DRF algorithm and possibly change the bandwidth and memory allocation. Thus, EPOCH_LEN should be longer than the time taken to perform these operations (around 3 μs with our implementation). Meanwhile, it is desirable to set a short EPOCH_LEN to quickly adapt to load variations and to update rate allocations approximately once per average RTT [155, 92]. Thus, we set the default value of EPOCH_LEN to 20 μs.

5.4.5 Virtual Memory System

sNIC's allow NTs to use off-chip, on-board memory. To isolate different applications' memory spaces and to allow the over-subscription of physical memory space in an sNIC, we build a simple page-based virtual memory system. NTs access on-board memory via a virtual memory interface, where each NT has its own virtual address space. Our virtual memory system translates virtual memory addresses into physical ones and checks access permissions with a single-level page table. We use huge pages (2 MB size) to reduce the amount of on-chip memory to store the page table. Physical pages are allocated on demand; when a virtual page is first accessed, sNIC allocates a physical page from a free list.

We further support the over-subscription of an sNIC's on-board memory, *i.e.*, an sNIC can allocate more virtual memory space than its physical memory space. When the physical memory is full, adding more NT would require shrinking memory already assigned to existing applications (§5.4.3). In this case, we reduce already assigned memory spaces by migrating memory pages to a remote sNIC, *i.e.*, swapping out pages. To decide what pages to swap out, we first use the DRF algorithm to identify what NT(s) should shrink their memory space. Within such an NT, we pick the least recently accessed physical page to swap out. Our virtual memory system tracks virtual memory accesses to capture per-page access frequency. It also transparently swaps in a page when it is accessed. If no other sNIC has free memory space when the sNIC needs to grow its virtual memory space, we reject requests to add new NTs or to enlarge existing NT's memory.

5.5 Distributed SuperNIC

The design discussion so far focused on a single sNIC. To enable better consolidation and network as a service, we build a rack-scale distributed sNIC platform that enables one sNIC to use other sNICs' resources. With this platform, a rack's sNICs can collectively provision for the maximum aggregated load of all the endpoints in the rack.

As discussed in §5.3, we support two types of sNIC pool topology. For the switch-attached topology, the ToR switch serves as the load balancer across different sNICs. It also decides which sNIC to launch a new instance of an NT with the goal of balancing traffic and efficiently utilizing sNIC hardware resources. Supporting the intermediate-pool topology where the ToR switch cannot perform the above tasks is more complex. Below we discuss our design for it.

SoftCores on the sNICs in the intermediate pool form a distributed control plane. They communicate with each other to exchange metadata and cooperate in performing distributed tasks. We choose this peer-to-peer design instead of a centralized one, because the latter requires another global manager and adds complexity and cost to the rack architecture. Every sNIC collects its FPGA space, on-board memory, and port bandwidth consumption, and it periodically sends this information to all the other sNICs in the rack. Each sNIC thus has a global view of the rack and can redirect traffic to other sNICs if it is overloaded. To redirect traffic, the sNIC's SoftCore sets a rule in the parser MAT to forward certain packets (*e.g.*, ones to be processed by an NT chain on another sNIC) to the remote sNIC.

If an sNIC is overloaded and no other sNICs currently have the NT chain that needs to be launched, the sNIC tries to launch the chain at another sNIC. Specifically, the sNIC's SoftCore first identifies the set of sNICs in the same rack that have available resources to host the NT chain. Among them, it picks one that is closest in distance to it (*i.e.*, fewest hops). The sNIC's SoftCore then sends the bitstreams of the NT chain to this picked remote sNIC, which launches the chain in one of its own free regions. When the original sNIC has a free region, it moves back the migrated

NT chain. If the NT chain is stateful, then the SoftCore manages a state migration process after launching the NT chain locally, by first pausing new traffic, then migrating the NT's states (if any) from the remote sNIC to the local sNIC.

5.6 Case Studies

We now present two use cases of sNIC that we implemented, one for disaggregated memory and one for regular servers.

5.6.1 Disaggregated Key-Value Store

We first demonstrate the usage of sNIC in a disaggregated environment by adapting a recent open-source FPGA-based disaggregated memory device called *Clio* [347]. The original Clio device hosts standard physical and link layers, a Go-Back-N reliable transport, and a system that maps keys to physical addresses of the corresponding values. Clients running at regular servers send key-value load/store/delete requests to Clio devices over the network. When porting to sNIC, we do not change the client-side or Clio's core key-value mapping functionality.

Disaggregating transport. The Go-Back-N transport consumes a fair amount of on-chip resources (roughly the same amount as Clio's core key-value functionality [131]). We move the Go-Back-N stack from multiple Clio devices to an sNIC and consolidate them by handling the aggregated load. After moving the Go-Back-N stack, we extend each Clio device's link layer to a reliable one (§5.3).

Disaggregating KV-store-specific functionalities. A unique opportunity that sNIC offers is its centralized position when connecting a set of endpoints, which users could potentially use to more efficiently coordinate the endpoints. We explore this opportunity by building a replication service and a caching service as two NTs in the sNIC.

For **replication**, the client sends a replicated write request with a replication degree K,

which the sNIC handles by replicating the data and sending them to K Clio devices. In comparison, the original Clio client needs to send K copies of data to K Clio devices or send one copy to a primary device, which then sends copies to the secondary device(s). The former increases the bandwidth consumption at the client side, and the latter increases end-to-end latency.

For **caching**, the sNIC maintains recently written/read key-value pairs in a small buffer. It checks this cache on every read request. If there is a cache hit, the sNIC directly returns the value to the client, avoiding the cost of accessing Clio devices. Our current implementation that uses simple FIFO replacement already yields good results. Future improvements like LRU could perform even better.

5.6.2 Virtual Private Cloud

Cloud vendors offer Virtual Private Cloud (VPC) for customers to have an isolated network environment where their traffic is not affected by others and where they can deploy their own network functions such as firewall, network address translation (NAT), and encryption. Today's VPC functionalities are implemented either in software [80, 255, 316] or offloaded to specialized hardware at the server [106, 107, 27]. As cloud workloads experience dynamic loads and do not always use all the network functions (§5.2), VPC functionalities are a good fit for offloading to sNIC. Our baseline here is regular servers running Open vSwitch (OVS) with three NFs, firewall, NAT, and AES encryption/decryption. We connect sNICs to both sender and receiver servers and then offload these three NFs to each sNIC as one NT chain.

5.7 Evaluation Results

We implemented sNIC on the HiTech Global HTG-9200 board [3]. Each board has nine 100 Gbps ports, 10 GB on-board memory, and a Xilinx VU9P chip with 2,586K LUTs and 43 MB BRAM. We implemented most of sNIC's data path in SpinalHDL [297] and sNIC's control

Table 5.1: SuperNIC FPGA Utilization. Shell cost in an FPGA. Most resources left for running NTs.

Module	Logic (LUT)	Memory (BRAM)
sNIC Core	4.36%	4.74%
Packet Store	0.91%	9.17%
PHY+MAC	0.72%	0.35%
DDR4Controller	1.57%	0.29%
MicroBlaze	0.25%	1.81%
Misc	1.52%	0.75%
Total	**9.33%**	**17.11%**

path in C (running in a MicroBlaze SoftCore [4] on the FPGA). Most data path modules run at 250 MHz. In total, sNIC consists of 8.5K SLOC (excluding any NT code). The core sNIC modules consume less than 5% resources of the Xilinx VU9P chip, leaving most of it for NTs (see Appendix for full report). To put it in perspective, the Go-back-N reliable transport we implement consumes 1.3% LUTs and 0.4% BRAM.

Environment. We perform both cycle-accurate simulation (with Verilator [8]) and real end-to-end deployment. Our deployment testbed is a rack with a 32-port 100 Gbps Ethernet switch, two HTG-9200 boards acting as two sNICs, eight Dell PowerEdge R740 servers, each equipped with a Xeon Gold 5128 CPU and an NVidia 100 Gbps ConnectX-4 NIC, and two Xilinx 10 Gbps ZCU106 boards running as Clio [131] disaggregated memory devices. Each sNIC uses one port to connect to the ToR switch and one port to connect to the other sNIC. Depending on different evaluation settings, an sNIC's downlinks connect to two servers or two Clio devices.

5.7.1 Overall Performance and Costs

CapEx cost saving. We compare the CapEx cost of sNIC's two architectures with no network disaggregation, traditional multi-host NIC, and traditional NIC-enhanced switches. All calculations are based on a single rack and include 1) endpoint NICs, 2) cables between endpoints, sNICs, and the ToR switch, 3) the ToR switch, and 4) cost of sNICs or multi-host

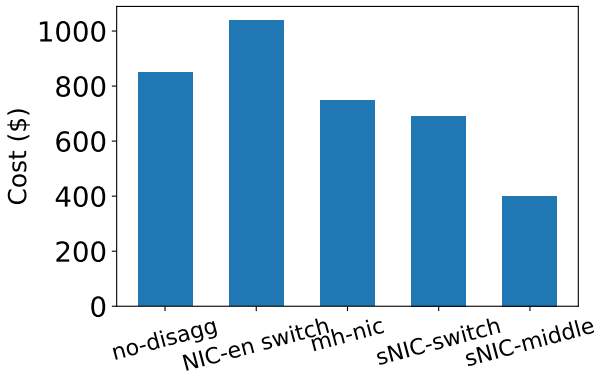


Figure 5.8: Per-Endpoint CapEx. A rack's network cost divided by endpoint count.

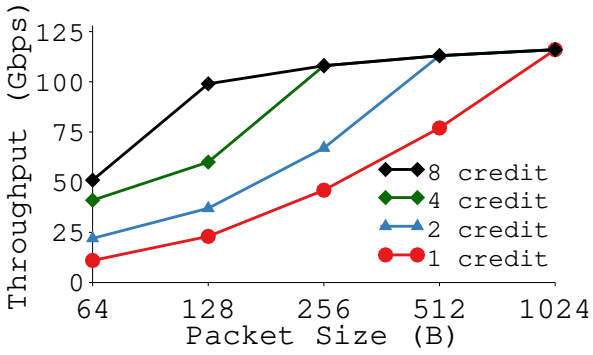


Figure 5.9: Throughput with different credits.

NICs. We use market price when calculating the price of regular endpoint NICs and cables. With sNIC, the endpoint NICs can be shrunken down to physical and link layers (based on our prototype estimation, it is roughly 20% of the original NIC cost of $500), and the cables connecting endpoints and sNICs in the middle-layer-pool architecture can be shortened (roughly 60% of the original cable cost of $100 [351]). We estimate the ToR-switch cost based on the number of ports it needs to support and a constant per-port cost of $250 [110].

To estimate the cost of an sNIC, we separate the non-NT parts and the NT regions. The former has a constant hardware cost, while the latter can be provisioned to the peak of aggregated traffic and NT usages, both of which are workload dependent. We use the peak-of-sum to

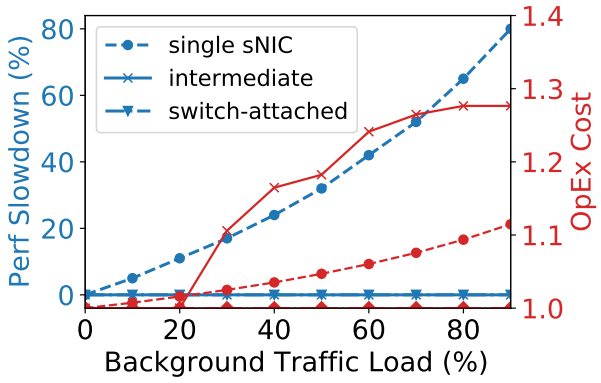


Figure 5.10: Distributed sNIC.

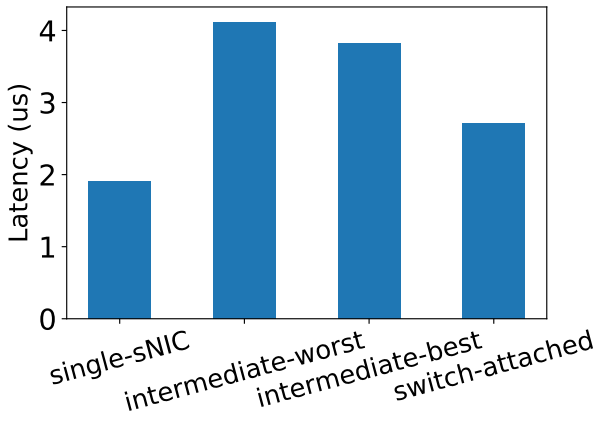


Figure 5.11: Topology Comparison.

sum-of-peak ratios obtained from the Facebook traces (§5.2.1). Today's multi-host NIC and NIC-enhanced switches do not consolidate traffic, and we assume that they will be provisioned for the sum-of-peak. See Appendix for detailed calculation.

Figure 5.8 plots the per-endpoint dollar CapEx cost. Overall, sNIC achieves **52% and 18% CapEx savings** for the middle-layer and switch-attached pool architecture compared to no disaggregation. Multi-host NIC and NIC-enhanced switches both have higher CapEx costs than sNIC, because of its high provisioning without auto-scaling. The NIC-enhanced switches are

even more costly than traditional racks mainly because of the added switch ports and NICs.

OpEx saving and single-sNIC performance. We compare a single sNIC connecting four endpoints with the baseline of no disaggregation when these endpoints each run its NTs on its own SmartNIC. We generate workloads for each endpoint based on the Facebook memcached dataset distributions [38]. For the per-endpoint SmartNIC, we statically allocate the hardware resources that can cover the peak load. Overall, we found that sNIC achieves **56% OpEx saving**, because sNIC dynamically scales the right amount of hardware resources for the aggregated load. sNIC only adds **only 4% performance overhead** over the optimal performance that the baseline gets with maximum allocation.

We then test the throughput a real sNIC board can achieve with a dummy NT. These packets go through every functional module of the sNIC, including the central scheduler and the packet store. We change the number of initial credits and packet size to evaluate their effect on throughput, as shown in Figure 5.9. These results demonstrate that our FPGA prototype of sNIC could reach more than 100 Gbps throughput. With higher frequency, future ASIC implementation could reach even higher throughput.

Next, we evaluate the latency overhead a real sNIC board adds. It takes 1.3 μs for a packet to go through the entire sNIC data path. Most of the latency is introduced by the third-party PHY and MAC modules, which could potentially be improved with real ASIC implementation and/or a PCIe link. The sNIC core only takes 196 ns. Our scheduler achieves a small, fixed delay of 16 cycles, or 64 ns with the FPGA frequency. To put things into perspective, commodity switch's latency is around 0.8 to 1 μs.

Distributed sNICs. To understand the effect of distributed sNIC pool, we compare the two pool topology with a single sNIC (no distributed support). Figure 5.10 shows the performance and OpEx cost. Here, we use the workload generated from the Facebook distribution as the foreground traffic and vary the load of the background traffic. As background load increases, a single sNIC gets more over-committed and its performance becomes worse. With distributed

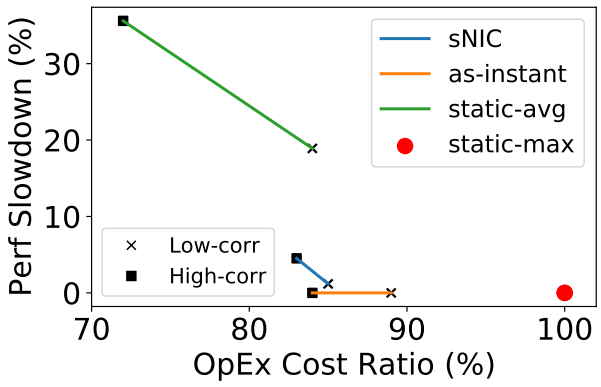


Figure 5.12: Performance and OpEx Overview. Lower is better for both axis.

sNIC, we use an alternative sNIC to handle the load, thus not impacting the foreground workload's performance. Note that the background workload's performance is not impacted either, as long as the entire sNIC pool can handle both workloads' traffic. Furthermore, these workloads are throughput oriented, and we achieve max throughput with distributed sNICs. As for OpEx, the intermediate-pool topology uses one sNIC to redirect traffic to the other sNIC. As the passthrough traffic also consumes energy, its total OpEx increases as the total traffic increases. The switch-attached topology does not have the redirecting overhead. The single sNIC also sees a higher OpEx as load increases because the workload runs longer and consumers more energy.

We then compare the two topologies of sNIC pool. Figure 5.11 shows the latency comparison. The intermediate-pool topology where we connect the sNICs using a ring has a range of latency depending on whether the traffic only goes through the immediately connected sNIC (single-sNIC) or needs to be redirected to another sNIC when the immediate sNIC is overloaded. Because of the ring connection, this other sNIC's distance to the immediately connected sNIC determines the additional latency incurred (intermediate-best and worst). In contrast, the switch-attached topology has a constant latency, even when one or multiple sNICs are overloaded. This is because the traffic always goes through the switch which directs it to the right sNIC.

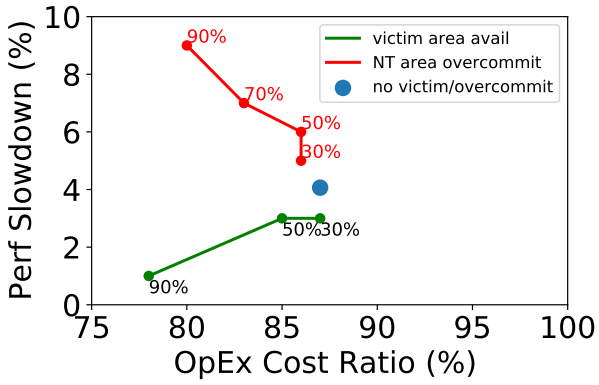


Figure 5.13: Single sNIC Sensitivity. Each lines is running a different set of experiments.

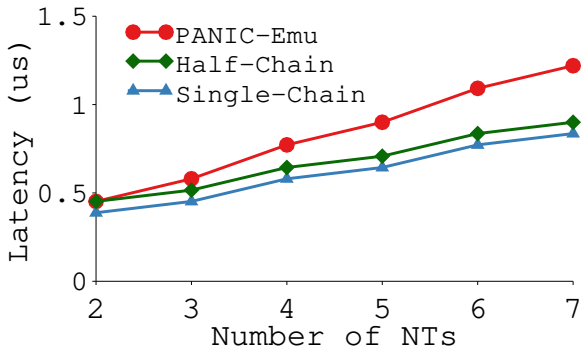


Figure 5.14: NT Chain.

5.7.2 Deep Dive into sNIC Designs

We now perform a set of experiments to understand the implications of sNIC's various designs in terms of performance and OpEx cost, also with the Facebook distribution.

Effect of auto-scaling. We compare our auto-scaled implementation of sNIC with two types of static allocations (*i.e.*, no load-based scaling): allocating for the highest load needs (*static-max*) and allocating for the average load needs (*static-avg*), and an unrealistic auto-scaled scheme which instantly scales the right amount of NT instances as load changes and incurs zero context switching overhead (*as-instant*). We generate two workloads using the Facebook

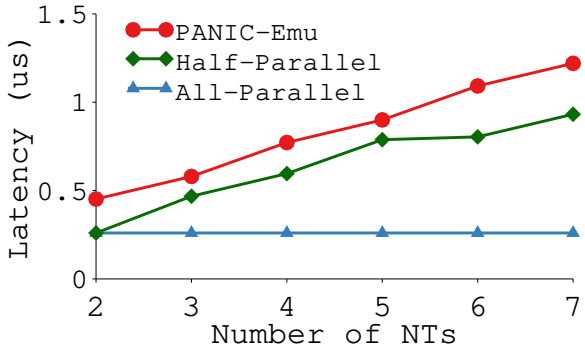


Figure 5.15: NT Parallelism.

distributions, one where different endpoints spike at similar time (high correlation) and one where they spike at different times (low correlation). Figure 5.12 shows the performance slowdown (compared to no network disaggregation) and OpEx costs (compared to static-max).

sNIC is at the Pareto frontier compared to the two static allocation schemes. Static-max has the best performance but the worst OpEx cost, as it pays for the peak hardware resources for the entire duration. In contrast, static-avg has the worst performance but best OpEx cost, since it only allocates the resource for the average usage for the entire duration. Compared to sNIC, as-instant only achieves slightly better performance with slightly more OpEx spending, as it tightly matches resources to the load which is unrealistic.

Comparing the two workloads, the low-correlation one always has better performance and more OpEx spending than the high-correlation one (except for as-instant which always yields best performance and static-max which always yields best performance and worst OpEx). This is because with low correlation, the aggregated traffic is more flattened out, which gives sNIC better chance to properly handle. As a result, sNIC scales the right amount of resources to satisfy the load's performance needs. When correlation is high (which is unlikely from our trace analysis in §5.2.1), there will be fewer but more intensive spikes. When sNIC is not fast or powerful enough to handle some of them, less resources is used but the performance goes down.

Effect of victim cache. To evaluate the effect of our victim-cache design, we set the baseline to be disabling victim cache (blue dot in Figure 5.13). We then change how often a de-scheduled NT can be kept around as a victim instead of being completely deleted (shown as percentage on the green line). This configuration models how often an sNIC's area is free to host victim NTs. As expected, the more de-scheduled NTs we can keep around, the better performance we achieve, with no victim cache (baseline) having the worst performance. The OpEx implication is less intuitive. Here, we only count the time and amount of NT regions that are actually accessed, as only those will cause the dynamic power (when idle, FPGA has a static power consumption regardless of how it is programmed). With fewer de-scheduled NTs kept around, more NTs need to be re-launched (through FPGA PR) when the workload demands them. These re-launching overhead causes the OpEx to also go up.

Effect of area over-commitment. We change the degree of area over-commitment by limiting how much hardware resources (*i.e.*, NT regions) the workload can use compared to the ideal amount of resources needed to fully execute it. As Figure 5.13 shows, as we increase the area over-commitment rate, we see worse performance and less resources (OpEx) used. Thus, our design uses distributed sNICs to avoid the over-commitment of a single sNIC.

NT chaining. To evaluate the effect of sNIC's NT-chaining technique, we change the length of NT sequence from 2 to 7 (as prior work found real NFs are usually less than 7 in sequence [301]). In comparison, we implemented PANIC's scheduling mechanism on our platform, so that everything else is the same as sNIC. We also evaluate the case where sNIC splits the chain into two sub-chains. Figure 5.14 shows the total latency of running the NT sequence with these schemes. sNIC outperforms PANIC because it avoids going through the scheduler during the sequence for single-chain and only goes through the scheduler once for half-chain.

NT-level parallelism. We then evaluate the effect of sNIC's NT-level parallelism by increasing the number of NTs that could run in parallel. We compare with PANIC (on our platform), which does not support NT-level parallelism. We also show a case where we split NTs

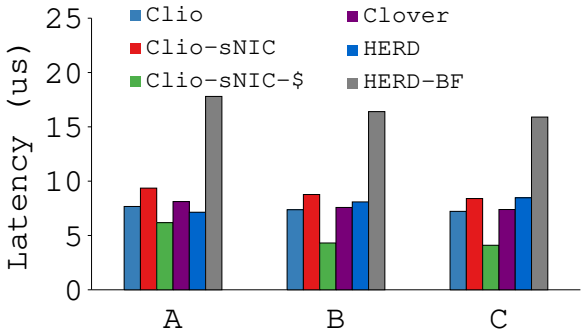


Figure 5.16: YCSB Latency.

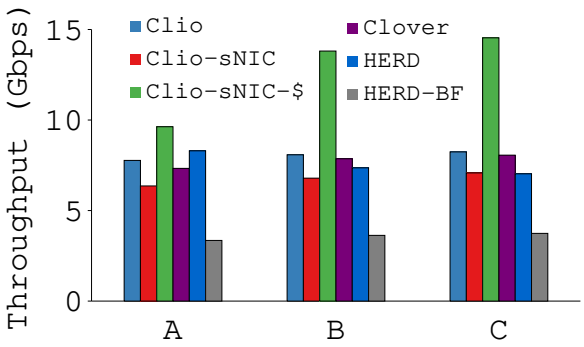


Figure 5.17: YCSB Throughput.

into two groups and run these groups as two parallel NT-chains (half-parallel). Figure 5.15 shows the total latency of these schemes. As expected, running all NTs in parallel achieves the best performance. The tradeoff is more NT region consumption. Half-parallel only uses two regions and still outperforms the baseline.

DRF Fairness. To evaluate the effectiveness of our scheduling policy, we ran the synthetic workloads as described in Figure 5.7 and use the default `EPOCH_LEN` of 20 μs and `MONITOR_PERIOD` of 10 ms. Figure 5.20 shows the resulting throughput timeline for different NTs of the two users. In between epoch 1 and 2, the loads of user2 increased to the second step. At the next epoch, we run DRF and adjust the allocation. After the DRF algorithm finishes (in

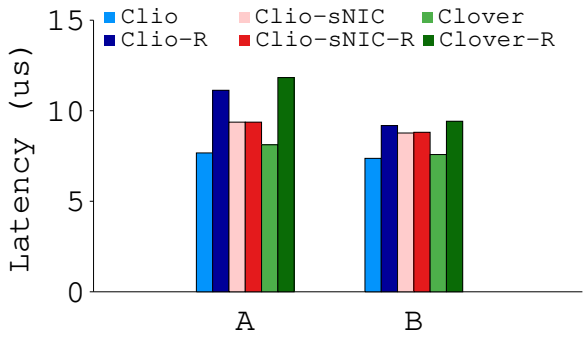


Figure 5.18: Replicated YCSB.

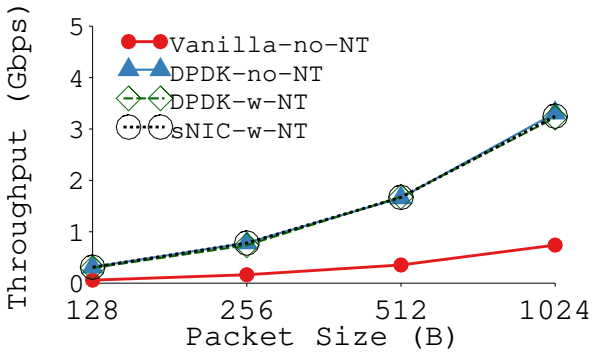


Figure 5.19: VPC Performance.

around 3 μs), user2 gets a higher (and fairer) allocation of NT2 and NT4, while user1's allocation decreases. After observing NT2 being overloaded for 10 ms, the sNIC scales out NT2 by adding one more instance of it at time epoch-503. After PR is done (in 5 ms), both user1 and user2's throughput increase.

5.7.3 End-to-End Application Performance

We now present our end-to-end application performance conducted on our rack testbed.

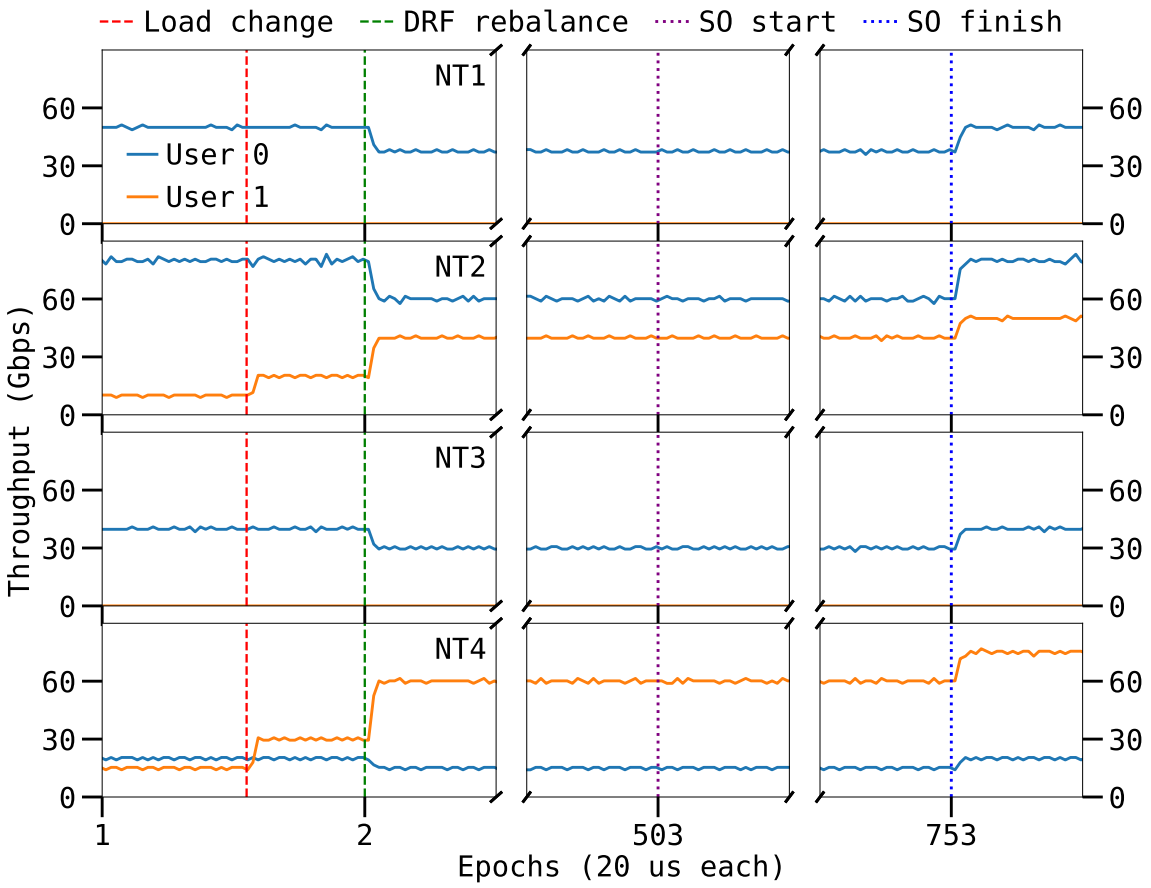


Figure 5.20: NT Scheduling. SO: Scaling Out.

Disaggregated Key-Value Store

In this set of experiments, we use one client server and two Clio devices. The Clio devices connect to one sNIC which connects to the ToR switch. We run YCSB's workloads A (50% set, 50% get), B (5% set, 95% get), and C (100% get) [75] for these experiments. We use 100K key-value entries and run 100K operations per test, with YCSB's default key-value size of 1 KB and Zipf accesses ($\theta = 0.99$).

Non-replicated YCSB performance and caching. We first evaluate the performance of running YCSB without replication using one client server and one Clio memory device.

Figure 5.16 and 5.17 plot the average end-to-end latency and throughput of running the YCSB workloads with (1) the original Clio, (2) Clio's Go-Back-N transport offloaded to sNIC (Clio-sNIC), (3) adding caching on top of Clio-sNIC (Clio-sNIC-$), (4) Clover [314], a *passive* disaggregated memory system where all processing happens at the client side and a global metadata server, (5) HERD [154], a two-sided RDMA system where both the client and memory sides are regular servers, and (6) HERD running on the NVidia BlueField SmartNIC [5] (HERD-BF). sNIC's performance is on par with Clio, Clover, and HERD, as it only adds a small overhead to the baseline Clio. With caching NT, sNIC achieves the best performance among all systems, esp. on throughput. This is because all links in our testbed are 100 Gbps except for the link to the 10 Gbps Clio boards. When there is a cache hit at the sNIC, we avoid going to the 10 Gbps Clio boards. HERD-BF performs the worst because of the slow link between its NIC and the ARM processor.

Replicated YCSB performance. We then test Clio, Clover, and Clio with sNIC with replicated write to two Clio devices. HERD does not support replication, and we do not include it here. Clover performs replicated write in a similar way as the baseline Clio, but with a more complex protocol. Figure 5.18 plots the average end-to-end latency with and without replicated writes using the YCSB A and B workloads. With sNIC's replication NT, the overhead that replication adds is negligible, while both Clio and Clover incur significant overheads when performing replication.

Virtual Private Cloud

We use one sender server and one receiver server, both running Open vSwitch (OVS) [255], to evaluate VPC. Our baseline is the default Open vSwitch that runs firewall, NAT, and AES. We further improve the baseline by running DPDK to bypass the kernel. In the sNIC setup, we connect the sender to an sNIC and the receiver to another sNIC. Each sNIC runs the three NFs as a chain. Figure 5.19 shows the throughput results. Overall, we find OVS to be a major performance

bottleneck in all the settings. Using DPDK improves OVS performance to some extent. Compare to running NTs at servers, offloading them to the sNIC improves throughput, but is still bounded by the OVS running at the endhosts.

5.8 Conclusion

We propose network disaggregation and consolidation by building SuperNIC, a new networking device specifically for a disaggregated datacenter. Our FPGA prototype demonstrates the performance and cost benefits of sNIC. Our experience also reveals many new challenges in a new networking design space that could guide future researchers.

5.9 Acknowledgments

Chapter 5, in part, has been submitted for publication of the material as it may appear in Yizhou Shan, Will Lin, Ryan Kosta, Arvind Krishnamurthy, Yiying Zhang, "Disaggregating and Consolidating Network Functionalities with SuperNIC", *arXiv, 2022*. The dissertation author was the primary investigator and author of this paper.

Chapter 6

Conclusion and Future Directions

The data center resource management problem is exacerbated by the fast-changing application demand and heterogeneous domain-specific accelerators. Innovations in the data center is hindered by the traditional monolithic servers. Hardware resource disaggregation is a proposal that breaks monolithic servers into independent network-attached resource pools, which promises to offer better manageability, high resource utilization, and better failure isolation. Despite the high hopes for hardware resource disaggregation, it was not clear how it could be deployed or whether it could deliver reasonable performance.

In this dissertation, we developed various systems transforming hardware resource disaggregation from a vague research proposal into one that is tangible, practical, deployable, and quantitatively shown to be beneficial. Specifically, we built a distributed persistent memory management system, a new operating system, two FPGA-based disaggregated hardware devices, and several companion distributed systems. As a result, we demonstrated the feasibility of resource disaggregation, presented several critical techniques for improving performance, and confirmed its advantages in providing better resource packing, failure isolation, and resource elasticity.

We made our initial effort building Hotpot, a logical disaggregation system for persistent memory that unifies distributed share memory and distributed storage system in one single layer.

However, we found ourselves still hindered by the inherent limitation of monolithic servers. This first trial reaffirmed the need to break and go beyond the traditional server boundary, i.e., physically disaggregate hardware resources. We then built LegoOS, the first operating system designed for managing various disaggregated resources at data center scale. We found that a complete disaggregation model in which compute and memory is completely separated is detrimental to performance. Having some local memory at the compute side can greatly improve the end-to-end performance without losing disaggregation's benefits. To avoid the emulation overheads found in LegoOS, we then built Clio, a low-cost, scalable, and flexible FPGA-based disaggregated memory system. Clio is the first hardware device specifically designed for memory disaggregation. In Clio, we co-designed networking transport, virtual memory system, and hardware. Clio overcomes the scalability bottleneck and any emulation overheads found in other related works. Finally, we tackled another data center's major resource, network, and proposed the concept of network disaggregation. We disaggregate network tasks from endpoints and consolidate them into a network resource pool, which provides network as a service to the connected endpoints. Our network resource pool consists of a distributed control plane with efficient, fair, and safe resource sharing, and SuperNIC, a new hardware-based programmable network device that consolidates network functionalities from multiple endpoints by fairly sharing limited hardware resources. We demonstrated SuperNIC's performance and cost benefits with real network functions and customized disaggregated applications.

6.1 Future Directions

We will end this dissertation with a few open questions and possible future directions raised during our work.

6.1.1 Fully Programmable Hardware Disaggregation

Data centers are becoming more intelligent than ever. Over the past several decades, researchers have proposed various schemes to offload computation to certain mediums so as to avoid data movement, achieve better coordination, and improve overall performance. Examples include but are not limited to active memory [343], active storage [268, 281], active pages [238], active storage/flash [311], active messages [321], active network [81], Processing-In-Memory [221], Near-Memory-Computing [292], and so on.

We make two key observations.

Our first observation is that the aforementioned offloading methods are in fact special cases under the hardware resource disaggregation scheme. Hardware resource disaggregation decomposes servers into independent network-attached devices. This dissertation repeatedly shows that adding the right amount of computation power to such devices can greatly improve end-to-end performance without losing disaggregation's cost and management benefits. For instance, one can use a disaggregated memory device to realize active memory.

Our second observation is that the emerging in-network computing is in fact also a special case under the hardware resource disaggregation scheme. Recently emerged in-network computing adds data path runtime programmability to both switches and end-host NICs. Along with an already fully distributed and programmable control path [103], the current data center network is close to an ideal active network setup in which one can flexibly offload, migrate, and balance application computation. The SuperNIC work further shows that we could view network as another dimension in the hardware resource spectrum, which is physically and logically separated from other types of resources.

Our key insight is that hardware resource disaggregation is a solution that unifies in-network computing and various offloading schemes, providing a fully programmable hardware basis for upper layer software. Under this scheme, all types of resources (e.g., compute, memory, storage, network) could operate on their own, each with a certain degree of programmability and

able to run user or vendor computation.

The remaining key question is how to best utilize this fully programmable disaggregated data center. More specifically, one should decide which computation to offload, where to offload onto, and when it should be migrated to other devices.

6.1.2 Selective, Dynamic Disaggregation and Programming Model

The above section projects a fully programmable hardware disaggregated data center in which all type of resources including network are programmable and independent from others. In this envisioned model, application developers and data center operators will have access to many types of resources sitting in different locations. Ideally, they could make fine-grained choices to find the best matching device for their computation. Since manual offloading and placement often times lead to worse performance and cost, prior work in this space has proposed dynamic offloading solutions that uses programming language framework or application hints. However, they either target a single SmartNIC [256, 259, 195], or a single middlebox and programmable switch [342, 63]. None of them is able to handle the complexity introduced by the fully programmable hardware disaggregation model.

We envision the following techniques and deem them as beneficial to work on top of a fully programmable disaggregated data center. First of all, we envision *Selective Disaggregation*, a static compile time mechanism or framework that once given an application as an input, would be able to divide the application into standalone codelets, and then decides the best matching hardware for each codelet. The framework would then automatically compile and deploy these codelets to corresponding hardware devices. Second, we envision *Dynamic Disaggregation*, a dynamic runtime system that could monitor the deployed codelets, and dynamically decide whether codelets should be migrated to another device with the same type or to a new type of device. Third, we envision a *Programmable Model* to facilitate the selective and dynamic disaggregation. At compile time, it should help selective disaggregation to break down program

into codelets, it should also help produce binaries (or bitstreams, objects) for each different hardware devices. We leave the details for future work.

6.1.3 Security

While we enjoy the convenience and benefits of cloud computing, the need for security and privacy is stronger. It is still an open question on how to ensure proper security or deliver confidential computing in a hardware disaggregated data center.

At first glance, both hardware and software operate in a more decentralized, distributed manner compared to the monolithic server model. Hence the first step in ensuring basic security is equipping disaggregated devices with authorization and authentication mechanisms to detect any forged requests, and encrypting all intra-cluster network traffic. For instance, augmenting network-attached FPGAs with access control modules [272], or hardening existing RDMA NICs [270, 294, 305]. Furthermore, each disaggregated device should have proper multi-tenancy mechanisms to isolate resources and avoid side channels. For instance, in Clio and SuperNIC, we added basic virtual memory subsystems to ensure memory safety and rely on compilation time design rule check to avoid side channels.

Delivering end-to-end confidential computing is a much more challenging task. Existing confidential computing solutions such as Intel SGX or ARM TrustZone target CPU-only environments, leaving disaggregated hardware devices unprotected. To realize confidential computing in a fully programmable disaggregated data center, we envision a two-step approach. First, we should add basic confidential solution to the disaggregated devices. For example, adding a trusted platform module to FPGA [266], GPU [137], and programmable switch. Second, we need a distributed security control system working with the aforementioned dynamic disaggregation system to orchestrate computation and data movement, ultimately ensuring no information leakage.

Bibliography

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

[2] Filesystem in Userspace. http://libfuse.github.io/.

[3] HTG-9200: Xilinx Virtex UltraScale+™ Optical Networking Development Platform. http://www.hitechglobal.com/Boards/UltraScale+_X9QSFP28.htm.

[4] MicroBlaze. https://en.wikipedia.org/wiki/MicroBlaze.

[5] NVIDIA BLUEFIELD DATA PROCESSING UNITS. https://www.nvidia.com/en-us/networking/products/data-processing-unit/.

[6] Open Compute Project (OCP). http://www.opencompute.org/.

[7] The CIFAR-10 dataset. https://www.cs.toronto.edu/~kriz/cifar.html.

[8] Verilator, the fastest verilog/systemverilog simulator.

[9] Wikipedia dump. https://dumps.wikimedia.org/.

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

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

[12] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*.

[13] Adrian M. Caulfield et. al. A cloud-scale acceleration architecture. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '16)*.

[14] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[15] Sandeep R Agrawal, Sam Idicula, Arun Raghavan, Evangelos Vlachos, Venkatraman Govindaraju, Venkatanathan Varadarajan, Cagri Balkesen, Georgios Giannikis, Charlie Roth, Nipun Agarwal, and Eric Sedlar. A Many-core Architecture for In-memory Data Processing. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '17)*.

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

[17] 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 *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*.

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

[19] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*.

[20] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. PIM-enabled Instructions: A Low-overhead, Locality-aware Processing-in-memory Architecture. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*.

[21] Alibaba. Alibaba Production Cluster Trace Data. `https://github.com/alibaba/clusterdata`.

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

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

[24] Amazon. Amazon EC2 Elastic GPUs. https://aws.amazon.com/ec2/elastic-gpus/.

[25] Amazon. Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/.

[26] Amazon. Amazon EC2 Root Device Volume. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/RootDeviceStorage.html#RootDeviceStorageConcepts.

[27] Amazon. AWS Nitro System. https://aws.amazon.com/ec2/nitro/.

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

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

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

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

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

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

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

[35] Y. Artsy, Hung-Yang Chang, and R. Finkel. Interprocess communication in charlotte. *IEEE Software*, Jan 1987.

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

[37] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, New York, NY, USA, 2012.

[38] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '12)*, London, United Kingdom, June 2012.

[39] Katelin Bailey, Luis Ceze, Steven D. Gribble, and Henry M. Levy. Operating System Implications of Fast, Cheap, Non-volatile Memory. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systemsi (HotOS '13)*, Napa, California, May 2011.

[40] Hitesh Ballani, Paolo Costa, Raphael Behrendt, Daniel Cletheroe, Istvan Haller, Krzysztof Jozwik, Fotini Karinou, Sophie Lange, Benn Thomsen, Kai Shi, and Hugh Williams. Sirius: A flat datacenter network with nanosecond optical switching. In *SIGCOMM 2020*. ACM, August 2020.

[41] Amnon Barak and Richard Wheeler. *MOSIX: An integrated unix for multiprocessor workstations*. International Computer Science Institute, 1988.

[42] Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. Popcorn: Bridging the programmability gap in heterogeneous-isa platforms. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*.

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

[44] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *Computer*, 40(12), December 2007.

[45] Forest Baskett, John H. Howard, and John T. Montague. Task Communication in DEMOS. In *Proceedings of the Sixth ACM Symposium on Operating Systems Principles (SOSP '77)*.

[46] Arkaprava Basu, Mark D. Hill, and Michael M. Swift. Reducing Memory Reference Energy with Opportunistic Virtual Caching. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*.

[47] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*.

[48] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Proceedings of the 2ed ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '90)*, Seattle, Washington, March 1990.

[49] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*.

[50] Laurent Bindschaedler, Ashvin Goel, and Willy Zwaenepoel. Hailstorm: Disaggregated compute and storage for distributed lsm-based databases. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, 2020.

[51] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. The End of Slow Networks: It's Time for a Redesign. *Proc. VLDB Endow.*, 9(7):528–539, March 2016.

[52] Roberto Bisiani and Mosur Ravishankar. PLUS: A Distributed Shared-Memory System. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA '90)*, Seattle, Washington, May 1990.

[53] David L. Black, Anoop Gupta, and Wolf-Dietrich Weber. Competitive Management of Distributed Shared Memory. In *Proceedings of the 34th IEEE Computer Society International Conference on COMPCON (COMPCON '89)*, San Francisco, California, March 1989.

[54] Mahdi Nazm Bojnordi and Engin Ipek. PARDIS: A Programmable Memory Controller for the DDRx Interfacing Standards. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*.

[55] L. Breslau, Pei Cao, Li Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: evidence and implications. In *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, 1999.

[56] Cache Coherent Interconnect for Accelerators, 2018. `https://www.ccixconsortium.com/`.

[57] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. Rethinking software runtimes for disaggregated memory. ASPLOS 2021, 2021.

[58] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Cascais, Portugal, October 2011.

[59] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. PolarFS: an ultra-low latency and failure resilient distributed file system for shared storage cloud database. *Proceedings of the VLDB Endowment (VLDB '18)*.

[60] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, et al. Polardb serverless: A cloud native database for disaggregated data centers. In *Proceedings of the 2021 International Conference on Management of Data*, 2021.

[61] A. Caulfield, P. Costa, and M. Ghobadi. Beyond smartnics: Towards a fully programmable cloud: Invited paper. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, 2018.

[62] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: Fault Containment for Shared-memory Multiprocessors. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*.

[63] X. Chen, H. Liu, D. Zhang, Z. Meng, Q. Huang, H. Zhou, C. Wu, X. Liu, and Q. Yang. Automatic performance-optimal offloading of network functions on programmable switches. *IEEE Transactions on Cloud Computing*.

[64] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and general distributed transactions using rdma and htm. In *Proceedings of the Eleventh European Conference on Computer Systems (EUROSYS '16)*, London, UK, April 2016.

[65] David Cheriton. The V Distributed System. *Commun. ACM*, 31(3), March 1988.

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

[67] Byung-Gon Chun, Frank Dabek, Andreas Haeberlen, Emil Sit, Hakim Weatherspoon, M. Frans Kaashoek, John Kubiatowicz, and Robert Morris. Efficient Replica Maintenance for Distributed Storage Systems. In *Proceedings of the 3rd Conference on Networked Systems Design and Implementation (NSDI '06)*, San Jose, California, May 2006.

[68] I-Hsin Chung, Bulent Abali, and Paul Crumley. Towards a Composable Computer System. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region (HPC Asia '18)*.

[69] Cisco, EMC, and Intel. The Performance Impact of NVMe and NVMe over Fabrics. `http://www.snia.org/sites/default/files/NVMe_Webcast_Slides_Final.1.pdf`.

[70] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines.

[71] CloudLab. `https://www.cloudlab.us/`.

[72] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*, New York, New York, March 2011.

[73] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Doug Burger, Benjamin C. Lee, and Derrick Coetzee. Better I/O through Byte-Addressable, Persistent Memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, Big Sky, Montana, October 2009.

[74] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*, New York, New York, June 2010.

[75] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, 2010.

[76] Paola Costa. Towards rack-scale computing: Challenges and opportunities. In *First International Workshop on Rack-scale Computing (WRSC '14)*.

[77] Paolo Costa, Hitesh Ballani, Kaveh Razavi, and Ian Kash. R2C2: A Network Stack for Rack-scale Computers. In *Proceedings of the ACM SIGCOMM 2015 Conference on SIGCOMM (SIGCOMM '15)*.

[78] CXL Consortium. `https://www.computeexpresslink.org/`.

[79] CXL Consortium. `https://www.computeexpresslink.org/`.

[80] 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 *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018.

[81] Rajdeep Das and Alex C. Snoeren. Enabling active networking on rmt hardware. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, HotNets '20, 2020.

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

[83] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. pages 137–150.

[84] Guiseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swami Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store.

[85] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and QoS-aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*.

[86] Gary Scott Delp. *The Architecture and Implementation of MEMNET: A High–speed Shared-memory Computer Communication Network*. PhD thesis, Newark, DE, USA, 1988. Order No. GAX88-24208.

[87] Fred Douglis and John K. Ousterhout. Process Migration in the Sprite Operating System. In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 18–25, Berlin, West Germany, September 1987. IEEE.

[88] DPDK. `https://www.dpdk.org/`.

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

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

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

[92] Nandita Dukkipati and Nick McKeown. Why flow-completion time is the right metric for congestion control. In *Proceedings of the ACM SIGCOMM 2006 Conference on SIGCOMM (SIGCOMM '06)*, SIGCOMM '06, 2006.

[93] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the EuroSys Conference (EuroSys '14)*, Amsterdam, The Netherlands, April 2014.

[94] Adam Dunkels. Design and Implementation of the lwIP TCP/IP Stack. *Swedish Institute of Computer Science*, 2001.

[95] Kevin Elphinstone and Gernot Heiser. From l3 to sel4 what have we learnt in 20 years of l4 microkernels? In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*.

[96] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*.

[97] Facebook. Introducing Lightning: A flexible NVMe JBOF. https://code.fb.com/data-center-engineering/introducing-lightning-a-flexible-nvme-jbof/.

[98] Facebook. Wedge 100: More open and versatile than ever. https://code.fb.com/networking-traffic/wedge-100-more-open-and-versatile-than-ever/.

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

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

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

[102] Seyed Kaveh Fayazbakhsh, Luis Chiang, Vyas Sekar, Minlan Yu, and Jeffrey C. Mogul. Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014.

[103] Andrew D. Ferguson, Steve Gribble, Chi-Yao Hong, Charles Killian, Waqar Mohsin, Henrik Muehe, Joon Ong, Leon Poutievski, Arjun Singh, Lorenzo Vicisano, Richard Alimi, Shawn Shuoshuo Chen, Mike Conley, Subhasree Mandal, Karthik Nagaraj, Kondapa Naidu Bollineni, Amr Sabaa, Shidong Zhang, Min Zhu, and Amin Vahdat. Orion: Google's Software-Defined networking control plane. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021.

[104] P. Ferreira and M. Shapiro. Larchant: Persistence by Reachability in Distributed Shared Memory through Garbage Collection. In *Distributed Computing Systems, 1996., Proceedings of the 16th International Conference on*, 1996.

[105] Paulo Ferreira, Marc Shapiro, Xavier Blondel, Olivier Fambon, João Garcia, Sytse Kloosterman, Nicolas Richer, Marcus Robert, Fadi Sandakly, George Coulouris, Jea Dollimore, Paulo Guedes, Daniel Hagimont, and Sacha Krakowiak. *PerDiS: Design, Implementation, and Use of a PERsistent DIstributed Store*. 2000.

[106] Daniel Firestone. VFP: A virtual switch platform for host SDN in the public cloud. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017.

[107] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre,

Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*.

[108] Brett D. Fleisch and Gerald J. Popek. Mirage: A Coherent Distributed Shared Memory Design. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP '89)*.

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

[110] FS. N8560-64C, 64-Port Ethernet L3 Data Center Switch, 64 x 100Gb QSFP28. `https://www.fs.com/products/110481.html`.

[111] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network Requirements for Resource Disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*.

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

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

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

[115] GenZ Consortium. `http://genzconsortium.org/`.

[116] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System.

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

[118] Phillip B. Gibbons, Michael Merritt, and Kourosh Gharachorloo. Proving Sequential Consistency of High-Performance Shared Memories. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '91)*, Hilton Head, South Carolina, July 1991.

[119] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation (OSDI '12)*.

[120] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. GraphX: Graph Processing in A Distributed Dataflow Framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, Colorado, October 2014.

[121] James R. Goodman. Coherency for Multiprocessor Virtual Address Caches. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems (ASPLOS '87)*.

[122] Google. Google Production Cluster Trace Data. `https://github.com/google/cluster-data`.

[123] Google. GPUs are now available for Google Compute Engine and Cloud Machine Learning. `https://cloudplatform.googleblog.com/2017/02/GPUs-are-now-available-for-Google-Compute-Engine-and-Cloud-Machine-Learning.html`.

[124] Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, London, UK, UK, 1978.

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

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

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

[128] Anubhav Guleria, J. Lakshmi, and Chakri Padala. Emf: Disaggregated gpus in datacenters for efficiency, modularity and flexibility. In *2019 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, 2019.

[129] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over Commodity Ethernet at Scale. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*, Florianopolis, Brazil, August 2016.

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

[131] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: A hardware-software co-designed disaggregated memory system. https://arxiv.org/abs/2108.03492.

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

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

[134] Hewlett-Packard. Memory Technology Evolution: An Overview of System Memory Technologies, 2010. http://h10032.www1.hp.com/ctg/Manual/c01552458.pdf.

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

[136] M Hosomi, H Yamagishi, T Yamamoto, K Bessho, Y Higo, K Yamane, H Yamada, M Shoji, H Hachino, C Fukumoto, et al. A Novel Nonvolatile Memory with Spin Torque Transfer Magnetization Switching: Spin-RAM. In *Electron Devices Meeting, 2005. IEDM Technical Digest. IEEE International*, pages 459–462, 2005.

[137] Tyler Hunt, Zhipeng Jia, Vance Miller, Ariel Szekely, Yige Hu, Christopher J. Rossbach, and Emmett Witchel. Telekine: Secure computing with cloud GPUs. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020.

[138] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. Netvm: High performance and flexible networking using virtualization on commodity platforms. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014.

[139] IBM. Crail Distributed File System. http://www.crail.io/.

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

[141] Intel. Intel Non-Volatile Memory 3D XPoint. http://www.intel.com/content/www/us/en/architecture-and-technology/non-volatile-memory.html?wapkw=3d+xpoint.

[142] Intel. Intel Omni-Path Architecture. https://tinyurl.com/ya3x4ktd.

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

[144] Intel, 2018. https://www.intel.com/content/www/us/en/high-performance-computing-fabrics/.

[145] Intel Corporation. Deprecating the PCOMMIT Instruction. https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction.

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

[147] JEDEC. HIGH BANDWIDTH MEMORY (HBM) DRAM JESD235A. https://www.jedec.org/standards-documents/docs/jesd235a.

[148] Wolfgang John, Joacim Halen, Xuejun Cai, Chunyan Fu, Torgny Holmberg, Vladimir Katardjiev, Mina Sedaghat, Pontus Sköldström, Daniel Turull, Vinay Yadhav, and James Kempf. Making Cloud Easy: Design Considerations and First Components of a Distributed Operating System for Cloud. In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '18)*.

[149] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Matt Dau Mike Daley, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Harshit Khaitan Alexander Kaplan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Matt Ross Jonathan Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Andy Swing Dan Steinberg, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*.

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

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

[152] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA Efficiently for Key-Value Services. In *Proceedings of the 2014 ACM conference on SIGCOMM (SIGCOMM '14)*, Chicago, Illinois, August 2014.

[153] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, Savanah, GA, November 2016.

[154] Kalia, Anuj and Kaminsky, Michael and Andersen, David G. Using RDMA Efficiently for Key-value Services. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14)*.

[155] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion control for high bandwidth-delay product networks. In *Proceedings of the ACM SIGCOMM 2002 Conference on SIGCOMM (SIGCOMM '02)*, SIGCOMM '02, 2002.

[156] 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 *2016 Design, Automation Test in Europe Conference Exhibition (DATE '16)*.

[157] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. Metron: NFV service chains at the true speed of the underlying hardware. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018.

[158] Antoine Kaufmann, SImon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High Performance Packet Processing with FlexNIC. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*.

[159] Stefanos Kaxiras and Alberto Ros. A New Perspective for Efficient Virtual-cache Coherence. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*.

[160] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA '92)*, Queensland, Australia, May 1992.

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

[162] R. E. Kessler and Miron Livny. An Analysis of Distributed Shared Memory Algorithms. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[163] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J. Rossbach. Sharing, protection, and compatibility for reconfigurable fabric with amorphos. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018.

[164] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash Storage Disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*.

[165] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash storage disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, 2016.

[166] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. ReFlex: Remote Flash $\approx$ Local Flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*.

[167] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. High-performance transactions for persistent memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, New York, NY, USA, 2016.

[168] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali G. Saidi, Steven Pelley, Sihang Liu, Peter M. Chen, and Thomas F. Wenisch. Delegated persist ordering. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*, 2016.

[169] Leonidas Kontothanassis, Galen Hunt, Robert Stets, Nikolaos Hardavellas, Michał Cierniak, Srinivasan Parthasarathy, Jr. Wagner Meira, Sandhya Dwarkadas, and Michael Scott. VM-Based Shared Memory on Low-Latency, Remote-Memory-Access Networks. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA '97)*, Denver, Colorado, June 1997.

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

[171] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. Do OS abstractions make sense on fpgas? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

[172] Orran Krieger and Michael Stumm. An Optimistic Algorithm for Consistent Replicated Shared Data. In *Proceedings of the 23rd Annual Hawaii International Conference on System Sciences (HICSS '90)*, Kailua-Kona, Hawaii, January 1990.

[173] John Kubiatowicz, David Bindel, Patrick Eaton, Yan Chen, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Westley Weimer, Chris Wells, Hakim Weatherspoon, and Ben Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage.

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

[175] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, A Social Network or A News Media? In *Proceedings of the 19th International Conference on World Wide Web (WWW '10)*, Raleigh, North Carolina, April 2010.

[176] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation (OSDI '12)*, Hollywood, California, October 2012.

[177] Butler W. Lampson. Atomic transactions. In *Distributed Systems - Architecture and Implementation, An Advanced Course*, London, UK, UK, 1981.

[178] Page Lawrence, Brin Sergey, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford University, 1998.

[179] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Phase Change Memory Architecture and the Quest for Scalability. *Commun. ACM*, 53(7):99–106, 2010.

[180] Benjamin C Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. Phase-change technology and the future of main memory. *IEEE micro*, 30(1):143, 2010.

[181] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed Virtual Disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '96)*.

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

[183] Myoung-Jae Lee, Chang Bum Lee, Dongsoo Lee, Seung Ryul Lee, Man Chang, Ji Hyun Hur, Young-Bae Kim, Chang-Jung Kim, David H Seo, Sunae Seo, et al. A Fast, High-Endurance and Scalable Non-Volatile Memory Device Made from Asymmetric Ta2O(5-x)/TaO(2-x) Bilayer Structures. *Nature materials*, 10(8):625–630, 2011.

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

[185] Sergey Legtchenko, Hugh Williams, Kaveh Razavi, Austin Donnelly, Richard Black, Andrew Douglas, Nathanael Cheriere, Daniel Fryer, Kai Mast, Angela Demke Brown, Ana Klimovic, Andy Slowey, and Antony Rowstron. Understanding Rack-Scale Disaggregated Storage. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '17)*.

[186] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[187] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, Shanghai, China, October 2017.

[188] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, 2016.

[189] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[190] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: High Precision Congestion Control. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19)*.

[191] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*.

[192] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. System-level Implications of Disaggregated Memory. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture (HPCA '12)*.

[193] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. PANIC: A high-performance programmable NIC for multi-tenant networks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

[194] LISA'17. Disaggregating the Network: Switching as a Service. `https://www.usenix.org/conference/lisa17/conference-program/presentation/schiff`.

[195] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading Distributed Applications onto SmartNICs Using IPipe. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19)*, Beijing, China, August 2019.

[196] Shai Bergman Matthias Hille Till Miemietz Nils Asmussen Michael Roitzsch Hermann Härtig Mark Silberstein Lluis Vilanova, Lina Maudlej. Slashing the disaggregation tax in heterogeneous data centers with fractos. EuroSys' 22, 2022.

[197] Virginia M. Lo. Operating Systems Enhancements for Distributed Shared Memory. *Advancies in Computers*, 39:191–237, December 1994.

[198] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. GraphLab: A New Framework For Parallel Machine Learning. In *Proceedings of the 26th Conference on Uncertainty in Artificial Intelligence (UAI '10)*, Catalina Island, California, July 2010.

[199] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning in the Cloud. *VLDB Endowment*, 5(8):716–727, April 2012.

[200] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: an rdma-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, Santa Clara, CA, 2017.

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

[202] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*, Chicago, Illinois, 2005.

[203] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*, Indianapolis, Indiana, June 2010.

[204] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. Clickos and the art of network function virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014.

[205] David Meisner, Brian T. Gold, and Thomas F. Wenisch. The powernap server architecture. *ACM Trans. Comput. Syst.*, February 2011.

[206] Mellanox. ConnectX-6 Single/Dual-Port Adapter supporting 200Gb/s with VPI. `http://www.mellanox.com/page/products_dyn?product_family=265&mtag=connectx_6_vpi_card`.

[207] Mellanox. Mellanox BuleField SmartNIC. `http://www.mellanox.com/page/products_dyn?product_family=275&mtag=bluefield_smart_nic`.

[208] Mellanox. Mellanox Innova Adapters. `http://www.mellanox.com/page/programmable_network_adapters?mtag=&mtag=programmable_adapter_cards`.

[209] Mellanox. Rdma aware networks programming user manual. `http://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf`.

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

[211] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. Atomic in-place updates for non-volatile main memories with kamino-tx. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, 2017.

[212] James Mickens, Edmund B. Nightingale, Jeremy Elson, Darren Gehring, Bin Fan, Asim Kadav, Vijay Chidambaram, Osama Khan, and Krishna Nareddy. Blizzard: Fast, Cloud-scale Block Storage for Cloud-oblivious Applications. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*.

[213] 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 *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, 2021.

[214] Dave Minturn. NVM Express Over Fabrics. 11th Annual OpenFabrics International OFS Developers' Workshop, March 2015.

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

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

[217] Jeffrey C. Mogul, Eduardo Argollo, Mehul Shah, and Paolo Faraboschi. Operating System support for NVM+DRAM Hybrid Main Memory. In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems (HotOS '09)*, Monte Verita, Switzerland, May 2009.

[218] MongoDB Inc. MongoDB. `http://www.mongodb.org/`.

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

[220] Timothy Prickett Morgan. Enterprises Get On The Xeon Phi Roadmap. `https://www.enterprisetech.com/2014/11/17/enterprises-get-xeon-phi-roadmap/`.

[221] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. A modern primer on processing in memory. *arXiv preprint arXiv:2012.03112*, 2020.

[222] Mihir Nanavati, Jake Wires, and Andrew Warfield. Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*.

[223] Dushyanth Narayanan and Orion Hodson. Whole-System Persistence. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '12)*, London, United Kingdom, March 2012.

[224] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-Tolerant Software Distributed Shared Memory. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC '15)*.

[225] Netronome. Agilio SmartNICs. `https://www.netronome.com/products/smartnic/overview/`.

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

[227] Nuno Neves, Miguel Castro, and Paulo Guedes. A checkpoint protocol for an entry consistent shared memory system. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, New York, NY, USA, 1994.

[228] Joel Nider and Alexandra (Sasha) Fedorova. The last cpu. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, 2021.

[229] Edmund B. Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. Helios: Heterogeneous Multiprocessing with Satellite Kernels. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*.

[230] Vlad Nitu, Boris Teabe, Alain Tchana, Canturk Isci, and Daniel Hagimont. Welcome to Zombieland: Practical and Energy-efficient Memory Disaggregation in a Datacenter. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*.

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

[232] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. Scale-out NUMA. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*.

[233] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. The Case for RackOut: Scalable Data Serving Using Rack-Scale Systems. In *Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC '16)*.

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

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

[236] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*.

[237] Open Coherent Accelerator Processor Interface, 2018. `https://opencapi.org/`.

[238] M. Oskin, F. T. Chong, and T. Sherwood. Active Pages: a computation model for intelligent memory. In *Proceedings. 25th Annual International Symposium on Computer Architecture (ISCA '98)*, Barcelona, Spain, August 1998.

[239] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments.

[240] Jiaxin Ou, Jiwu Shu, and Youyou Lu. A high performance file system for non-volatile main memory. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016.

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

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

[243] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch. The Sprite Network Operating System. *Computer*, 21(2), February 1988.

[244] Jiannan Ouyang, Brian Kocoloski, John R. Lange, and Kevin Pedretti. Achieving Performance Isolation with Lightweight Co-Kernels. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '15)*.

[245] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: A framework for nfv applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015.

[246] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. Netbricks: Taking the v out of NFV. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[247] David C. Parkes, Ariel D. Procaccia, and Nisarg Shah. Beyond dominant resource fairness: Extensions, limitations, and indivisibilities. *ACM Trans. Econ. Comput.*, March 2015.

[248] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, Piscataway, NJ, USA, 2014.

[249] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, Minneapolis, Minnesota, USA, 2014.

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

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

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

[253] Peter X. Gao and Akshay Narayan and Sagar Karandikar and Joao Carreira and Sangjin Han and Rachit Agarwal and Sylvia Ratnasamy and Scott Shenker. Network Requirements for Resource Disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, Savanah, GA, November 2016.

[254] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, Saint-Malo, France, October 1997.

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

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

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

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

[259] Yiming Qiu, Jiarong Xing, Kuo-Feng Hsu, Qiao Kang, Ming Liu, Srinivas Narayana, and Ang Chen. Automated smartnic offloading insights for network functions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, 2021.

[260] Moinuddin K Qureshi, Michele M Franceschini, Luis A Lastras-Montaño, and John P Karidis. Morphable memory system: a robust architecture for exploiting multi-level phase change memories. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '07)*, June 2010.

[261] Moinuddin K. Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, New York, NY, USA, 2009.

[262] Umakishore Ramachandran and M. Yousef A. Khalidi. An Implementation of Distributed Shared Memory. *Software:Practice and Experience*, 21(5):443–464, May 1991.

[263] Parthasarathy Ranganathan, Daniel Stodolsky, Jeff Calow, Jeremy Dorfman, Marisabel Guevara, Clinton Wills Smullen IV, Aki Kuusela, Raghu Balasubramanian, Sandeep Bhatia, Prakash Chauhan, Anna Cheung, In Suk Chong, Niranjani Dasharathi, Jia Feng, Brian Fosco, Samuel Foss, Ben Gelb, Sara J. Gwin, Yoshiaki Hase, Da-ke He, C. Richard Ho, Roy W. Huffman Jr., Elisha Indupalli, Indira Jayaram, Poonacha Kongetira, Cho Mon Kyaw, Aaron Laursen, Yuan Li, Fong Lou, Kyle A. Lucke, JP Maaninen, Ramon Macias, Maire Mahony, David Alexander Munday, Srikanth Muroor, Narayana Penukonda, Eric Perkins-Argueta, Devin Persaud, Alex Ramirez, Ville-Mikko Rautio, Yolanda Ripley, Amir Salek, Sathish Sekar, Sergey N. Sokolov, Rob Springer, Don Stark, Mercedes Tan, Mark S. Wachsler, Andrew C. Walton, David A. Wickeraad, Alvin Wijaya, and Hon Kwan Wu. Warehouse-scale video acceleration: Co-design and deployment in the wild. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021.

[264] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture (HPCA '07)*.

[265] Richard F. Rashid and George G. Robertson. Accent: A Communication Oriented Network Operating System Kernel. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles (SOSP '81)*.

[266] Wei Ren, Junhao Pan, and Deming Chen. Accguard: Secure and trusted computation on remote fpga accelerators. In *2021 IEEE International Symposium on Smart Electronic Systems (iSES) (Formerly iNiS)*, 2021.

[267] G. C. Richard and M. Singhal. Using logging and asynchronous checkpointing to implement recoverable distributed shared memory. In *Proceedings of 1993 IEEE 12th Symposium on Reliable Distributed Systems*, Oct 1993.

[268] Erik Riedel, Garth Gibson, and Christos Faloutsos. Active Storage For Large-Scale Data Mining and Multimedia.

[269] Brian M. Rogers, Anil Krishna, Gordon B. Bell, Ken Vu, Xiaowei Jiang, and Yan Solihin. Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*.

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

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

[272] Daniel Rozhko and Paul Chow. The network management unit (nmu): Securing network access for direct-connected fpgas. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, 2019.

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

[274] Stephen M. Rumble. Infiniband Verbs Performance. `https://ramcloud.atlassian.net/wiki/display/RAM/Infiniband+Verbs+Performance`.

[275] George Samaras, Kathryn Britton, Andrew Citron, and C. Mohan. Two-phase commit optimizations and tradeoffs in the commercial environment. In *Proceedings of the Ninth International Conference on Data Engineering*, Washington, DC, USA, 1993.

[276] Russel Sandberg. The Design and Implementation of the Sun Network File System. In *Proceedings of the 1985 USENIX Summer Technical Conference*, pages 119–130, Berkeley, CA, June 1985.

[277] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, Cambridge, Massachusetts, USA, 1996.

[278] Bianca Schroeder and Garth A. Gibson. Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You? In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST '07)*.

[279] Malte Schwarzkopf, Matthew P. Grosvenor, and Steven Hand. New Wine in Old Skins: The Case for Distributed Operating Systems in the Data Center. In *Proceedings of the 4th Asia-Pacific Workshop on Systems (APSys '13)*.

[280] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu Shi. Design and implementation of a consolidated middlebox architecture. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, 2012.

[281] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A User-Programmable SSD. In

Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14).

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

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

[284] Yizhou Shan, Will Lin, Ryan Kosta, Arvind Krishnamurthy, and Yiying Zhang. Disaggregating and consolidating network functionalities with supernic. In *arXiv*, 2022.

[285] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed Shared Persistent Memory. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*.

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

[287] Marc Shapiro and Paulo Ferreira. Larchant-rdoss: A distributed shared persistent memory and its garbage collector. Technical report, Ithaca, NY, USA, 1994.

[288] Marc Shapiro, Sytse Kloosterman, Fabio Riccardi, and The Perdis. Perdis - a persistent distributed store for cooperative applications. In *In Proc. 3rd Cabernet Plenary W*, 1997.

[289] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. *SIGCOMM Comput. Commun. Rev.*, 2012.

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

[291] Mark Silberstein. Accelerators in data centers: the systems perspective. `https://www.sigarch.org/accelerators-in-data-centers-the-systems-perspective/`.

[292] 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. *Microprocessors and Microsystems*, 71:102868, 2019.

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

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

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

[296] Alan Jay Smith. Cache Memories. *ACM Comput. Surv.*, 14(3), September 1982.

[297] SpinalHDL. SpinalHDL. `https://github.com/SpinalHDL/SpinalHDL`.

[298] M. Stumm and Songnian Zhou. Fault tolerant distributed shared memory algorithms. In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing 1990*, Dec 1990.

[299] Michael Stumm and Songnian Zhou. Algorithms Implementing Distributed Shared Memory. *IEEE Computer*, 23(5):54–64, May 1990.

[300] Michael Stumm and Songnian Zhou. Fault Tolerant Distributed Shared Memory Algorithms. In *Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing (IPDPS '90)*, Dallas, Texas, December 1990.

[301] Chen Sun, Jun Bi, Zhilong Zheng, Heng Yu, and Hongxin Hu. Nfp: Enabling network function parallelism in nfv. SIGCOMM '17, 2017.

[302] Kosuke Suzuki and Steven Swanson. The non-volatile memory technology database (nvmdb). Technical Report CS2015-1011, Department of Computer Science & Engineering, University of California, San Diego, May 2015.

[303] Andrew S. Tanenbaum, M. Frans Kaashoek, Robbert Van Renesse, and Henri E. Bal. The Amoeba Distributed Operating System&Mdash;a Status Report. *Comput. Commun.*, 14(6), August 1991.

[304] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, and Sape J. Mullender. Experiences with the Amoeba Distributed Operating System. *Commun. ACM*, 33(12), December 1990.

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

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

[307] Everspin Technologies. NVMe Storage Accelerator Series. `https://www.everspin.com/nvme-storage-accelerator-series`.

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

[309] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-Based Service Level Agreements for Cloud Storage. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Farmington, Pennsylvania, November 2013.

[310] Shelby Thomas, Geoffrey M. Voelker, and George Porter. CacheCloud: Towards Speed-of-light Datacenter Communication. In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '18)*.

[311] Devesh Tiwari, Simona Boboila, Sudharshan S. Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter J. Desnoyers, and Yan Solihin. Active flash: Towards energy-efficient, in-situ data analytics on extreme-scale machines. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, 2013.

[312] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E. Porter. A Study of Modern Linux API Usage and Compatibility: What to Support when You'Re Supporting. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*.

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

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

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

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

[317] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys '15)*.

[318] VMware. Virtual SAN. `https://www.vmware.com/products/vsan.html`.

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

[320] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*, New York, New York, March 2011.

[321] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992.

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

[323] Michael Walfish, Jeremy Stribling, Maxwell Krohn, Hari Balakrishnan, Robert Morris, and Scott Shenker. Middleboxes no longer considered harmful. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation - Volume 6*, OSDI'04, 2004.

[324] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A memory-disaggregated managed runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*.

[325] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. P4fpga: A rapid prototyping framework for p4. In *Proceedings of the Symposium on SDN Research*, SOSR '17, page 122–135, 2017.

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

[327] W. H. Wang, J.-L. Baer, and H. M. Levy. Organization and Performance of a Two-level Virtual-real Cache Hierarchy. In *Proceedings of the 16th Annual International Symposium on Computer Architecture (ISCA '89)*.

[328] Andrew Warfield, Russ Ross, Keir Fraser, Christian Limpach, and Steven Hand. Parallax: Managing Storage for a Million Machines. In *Proceedings of the 10th Conference on Hot Topics in Operating Systems (HotOS '05)*.

[329] David Wentzlaff, Charles Gruenwald, III, Nathan Beckmann, Kevin Modzelewski, Adam Belay, Lamia Youseff, Jason Miller, and Anant Agarwal. An Operating System for Multicore and Clouds: Mechanisms and Implementation. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*.

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

[331] Xiaojian Wu and A.L.N. Reddy. Scmfs: A file system for storage class memory. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*, Nov 2011.

[332] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, March 1995.

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

[334] J Joshua Yang, Dmitri B Strukov, and Duncan R Stewart. Memristive devices for computing. *Nature nanotechnology*, 8(1):13–24, 2013.

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

[336] Yiying Zhang and Steven Swanson. A Study of Application Performance with Non-Volatile Main Memory. In *Proceedings of the 31st IEEE Conference on Massive Data Storage (MSST '15)*, Santa Clara, California, June 2015.

[337] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI '12)*, San Jose, California, April 2012.

[338] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. The End of a Myth: Distributed Transactions Can Scale. *Proc. VLDB Endow.*, 10(6):685–696, February 2017.

[339] Gerd Zellweger, Simon Gerber, Kornilios Kourtis, and Timothy Roscoe. Decoupling cores, kernels, and operating systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI '14)*.

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

[341] Irene Zhang, Naveen Kr Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R K Ports. Building Consistent Transactions with Inconsistent Replication. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, Monterey, California, October 2015.

[342] Kaiyuan Zhang, Danyang Zhuo, and Arvind Krishnamurthy. Gallium: Automated software middlebox offloading to programmable switches. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, 2020.

[343] Lixin Zhang, Zhen Fang, M. Parker, B.K. Mathew, L. Schaelicke, J.B. Carter, W.C. Hsieh, and S.A. McKee. The impulse memory controller. *IEEE Transactions on Computers*, 50(11):1117–1132, 2001.

[344] Qiao Zhang, Guo Yu, Chuanxiong Guo, Yingnong Dang, Nick Swanson, Xinsheng Yang, Randolph Yao, Murali Chintalapati, Arvind Krishnamurthy, and Thomas Anderson. Deepview: Virtual Disk Failure Diagnosis and Pattern Detection for Azure. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*.

[345] Yang Zhang, Bilal Anwer, Vijay Gopalakrishnan, Bo Han, Joshua Reich, Aman Shaikh, and Zhi-Li Zhang. Parabox: Exploiting parallelism for virtual network functions in service chaining. In *Proceedings of the Symposium on SDN Research*, SOSR '17, 2017.

[346] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. Mojim: A Reliable and Highly-Available Non-Volatile Memory System. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*, Istanbul, Turkey, March 2015.

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

[348] Songnian Zhou, Michael Stumm, Kai Li, and David Wortman. Heterogeneous Distributed Shared Memory. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):540–554, September 1992.

[349] Yuanyuan Zhou, Liviu Iftode, and Kai Li. Performance Evaluation of Two Home-based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, Seattle, Washington, USA, 1996.

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

[351] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Xuan Kelvin Zou, Hang Guan, Arvind Krishnamurthy, and Thomas Anderson. RAIL: A case for redundant arrays of inexpensive links in data center networks. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017.

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