BUILDING FAST, SCALABLE, LOW-COST, AND SAFE RDMA SYSTEMS IN DATACENTERS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Shin-Yeh Tsai

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2019

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

Dr. Yiying Zhang, Chair

School of Electrical and Computer Engineering

Dr. Mathias Payer

Department of Computer Science

Dr. Tiark Rompf

Department of Computer Science

Dr. He Wang

Department of Computer Science

Approved by:

Dr. Voicu S. Popescu

Department of Computer Science

Dedicated to my family for their love and support

ACKNOWLEDGMENTS

This dissertation would not have been possible without the support from many people in my life. First of all, I would like to express my sincerest gratitude to my advisor Professor Yiying Zhang for her support, patience, and listening. I really appreciate that she gave me a chance to do research with her. By working with her, I have learned how to discover a research problem, how to build a system to verify ideas, and how to take broad, high-level ideas and to be able to focus on those ideas in a more nuanced, focused way.

I still remember the time that Professor Zhang and I worked together for my first research project, and the impressive moment that we were notified the project is accepted to be published. In the early stage of the project, she gave me tremendous suggestions to improve the designs. When we were consolidating our ideas and writing the paper, she taught me how to write in a technical style and helped me to polish and improve the contents. After the paper is accepted, she taught me how to prepare an excellent presentation and how to illustrate our ideas in the presentation clearly. I have learned so much from Professor Zhang, and I would not be able to complete this work without her gratuitous support.

I would also like to thank the members of my committee: Professor Mathias Payer, Professor Tiark Rompf, and Professor He Wang. Their feedback about my dissertation and research has made my work significantly stronger. I would particularly like to thank Professor Payer for his insightful comments and suggestions. It was an enjoyable experience to collaborate with him to discover exciting research ideas.

It was also a pleasure to be a member of an awesome research group, WukLab. I thank them for their input and support of my work. Their feedback about my ideas has made my work better.

Finally, I am immensely grateful to my family, especially my parents, who encouraged me to pursue this work and provide their unconditional love through the whole process. I acknowledge my mom for her endless patience and understanding. I miss my maternal grandfather and my paternal grandmother so much. The life lessons they taught me and what I have learned during my Ph.D. journey will be in my heart forever. I dedicate this dissertation to my family.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

ABSTRACT

Tsai, Shin-Yeh Ph.D., Purdue University, August 2019. Building Fast, Scalable, Low-Cost, and Safe RDMA Systems in Datacenters. Major Professor: Yiying Zhang.

Remote Direct Memory Access, or *RDMA*, is a technology that allows one computer server to direct access the memory of another server without involving its CPU. Compared with traditional network technologies, RDMA offers several benefits including low latency, high throughput, and low CPU utilization. These features are especially attractive to datacenters, and because of this, datacenters have started to adopt RDMA in production scale in recent years.

However, RDMA was designed for confined, single-tenant, High-Performance-Computing (HPC) environments. Many of its design choices do not fit datacenters well, and it cannot be readily used by datacenter applications. To use RDMA, current datacenter applications have to build customized software stacks and fine-tune their performance. In addition, RDMA offers limited scalability and does not have good support for resource sharing or protection across different applications.

This dissertation sets out to seek solutions that can solve issues of RDMA in a systematic way and makes it more suitable for a wide range of datacenter applications.

Our first task is to make RDMA more scalable, easier to use, and have better support for safe resource sharing in datacenters. For this purpose, we propose to add an *indirection layer* on top of native RDMA to virtualize its low-level abstraction into a high-level one. This indirection layer safely manages RDMA resources for different datacenter applications and also provide a means for better scalability.

After making RDMA more suitable for datacenter environments, our next task is to build applications that can exploit all the benefits from (our improved) RDMA. We designed a set of systems that store data in remote persistent memory and let client

machines access these data through pure one-sided RDMA communication. These systems lower monetary and energy cost compared to traditional datacenter data stores (because no processor is needed at remote persistent memory), while achieving good performance and reliability.

Our final task focuses on a completely different and so far largely overlooked one — security implications of RDMA. We discovered several key vulnerabilities in the one-sided communication pattern and in RDMA hardware. We exploited one of them to create a novel set of remote side-channel attacks, which we are able to launch on a widely used RDMA system with real RDMA hardware.

This dissertation is one of the initial efforts in making RDMA more suitable for datacenter environments from scalability, usability, cost, and security aspects. We hope that the systems we built as well as the lessons we learned can be helpful to future networking and systems researchers and practitioners.

1 INTRODUCTION

Remote Direct Memory Access, or *RDMA*, is a technology that lets a computer server directly access the memory of another server without involving the CPU of this other server. Apart from this access pattern (commonly called *one-sided communication*), RDMA also allows applications to issue requests directly from the user space to the NIC, bypassing the OS kernel, and in doing so RDMA avoids any memory copying. With one-sided communication, kernel bypassing, and memory zero copy, the RDMA technology enables network communication with high throughput, low latency, and low CPU utilization.

The RDMA technology was originally designed for the high-performance computing (HPC) environments and has been widely adopted in these environments over the past two decades. In recent years, there has been a dramatically increasing interest in adopting RDMA in datacenters. Major cloud vendors like Microsoft Azure [1] and Alibaba Cloud [2] have all deployed RDMA in production scale. RDMA is especially attractive in datacenter environments for three reasons.

First, datacenter applications are becoming more distributed [3–8] and *disaggregated* [9–16]. Network communication is a key factor in almost all modern datacenter applications' performance. While most of today's datacenters run on 40 Gbps Ethernet-based network with $\sim$100 μs round-trip time (RTT) [17, 18], the latest RDMA technology can reach 200 Gbps throughput and sub-0.6 μs RTT [19], thanks to its kernel bypassing and memory zero-copy features. Both throughput-oriented and latency-bound datacenter applications can benefit from the network performance improvements offered by RDMA.

Second, big data gives rise to a host of applications that access large amounts of (in-memory) data during their computation: data analytics, graph computation, deep learning, and in-memory databases, to name just a few. Their memory capacity

needs often exceed what a single machine can offer in today's datacenters [20–25]. For example, computation on a big social-network graph can take more than 5 TB memory [22]. Unfortunately, memory in a single computer is facing a capacity wall and a bandwidth wall, and it's hard to meet the requirements of many memory-hunger applications. At the same time, memory (and CPU) utilization across different machines is imbalanced, causing resource wastes in datacenters. Both these two problems can be solved by allowing applications to go beyond a single machine's memory and access *remote memory*, the key feature RDMA provides.

Third, certain tasks and applications in datacenters only need locations to store data without the need to perform any computation on them at these locations. For example, storage systems and databases in datacenters often keep multiple copies of data for better reliability and availability. With a primary-backup model, only a primary server performs application computation of data, and other servers (passively) receives copies of the data. RDMA's one-sided communication pattern benefits these datacenter systems by largely reducing the CPU utilization of the backup servers, which in turn saves energy consumption of the whole datacenter.

Despite of these promising usage cases of RDMA, moving RDMA into datacenters faces key challenges. RDMA was designed for HPC environments that are often single-purposed, single-tenant, and have small scales. RDMA in its native form (*i.e.*, unmodified RDMA hardware, driver, and default libraries) is not a good fit for the more general-purpose, multi-tenant, and large-scale datacenter environments.

Native RDMA is hard to use and inflexible; it does not scale well; it only offers limited options in sharing resources, performance isolation, and resource protection; and it has various security vulnerabilities. Because of these issues, datacenter applications cannot and should not use RDMA as is. A common practice currently is to build customized RDMA-based software stacks for each datacenter application and fine-tune its performance [26–38].

This dissertation aims to make RDMA more suitable for datacenter applications in a systematic way. We revisited many design decisions in native RDMA and built a

generic layer on top of native RDMA to make it more scalable and easier to use, with better support for resource sharing, performance isolation, and resource protection. We further built a set of remote-memory systems on top of this improved RDMA abstraction and ported several applications to it. Finally, we explored security vulnerabilities in both hardware that implements RDMA and the more generic one-sided communication pattern.

This dissertation significantly advances state of the art in RDMA software, transforming RDMA from a technology designed for restricted, HPC environments to one that is readily useful for a wide range of datacenter applications, all without changing any of RDMA-based hardware or the RDMA network protocol. Below, we give a brief overview of the three main projects in this dissertation.

1.1 Indirection Layer on RDMA for Better Datacenter Support

For RDMA to be successful in datacenters, it first needs to be scalable, easy-to-use, and safe to share across different applications. The first part of this dissertation focuses on improving the usability and scalability of RDMA for datacenter environments.

The design of native RDMA offloads the whole network stack and complex metadata to NIC hardware. While this approach minimizes software overhead, it inevitably limits the flexibility of RDMA and makes RDMA NICs the scalability bottleneck.

We propose to add a software indirection layer on top of RDMA hardware to *virtualize* the low-level, inflexible native RDMA abstraction into a flexible and easy-to-use one that can better support datacenter applications. We build such an indirection layer in the kernel space, which manages and safely shares RDMA resources across applications. This system not only makes RDMA a better fit for datacenter applications but also preserves most of native RDMA's superior performance and significantly improve its scalability.

1.2 RDMA-Based Data Store with Remote Persistent Memory

The indirection layer we built (as the first part of this dissertation) makes RDMA more flexible, scalable, easy-to-use, and suit datacenter environments better. The next question we seek to answer is *how to build applications on top of the (improved) RDMA stack*.

Our major efforts here is on building distributed RDMA-based data stores, an important class of systems that datacenter applications leverage for storing and accessing their data. Our focus here is to not only seek solutions that have good performance and scalability, but also ones that have low monetary and energy cost and can sustain system crashes.

To achieve these goals, we push the one-sided communication idea to an extreme by proposing to completely remove the CPU at where data is stored, making these data accessible only from one-sided RDMA. We further propose to store these data in *persistent memory*, which are denser than DRAM and can sustain power loss. With these ideas, we designed a set of novel RDMA-based data store systems with remote persistent memory. These systems offer great performance, low cost, reliability, and high availability.

1.3 Security Implications of One-Sided Communication and RDMA

The first two parts of this dissertation and most of existing RDMA research and production systems focus on the performance, scalability, and programmability of RDMA, leaving one important aspect yet to be explored — security. The third part of this dissertation explores various security implications of RDMA systems and one-sided communication in datacenter and cloud settings.

Among these security implications, the most notable one is vulnerabilities related to RDMA's hardware design. RDMA NICs caches various metadata in a limited-size on-NIC SRAM. We discovered new timing side channels that exploit this mechanism of RDMA NICs and built real side-channel attacks on a widely used RDMA data store

system with three generations of RDMA NICs. With these attacks, an attacker can steal the access information of a victim without accessing either the victim's machine or the machine that stores the data. As the first work to explore vulnerabilities in RDMA hardware, we raise awareness of the security aspects of RDMA and provide several promising directions of mitigation techniques.

1.4 Chapter Overview

The rest of this dissertation is organized as follows.

- **Background:** Chapter 2 provides a background on one-sided communication, RDMA architecture and applications, persistent memory, and side-channel attacks.
- **Indirection Layer on RDMA for Better Datacenter Support:** Chapter 3 presents LITE, the indirection layer we built in the Linux kernel which virtualizes native RDMA into a flexible, high-level, easy-to-use abstraction and allows applications to safely share resources.
- **RDMA-Based Data Store with Remote Persistent Memory:** Chapter 4 describes DPM, a set of systems that treat remote persistent memory as "dumb" (*i.e.*, with no processing power) and access these remote persistent memory devices via one-sided communication.
- **Security Implications of One-Sided Communication and RDMA:** Chapter 5 discusses various security implication of RDMA and the one-sided communication pattern. The focus of this chapter is a set of RDMA-based remote side-channel attacks. We also discuss potential mitigation techniques and the opportunities of leveraging RDMA to enhance security.
- **Related Work:** Chapter 6 briefly introduces other research efforts in datacenter RDMA systems and applications. We also present related research work in persistent memory data store systems and remote side-channel attacks.

- **Conclusion:** Chapter 7 concludes this dissertation with a summary of all the chapters and the lessons we have learned.

2 BACKGROUND

This chapter provides the background of various aspects of this dissertation. We first provide an overview of RDMA and its current usages in datacenters (§2.1), then introduce various distributed in-memory data store systems in datacenters (§2.2), and finally discusses side-channel attacks and its countermeasures (§2.3).

2.1 Remote Direct Memory Access

This section provides a background of RDMA. We first give an overview of the RDMA technology and introduce the abstraction of native RDMA. We then discuss RDMA NICs where most RDMA functionalities are implemented and provide our performance benchmark results of four different RDMA NICs. Finally, we describe RDMA-based applications in datacenter and the deployment of RDMA in datacenter.

2.1.1 Architecture and Abstraction

Remote Direct Memory Access, or *RDMA*, is a network technology designed to offer low-latency, high-throughput, and low-CPU-utilization network communication. Its main idea is to allow one machine to directly access the memory of another machine without involving the CPU of this other machine, a communication pattern called *one-sided communication*. RDMA offers superior performance compared with traditional network technologies because of three main technologies. First, one-sided RDMA requests bypass the CPU of the receiver. Second, applications issue RDMA requests directly from user space, bypassing kernel and avoiding kernel trap cost on the data path. Third, RDMA avoids memory copying when performing network communication (a technique called *zero-copy*).

There are three implementations of RDMA: InfiniBand (IB) [39, 40], Internet Wide Area RDMA Protocol (*iWARP*) [41], and RDMA over Converged Ethernet (*RoCE*) [42, 43]. All implementations follow the standard RDMA protocol [44]. IB is a switched network that is specifically designed for RDMA. iWARP and RoCE are two Ethernet-based technologies. iWARP is defined based on four IETF-standards and it relies on congestion-aware protocols such as TCP and STCP to deliver reliable RDMA services. iWARP typically requires implementing the whole TCP network stack in RDMA NICs, because the kernel implementation of the TCP network stack is a performance bottleneck in a high-speed network. RoCE implements the RDMA protocol over standard Ethernet (*RoCEv1*) and UDP (*RoCEv2*), and is the preferred technology in existing datacenters [18]. Because of its performance and cost benefits [18, 45, 46], RDMA has been deployed in large scale in datacenters like Microsoft Azure [1] and Alibaba Cloud [2].

To find out the owning cost of RDMA networks in datacenters, we study the market prices of RDMA NICs and RDMA switches. We use RoCE as an example since most of the datacenter networks already adopt Ethernet-based network, and RoCE offers better performance than iWARP [46, 47]. Lots of the existing commodity Ethernet switches [48–50] already support Priority Flow Control (PFC) which enables lossless Ethernet for RoCE. Thus, it does not need to replace existing Ethernet switches to adopt RoCE network. On the other hand, RoCE NICs are price competitive to regular Ethernet NICs [51, 52] especially in 40 Gbps — the most popular network bandwidth in datacenters [18]. Therefore, in datacenters, the owning cost of RoCE network is similar to the owning cost of existing Ethernet-based network because the price of RoCE NICs is similar to the price of regular Ethernet NICs and datacenters do not need to purchase a set of new Ethernet switches.

The standard interface of native RDMA is a set of operations collectively called *Verbs*. Native RDMA allows accesses from both user space and kernel space using Verbs. *libibverbs* is an open-source low-level library which defines a set of required APIs. Developers build RDMA-based applications through Verbs and libibverbs, and

both Verbs and libibverbs can support different RDMA implementations (*e.g.*, RoCE, iWARP, and RDMA) which facilitates the development of RDMA-based applications.

RDMA supports both one-sided and two-sided communication. One-sided RDMA operations directly access memory at a remote node without involving the remote node's CPU, similar to DMA on a single machine. Two-sided RDMA operations inform the remote node of a delivered message and involve both sender and receiver processing either through signaling or polling, similar to `send`/`recv` in traditional network messaging.

RDMA communication is implemented using various types of *queues* including send, receive, and completion queues, and applications use these queues to post requests and to learn when a request has completed or been received.

To perform a one-sided RDMA operation, an application process at a receiver node needs to first allocate a consecutive virtual memory space and then use the virtual memory address range to register a *memory region*, or *MR*, with the RDMA NICs. An application can register multiple MRs over the same or different memory spaces. The RDMA NIC will assign a pair of local and remote protection keys (called *lkey* and *rkey*) to each MR. This application then conveys the virtual address of the MR and its *rkey* to processes running on other nodes. After building connections between these other nodes (senders) and the node that the MR-registering application runs on (receiver), these processes can use 1) a virtual memory address that falls in the MR's virtual memory address range, 2) a size, and 3) the *rkey* of the MR to perform one-sided RDMA `read` and `write`. In RDMA's term, a connection is called a *Queue Pair*, or *QP*. The local host also needs to register a local MR for the read/write buffer and can then perform RDMA operations by posting requests on a send queue (*SQ*). The RDMA read/write operation returns as soon as the request is sent to RDMA NIC. Applications have to separately poll a send *completion queue* (*CQ*) to know when the remote data has been read or written.

To perform a two-sided RDMA operation, the remote host needs to pre-post receive buffers to a receive queue (*RQ*) before the local host can send a message. The remote host polls the receive CQ to identify a received message coming.

RDMA supports reliable and unreliable connections (*RC* and *UC*) and unreliable datagram (*UD*). UD mode has the best scalability because UD mode supports the communication between one UD QP to multiple UD QPs, but UD mode only supports send/recv operations. UC mode supports send/recv and write operations, but the connection semantic limits the scalability to the communication between one UC QP to one UC QP. RC mode supports all RDMA operations including send/recv, read/write, and atomic operations, and RC mode has the same scalability as UC mode although RC mode has more processing overhead on RDMA NICs [33].

2.1.2 RDMA NICs

RDMA NICs, or *RNIC*s, are where most RDMA functionalities are implemented. They usually contain complex hardware logic that implements the RDMA protocol and some SRAM to store metadata, and they are often connected to the host's PCIe bus (allowing RNICs access to main memory through DMA). Because of the need to bypass the kernel and receiver's CPU, most RDMA functionalities and data structures have to be offloaded to the RNIC hardware.

An RNIC's on-board SRAM stores three types of metadata. First, it stores metadata for each QP in its memory. Second, it stores *lkey*s, *rkey*s, and virtual memory addresses for all registered MRs. Third, it caches page table entries (PTEs) for MRs to obtain the DMA address of an RDMA request from its virtual memory address. RNICs have a limited amount of on-board SRAM which can only hold metadata for hot data. When the SRAM is full, an RNIC will evict its cached metadata to the main memory on the host machine, and on a future access, fetch the evicted metadata from the host main memory back through the PCIe bus. The SRAM architecture is

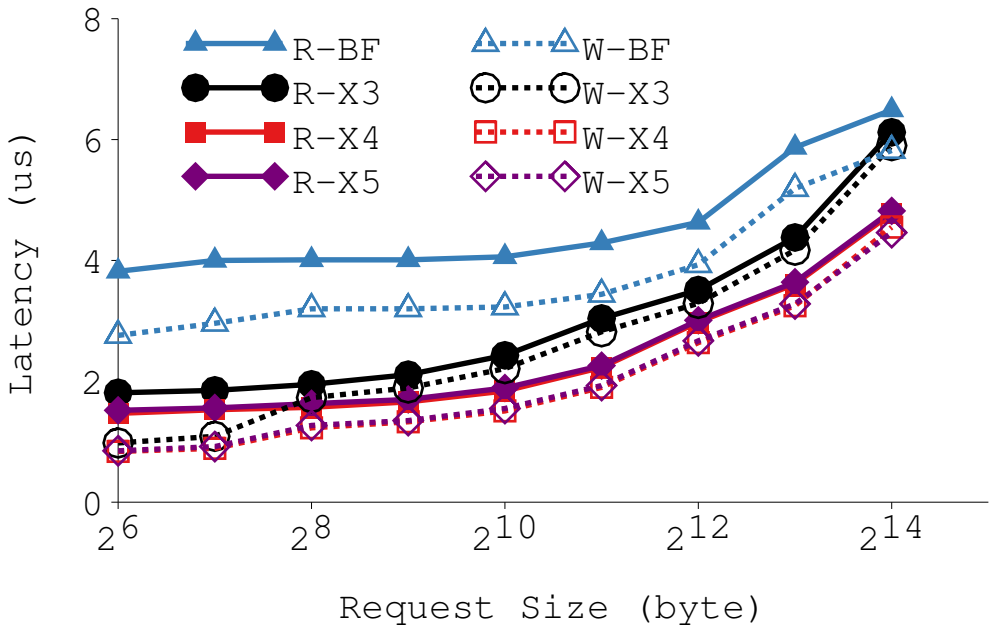


Figure 2.1.: **Latency of RDMA Requests.** *R- and W- represent READ and WRITE request respectively. X3, X4, and X5 represent ConnectX series RNICs, and BF represents BlueField. All of them are connected with a 100 Gbps InfiniBand switch.*

vendor-specific and not disclosed or specified in the RDMA standard. We benchmark the performance of the state-of-the-art RNICs in Section 2.1.3.

2.1.3 RDMA Microbenchmarks

To better understand the performance of RDMA and to provide a baseline for evaluations in this dissertation, we show the microbenchmarks of the native RDMA among four different Mellanox RNICs: ConnectX-3 (40 Gbps) [53], ConnectX-4 (100 Gbps) [54], ConnectX-5 (100 Gbps) [55], and BlueField (100 Gbps) [56]. We connect all the RNICs with a 100 Gbps InfiniBand switch. ConnectX-3, ConnectX-4, and ConnectX-5 are ASIC-based NICs, and BlueField is a SoC-based SmartNIC which has on-board programmable processors. SmartNICs are more expensive than ASIC-based NICs, but SmartNICs can accelerate several applications (*e.g.*, encryption, decryption, firewall, etc) and reduce host CPU utilization.

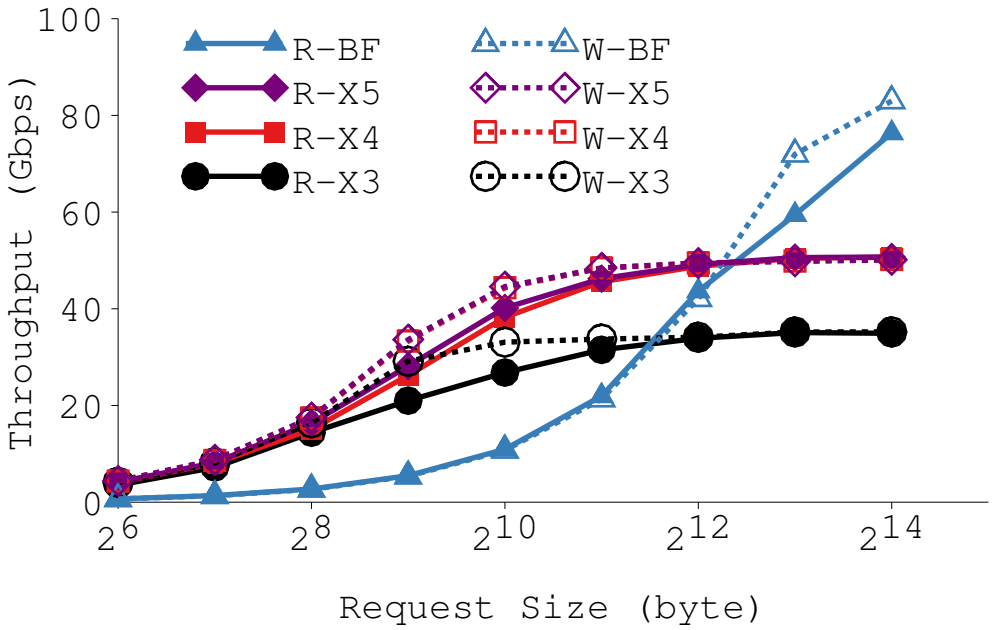


Figure 2.2.: **Throughput of RDMA Requests.** *R- and W- represent READ and WRITE request respectively. X3, X4, and X5 represent ConnectX series RNICs, and BF represents BlueField. All of them are connected with a 100 Gbps InfiniBand switch.*

We use OpenFabrics *perftest* software to measure the latency and the throughput of the above four RNICs. Limited by the type of machines we have in our lab cluster, all RNICs can only connect to these machines via a PCIe 3.0 ×8 bus. However, ConnectX-4 and ConnectX-5 require at least PCIe 3.0 ×16 bus to sustain the 100 Gbps bandwidth. Therefore, the throughput of these two RNICs are capped by the PCIe bandwidth of the host machines. For BlueField, we use the on-board SoCs to issue and process the requests. Thus, the RDMA requests access on-board memory through on-board PCIe 4.0 ×16 bus without being capped by the PCIe bandwidth of the host machines.

Figure 2.1 shows the latency of RDMA requests as request size increases. For all different RNICs, read request has higher latency than write request because of the implementation of RDMA transmission protocol [33]. ConnectX-4 and ConnectX-5 are the new generation RNICs which outperform ConnectX-3 in latency especially

when the request size is big. Although BlueField uses the same chip as ConnectX-5, BlueField has a higher latency because the processing speed of BlueField is slower than the regular ASIC-based RNICs.

Figure 2.2 takes a closer look at the throughput of RNICs against request size. BlueField has the highest throughput because BlueField supports the highest maximum bandwidth and the on-board PCIe bandwidth can sustain the bandwidth. Although ConnectX-4 and ConnectX-5 have the same maximum bandwidth as BlueField, the throughput of ConnectX-4 and ConnectX-5 is capped by the PCIe bandwidth of the host machines. ConnectX-3 has the lowest throughput because the maximum bandwidth of ConnectX-3 is only 40 Gbps. Read and Write requests have similar throughput. When the request size is small (*e.g.*, smaller than 4096), the throughput of BlueField is lower than ConnectX-4 and ConnectX-5 because of the limited processing speed of BlueField. When the request size increases, BlueField has the highest throughput because the bandwidth is not capped by the on-board PCIe bandwidth, but the throughputs of ConnectX-4 and ConnectX-5 are capped by the attached PCIe 3.0 $\times 8$ bus. There is no significant difference between read and write request in the throughput test because the parallel processing within the throughput benchmark amortizes the difference between two protocols.

2.1.4 RDMA in Datacenter Applications

The past two decades have seen a growing usage of RDMA in HPC environments [57–59]. In recent years, there is an emerging trend in using RDMA in datacenter environments from both industry and academia [60–63]. Recent RDMA-based applications include in-memory key-value stores [29–35, 64–66], in-memory databases and transactional systems [26, 67, 68], graph processing systems [28, 69], distributed machine learning systems [70], consensus implementations [71, 72], distributed non-volatile memory systems [36, 37, 73], and remote swap systems [63, 74]. Most of these

applications use both one-sided and two-sided RDMA operations, with some being pure one-sided [26, 27].

We believe that the use of RDMA in datacenters will continue increasing in the future because of application needs support from hardware and network. Many modern datacenter applications demand fast access to a vast amount of data, most conveniently and efficiently as in-memory data. With memory on a single machine facing its wall [75], these applications can largely benefit from fast, direct access to remote memory [25, 63, 76]. At the same time, more hardware in datacenters is adding the support for direct RDMA accesses, such as NVMe over Fabrics [77, 78] and GPU Direct [79–82]. Thus, there will be both high demand and more support for the low-latency, RDMA-equipped technologies in datacenters, and we believe that the trend of using RDMA to build datacenter systems will only increase in incoming years.

2.2 Datacenter In-Memory Data Stores

This section presents background on different types of data stores including in-memory data stores, in-PM data stores, and RDMA data stores in the datacenter settings.

2.2.1 Distributed and Remote DRAM-Based Data Stores

Existing in-memory data store systems can be categorized into two types of models. The first model is *distributed memory*, which combines the memory of nodes in a cluster into a virtual memory pool [25, 37, 83–89]. Processes running on any node in the cluster can allocate and access memory in this global pool. However, usage of these distributed memory systems has been limited, mainly because their high network and software overheads are not acceptable for most memory-based applications. Distributed memory/PM allows a process to use memory larger than local main memory and greatly improves resource utilization. However, it requires a sig-

nificant amount of CPU time on each machine to participate in a distributed data access protocol.

The second model is a *remote memory* model which separates servers in a cluster into two groups, compute nodes and memory nodes [26, 27, 90–92]. Data is stored and managed at the memory nodes and compute nodes issue network requests to fetch or store data at memory nodes. This model requires CPUs at memory nodes to process memory requests and perform various metadata and control tasks. Every memory has a local CPU which performs memory management.

2.2.2 Distributed Persistent Memory Data Stores

Non-volatile memory technologies such as 3DXpoint [93], phase change memory (*PCM*), spin-transfer torque magnetic memories (*STTMs*), and the memristor provide byte addressability, persistence, and latency that is within an order of magnitude of DRAM [94–101]. In addition, NVM consumes significantly lower energy compared to DRAM [102–104]. NVMs can attach directly to the main memory bus and we call such DIMM-based NVMs Persistent Memory or *PM* in this dissertation. PM has attracted extensive research efforts in the past decade, most of which were designed for single-node environments [29, 105–113].

Recently, because of the need to deploy PM in datacenter environments [114], new research interests arise in the distributed PM area [36, 37, 73, 115]. Mojim [36] uses asynchronous primary-backup replication to provide reliability and availability to PM. Hotpot [37] and Octopus [73] have all taken a model where each node in a cluster includes some amount of PM used to store data that can be accessed both locally and by other nodes. This distributed PM model not only increases CPU utilization but also makes PM deployment hard. To deploy PM in datacenters, one has to find empty DIMM slots in existing servers there or purchase new servers to host PM.

2.2.3 RDMA Data Stores

In addition to the general discussion of RDMA applications in Section 2.1, this section focuses on RDMA data stores specifically. Recently, with the increasing popularity of low-latency RDMA networks in datacenters [29, 33, 116], several recent in-memory data stores such as key-value stores [29–31, 34, 35, 64, 65], in-memory databases and transactional systems [26, 27, 67, 68], and remote swap systems [63, 74] use RDMA to perform their network communication. Most of them use one-sided RDMA for reads and two-sided RDMA for writes. To achieve low-latency performance, they usually use busy-polling threads to receive incoming two-sided RDMA requests. Wei *et al.* summarize the design tradeoffs of one-sided vs. two-sided RDMA for transaction systems [68].

Existing RDMA-based distributed data stores perform management tasks such as memory allocation, garbage collection, and load balancing at CPUs in data-hosting nodes [26, 29]. Consequently, even when one-sided RDMA operations help reduce CPU utilization, practical RDMA-based data stores still require a CPU and a significant amount of energy at each data-hosting machine. For example, FaRM [29], an RDMA-based distributed memory system that tries to use as much one-sided communication as possible, still requires processing power to perform metadata operations and certain steps in its write replication protocol.

2.3 Side-Channel Attacks

Traditional computer security attacks target the weakness in a computer system itself. In contrast, side-channel attacks exploit information leakages in aspects beyond the targeted system itself (thus the name "side channel"). Such side channels include power usage, electromagnetic emissions, timing difference, and acoustic emissions [117, 118].

In recent years, a host of attacks that exploit various hardware features to establish side channels have been proposed. CPU-cache-based side-channel attacks such

as Prime+Probe [119–127], Evict+Reload [128], Flush+Reload [129], and Flush+Flush [127, 128] can leak victim's memory access patterns at fine granularity. CPU-cache-based side channels are also the key enabling factors in attacks like Meltdown [130], Spectre [131], and Foreshadow [132]. Other than CPU caches, TLB [133] or port contention [134] also expose hardware-based side channels.

Side-channel attacks raised key concerns in cloud environments where one tenant can steal information from other tenants when they share the same physical resource [122, 135–137] or the same service [138]. Most notable and impactful side-channel related attacks are Spectre [131] and Meltdown [130], which exploit vulnerabilities in most modern processors to steal data that attackers otherwise do not have access to. They resulted in patching Linux machines running in almost all major datacenters and causing application performance slowdown of as high as 30% [139].

There are two major ways to mitigate side-channel attacks in system design. The first way is to hide leaked information (*e.g.*, re-implement critical components or change cryptographic primitives). The second way is to remove the correlation between each execution cycle (*e.g.*, randomization or introducing noise). System developers frequently use noise, randomization, and resource isolation to hide information and eliminate the correlation [118, 129, 140, 141]. However, these countermeasures mostly come with latency and throughput overhead, which is critical to datacenter applications. Datacenter environments share physical resources for better resource utilization, and this colocation makes datacenter applications even more vulnerable to side-channel attacks.

3 INDIRECTION LAYER ON RDMA FOR BETTER DATACENTER SUPPORT

RDMA was originally designed for HPC environemnts. Many design choices in RDMA suit a confined, single-purpose environment like HPC well, but native RDMA is not a good fit for the more general-purposed, heterogeneous, and large-scale datacenter environments because of the following three reasons.

First, there is a fundamental mismatch between the abstraction native RDMA provides and what datacenter applications desire. Datacenter applications usually build on high-level abstractions, but native RDMA provides a low-level abstraction that is close to hardware primitives. As a result, it is not easy for datacenter applications to use RDMA and even more difficult for them to exploit all the performance benefits of RDMA. Most RDMA-based datacenter applications require customized RDMA software stacks [29–33, 66], significant amounts of application adaptation [26–28], or changes in RDMA drivers [29, 66].

Second, RDMA manages and protects resources at the hardware level, and it lets user-level applications directly issue requests to RDMA NICs (called RNICs) bypassing kernel. This design causes at least three drawbacks for datacenter applications: lack of resource sharing, insufficient performance isolation, and inflexible protection.

The third issue of native RDMA is that the current architecture of RDMA cannot provide performance that scales with datacenter applications' memory usage. When bypassing kernel, RDMA inevitably adds burden to RNICs by moving privileged operations and metadata to hardware. For example, RNICs store protection keys and cache page table entries for user memory regions in its SRAM. It is essentially difficult for this architecture to meet datacenter applications' memory usage demand since the increase of on-RNIC memory capacity is slow and is cost- and energy-inefficient.

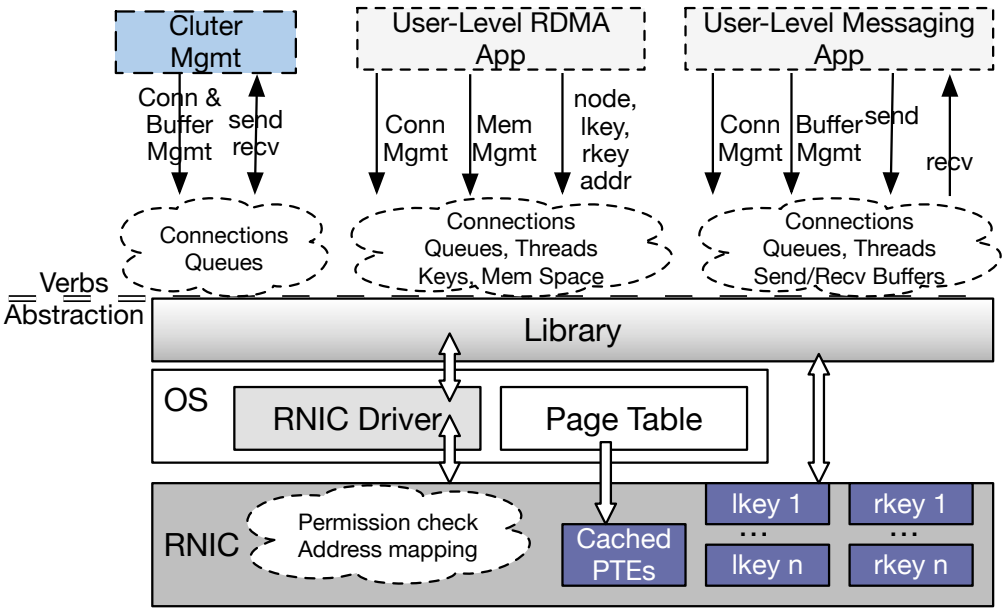


Figure 3.1.: **Traditional RDMA Stack.**

This chapter presents *LITE*, a *Local Indirection TiEr* in the kernel space to virtualize and manage RDMA for datacenter applications. LITE organizes memory as virtualized memory regions and supports a rich set of APIs including various memory operations, RPC, messaging, and synchronization primitives. Being in the kernel space, LITE safely manages privileged resources, provides flexible protection, and guarantees performance isolation across applications. Figures 3.1 and 3.2 illustrate the architecture of native RDMA and LITE.

Our approach of *onloading* [142] functionalities into kernel is the opposite to RDMA and many other networking systems' approach of offloading functionalities into hardware [143–145]. It is widely held that RDMA achieves its low-latency performance by directly accessing remote memory, bypassing kernel, and zero memory copy. We revisited these three techniques and found that with a good design, using a kernel-level indirection layer can preserve RDMA's performance benefits *and* avoid the drawbacks of native RDMA caused by kernel bypassing.

First, we add a level of indirection only at the local node and still ensure that one-sided RDMA operations directly access remote memory. Second, we onload only the

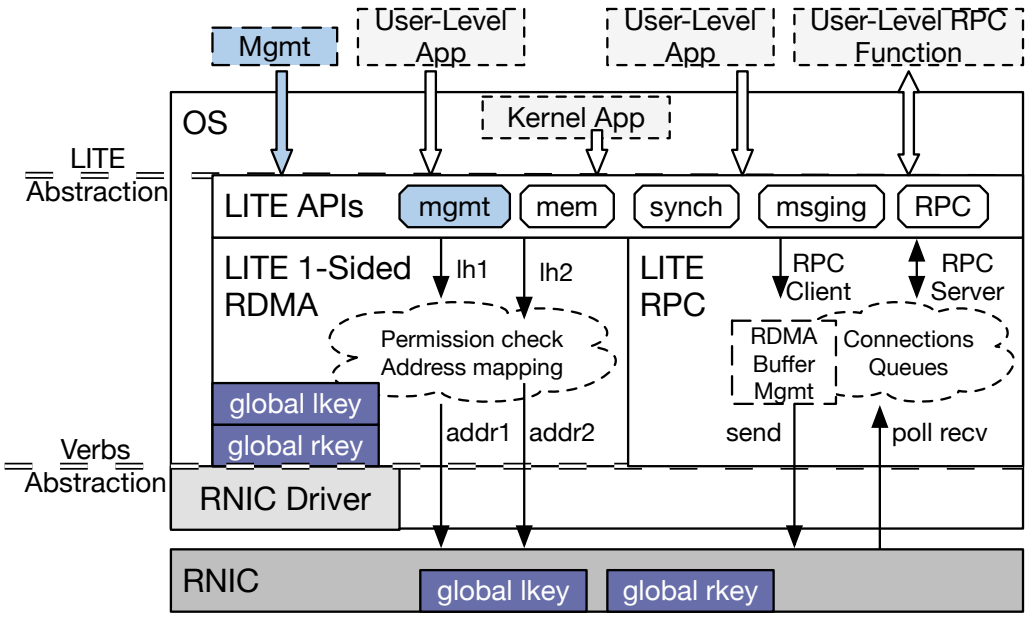


Figure 3.2.: **LITE Architecture.**

management of privileged resources from hardware to kernel and leave the rest of the network stack at hardware. Doing so not only preserves native RDMA's performance but also solves the performance scalability issues caused by limited on-RNIC SRAM. Third, we avoid memory copy between user and kernel spaces by addressing user memory directly with physical addresses. Finally, we designed several optimization techniques to minimize system call overhead.

Internally, LITE consists of an RDMA stack and an RPC stack. The RDMA stack manages LITE's memory abstraction by performing its own address mapping and permission checking. With this level of indirection, we safely remove these two functionalities and the resulting scalability bottlenecks from RNICs without any modification on RNICs or drivers. LITE implements RPC with a new mechanism based on two-sided RDMA and it achieves flexibility, good performance, low CPU utilization, and efficient memory space usage at the same time. On top of these two stacks, we implement a set of extended higher-level functionalities and QoS mechanisms.

Our evaluation shows that compared to native RDMA and existing solutions that are customized to certain applications [32, 66], LITE delivers similar latency and

throughput, while improving flexibility, performance scalability, CPU utilization, resource sharing, and quality of service.

We further demonstrate the ease-of-use, flexibility, and performance benefits of LITE by building four distributed applications on LITE: an atomic logging system, a MapReduce system, a graph engine, and a kernel-level Distributed Shared Memory (DSM) system. These systems are easy to build and perform well. For example, our implementation of graph engine has only 20 lines of LITE code, which encapsulate all the network communication functionalities. While using the same design as PowerGraph [6], this LITE-based graph engine outperforms PowerGraph by 3.5× to 5.6×. Our LITE-based MapReduce is ported from a single-node MapReduce implementation [146] with 49 lines of LITE code, and it outperforms Hadoop [4] by 4.3× to 5.3×.

In summary, we propose to build a new abstraction, LITE, which *virtualizes* the low-level, inflexible native RDMA abstraction into a flexible and easy-to-use one that can better support datacenter applications, and to build this virtualization layer in the kernel space to manage and safely share RDMA resources across applications.

Overall, LITE offers the following three key functionalities:

- Virtualizes RDMA with a generic kernel-level indirection layer for datacenter RDMA applications.
- Provides a set of mechanisms to minimize the performance overhead of kernel-level indirection and demonstrated the possibility of virtualizing RDMA while preserving (or even improving) its performance.
- Solves all the three issues of native RDMA for datacenter applications. Datacenter applications can easily use the abstraction layer to perform low-latency network communication and distributed operations.

3.1 Issues of RDMA

This section introduces the limitations of using RDMA in datacenter applications. Many design choices of RDMA work well in a controlled, specialized environment like HPC. However, datacenter environments are different in that they need to support heterogeneous, large-scale, fast-evolving applications. There are several issues in using native RDMA for datacenter applications as we will introduce in Section 3.1.1, Section 3.1.2, and Section 3.1.3.

3.1.1 Issue 1: Mismatch in Abstractions

A major reason why RDMA is not easy to use is the mismatch between its abstraction and what datacenter applications desire. Unlike the HPC environment where developers can carefully tune one or very few applications to dedicated hardware, datacenter application developers desire a high-level, easy-to-use, flexible abstraction for network communication so that they can focus on application-specific development. Originally designed for the HPC environment, native RDMA uses a low-level abstraction that is close to hardware primitives and is difficult to use [32]. Applications have to explicitly manage various types of resources and go through several non-intuitive steps to perform an RDMA operation, as explained in Section 2.1. It is even more difficult to optimize RDMA performance. Users need to properly choose from different RDMA operation options and tune various configurations, sometimes even to adopt low-level optimization techniques [29, 32, 66].

An API wrapper on top of the native RDMA interface such as Rsocket [147] can translate certain RDMA APIs into high-level ones. However, such a simple wrapper is far from enough for datacenter applications. For example, RDMA memory regions are created and accessed using virtual memory addresses in application process address spaces. The use of virtual memory addresses requires RNICs to store page table entries (PTEs) for address mapping (Section 3.1.2), makes it hard to share resources across processes (Section 3.1.3), and does not sustain process crashes. API wrappers

cannot solve any of these issues since they do not change the way RNICs use virtual memory addresses.

3.1.2 Issue 2: Unscalable Performance

When bypassing kernel, privileged operations and data structures are offloaded to the hardware RNIC. With limited on-RNIC SRAM, it is fundamentally hard for RDMA performance to scale with respect to three factors: the amount of MRs, the total size of MRs, and the total number of QPs.

First, RNICs store *lkey*s, *rkey*s, and virtual memory addresses for all registered MRs. As the number of MRs increases, RNICs will soon (above 100 MRs in our experiments) face memory pressure (Figure 3.4). Since each MR is a consecutive virtual memory range and supports only one permission, not being able to use many MRs largely limits the flexibility of RDMA. For example, many key-value store systems use non-consecutive memory regions to store data. Memcached [90] performs on-demand memory allocation in 1 MB units and optimizes memory allocation using 64 MB pre-allocated memory blocks. It would require at least 1000 MRs for 64 GB data even with pre-allocation. Masstree [148] uses a separate memory region for each value and can take up to 140 million memory regions. These scenarios use MRs far more than what an RNIC can handle without losing performance. Using bigger MRs can reduce the total number of MRs, but requires substantial application changes and can cause memory space waste [149].

Second, RNIC caches PTEs for MRs to obtain the DMA address of an RDMA request from its virtual memory address. When there is a PTE miss in the RNIC, the RNIC will fetch the PTE from the host OS. When the total size of registered MRs exceeds what an RNIC can handle (above 4 MB in our experiments in Figure 3.5), thrashing will happen and degrade performance. Unfortunately, most datacenter applications use large amount of memory. For example, performing PageRank [150] on a 1.3 GB dataset using GraphX [151] and Spark [3] will need 12 GB and 16 GB memory

heaps respectively [63]. FaRM [29] uses 2 GB huge pages to mitigate the scalability issue of MR size. However, using huge pages will result in increased memory footprints, physical memory fragmentation, false memory sharing, and degraded NUMA performance [152–154].

Finally, RNIC stores metadata for each QP in its memory. RDMA performance drops when the number of QPs increases [29], largely limiting the total number of nodes that can be connected through RC in an RDMA cluster. FaSST [66] used UD to reduce the number of QPs. But UD is unreliable and does not support one-sided RDMA.

Datacenter applications often require the above three types of scalability. Even if a single application's scale is small, the combination of multiple applications will likely cause scalability issues. The speed of on-RNIC memory increase falls behind the increasing scalability of datacenter applications. Moreover, large on-RNIC memory is cost- and energy-inefficient [142]. We believe that offloading all priviledged functionalities and metadata to hardware is not and will not be a viable way to use RDMA for datacenter applications. Rather, the RDMA software and hardware stacks need to be restructured.

3.1.3 Issue 3: Lack of Resource Sharing, Isolation, and Protection

Native RDMA does not provide any mechanisms to safely share resources such as QPs, CQs, memory buffers, polling threads across different applications; it only provides a mechanism to share receive queues (called SRQ) within a process. Each application process has to build and manage its own set of resources.

The lack of resource sharing makes the performance scalability issue described above even worse. For example, each pair of processes on two nodes need to build at least one QP to perform RC operations. To perform polling for two-sided RDMA, each node needs at least one thread per process to busy poll separate receive CQs. Sharing resources within a process [29] improves scalability, but has limited scope.

Without global management of resources, it is also hard to isolate performance and deliver quality of service (QoS) to different applications. For example, an application can simply register a huge amount of MRs to fill RNIC's internal memory and impact the performance of all other applications using the RNIC.

RDMA protects MR with *lkey*s and *rkey*s. However, such protection is not flexible. Each MR can only be registered with one permission, which is used by all applications to access the MR. To change the permission of an MR, it needs to be de-registered and registered again. Moreover, native RDMA relies on user applications to pass *rkey*s and memory addresses of MRs between nodes. Unencrypted *rkey*s and addresses can cause security vulnerability [155].

3.2 Virtualizing RDMA in Kernel: a Design Overview

> *"All problems in computer science can be solved by another level of indirection"* — often attributed to Butler Lampson, who attributes it to David Wheeler

The issues of native RDMA outlined in the previous section, namely 1) no high-level abstraction, 2) unscalable performance due to easily-overloaded on-RNIC memory, and 3) lack of resource management, are orthogonal to each other. However, they all point to the same solution: a virtualization and management layer for RDMA. Such a layer is crucial to make RDMA practical for datacenter applications. This section discusses why and how we add a kernel-level indirection, the challenges of adding such an indirection layer, and an overview of the design and architecture of LITE.

3.2.1 Kernel-Level Indirection

Many of RDMA's issues discussed in Section 3.1 have been explored decades ago, with different types of hardware resources. Hardware devices such as DRAM and disks expose low-level hardware primitives that are difficult and unsafe to use directly by

applications. Virtual memory systems and file systems solve these issues by virtualize, protect, and manage these hardware resources in the kernel space. We believe that we can use the same classic wisdom of *indirection* and *virtualization* to make native RDMA ready for datacenter application usage.

We propose to virtualize RDMA using a level of indirection in the kernel space. An indirection layer can transform native RDMA's low-level abstraction into a high-level, easy-to-use abstraction for datacenter applications. A kernel-level indirection layer can safely manage all privileged resources. It can thus move metadata and operations from hardware to software. Doing so largely reduces the memory pressure of RNICs and improves the scalability of RDMA-based datacenter applications that is currently bottlenecked by on-RNIC SRAM size. Moreover, a kernel indirection layer can serve both kernel-level applications and user-level applications.

3.2.2 Challenges

Building an efficient, flexible kernel-level RDMA indirection layer for datacenter applications is not easy. There are at least three unique challenges.

The biggest challenge is *how to preserve the performance benefits of RDMA while adding the indirection needed to support datacenter applications?*

Next, *how can we make LITE generic and flexible while delivering good performance?* Both LITE's abstraction and its implementation need to support a wide range of datacenter applications. LITE also needs to let applications safely and efficiently share resources. Unlike previous works [29, 30, 32, 33, 66], we cannot use abstractions or optimization techniques that are tailored towards a specific type of application.

Finally, *can we add kernel-level indirection without changing existing hardware, driver, or OS?* To make it easier to adopt LITE, LITE should not require changes to existing system software or hardware. Ideally, it should be contained in a stand-alone kernel loadable module.

3.2.3 LITE Overall Architecture

LITE uses a level of indirection in the kernel space to virtualize RDMA. It manages and virtualizes RDMA resources for all applications that use LITE (applications that do not want to use LITE can still access native RDMA directly on the same machine). LITE talks to RDMA drivers and RNICs using the standard Verbs abstraction. Figure 3.2 presents LITE's overall architecture. We implemented LITE as a loadable kernel module in the 3.11.1 Linux kernel with around 15K lines of code.

Overall, LITE achieves the following design goals.

- LITE provides a flexible and easy-to-use abstraction to a wide range of datacenter applications.
- LITE preserves RDMA's three performance benefits: low latency, high bandwidth, and low CPU utilization.
- LITE's performance scales better than native RDMA.
- LITE offers fine-grained and flexible protection.
- It is efficient to share RDMA resources and easy to isolate performance with LITE.
- LITE needs no hardware, driver, or OS changes.

LITE supports three types of interfaces: memory-like operations, RPC and messaging, and synchronization primitives, and it supports both kernel-level and user-level applications. We selected these semantics because they are familiar to datacenter application programmers. Most of LITE APIs have their counterparts in existing memory, distributed, and networking systems. Table 1 lists LITE's major APIs.

Internally, LITE consists of two main software stacks: a customized implementation of one-sided RDMA operations (Section 3.3), and a stack for RPC functions and messaging based on two-sided RDMA operations (Section 3.4). These parts share many resources, such as QPs, CQs, and LITE internal threads (Section 3.5).

Table 3.1.: **Major LITE APIs.** *Only the APIs in bold have their counterparts in native Verbs.*

	API	Explanation	Analogy
	LT_join	Start LITE and join cluster	
Memory	***LT_read***	RDMA read from space in an LMR	mem load
	LT_write	RDMA write to space in an LMR	mem store
	LT_malloc	allocate an LMR at node(s)	malloc
	LT_free	free an LMR and notify others	free
	LT_(un)map	open/close an LMR with name	m(un)map
	LT_memset	set space in an LMR with value	memset
	LT_memcpy	copy content from LMR to LMR	memcpy
	LT_memmove	move data from LMR to LMR	memmove
RPC/Msg	*LT_regRPC*	register an RPC func with an ID	RPC register
	LT_RPC	calls a remote RPC function with ID	RPC call
	LT_recvRPC	receives next RPC call with RPC ID	RPC receive
	LT_replyRPC	replies an RPC call with retsults	RPC return
	LT_send	send data to a remote node	send msg
Sync	*LT_(un)lock*	lock/unlock a distributed lock	pthread_lock
	LT_barrier	wait until a set of nodes reach barrier	pthread_barrier
	LT_fetch-add	atomically fetch data and add value	fetch&add
	LT_test-set	atomically test data and set value	test&set

Our security model is to trust LITE and not any users of LITE. For example, LITE disallows users from passing MR information themselves and thus avoids the security vulnerability of passing *rkey*s in plain text (Section 3.1.3). Improvement to our current security model is possible, for example, by reducing the physical memory space LITE controls or utilizing techniques like MPK [156]. We leave it for future work.

A LITE cluster consists of a set of LITE nodes, each running one instance of LITE. We provide a management library that manages the LITE cluster membership. It can run on one node or a high-availability node pair, and all the states it maintains can be easily reconstructed upon failure restart.

3.2.4 LITE Design Principles

Before our detailed discussion of LITE internals, we outline the design principles that help us achieve our design goals.

Generic layer with a virtualized, flexible abstraction. LITE provides a flexible, high-level abstraction that can be easily used by a wide range of applications. We designed LITE APIs to be a set of common APIs that datacenter applications can further build their customized APIs on top of. Specifically, we *offload* memory, connection, and queue management from applications to LITE. This minimal set of LITE APIs incorporate protection and moving them to the user space can cause security vulnerabilities or require hardware-assisted mechanisms that are not flexible [157, 158].

Avoid redundant indirection in hardware by onloading software-efficient functionalities. Using a kernel-level indirection does not necessarily mean adding one level of indirection. We remove the indirection that currently exists in hardware RNIC to avoid *redundant* indirection. Specifically, we *onload* two functionalities from hardware to LITE: memory address mapping and protection. We leave the rest of the RDMA stack such as hardware queues at RNIC. Removing these two functionalities from hardware not only eliminates the overhead of redundant indirection, but also minimizes hardware memory pressure, which in turn improves RDMA's scalability with respect to applications' memory usage (Section 3.1.2). Meanwhile, onloading only these two functionalities does not add much burden to the host machine and preserves RDMA's good performance, as we will see in Section 3.3.2.

Only adding indirection at the local side for one-sided operations. Direct remote memory accesses with one-sided RDMA eliminates CPU utilization at the remote node completely. To retain this benefit, we propose to only add an indirection layer at a local node. As we will show in Section 3.3, the local indirection layer is all that is needed to solve the issue of native one-sided RDMA operations.

Avoid hardware or driver changes. Removing hardware-level indirection without changing hardware is desired but is not easy. Fortunately, we identified a way to

interact with RNIC with physical memory addresses. Based on this mechanism, we propose a new technique to minimize hardware indirection without changing hardware (Section 3.3.1).

Hide kernel cost. We design several techniques to *hide* system call overhead by moving most of the overhead off performance-critical paths (Section 3.4.2). Unlike previous light-weight system call solutions [159], our approach does not require any change in existing OS. LITE avoids memory copy using address remapping and scatter-gather lists. Finally, LITE lets the sending-side application threads run to the end to avoid any thread scheduling costs.

The next few sections are organized as follows. Section 3.3 and Section 3.4 describe in detail LITE's one-sided RDMA and RPC stacks. Section 3.3 also presents a new virtualized memory abstraction LITE uses. Section 3.5 discusses LITE's resource sharing and QoS mechanisms. Section 3.6 briefly describes several extended functionalities we add on top of the RDMA and the RPC stacks in LITE.

All our experiments throughout the rest of the chapter were carried out in a cluster of 10 machines, each equipped with two Intel Xeon E5-2620 2.40GHz CPUs, 128 GB DRAM, and one 40 Gbps Mellanox ConnectX-3 NIC. A 40Gbps Mellanox InfiniBand Switch connects these machines' IB links.

3.3 LITE Memory Abstraction and RDMA

This section presents LITE's memory abstraction and its mechanism to support flexible one-sided RDMA operations. LITE manages address mapping and permission checking in the kernel and exposes a flexible, virtualized memory abstraction to applications. LITE adds a level of indirection only at the request sending side. Like native RDMA, one-sided RDMA operations with LITE does not involve any remote CPU, kernel, or LITE. But with the indirection layer at the local side, LITE can support more flexible, transparent, and efficient memory region management and access control.

3.3.1 LITE Memory Abstraction and Management

LITE's memory abstraction is based on the concept of LITE Memory Regions (*LMRs*), virtual memory regions that LITE manages and exposes to applications. An LMR can be of arbitrary size and can have different permissions to different users. Internally, an LMR can map to one or more physical memory address ranges. An LMR can even spread across different machines. This flexible physical location mapping can be useful for load balancing needs.

LMR handler. LITE hides the low-level information of an LMR (e.g., its location) from users and only exposes one entity — the LITE handler, or *lh*. *lh* can be viewed as a capability [160, 161] to an LMR that encapsulates both permission and address mapping. LITE allows users to set different types of permissions to different users, such as *read*, *write*, and *master* (we will explain master soon). In using *lh*, LITE provides the transparency that RDMA lacks. Native RDMA operations require senders to specify the target node, virtual address, and *rkey* of an MR. LITE hides all these details from senders behind the *lh* abstraction. Only an LMR's master knows which node(s) an LMR is on. *lh* is all that users need to perform LITE operations. However, an *lh* is meaningless without LITE and users cannot use *lh*s to directly access native MRs.

We let users associate their own "name" with an LMR, for example, a global memory address in a DSM system or a key in a key-value store system. Names need to be unique within a distributed application. Other users can acquire an *lh* of the LMR from LITE using this name via *LT_map*. This naming mechanism gives applications full flexibility to impose any semantics they choose on top of LITE's memory abstraction.

***lh* mapping and maintenance.** LITE manages the mapping from an *lh* to its physical memory address(es) and performs permission checking for each LMR before issuing native RDMA operations to RNICs. LITE maintains *lh* mappings and permissions at the node that accesses this LMR instead of at where the LMR resides,

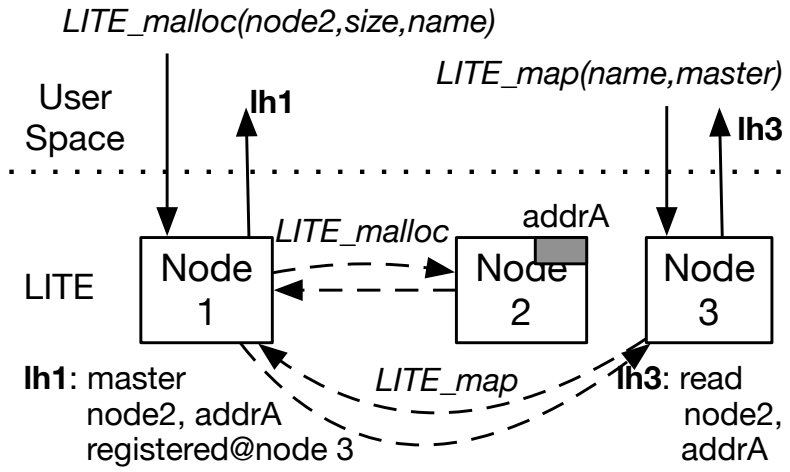


Figure 3.3.: **LITE *lh* Example.** *Node1 allocates an LMR from node2 and is the master of it. Node3 maps the LMR with read permission by contacting Node1.*

since we want to avoid any indirection at the remote node and retain RDMA's direct remote access. Figure 3.3 shows an example of using and managing *lh*s for an LMR.

An *lh* of an LMR is local to a process on a node; it is invalid for other processes or on other nodes. Unlike native RDMA which lets users pass *rkey* and MR virtual memory address across nodes to access an MR, LITE prevents users from directly passing *lh*s to improve security and to simplify LMR's usage model. All *lh* acquisition has to go through LITE using *LT_map*. LITE always generates a new *lh* for a new acquisition.

Master role. Although LITE hides most details of LMR such as its physical location(s) from users, it opens certain LMR management functionalities to a special role called *master*. The user that creates an LMR is its master. A master can choose which node(s) to allocate an LMR during *LT_malloc*. We also allow a master to register already-allocated memory as an LMR.

Master can move an existing LMR to another node. Master maintains a list of nodes that have mapped the LMR, so that when the master moves or frees (*LT_free*) the LMR, LITE at these nodes will be notified. The master role can use these functionalities to easily perform resource management and load balancing.

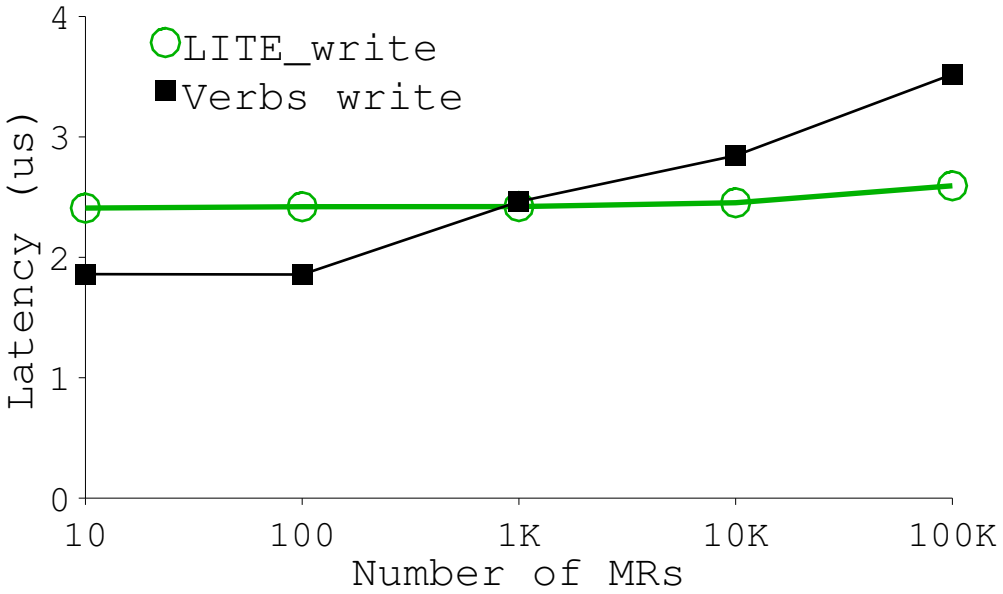


Figure 3.4.: **RDMA Write Latency against Num of (L)MRs.** *Each (L)MR is 4 KB. Each write is 64B and its location is randomly chosen from all (L)MRs.*

Only a master can grant permissions to other users. To avoid the allocator of an LMR being the performance bottleneck or the single point of failure, LITE supports more than one master of an LMR. A master role can grant the master permission to any other user.

Non-master map and unmap LMR. A user who wants to access an LMR first needs to acquire an *lh* by asking a master using *LT_map*. At the master node, LITE checks permission and replies the requesting node with the location of the LMR. LITE at the requesting node then generates a new *lh* and establishes its mapping and permissions. LITE stores all the metadata of an *lh* at the requesting node to avoid extra RTTs to master when users access LITE. After *LT_map*, users can perform LITE memory APIs in Table 3.1 using the *lh* and an offset. To unmap an LMR, LITE removes the user's *lh* and all its associated metadata and informs the master.

Avoiding RNIC indirection. With LITE managing LMR's address mapping and permission checking, we want to remove the *redundant* indirection in RNICs and reduce its memory pressure. However, without changing hardware, LITE can only

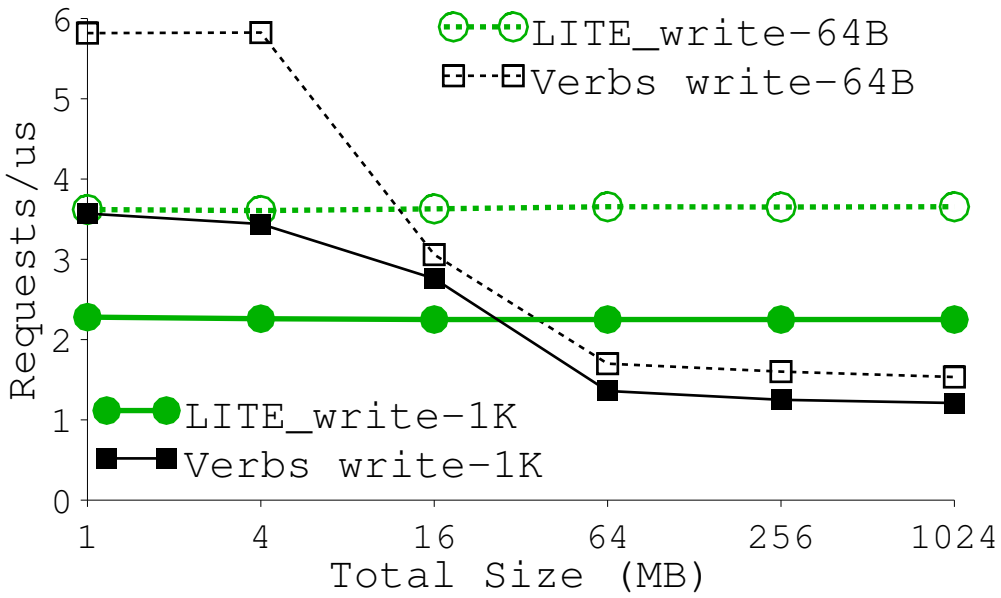


Figure 3.5.: **RDMA Write Throughput against (L)MR Size.** *Each run uses one MR. Each RDMA operation randomly writes 64B or 1KB.*

perform native RDMA operations with real MRs, which are mapped and protected in RNICs.

Fortunately, there is an infrequently used API that RDMA Verbs supports — the kernel space can register MRs with RNICs directly using physical memory addresses. LITE leverages this API to register only one MR with RNIC that covers the whole physical main memory. Using this global MR internally offers several benefits.

First, it eliminates the need for RNICs to fetch or cache PTEs. With native RDMA, an RNIC needs to map user-level virtual memory addresses to physical ones using their PTEs before performing an DMA operation. Since LITE registers the global MR with physical memory addresses, RNICs can directly use physical addresses without any PTEs. This technique largely improves LITE's performance scalability with LMR size (section 3.3.2).

Second, for the global MR, LITE registers only one *lkey* and one *rkey* with RNIC and uses the global *lkey* and *rkey* to issue all RDMA operations to RNIC. With only

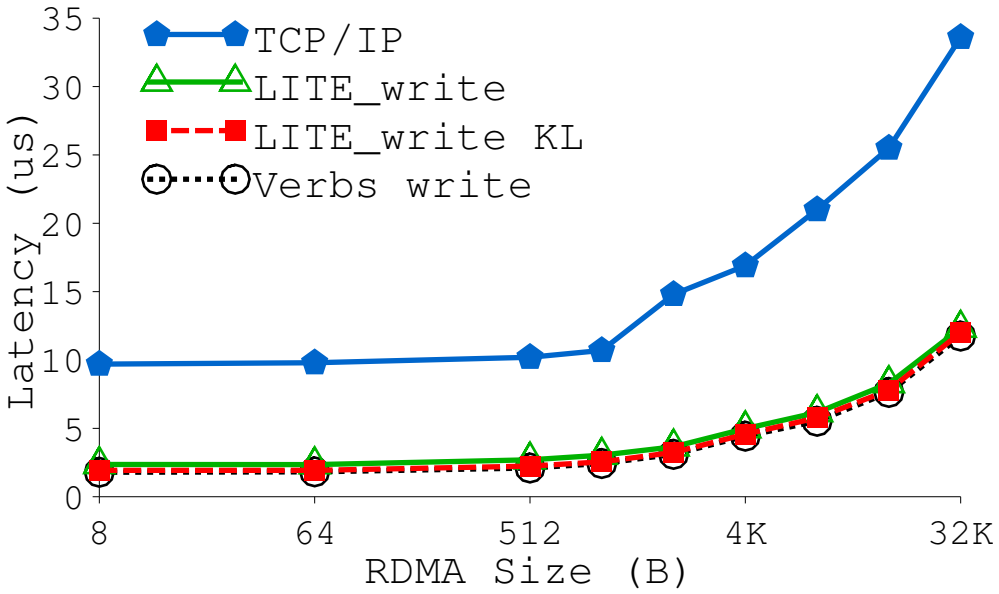


Figure 3.6.: **LITE and Native RDMA Write Latency.** *KL denotes kernel-level invocation. Lines without KL denotes user-level.*

one global *lkey* and *rkey* at the RNIC, LITE has no scalability issue with the amount of LMRs (section 3.3.2).

Finally, a subtle effect of using physical addresses is the avoidance of a costly memory pinning process during the creation of LMR. When an MR is created, native RDMA goes through all its memory pages and pins them in main memory to prevent them from being swapped out during RDMA operations [162,163]. In contrast, LITE does not need to go through this process since it allocates physical memory regions for LMRs.

However, using physical addresses to register a global MR with RNIC has one potential problem. LITE has to issue RDMA operations to the RNIC using physical memory addresses, and the memory region in an RDMA operation needs to be physically consecutive. These two constraints imply that each LMR also has to be physically consecutive. But allocating large physically consecutive memory regions can cause external fragmentation.

To solve this problem, we utilize the flexibility of the LMR indirection and spread large LMRs into smaller physically-consecutive memory regions. When a user performs a LITE read or write operation to such an LMR, LITE will issue several RDMA operations at the different physical memory regions. In our experiments, this technique scales well and has only less than 2% performance overhead compared to performing an RDMA operation on a huge physically consecutive region (e.g., 128 MB), while the latter will cause external fragmentation. When an LMR is small, LITE still allocates just one consecutive physical memory.

3.3.2 LITE RDMA Benefits and Performance

LITE executes one-sided *LT_read* and *LT_write* by issuing native one-sided RDMA read or write to RNICs after performing address translation and permission checking. Since LITE directly uses physical memory addresses to perform native RDMA operations, there is no need for any memory copy and LITE retains RDMA's zero copy benefit. LITE lets the requesting application thread run to the end and incurs no scheduling costs. Different from native RDMA read and write, *LT_read* and *LT_write* return only when the data has been read or written successfully. So users do not need to separately poll the completion status. LITE's one-sided RDMA achieves several benefits.

First, LITE's memory abstraction allows more flexibility in LMR's physical location(s), naming, and permissions. The LMR indirection also adds a high-degree of transparency, yet still letting masters perform memory resource management and load balancing.

Second, unlike previous solutions [29], LITE does not require any change in RDMA drivers or the OS and is implemented completely in a loadable kernel module.

Finally, LITE solves the performance scalability issues of native RDMA in section 3.1.2, while still delivering close-to-raw-RDMA performance when the scale is

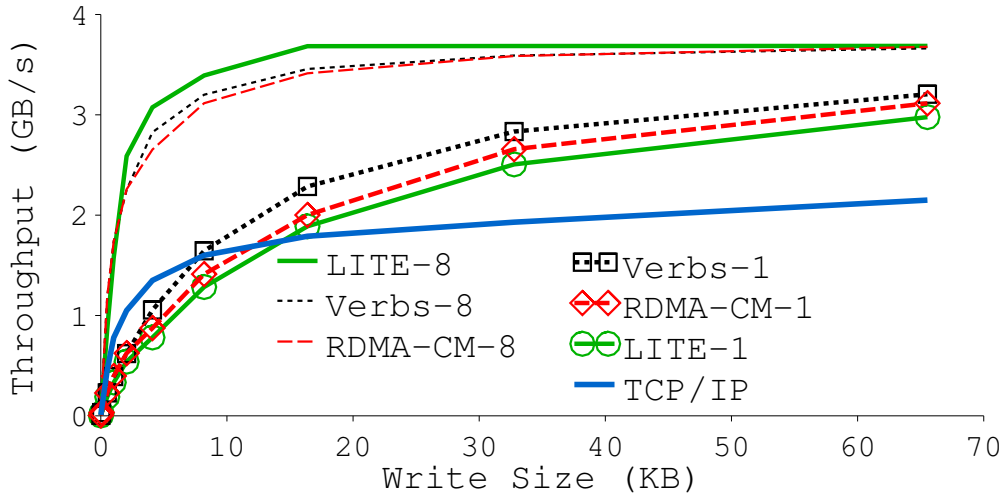


Figure 3.7.: **LITE RDMA Throughput.** *The number at the end of each line label represents the level of parallelism in request issuing. TCP/IP uses 1 thread with tcp_bw test in qperf.*

small. We expect datacenter environments to have large scale, making LITE a better performant choice than native RDMA.

Figure 3.4 presents the latency of *LT_write* and native RDMA write as the number of LMRs or MRs increases. Figure 3.5 shows the throughput of *LT_write* and native RDMA write as the size of an LMR or MR increases. Read performance has similar results. Native RDMA's performance drops quickly with the amount and the size of MRs, while LITE scales well with both the number of LMRs and the total size of LMR and outperforms native RDMA when the scale is big.

Figure 3.6 and Figure 3.7 take a closer look at the latency and throughput comparison of LITE, native RDMA write, and TCP/IP. We use *qperf* [164] to measure the performance of TCP/IP on InfiniBand (via IPoIB [165]). Even with a small scale, LITE's performance is close and sometimes better than native RDMA. LITE's kernel-level RDMA performance is almost identical to native RDMA, and its user-level RDMA has only a slight overhead over native RDMA. With more threads, LITE's throughput is slightly better than native RDMA's. TCP/IP's latency is always higher than both native RDMA and LITE, and TCP/IP's throughput is mostly lower than

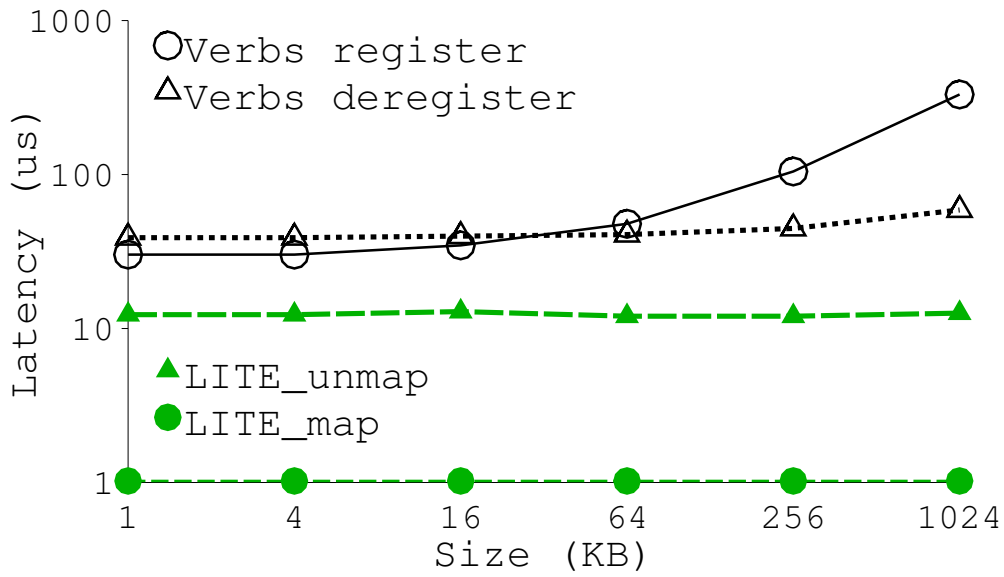


Figure 3.8.: **(De)Registering (L)MR Latency under LITE and native RDMA.** *This LT_map targets to a local LMR. Verbs register opens a MR to allow RDMA device to access this memory region. Verbs deregister remove the access permission and releases the resource of MR. LT_map opens an LMR with a name and register a memory space under the LMR. LT_unmap closes the LMR and release resources. LT_unmap includes RTT to communicate with all registered LITE nodes.*

them. When request size is between 128 B and 1 KB, TCP/IP's throughput is slightly higher than RDMA, because *qperf* executes requests in a non-blocking way, while our RDMA experiments run in a blocking way.

LITE's LMR register and de-register processes are also much faster and scale better with respect to MR size than native RDMA, as shown in Figure 3.8. As explained earlier in this section, native RDMA pins (unpins) each memory page of an MR in main memory during registering (de-registering), while LITE avoids this costly process.

3.4 LITE RPC

The previous section describes LITE's memory abstraction and basic memory-like operations in Table 3.1. In addition to one-sided RDMA and memory-like operations,

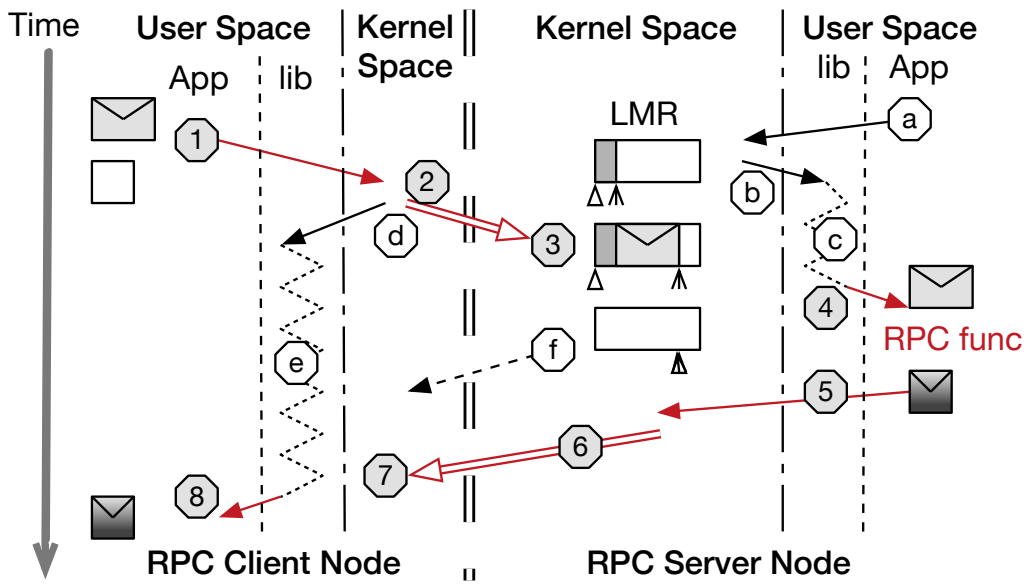


Figure 3.9.: **LITE RPC Mechanism.** *Red arrows represent the performance critical path of LT_RPC.*

LITE still supports traditional two-sided messaging. But the main type of two-sided operations we focus on is RPC. This section discusses LITE's RPC implementation and evaluation (second part of Table 3.1).

We believe that RPC is useful in many distributed applications, for example, to implement a distributed protocol, to perform a remote function, or simply to send a message and get a reply. We propose a new two-sided RDMA-based RPC mechanism and a set of optimization techniques to reduce the cost of kernel crossings during RPC. The LITE RPC interface is similar to traditional RPC [166]. Each RPC function is associated with an ID which multiple RPC clients can *bind* to and multiple RPC server threads can execute.

3.4.1 LITE RPC Mechanism

RDMA-write-imm-based Communication. We propose a new method to build RDMA-based RPC communication: using two RDMA *write-imm* operations. Write-imm is a Verb that is similar to RDMA write. But in addition to performing a direct

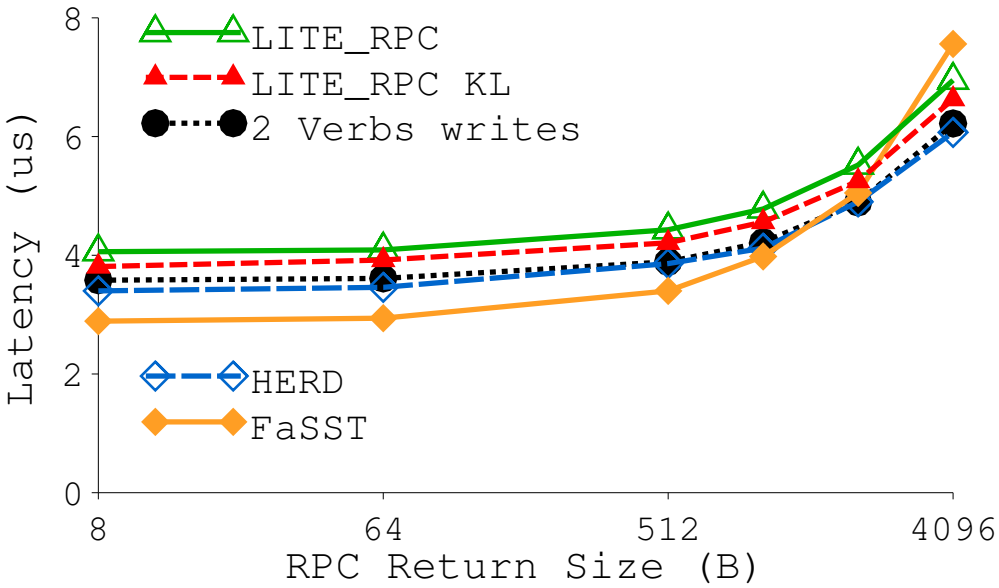


Figure 3.10.: **RPC Latency Comparison.** *The RPC input size is always 8B. FaRM uses two RDMA writes to implement message passing primitive. Sender uses one RDMA write to write to a buffer, and receiver polls the buffer to get message. We use two RDMA writes to emulate a RPC call in FaRM.*

write into remote memory, it also sends a 32-bit *immediate (IMM)* value and notifies the remote CPU of the completion of the write by placing an IMM entry in the remote node's receive CQ.

LITE performs one write-imm for sending the RPC call input and another write-imm for sending the reply back. LITE writes RPC inputs and outputs in LMRs and uses the 32-bit IMM value to pass certain internal metadata. To achieve low latency and low CPU utilization, LITE uses one shared polling thread to busy poll a global receive CQ for all RPC requests. LITE periodically posts IMM buffers in the receive queue in the background.

LITE RPC Process. When an RPC client node first requests to bind with an RPC function, LITE allocates a new internal LMR (e.g., of 16 MB) at the RPC server node. A header pointer and a tail pointer indicate the used space in this LMR. The client node writes to the LMR and manages the tail pointer, while the server node reads from the LMR and manages the header pointer.

Figure 3.9 illustrates the process of an *LT_RPC* call. The RPC client calls *LT_RPC* with function input and a memory space address for the return value ①. LITE uses write-imm to write the input data and the address of the return memory space to the tail pointer position of the LMR at the server node ②. LITE uses the IMM value to include the RPC function ID and the offset where the data starts in the LMR.

At the server node, a user thread calls *LT_recvRPC* to receive the next RPC call request. When the server node polls a new received request from the CQ, it parses the IMM value to obtain the RPC function ID and data offset in the corresponding LMR ③. LITE moves the data from the LMR to the user memory space specified in the *LT_recvRPC* call and returns the *LT_recvRPC* call ④. The LMR header pointer is adjusted at this time and another background thread sends the new header pointer to the client node ⓕ. The RPC server thread performs the RPC function after ④ and calls *LT_replyRPC* with the function return value ⑤. LITE writes this value to the client node at the address specified in *LT_RPC* using write-imm ⑥. The client node LITE returns the *LT_RPC* call ⑧ when it polls the completion of the write ⑦.

To improve *LT_RPC* throughput and reduce CPU utilization, we do not poll the sending state of any write-imm. LITE relies on the RPC reply (⑦) to detect any failure in the RPC process including write-imm errors; if LITE does not receive a reply within a certain period of time, it will return a time-out error to user. Thus, we can safely remove the check of sending states.

3.4.2 Optimizations between User-Space and Kernel

A straightforward implementation of the above LITE RPC process would involve three system call overhead (*LT_RPC*, *LT_recvRPC*, and *LT_replyRPC*) and six user-kernel-space crossings, costing $\sim 0.9\,\mu s$. We proposed a set of optimization techniques to reduce this overhead for LITE RPC. The resulting RPC process only incurs two user-to-kernel crossings ① ⑤, or $\sim 0.17\,\mu s$ in our experiments.

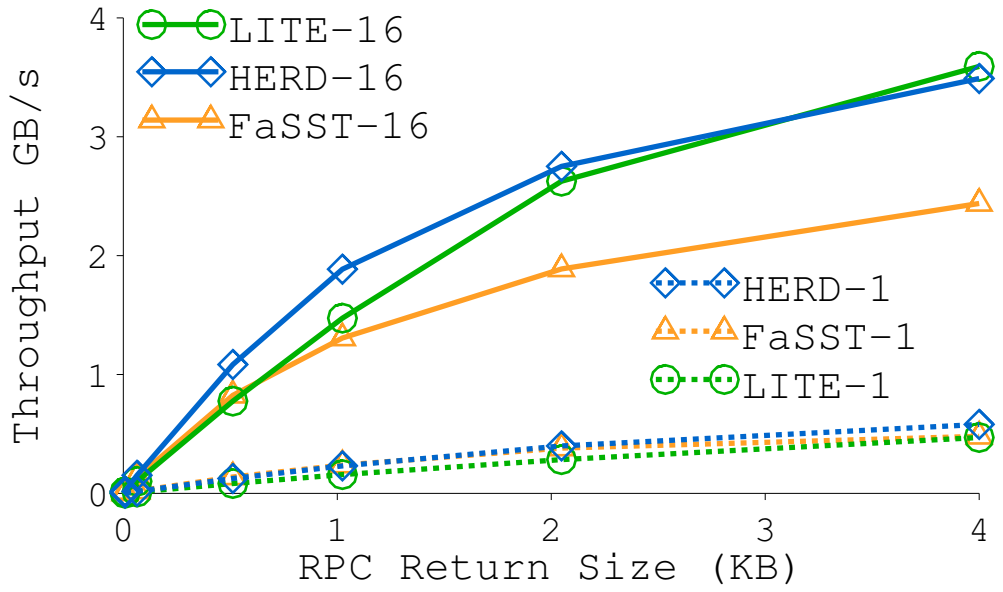


Figure 3.11.: **RPC Throughput.** *RPC throughputs using 16 and 1 concurrent RPC clients and servers. RPC input size is always 8 bytes.*

LITE hides the cost of returning a system call from the kernel space to the user space by removing this kernel-to-user crossing from the performance critical path. When LITE receives an *LT_RPC* or *LT_recvRPC* system call, it immediately returns to the user space without waiting for the results ⓑ ⓓ. But instead of returning the thread to the user, LITE returns it to a *LITE user-level library*.

We use a small memory space (one page) that is shared between kernel and an application process to indicate the ready state of a system call result, similar to the shared memory space used in previous light-weight system calls [159, 167]. When the LITE user-level library finds the result of a system call being ready, it returns the system call to the user ④ ⑧.

To further improve throughput, we provide an optional LITE API to send a reply and wait for the next received request. This API combines *LT_replyRPC* and *LT_recvRPC* in order to remove steps ⓐ and ⓑ. This interface is useful for RPC servers that continuously receives RPC requests.

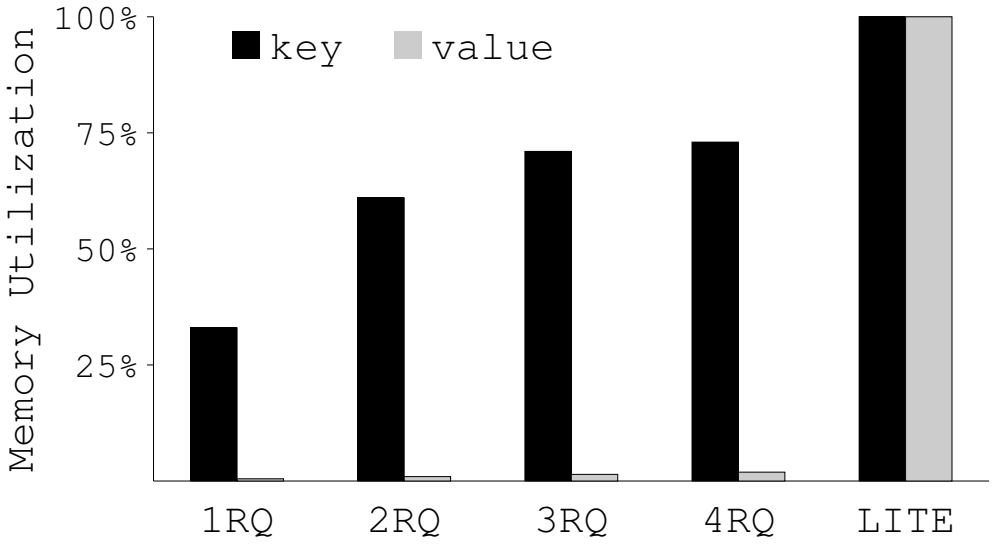


Figure 3.12.: **LITE RPC Memory Utilization.** *The first 4 set of bars represent send-based RPC with different number of varying-sized RQs.*

LITE minimizes CPU utilization when performing the above optimizations. Unlike previous solutions that require multiple kernel threads to reduce system call overhead [159] or require changes in system call interface [159, 168], LITE does not need any additional kernel (or user-level) threads or any interface changes. Furthermore, the LITE library uses an adaptive way to manage threads. It first tries to busy check the shared state. If it does not get any ready state shortly, LITE library will put the user thread to sleep and lazily checks the ready state (ⓒ ⓔ).

In addition to minimizing system call overhead, LITE also minimizes the cost of memory copying between user and kernel spaces. LITE avoids almost all memory copying by directly addressing a user-level memory buffer using its physical address (① ⑤ ⑧). LITE only does one memory move to move data from the LMR at the server node to the user-specified receive buffer. From our experimental results, doing so largely improves RPC throughput by releasing the internal LMR space as soon as possible.

3.4.3 LITE RPC Performance and CPU Utilization

LITE provides a general layer that supports RPC operations of different applications. LITE RPC offers low-latency, high-throughput, scalable performance, efficient memory usage, and low CPU utilization. Figures 3.10 and 3.11 demonstrate *LT_RPC*'s latency and throughput and how they compare to other systems.

The user-level *LT_RPC* has only a very small overhead over the kernel-level one. To further understand the performance implications of LITE RPC, we break down the total latency of one LITE RPC call. Of the total 6.95 μs spent on sending 8B key and return a 4 KB page in an *LT_RPC* call, metadata handling including mapping and protection checking takes less than 0.3 μs. The kernel software stacks of *LT_recvRPC* and *LT_replyRPC* take 0.3 μs and 0.2 μs respectively. The cost of user-kernel-space crossings is 0.17 μs.

To compare LITE with related efforts, we quantitatively and qualitatively compare it with several existing RDMA-based systems and their RPC implementations. FaRM [29] uses RDMA write as their message-passing mechanism. RPC could be implemented on top of FaRM with two RDMA writes. Since FaRM is not open-source, we directly compare LITE's RPC latency to the summation of two native RDMA writes' latency which is a lower bound but is not enough to build a real RPC operation. LITE has only a slight overhead over two native RDMA writes.

To evaluate the performance of LITE, we next compare LITE with two open-source RDMA-based RPC systems, HERD [32] and FaSST [66], using their open-source implementations. HERD implements RPC with a one-sided RDMA write for RPC call and one UD send for RPC return. HERD's RPC server threads busy check RDMA regions to know when a new RPC request has arrived. In comparison, LITE uses write-imm and polls the IMM value to receive an RPC request. The latency of checking one RDMA region is slightly faster than LITE's IMM-based polling (as with the microbenchmark results in Figure 3.10). However, in practice, HERD's mechanism does not work for our purpose of serving datacenter applications, since it

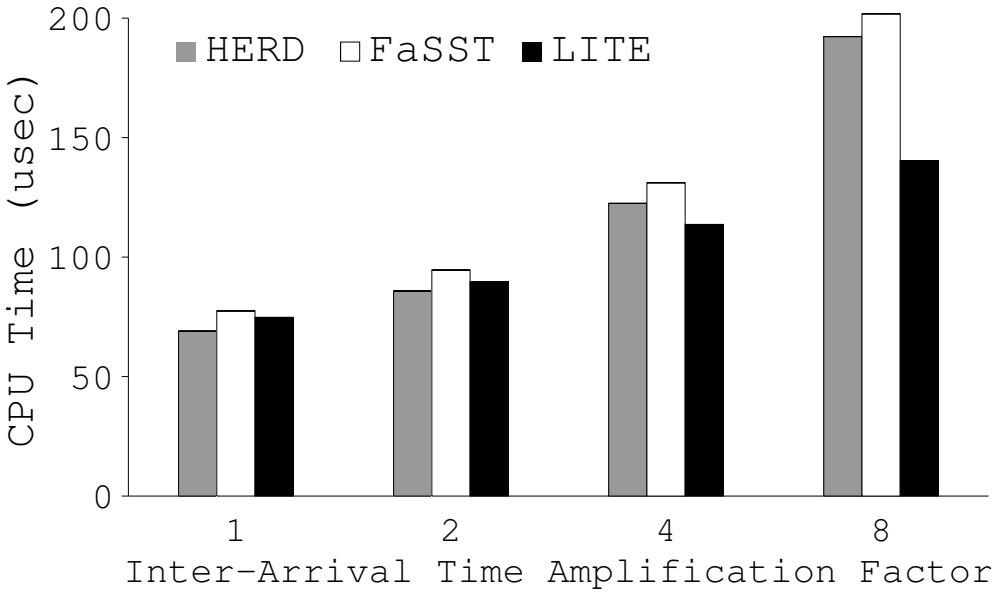


Figure 3.13.: **CPU Usage with Facebook Distribution.** *Average CPU time per request when changing the Facebook distribution inter-arrival time with an amplification factor.*

needs to busy check different RDMA regions for all RPC clients, causing high CPU or performance overhead. LITE only checks one receive CQ which contains the IMM values for all RPC clients.

FaSST is another RPC implementation using two UD sends. LITE has better throughput than FaSST and better latency when RPC size is big. FaSST uses a master thread (called coroutine) to both poll the receive CQ for incoming RPC requests and execute RPC functions. Our evaluation uses FaSST's benchmark which performs a dummy zero-length RPC function. In practice, executing an arbitrary RPC function in the polling thread is not safe and will cause throughput bottlenecks. LITE lets user threads execute RPC functions and ensures that the polling thread is lightweight.

Moreover, LITE is more space efficient than send-based RPC implementations. To use send, the receiving node needs to pre-post receive buffers that are big enough to accommodate the maximum size of all RPC data, causing huge amount of wasted

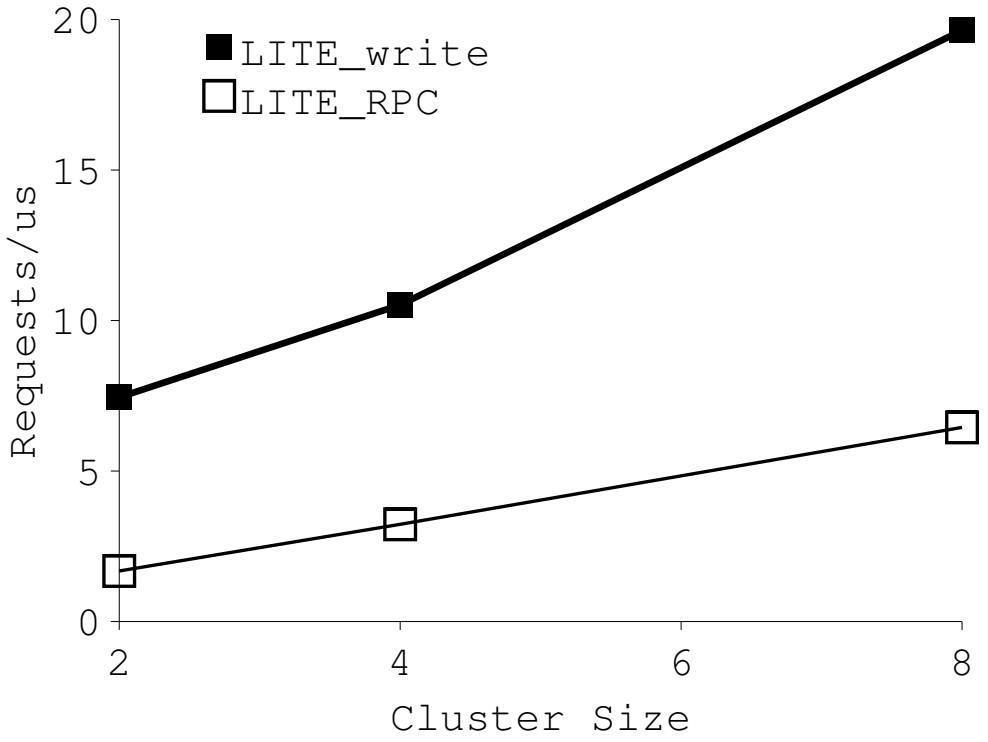


Figure 3.14.: **Scalability of LITE RDMA and RPC.** *Each node runs 8 threads. LT_write writes 64B. LT_RPC sends 64B and gets 8B reply.*

memory space [169]. In comparison, with write-imm, LITE does not need receive buffers for any RPC data. Figure 3.12 presents the memory space utilization with LITE RPC and *send* under the Facebook key-value store distributions [170]. For the send-based RPC in comparison, we already use a memory space optimization technique that posts receive buffers of different sizes on different RQs and chooses the most space-efficient RQ to send the data to [169]. Still, LITE is significantly more space-efficient than send-based RPC, especially for sending big data (values in the Facebook key-value store workload).

Finally, we evaluate the CPU utilization of LITE and compare it with HERD and FaSST. We first use a simple workload of sending 1000 RPC requests per second using 8 threads between two nodes. LITE's total CPU time is 4.3 seconds, while HERD and FaSST use 8.7 and 8.8 seconds.

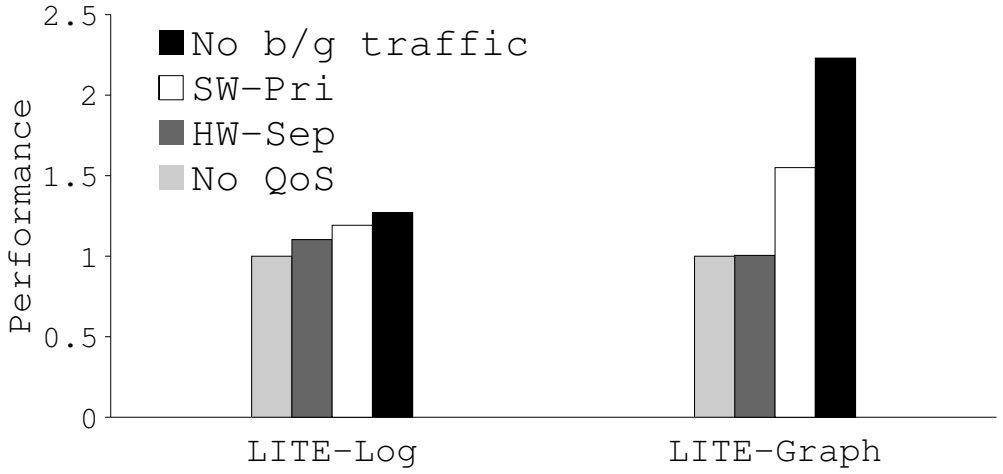


Figure 3.15.: **LITE QoS with Real Applications.** *No b/g traffic represents running LITE-Graph and LITE-Log without low-priority background traffic. No QoS is used as a baseline of performance (higher is better).*

We then implemented a macrobenchmark that performs RPC calls with inter-arrival time, input and return value sizes following the Facebook key-value store distribution [170]. In order to evaluate CPU utilization under different load, we multiply the inter-arrival time of the original distribution with a factor of 1× to 8×. Figure 3.13 plots the average CPU time per request of LITE, HERD, and FaSST when performing 100,000 RPC calls. When the workload is light (*i.e.*, larger inter-arrival time), LITE uses less CPU than both HERD and FaSST, mainly because of LITE's adaptive thread model that lets threads sleep. When the workload is heavy, LITE's CPU utilization is better than FaSST and is similar to HERD.

3.5 Resource Sharing and QoS

This section discusses how LITE shares resources and guarantees quality of service (QoS) across different applications. Our emphasis is to demonstrate the possibility and flexibility of using LITE to share resources and to perform QoS, but not to find the best policy of resource sharing or QoS.

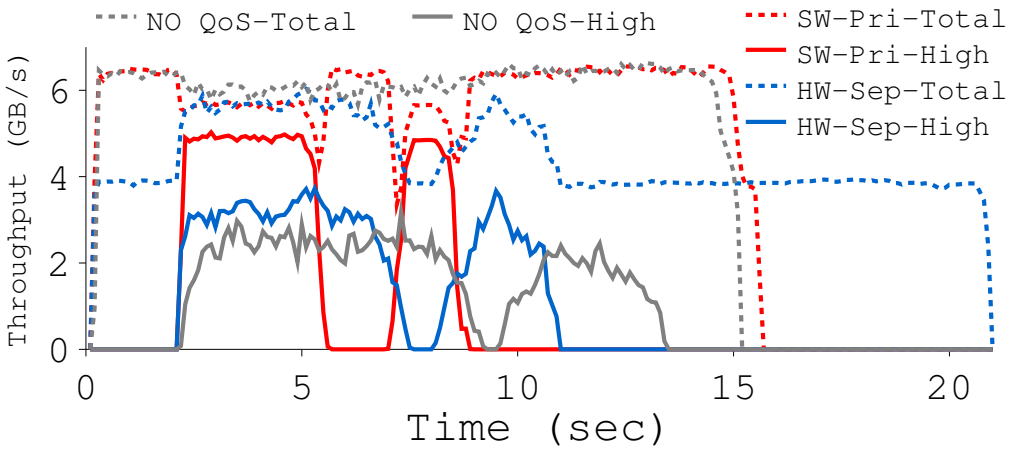


Figure 3.16.: **LITE QoS under Synthetic Workload.** *20 low-priority threads start from time 0, each running 600K requests (5 doing 4 KB LT_write, 5 doing 8 KB LT_write, 5 doing 4 KB LT_read, and 5 doing 4 KB LT_read). After 2 seconds, 20 high-priority threads join the system, each running 200K requests (10 doing 4 KB LT_write and 10 doing 4 KB LT_read). After finishing 200K requests, 8 of these 20 high-priority threads will sleep for 2 seconds and start running another 100K high-priority requests.*

3.5.1 Resource Sharing

LITE shares many types of resources across user applications and between LITE's one-sided RDMA and RPC components. In total, LITE uses $K \times N$ QPs and one busy polling thread for a shared receive CQ per node, where N is the total number of nodes. K is a configurable factor to control the tradeoff between total system bandwidth and total number of QPs; from our experiments $1 \leq K \leq 4$ gives best performance. Being able to share resources largely improves the scalability of LITE and minimizes hardware burden. In comparison, a non-sharing implementation on Verbs would need $2 \times N \times T$ QPs, where T is number of threads per node. FaRM [29] shares QPs within an application and requires $2 \times N \times T/q$ QPs, where q is a sharing factor. FaSST [66] uses T UD QPs; UD is unreliable and does not support one-sided RDMA operations. None of these schemes share QPs or polling threads across applications. Figure 3.14 shows that LITE one-sided RDMA and *LT_RPC* both scale well with number of nodes.

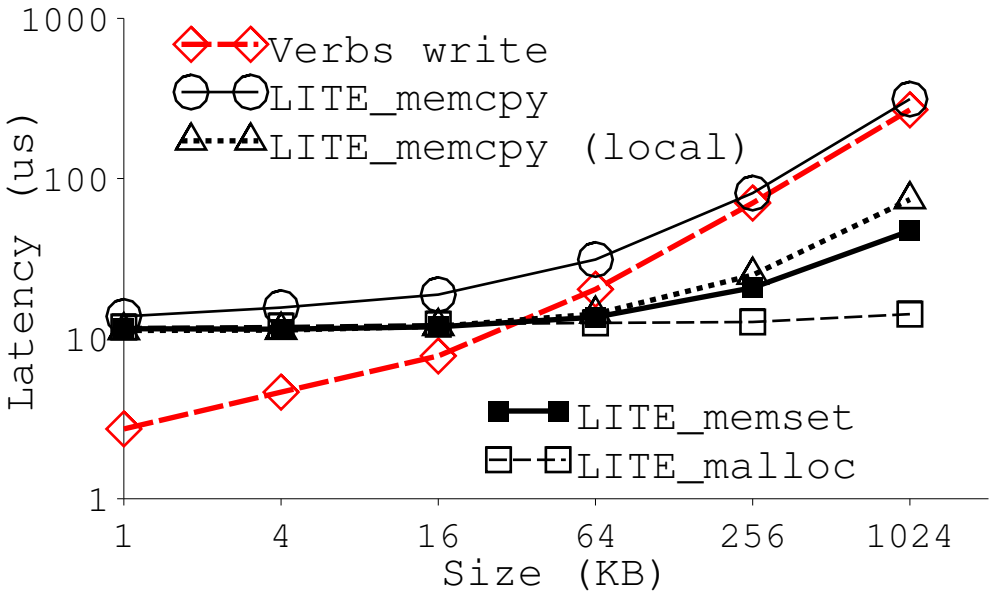


Figure 3.17.: **LITE Memory Operations Latency.** *LT_memcpy (local) represents LT_memcpy between two LMRs on the same node. LT_memmove is the same as LT_memcpy.*

3.5.2 Resource Isolation and QoS

When sharing resources, LITE ensures that user data is properly protected and that different users' performance is isolated from each other. We explored two approaches of delivering QoS. The first way (*HW-Sep*) relies on hardware resource isolation to achieve QoS. Specifically, HW-Sep reserves different QPs and CQs for different priority levels, *e.g.*, three QPs for high-priority requests and one QP for low-priority ones. Jobs under a specific priority can only use resources reserved for that priority.

The second approach (*SW-Pri*) performs priority-based flow and congestion control at the sending side using a combination of three software policies: 1) when the load of high-priority jobs is high, rate limit low-priority jobs (*i.e.*, reducing their sending speed); 2) when there is no or very light high-priority jobs, do not rate limit low-priority jobs; and 3) when the RTT of high-priority jobs increases, rate limit low-priority jobs. We chose these three policies to demonstrate that it is easy and flexible

to implement various flow-control policies with LITE; the first two policies are based on sending-side information and the last one leverages receiver-side information.

We evaluated HW-Sep and SW-Pri first with a synthetic workload that has a mixture of high-priority and low-priority jobs performing *LT_write* and *LT_read* of different request sizes. Figure 3.16 details this workload and plots the performance of HW-Sep, SW-Pri, and no QoS over time. As expected, without QoS, high-priority jobs can only use the same amount of resources as low-priority jobs, and thus are only able to achieve half of the total bandwidth. SW-Pri achieves high aggregated bandwidth that is close to the "no QoS" results, while being able to guarantee the superior performance of high-priority jobs. HW-Sep's QoS is worse than SW-Pri and its aggregated performance is the worst among the three. This is because low-priority jobs cannot use the resources HW-Sep reserves for high-priority jobs even when there is no high-priority jobs, thus limiting the total bandwidth achievable by HW-Sep. This result hints that a pure hardware-based QoS mechanism (HW-Sep) cannot provide the flexibility and performance that a software mechanism like SW-Pri offers.

Next, we evaluated LITE's QoS with real applications. We ran two applications that we built, LITE-Graph (Section 3.7.3) and LITE-Log (Section 3.7.1), with high priority, and a low-priority background task of constantly writing data to four nodes. Figure 3.15 compares the performance of LITE-Graph and LITE-Log under HW-Sep, SW-Pri, and no QoS. Similar to our findings from synthetic workloads, SW-Pri achieves better QoS than HW-Sep with real applications as well. Compared to LITE-Log, LITE-Graph is less affected by QoS because it is more CPU-intensive than LITE-Log.

3.6 Extended Functionalities

This section presents the implementation and evaluation of LITE's extended functionalities that we built on top of LITE's RDMA and RPC stacks. These function-

alities include memory-like operations that we did not cover in Section 3.3 (rest of the Memory part of Table 3.1) and synchronization operations (the Sync part of Table 3.1).

3.6.1 Memory-Like Operations

In addition to RDMA read and write, LITE supports a rich set of memory-like APIs that have similar interfaces as traditional single-node memory operations, including memory (de)allocation, set, copy, and move. To minimize network traffic, LITE internally uses its RPC interface to implement most LITE memory APIs. Figure 3.17 presents the latency of *LT_malloc*, *LT_memset*, *LT_memcpy*, and *LT_memmove* (Table 3.1) as LMR size grows.

Applications call these memory-like APIs using *lh*s, in a similar way as how they call POSIX memory APIs using virtual memory addresses. For example, applications specify a source *lh* and a destination *lh* to perform *LT_memcpy* and *LT_memmove*. LITE implements *LT_memcpy* and *LT_memmove* by sending an *LT_RPC* to the node that stores the source LMR. This node will perform a local *memcpy* or *memmov* if the destinate LMR is at the same node. Otherwise, it will perform an *LT_write* to the destinate LMR which is at a different node. Afterwards, this node will reply to the requesting node, completing the application call.

LITE implements *LT_memset* by sending a command to the remote node that stores the LMR, which performs a local *memset* and replies. An alternative way to implement *LT_memset* is to perform an *LT_write* to the MR with the value to be set. This alternative approach is worse than our implementation of *LT_memset* when the LMR size grows, since it sends more data over the network.

3.6.2 Synchronization and Atomic Primitives

LITE provides a set of synchronization and atomic primitives, including *LT_lock*, *LT_unlock*, *LT_barrier*, *LT_fetch-add*, and *LT_test-set* (Table 3.1). The last two are

direct wrappers of their corresponding native Verbs. We added distributed locking and distributed barrier interfaces to assist LITE users in performing distributed coordination.

We used an efficient implementation of LITE locking that balances lock operation latency and network traffic overhead. A LITE lock is simply a 64-bit integer value in an internal LMR and each lock has an owner node. The *LT_lock* operation first uses one *LT_fetch-add* to try to acquire the lock. If a lock is available, this acquiring process is very fast (2.2 μs in our experiment). Otherwise, LITE will send an *LT_RPC* to the owner of the lock who maintains a FIFO queue of all users waiting on the lock. By maintaining a FIFO wait queue, LITE minimizes network traffic by only waking up and granting a lock to one waiting user once the lock is available. Our experiment shows that LITE lock scales well with number of contending threads and nodes.

3.7 LITE Applications

To demonstrate the ease-of-use, flexibility, and superior performance of LITE, we developed four datacenter applications on top of LITE. This section describes how we built or ported these applications and their performance evaluation results. We summarize our experience in building applications on LITE at the end of this section.

3.7.1 Distributed Atomic Logging

LITE-Log is a simple distributed atomic logging system that we built using LITE's memory APIs. With LITE-Log, we push the "one-sided" concept to an extreme: the creation, maintenance, and access of a *global log* are performed all from remote. This one-sided LITE-Log has complete transparency to its users and is very easy to use.

An *allocator* creates a global log as an LMR and several metadata variables also as LMRs using *LT_malloc*. A set of *writers* commit transactions to the log, and a log *cleaner* periodically cleans the log. The same node can run more than one role.

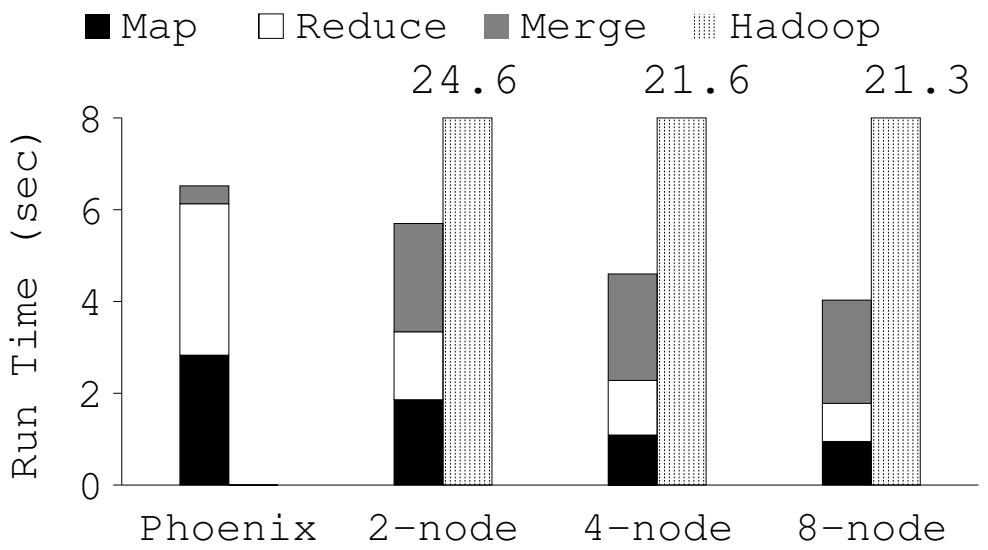


Figure 3.18.: **MapReduce Performance.** *The left bars in the 2,4,8-node groups are LITE-MR.*

Writes to the log (log entries) are buffered at a local node until a *commit* time. At commit time, the writer first reserves a consecutive space in the log for its transaction data by testing and adjusting metadata of the log using *LT_fetch-add*. The writer then writes the transaction with *LT_write* to the reserved log space. The log cleaner runs in thebackground to clean up the global log with the help of *LT_read*, *LT_fetch-add*, and *LT_test-set*.

Our experiments show that LITE-Log can achieve 833K transaction commit per second when two nodes concurrently commit single-entry (of 16B) transactions and LITE-Log's transaction commit throughput scales with number of nodes and size of transaction.

3.7.2 MapReduce

LITE-MR is our implementation of MapReduce [171] on LITE. We ported LITE-MR from Phoenix [146], a single-node multi-threaded implementation of MapReduce. We spread the original Phoenix mapper and reducer threads onto a set of worker nodes

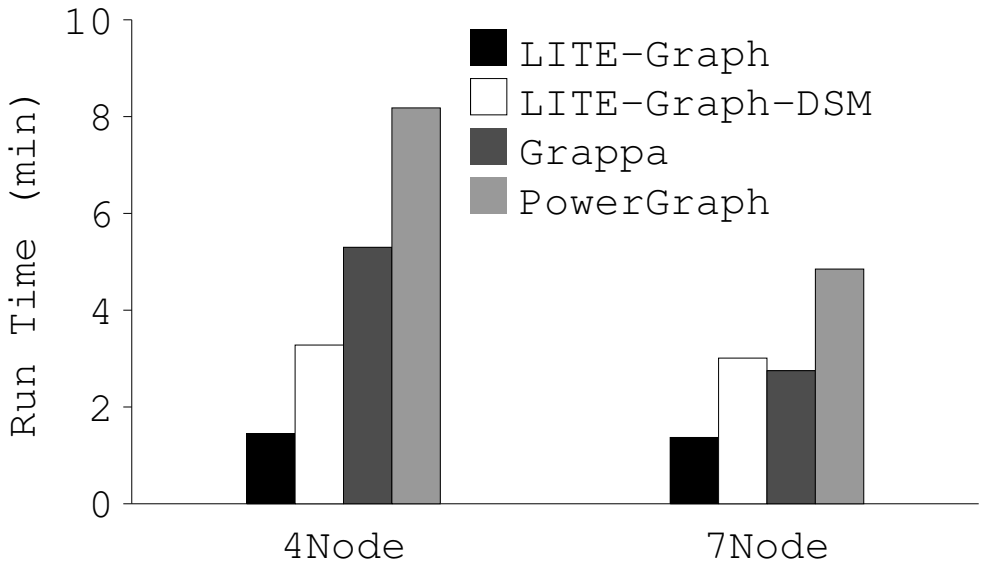


Figure 3.19.: **PageRank Performance.** *Each node runs four threads.*

and use a separate node as the master node. The master enforces the original Phoenix job splitting policy but splits tasks to multiple worker nodes. We implemented LITE-MR's network communication using *LT_read* and *LT_RPC*.

Same as MapReduce, LITE-MR uses three phases: map, reduce, and merge. In the map phase, each worker thread performs map tasks assigned by the master. After completing all map tasks, a worker thread combines all the intermediate results into a set of finalized buffers. It then registers one LMR per finalized buffer with an identifier and sends all the identifiers to the master.

In the reduce phase, the master sends the identifiers collected in the map phase to the reduce worker threads. The worker threads use these identifiers to directly read the map results from the mapper nodes using *LT_read*. After completing all reduce tasks, a reduce worker thread combines the results of all its tasks. It registers the combined buffer with one LMR and reports its associated identifier to the master. The merge phase works in a similar way as the reduce phase; each merge worker threads read reduce results with *LT_read*. At the end of Merge phase, the master node reads the final results with *LT_read* and reports the results to the user.

Figure 3.18 presents the WordCount run time of the Wikimedia workloads [172] using Phoenix, LITE-MR, and Hadoop [4]. For all the schemes, we use the same number of total threads. Phoenix uses a single node, while LITE-MR and Hadoop use 2, 4, and 8 nodes. LITE-MR outperforms Hadoop by 4.3× to 5.3×. We run Hadoop on IPoIB, which performs much worse than LITE's RDMA stack.

Surprisingly, with the same amount of total threads, LITE-MR also outperforms Phoenix even though LITE involves network communication and Phoenix only accesses shared memory on a single node. We break down Phoenix and LITE-MR's run time into different phases and found that LITE-MR's map and reduce phases are shorter than Phoenix's. In these phases, only reducers read data from mappers. We made a simple change to modify Phoenix's global tree-structured index to a per-node index to run LITE-MR on distributed nodes. The gain of this change is larger than the cost of network communication in LITE. However, using the same split index in Phoenix affects Phoenix's multicore optimizations for local threads. LITE-MR's merge phase performs worse than Phoenix's because all data to be merged is on a single node with Phoenix while they are on different nodes with LITE-MR. Both LITE-MR and Phoenix use 2-way merge and requires multiple rounds of reading and writing data. However, this cost is the result of performing distributed merging, not because of using LITE. Finally, LITE-MR performs better with more nodes, because the total number of LMRs stay the same but the cost of mapping and unmapping LMRs is amortized across more nodes.

3.7.3 Graph Engine

We implemented a new graph engine, LITE-Graph, based on the design of PowerGraph [6]. Like PowerGraph, it organizes graphs with vertex-centric representation. It stores the global graph data in a set of LMRs and distributes graph processing load to multiple threads across LITE nodes. Each thread performs graph algorithms

on a set of vertices in three steps: gather, apply, and scatter, with the optimization of delta caching [6].

After each step, we perform an *LT_barrier* to only start the next step when all LITE nodes have finished the previous step. At the scatter step, LITE-Graph uses *LT_read* and *LT_write* to read and update the global data stored in LMRs. To ensure consistency of the global data, we can only allow one write at a time. LITE-Graph uses *LT_lock* to protect the update to each LMR. With this implementation, splitting the global data into more LMRs can increase parallelism and the total throughput of LITE-Graph.

We perform PageRank [150] on the Twitter dataset (1 M vertices, 1 B directed edges) [173] using LITE-Graph, PowerGraph, and Grappa [25]. Grappa is a DSM system that uses a customized IB-based network stack to aggregates network requests. PowerGraph uses IPoIB on InfiniBand. Figure 3.19 shows the total run time of these systems using four nodes and seven nodes, each node running four threads. Compared to PowerGraph and Grappa, LITE-Graph performs significantly better, mostly due to the performance advantage of LITE's stack over IPoIB and Grappa's networking stack.

3.7.4 Kernel-Level DSM

LITE-DSM is a kernel-level DSM system that we built in Linux on LITE. It supports multiple concurrent readers and a single writer (MRSW) and the release consistency level (using two operations to *acquire* and *release* a set of data for writing). LITE-DSM designates a home node for each memory page like HLRC DSM systems [84, 174]. Currently, it assigns home node in a round robin fashion. At releasing time, dirty data is pushed to the home node, which informs all nodes that have a cached copy to invalidate the data.

LITE-DSM hides all its operation and the globally shared memory space from users by intercepting the kernel page fault handler to perform remote operations if

needed. Users on a set of nodes open LITE-DSM by first agreeing on a range of reserved global virtual addresses that are the same on all nodes using *LT_RPC*.

A remote page read in LITE-DSM does not need to inform the home node, since multiple readers can read at the same time. Thus, LITE-DSM uses the one-sided *LT_read* to perform a remote page read. The acquire and release operations both involve distributed protocols to invalidate or update data and metadata. Thus, we use *LT_RPC* to implement LITE-DSM protocols to exchange as much information as possible in a single round trip.

In building these distributed protocols, we found the need of a multicast function [61, 175, 176]. We extended LITE APIs to include a new API that sends RPC to multiple RPC server machines. Since multicast is not our focus, we use a simple implementation by generating concurrent *LT_RPC* requests to the destinations and replying to the RPC client after all the desinations reply.

We evaluated LITE-DSM with sequential and random read, write, and sync operations on four machines. As expected, reads have the lowest latency, 12.6 μs and 17.2 μs for random and sequential 4 KB read. Sync is more costly, taking 9.2 μs and 74.3 μs to begin and commit 10 dirty 4 KB pages.

We further built a user-space graph engine, LITE-Graph-DSM, on top of LITE-DSM using a similar design as LITE-Graph. LITE-Graph-DSM performs native memory loads and stores in the distributed shared memory space provided by LITE-DSM instead of LITE memory operations. As shown in Figure 3.19, LITE-Graph-DSM's performance is worse than LITE-Graph because of the overhead caused by the additional DSM layer. LITE-Graph-DSM still outperforms PowerGraph significantly and is similar or better than Grappa.

3.7.5 Programming Experience

Overall, we find LITE very simple to use and it provides all the network functionalities that our applications need. Table 3.2 lists the lines of code (LOC) and

Table 3.2.: **LITE Application Implementation Effort.** **LITE-MR ports from the 3000-LOC Phoenix with 600 lines of change or addtion.*

Application	LOC	LOC using LITE	Student Days
LITE-Log	330	36	1
LITE-MR	600*	49	4
LITE-Graph	1400	20	7
LITE-DSM	3000	45	26
LITE-Graph-DSM	1300	0	5

graduate student days to implement the applications on LITE. Most of the code and implementation efforts are on the applications themselves instead of on LITE. There is no need for any other networking code apart from the use of LITE APIs. In comparison, we spent 4 months and 4500 LOC building an RDMA stack and optimizing it for our previous in-house DSM system.

Using LITE requires no expert knowledge in RDMA. LITE-Log and LITE-MR were built by the same student that built LITE, while the rest were built by one who has no knowledge about LITE internals. They were able to build applications at similar speed and ease.

Overall, we find LITE's abstraction very flexible. For example, the "name" LITE-Graph associates with its LMR is the vertex index, and the "name" LITE-DSM uses is the global virtual memory address in DSM. In general, LITE's memory APIs are a good fit for accessing data fast and its synchronization APIs are helpful in offering synchronized accesses. Building applications using these APIs is to a large extent similar to building a single-machine shared-memory application. LITE RPC is a better fit for exchanging metadata and implementing complex distributed protocols.

After choosing the right LITE APIs (e.g., memory vs. RPC), optimizing LITE-based application performance is easy and mostly does not involve networking optimizations. For example, LITE-Graph improves performance by using finer-grained LITE locks to increase parallelism, the same concept commonly used in multi-threaded applications.

Finally, it is easy to run multiple applications together on LITE, while each application's implementation process involves no other applications.

3.8 Moving LITE to User Space

LITE was designed for general datacenter usages; it provides scalability, flexibility, ease-of-programming, safe sharing, and quality of service. Our decision of building LITE as a kernel-level indirection layer fits applications that require stronger security guarantees. However, it comes with two tradeoffs. First, it is inevitable to incur the latency overhead caused by kernel trap. Second, maintaining and managing kernel modules is more difficult than user-level code.

After developing LITE in the kernel space, we explored options to migrate it to the user space for applications that are latency sensitive and can accept weaker security guarantees. Specifically, we created a user-level version of LITE called *U-LITE* that maintains most of LITE's high-level abstraction and scalability benefits.

Applications use U-LITE by including it as a library. U-LITE registers a big memory space with memory huge page during initialization. It also manages memory spaces and address mappings for LMRs. Currently, we do not support RPC with U-LITE or resource sharing across different application processes, and leave them for future work.

3.9 Conclusion

We presented LITE, a Local Indirection TiEr in the OS to virtualize and manage RDMA for datacenter applications. LITE solves three key issues of native RDMA when used in the datacenter environments: mismatched abstraction, unscalable performance, and lack of resource management. LITE demonstrates that using a kernel-level indirection layer can preserve native RDMA's good performance, while solving its issues. We performed extensive evaluation of LITE and built four datacenter applications on LITE. Overall, LITE is both easy to use and performs well.

4 RDMA-BASED DATA STORE WITH REMOTE PERSISTENT MEMORY

The previous chapter presented LITE, an indirection tier that virtualizes and manages RDMA for datacenter applications. LITE largely improves the usability and scalability of RDMA, while preserving RDMA's good performance and providing means for safe RDMA resource sharing. A natural question that follows is *how to build applications on top of the (improved) RDMA abstraction*. While the previous chapter described our efforts in porting several different applications to LITE, this chapter focuses on one important type of system in datacenters — in-memory data stores and presents our efforts in building RDMA-based in-memory data stores.

Many datacenter applications use in-memory data stores such as in-memory key-value stores [29, 31, 33], databases [26, 177], storage systems [92], and caching [91] for fast data accesses. With in-memory data stores being a critical system layer in datacenters, various research efforts have been put into building them. However, most of the existing work focus on performance and optimization, ignoring two important system design factors: monetary costs and manageability.

We set our target to study a different direction: build a low-cost, scalable data stores. Two main factors contribute to the owning and running cost (CapEx and OpEx) of existing in-memory data stores. First, large DRAM used to store data is expensive and consumes a significant amount of energy. Second, all in-memory data stores that we know of require processing power at the server [29, 31, 33, 92] or hardware [64, 178, 179] that host the data, adding both CapEx and OpEx to datacenters.

To cut the cost of DRAM, we adopt the upcoming persistent memory (*PM*) [180, 181], which offers 60% lower energy and scales 2-4× than DRAM besides being non-volatile [102, 103, 182]. To cut the cost of processing power, we exploit one-sided network communication of RDMA. RDMA allows one machine to access another machine's memory without involving the latter's CPU.

One-sided RDMA accesses can eliminate CPU utilization completely. Several recent in-memory data stores adopted one-sided RDMA read operations for read accesses, but their write accesses and data management still require processing power. We explore the possibility of removing processing units entirely from the memory nodes and perform *all* access and management operations from remote.

The resulting system treats PM as a passive, or "dumb", media to store data (persistently), and we call it *dumb persistent memory*, or *DPM*. DPM can be constructed with DIMM-attached PM [180, 181] and an RNIC in a regular server or in a *disaggregated* way [38, 183] with a device to host PM and an RNIC. Being able to remove processing needs in DPM can free up a server's CPU to perform other tasks or can make a disaggregated device easy and cheap to build. In addition, DPM can improve the manageability of a data store by controlling it in a centralized manner.

DPM is a cost-effective system because DPM only needs a network interface, a hardware PM controller, and PM; it requires no server packaging or any processors. Datacenters owners can use normal servers as compute nodes, or CN, and store data in DPM.

However, removing processing units from where data is hosted is not easy. Existing in-memory data stores heavily rely on processing power for both the data path and the control path. Without any processing power, accesses to DPMs have to come all from the network, which makes concurrent writes especially hard; DPMs cannot perform any management tasks of its own memory resources; and each DPM can fail independently.

To confront the unique challenges of DPM and to explore the design tradeoffs of the DPM model, this chapter presents three architectures of organizing DPMs (Figure 4.1(c) to 4.1(e)).

Our core design philosophy across these different DPM models is to build concurrent data stores that target read-most workloads where concurrent writes can happen but not often. This is a common workload pattern in many datacenter environments [170]. We demonstrate that for read-most workloads, it is possible to remove

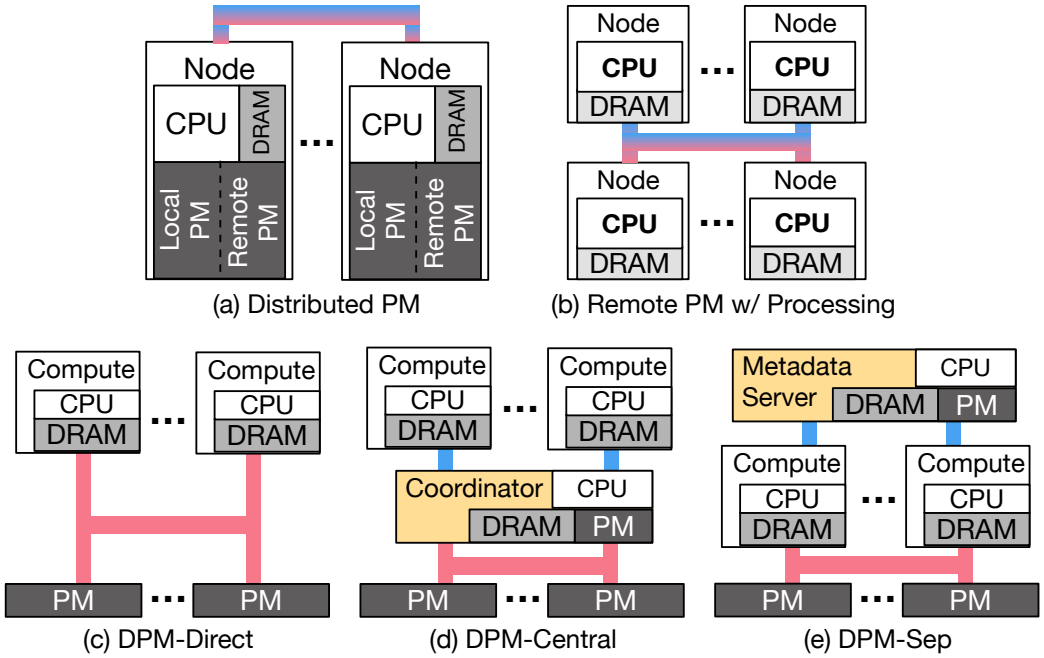


Figure 4.1.: **PM Organization Comparison.** *Blue bars indicate two-way communication and pink ones indicate one-way communication. Bars with both blue and pink mean support for both.*

processing units all together while preserving the same performance, but doing so requires new designs in both the data path and the control path.

We start our exploration of DPM organizations with a simple architecture, *DPM-Direct*, where compute nodes (*CN*s) directly access DPMs with one-sided operations. The second architecture, *DPM-Central*, uses a central server (the *coordinator*) to orchestrate the accesses from CNs to DPMs and to manage DPM resources. CNs can talk to the coordinator with two-sided communication and the coordinator accesses DPMs with one-sided communication. Our third architecture, *DPM-Sep*, separates the data plane from the control plane. CNs directly access DPMs for all data operations. We use a metadata server (*MS*) to handle all metadata and control operations. CNs talk to the MS with two-sided communication.

Based on these three architectures, we designed three atomic, crash-consistent DPM data store systems. All these systems provide the same guarantees that when writing to a data entry, the data entry either has all new data (if the write is success-

fully committed) or all old data (if the write fails), and that CNs only read committed data (*i.e.*, the read-committed isolation level). These properties hold even when a DPM crashes during the write and recovers afterwards.

With the DPM-Direct architecture, all data and control paths have to be conducted by CNs in a distributed manner. We design a data store, *DirectDS*, for optimized read performance and minimal management/control tasks. To avoid space allocation on each write, DirectDS assigns two spaces for each data entry during creation time, one to write uncommitted new data and one to store committed data. DirectDS uses error detection code to achieve single network round trip (RTT) reads and use a customized distributed locking mechanism for writes.

On top of DPM-Central, we propose *CentralDS*. CentralDS leverages the centralized coordinator to perform space management, to store metadata, and to serve as the serializing point for concurrent accesses. CNs send RPC read/write requests to the coordinator, which uses local locks to protect concurrent accesses, reads/writes data to DPM, and updates metadata locally.

On top of DPM-Sep, we propose *SepDS*. On the data path, SepDS performs out-of-place writes that are similar to log-structured writes. We use a novel data structure that enables CNs to efficiently locate the latest data entry without the need to communicate with the MS. This design achieves 1-RTT read performance when there is no concurrent writes, while ensuring correct concurrent writes with satisfactory performance. We move all metadata and control operations off performance critical path. The MS stores all metadata and CNs caches hot metadata. To minimize the need for CNs to communicate with MS, CNs inform the MS to reclaim old data entries in a lazy, batched manner. We also *completely* eliminate the need for the MS to communicate with DPMs; it manages DPM space and performs control tasks like load balancing without accessing them.

To sustain non-transient DPM failures and to provide high availability, it is not enough to store data in just one DPM. For each of the three systems, we add the support of data and metadata replication. We also leverage the data redundancy to

provide better load balancing — we dynamically choose which DPM to write/read data replicas based on the loads of each DPM.

We evaluate the three DPM data stores using real servers as CNs, the coordinator, the MS, and DPM devices, all connected with RDMA network, and we emulate PM with DRAM. We perform a systematic, extensive set of experiments to evaluate the latency, throughput, scalability, CPU utilization, and the CapEx and OpEx of the three DPM data stores using microbenchmarks and YCSB workloads [184, 185].

Our evaluation results demonstrate that we can achieve the same or better performance and scalability as traditional distributed and remote non-DPM in-memory stores (Figure 4.1(a) and Figure 4.1(b)) when reducing OpEx by 30% to 99% and CapEx by 75%. Our evaluation also provides a detailed tradeoff analysis of the three DPM architectures and reveals pitfalls in designing DPM systems. For example, while DPM-Central delivers great concurrent write performance and simplifies the control plane, its read performance and CapEx/OpEx are the worst among the three DPM designs.

Overall, DPM is a new model of low-cost in-memory data stores using dumb persistent memory and exploiting one-sided network communication of RDMA. DPM offers several key benefits:

- Without the need for a processor or a server to host DPM, the monetary and energy cost of DPM is low.
- The DPMmodel is designed for datacenter workload — read-most workload.
- Unlike the alternative approach of attaching PM to a server, DPMs can be integrated into current datacenters without any disruption to existing servers.
- Multiple compute nodes can share one DPM device and one compute node can store data on multiple DPMs. Doing so enables better resource packing than attaching and confining the usage of PM to a single node.

- The DPM model offers great elasticity since DPMs can be freely added, removed, and replaced. The amount of compute nodes and DPMs can scale *independently*.

4.1 DPM Overview

Before presenting the design of DPM data stores, we first give an overview of existing distributed memory systems, DPM, and the three architectures we propose.

Traditional distributed storage or distributed memory systems can be categorized into two types of models as discussed in Section 2.2.1. The first model is *distributed memory* (Figure 4.1(a)), which combines the memory of nodes in a cluster into a virtual memory pool. The second model is a *remote memory* model which separates servers in a cluster into two groups, compute nodes and memory nodes (Figure 4.1(b)).

4.1.1 DPM Deployment and Architectures

A natural way to deploy PM in datacenters is to insert them to regular servers' DIMM slots. Under this deployment, we can build low-cost PM-based data stores by treating PM dumb and let remote servers access it through one-sided RDMA.

Another way to deploy PM is to host them in *disaggregated* devices. Similar to disaggregated memory [183, 186] and other resource disaggregation systems [38, 187, 188], disaggregated PM devices attach directly to the network and can be accessed by remote servers over the network. With the DPM approach, these devices do not need any local processing units; they only need to connect PM directly to a network interface on board (with possibly a small hardware memory controller). *The Machine* project from HPE [22, 187, 189–192] also uses the disaggregated approach to organize PM. The Machine organizes a rack by connecting a pool of SoCs to a pool of PMs through a specialized cache-coherent network layer [193]. Although being a significant initial step in disaggregated PM research, the Machine only explores one design choice and relies heavily on special network to access and manage disaggregated PM.

We propose three architectures to organize DPM and they work for both server-hosted DPM and disaggregated DPM, over general-purpose RDMA network. The first architecture, DPM-Direct, directly connects CNs to DPMs and has CNs perform all data and control plane operations through one-sided RDMA operations to DPMs (Figure 4.1(c)). DPM-Central connects all CNs and DPMs to a central coordinator, which is involved in both data and control planes (Figure 4.1(d)). Finally, DPM-Sep separates data plane and control plane by performing the former directly from CNs to DPMs and the latter handled by one or more global metadata servers (Figure 4.1(e)).

4.1.2 DPM Benefits and Challenges

DPM offers a cost-effective way to deploy PM in datacenters. The DPM approach can largely reduce the total CPU utilization by not requiring any processing power at where the data is hosted. As we will see in Section 4.3, our DPM-based data stores reduces CPU utilization by 80% to 85% compared to the traditional remote memory architecture. In addition, it offers an easy and low-cost option to build disaggregated PM devices. Moreover, by extracting metadata and control planes from disaggregated DPM devices, it largely improves the manageability of the system.

Notice that there is a subtle but important difference between the DPM approach and approaches to reduce processing time. The DPM approach pushes for the complete elimination of processing needs at where data is hosted. In doing so, processing units such as CPU, ASIC, FPGA [194], and SoC [56] can be removed entirely, reducing not only the cost of these units themselves but also device PCB materials and the developing cost of software running on these units [195].

Although DPM offers many benefits, it is only attractive when there is no or minimal performance lost compared to other more expensive solutions. Building a DPM data store system that can lower the cost but maintain the performance of non-DPM systems is hard. Different from traditional distributed storage and memory systems, DPMs can only be accessed *and* managed from remote. A major technical hurdle is

Table 4.1.: **Design Comparison of DPM Data Stores.** *The R-RTT and W-RTT columns show the number of RTTs required to perform a read and a write (with replication). All RTT values are measured when there is no contention. RTTs in distributed PM's read/write, N, depends on protocols and whether data is local. The Scalability column shows if a system is scalable with the number of CNs, the number of DPMs, both, or neither. † only scalable when there is no contention. The metadata columns show the space needed to store the metadata of a data entry.*

System	R-RTT	W-RTT	Scalability	Metadata
Distributed PM	0-N	0-N	Neither	large
Remote PM w/ CPU	1	1	w/ cost	small
DirectDS	1	6(6)	w/ DPM†	large
CentralDS	2	3(3)	Neither	small
SepDS	1	3(4)	w/ both	medium

in providing good performance with concurrent data accesses. The lack of processing power at DPMs makes it impossible to orchestrate (e.g., serialize) concurrent accesses there. Managing distributed PM resources without any DPM-local processing is also hard and when performed improperly can largely hurt foreground performance. In addition, DPMs can fail independently and such failures have to be handled properly to ensure data reliability and high availability.

4.1.3 Design DPM for Read-Most Data Stores

To confront the challenges of DPM, we propose to design distributed data stores for *read-most* workloads, the most common access pattern in datacenters [170]. Such data stores should still support concurrent writes and ensure data consistency, but their performance should be optimized for reads. We demonstrate that such data stores can be built with no processing units at where data is hosted. By not optimizing for concurrent write performance, we can have everything else: low CapEx and OpEx cost, good performance under low write contention, scalability, manageability, reliability, and high availability.

Table 4.2.: **Cost Comparison of DPM Data Stores.** *The CapEx column represents dollar cost to build eight CNs and eight PMs, and the OpEx column shows the energy cost to run 1 billion operations. Section 4.3 discusses the details of CapEx and OpEx calculation.*

System	CapEx ($)	OpEx ($/BOP)	Performance
Distributed PM	30400	2.4547	Good only when accessing data on local node
Remote PM w/ CPU	60800	0.0139	Good overall
DirectDS	38520	0.0155	Best for small-sized read
CentralDS	42320	0.0452	Not good for read-intensive traffic
SepDS	42320	0.0061	Good overall (unless high write contention)

4.2 DPM Data Stores

This section first describes the interface of all our DPM data stores and their common features. We then present the three data stores we built with the three architectures presented in Section 4.1.1, DirectDS, CentralDS, and SepDS. Figure 4.2 illustrates the read and write operation flow of these systems and Tables 4.1 and 4.2 summarize the tradeoffs of these systems. Finally, we discuss failure handling and load balancing in these data stores. We implemented all our data store systems in user space on Linux.

4.2.1 System Interface and Overview

Interface and guarantees. The current data model that our three data stores support is a key-value store, but these systems can be extended to other data models. Users can create, read (get), write (put), and delete a key-value entry. Different CNs can have shared access to the same data. We manage the consistency of concurrent data accesses in software instead of relying on any hardware-provided coherence like [193, 196, 197].

All our DPM data stores ensure atomicity of an entry across concurrent readers and writers. A successful write indicates that the data entry is committed (atomically), and reads only see committed value. We choose single-entry atomic write and read committed because these consistency and isolation levels are widely used in many data store systems [31, 33] and can be extended to other levels.

Since our DPM systems store persistent data, it is important to provide data reliability and high availability. Our DPM systems guarantee the consistency of data when crashes happen. After restart, each data entry is guaranteed to either only have new data values or old ones. In addition, all our three systems provide replication across DPMs to ensure that data is still available even after losing $N-1$ DPMs (when the degree of replication is N).

Network layer. We choose RDMA as the network layer that connects all servers and DPMs. We use RDMA's RC (Reliable Connection) mode which supports one-sided RDMA operations and ensures lossless and ordered packet delivery. Similar to prior solutions [29,32,33,116], we solve RDMA's scalability issues using memory huge page to register memory regions with RDMA NICs.

Ensuring data persistence. For data to be persistent in DPM, it is not enough to just perform a remote write. After a remote write (*e.g.*, RDMA write), the data can be in NIC, PCIe hub, or PM. Only when the data is written to PM can it sustain power failure. To ensure this data persistence, we follow the guidance of SNIA [198] and Mellanox [39, 44] by performing a remote read to ensure that data is actually in PM. Since we use RDMA RC which guarantees ordered data delivery and PCIe also follows ordering [199], we only read the last byte of a data entry to verify its persistence [198].

4.2.2 Direct Connection

The DPM-Direct architecture (Figure 4.1(c)) connects CNs directly to DPMs. CNs perform un-orchestrated, direct accesses to DPMs using RDMA one-sided oper-

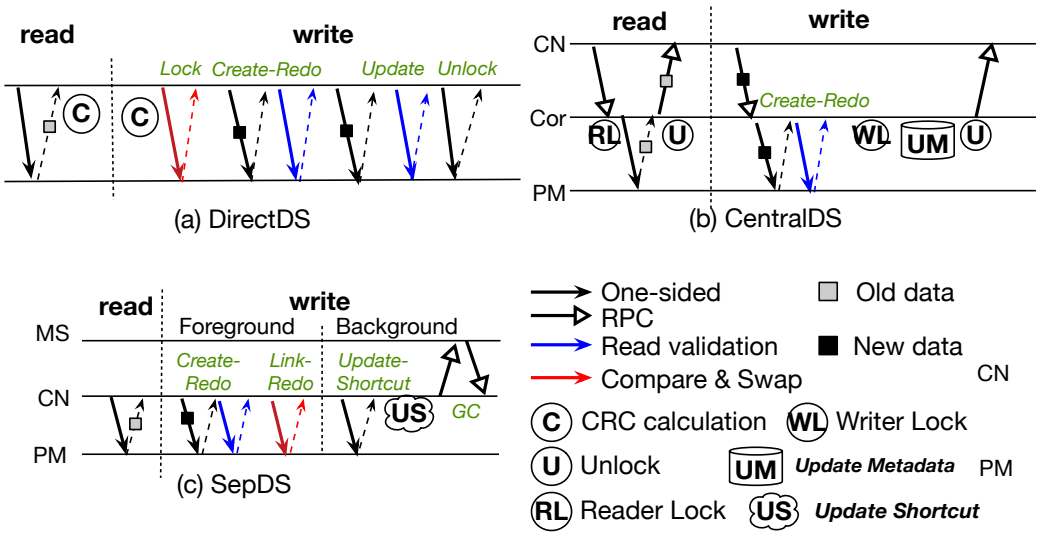


Figure 4.2.: **Read/Write Protocols of DPM Systems.**

ations. Under DPM-Direct, performing metadata and control operations from CNs is hard and costly (*e.g.*, by performing distributed coordination across CNs).

Control plane. We made two design choices to ease the control path of DirectDS. First, we pre-assign two spaces for each data entry, one to store committed data where reads go to (we call it the *committed space*) and one to store in-flight, new data (*uncommitted space*). These spaces are statically allocated at data entry creation time by CNs. Doing so avoids dynamic space allocation and de-allocation. We currently use a distributed consensus protocol across CNs to perform space allocation and de-allocation and use Memcached [90] to store metadata. Other distributed consensus and metadata management systems can also work. Second, to avoid reading and writing metadata from DPMs and the cost of ensuring metadata consistency under concurrent accesses, CNs in DPM-Direct locally store all the metadata of key-value entries, including the key of a value and the location of its committed and uncommitted spaces.

Data plane. A straightforward method to ensure safe concurrent read and write accesses is to use distributed locks and lock a data entry before accessing it. Doing so causes two network round trips (RTTs) of lock and unlock for each data access. To

improve read performance, we adopt a lock-free read mechanism with the help of error detecting codes. Figure 4.2(a) illustrates the read and write protocol of DirectDS.

To read a data entry, a CN uses its stored metadata to find the location of the data entry's committed space (and the first 8-byte lock). Then, the CN simply issues an RDMA read to fetch the data and calculates and validates its CRC afterwards. The read latency of DirectDS is one RTT plus the CRC calculation time.

To write a data entry, a CN first calculates and attaches a CRC to the new data entry. It then locates the entry and locks it. After acquiring the lock, the CN writes the new data (and CRC) to the un-committed space. To ensure data persistence, the CN issues an RDMA read to the last byte of the un-committed space to validate that it is actually written to PM [198,200]. This uncommitted data serves as the *redo* copy that will be used during recovery if a crash happens. The CN then writes the new data to the committed space with an RDMA write and validates it with an RDMA read. At the end, the CN releases the lock. The total write latency is 6 RTTs (when no contention), two of which involve data read/write.

We implemented our distributed locking using RDMA one-sided operations. We associate an 8-byte value at the beginning of each data entry to implement its lock. To acquire the lock, a CN performs a one-sided RDMA `c&s` (compare-and-swap) operation to the value (*e.g.*, comparing whether the value is 0 and if so setting it to 1). To release the lock, the CN simply performs an RDMA write and sets the value to 0.

Our lock implementation leverages the unique feature of the DPM model that all memory accesses to DPMs come from the network (*i.e.*, the NIC). Without processor's accesses to memory, DMA guarantees that network atomic operations like `c&s` are atomic [144, 201]. Note that an RDMA `c&s` operation to an in-memory value which can also be accessed locally at the same time does not guarantee the atomicity of the value [116, 144, 202], and thus it cannot be used in distributed PM systems in the same way.

Discussion. As we will see in Section 4.3, DirectDS delivers very good read performance when read size is small. since it only requires one lock-free RTT and it is fast to calculate small CRC. Its write performance is much worse because of the high RTTs and lock contention on writes to the same data entries. Its scalability is also limited because of lock contention during concurrent writes.

Moreover, DirectDS also requires large space for both data and metadata. For each data entry, it doubles the space because of the need to store two copies of data. The metadata overhead is also high, since CNs have to store all metadata.

4.2.3 Connecting Through Coordinator

Most limitations of DPM-Direct come from the fact that there is no central coordination of data, metadata, or control operations. For example, DPM-Direct systems have to write data twice, once to the un-committed and once to the committed space, because CNs in DPM-Direct only know a fixed location to read committed data. The DPM-Central architecture (Figure 4.1(c)) takes the opposite design choice and uses a central *coordinator* to orchestrate all data accesses and to perform metadata and management operations. All CNs send RPC requests to the coordinator (we use the HERD [32, 33] RPC system for this purpose, but other RPC systems [203, 204] can also work). The coordinator handles RPC requests by performing one-sided requests to DPMs. For better throughput, we use multiple RPC handling threads at the coordinator.

Since all requests go through the coordinator, it can serve as the serialization point for concurrent accesses to a data entry. We simply use a local read/write lock for each data entry at the coordinator as the synchronization of multiple coordinator threads. In addition to orchestrating data accesses, the coordinator performs all space allocation and de-allocation of data entries. The coordinator uses its local PM to persistently store all the metadata for a data entry including its key, its location, and a read/write lock. With the coordinator handling all read requests, it can freely direct

a read to the latest location of committed data. Thus, it does not need to maintain the same location for committed data and changes the location of committed data after each write.

To perform a read, a CN sends an RPC read request to the coordinator. The coordinator finds the location of the entry's committed data using its local metadata, acquires its local lock of the entry, reads the data from the DPM using a one-sided RDMA read, releases the lock, and finally replies to the CN's RPC request. The end-to-end read latency a CN observes is 2 RTTs, and both RTTs involve sending data.

When receiving a write request from a CN, the coordinator allocates a new space in a DPM for the new data. It then writes the data and validates it with an RDMA read. Note that we do not need to lock (either at coordinator or at DPM) during this write, since it is an out-of-place write to a location that is not exposed to any other coordinator RPC handlers.

After successfully verifying the write, the coordinator updates its local metadata of where the committed version of the data entry is and flushes this new metadata to its local PM for crash recovery (by performing CPU cache flushes and memory barrier instructions [205]). Since concurrent coordinator RPC handlers can update the same information of where the latest data entry is, we use a local lock to protect this metadata change. The total write latency without contention is 3 RTTs, with two of them containing data and one for validation.

Discussion. CentralDS largely reduces write RTTs over DirectDS and thus has good write performance when the scale of the cluster is small. However, from our experiments, the coordinator soon becomes the performance bottleneck when either the number of CNs increases or the number of DPMs increases. CentralDS's read performance is also worse than DirectDS with the extra hop between a CN and the coordinator. In addition, the CPU utilization of the coordinator is high, since it needs to have a high amount of RPC handlers to sustain parallel requests from CNs

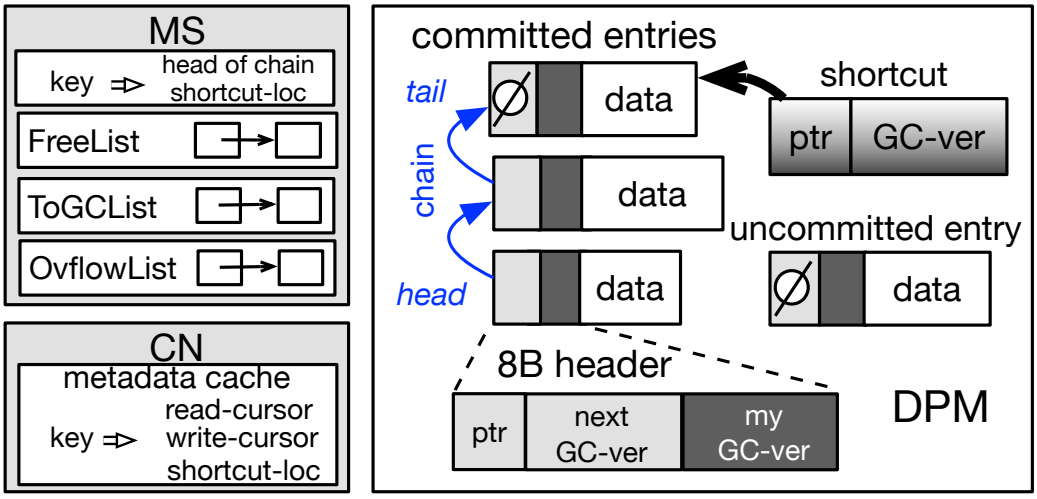


Figure 4.3.: **SepDS System Design.**

(Section 4.3). However, unlike DPM-Direct, CNs in the DPM-Direct architecture does not need to store any metadata.

4.2.4 Separating Data and Control

DPM-Direct has good read performance (when data size is small) but has poor write performance and costly metadata/control plane. CentralDS improves DPM-Direct's write performance and manageability but suffers from the scalability bottleneck of the central coordinator. To solve these problems of the first two DPM architectures, we propose a third architecture, DPM-Sep (Figure 4.1(d)), and a data store designed for it, *SepDS*. The main idea of DPM-Sep is to separate the data plane from the control plane. It lets CNs directly access DPMs for all data operations and uses a metadata server (MS) for all control plane operations.

The MS stores metadata of all data entries in its local PM. We keep the amount of metadata small, and 1 TB of PM (a conservative estimation of the size of PM a server can host) can store metadata for 64 TB data at the granularity of 1 KB per data entry. CNs cache metadata of hot data entries; under memory pressure, CNs

will evict metadata according to an eviction policy (we currently support FIFO and LRU).

SepDS aims to deliver scalable, low-latency, high-throughput performance under low write contention at the data plane and to avoid the MS being the bottleneck at the control plane. Our overall approaches to achieve these design goals include: 1) moving all metadata operations off performance critical path, 2) using lock-free data structures to increase scalability, 3) employing optimization mechanisms to reduce network round trips for data accesses, and 4) leveraging the unique atomic data access guarantees of DPM. Figure 3 illustrates the data structures used in SepDS.

Data Plane

To achieve our data plane design goal, we propose a new mechanism to perform lock-free, fast, and scalable reads and writes. The basic idea is to allow multiple committed versions of a data entry in DPMs and to link them into a *chain*. Each committed write to a data entry will move its latest version to a new location. To avoid the need to update CNs with the new location, we use a self-identifying data structure to let CNs be able to find the latest version.

We include a *header* with each version of a data entry, which contains a pointer and some metadata bits used for garbage collection. The pointers chain all versions of a data entry together in the order that they are written. A `NULL` pointer indicates that the version is the latest.

A CN acquires the header of the chain head from the MS at the first access to a data entry. It then caches the header locally to avoid the overhead of contacting MS on every data access. As a CN reads or writes an entry, it advances its cached header. We call a CN-cached header a *cursor*.

Read. SepDS reads are lock-free. To read a data entry, the CN performs a *chain walk*. The chain walk begins with fetching the data entry its current cursor points to. It then follows the pointer in the following entries until it reaches the last entry.

All steps in the chain walk use one-sided RDMA reads. After a chain walk, the CN updates its cursor to the last entry.

A chain walk can be slow with long chains when a cursor is not up to date [206]. Inspired by skip-list [207], we propose to solve this issue by using a *shortcut* to directly point to a newer entry. The shortcut of a data entry is stored in DPM and the location of the shortcut never changes during the lifetime of the data. MS stores the locations of all shortcuts and CNs cache the hot ones. Shortcuts are *best effort* in that they are intended but not enforced to always point to the last version of an entry.

The CN issues a chain walk read and a shortcut read in parallel. It returns to user when the faster one returns and discards the other result. Note that we do not replace chain walks completely with shortcut reads, since shortcuts are updated asynchronously in the background and may not be updated as fast as the cursor. When the CN has a pointer that points to the latest version of data, a read only takes 1 RTT.

Write. SepDS never overwrites existing data entries and performs a lock-free out-of-place write before linking the new data to an entry chain. To write a data entry, a CN first selects a free DPM buffer assigned to it by MS in advance (see § 4.2.4). It performs a one-sided RDMA write to write the new data to this buffer and then issues a read of the last byte to ensure that the data is written in PM. Afterwards, the CN performs an RDMA `c&s` operation to link this new entry to the tail of the entry chain. Specifically, the `c&s` operation is on the header that CN's cursor points to. It compares if the pointer in the header is `NULL` and swaps the pointer to point to the new entry. If the `c&s` succeeds, we treat this data as *committed* and return the write request to the user. If the pointer is not `NULL`, it means that the cursor does not point to the tail of the chain and we will do a chain walk to reach the tail and then do another `c&s`.

Afterwards, the CN uses a one-sided RDMA write to update the shortcut of the entry to point to the new data entry. This step is off the performance critical path. The CN also updates its cursor to the newly written data entry. We do not invalidate

or update other CNs' cursors at this time to improve the scalability and performance of SepDS.

SepDS' chained structure and write mechanism ensure that writers do not block readers and readers do not block writers. They also ensure that readers can only view committed data. Without high write contention to the same data entry, one write takes only 3 RTTs.

Retire. After committing a write, a CN can *retire* the old data entry, indicating that the entry space can be reclaimed. To improve performance and minimize the need to communicate with the MS, CNs perform lazy, asynchronous, batched retirement of old data entries in the background. We further avoid the need for MS to invalidate CN-cached metadata using a combination of timeout and epoch-based garbage collection.

Control Plane

CNs communicate with the MS using two-sided operations for all metadata operations. The MS performs all types of management of DPMs. It manages physical memory space of DPM, stores the location and shortcut of a data entry. We carefully designed these MS functionalities to achieve good performance and scalability.

Space allocation. With the data plane out-of-place write model, SepDS has high demand for DPM space allocation. We use an efficient space allocation mechanism where MS packages free space of all DPMs into chunks. Each chunk hosts the same size of data entries and different chunks can have different data sizes, similar to FaRM [29] and Hoard [208]. Instead of asking for a new free entry before every write, each CN requests multiple entries at a time from the MS in the background. This approach moves space allocation off the critical path of writes and is important to deliver good write performance.

Garbage collection. SepDS' append-only chained data structure makes its writes very fast. But like all other append-only or log-structured data stores, SepDS needs to garbage collect (GC) old data. We designed a new efficient GC mechanism that

does not involve any data movement or communication to DPM and minimizes the communication between MS and CNs.

The basic flow of GC is simple: the MS keeps busy checking and processing incoming retire requests from CNs. The MS decides when a data entry can be reclaimed and puts a reclaimed entry to a free list (*FreeList*). It gets free entries from this list when CNs request for more free buffers. A reclaimed entry can be used by any CN for any new entry, as long as the size fits.

Although the above strawman GC implementation is simple, making GC work correctly, efficiently, and scale well is challenging. First, to achieve good GC performance, we avoid the invalidations of CN cached cursors after reclaiming entries so as to minimize the network traffic between the MS and CNs. However, with the strawman GC implementation, CNs' outdated cursors can cause failed chain walks. We solve this problem using two techniques: 1), the MS does not clear the header (or the content) of a data entry after reclaiming it, and 2), we assign a *GC version* to each data entry. The MS increases the GC version number after reclaiming a data entry. It gives this new GC version together with the location of the entry when assigning the entry as a new free buffer to a CN, A. Before CN A uses the entry for its new write, the entry content at the DPM still has old header and data (with old GC version). Other CNs that have cached cursors to this entry can thus still use the old pointer to perform chain walk. CNs differentiate if an entry is its intended data or has already been reclaimed and reused for other data by comparing the GC version in its cached cursor and the one it reads from the DPM. After CN A writes the new data with the new GC version number, other CNs that have the old cursors will have a mismatched GC version and discard the entry and invalidates their cursors. Doing so not only avoids the need for MS to invalidate cursor caches on CNs, but also eliminates the need for MS to access DPMs during GC.

The next challenge is related to our targeted guarantee of read isolation and atomicity (*i.e.*, readers should always read the data that is consistent to its metadata header). An inconsistent read can happen if the read to a data entry takes long and

during the reading time, this entry has been reclaimed and used to write a new data entry. We use a read timeout scheme similar to [29]. CNs abort a read operation after T_r, an agreed value among CNs and the MS. The MS delays the actual reclamation of an entry to only T_r time after it receives the retire request of the entry. Specifically, the MS leaves the entry in a *ToGCList* for T_r and then moves it to the FreeList.

The final challenge is the overflow of GC version numbers. We can only use limited number of bits for GC version in the header of a data entry (currently 8 bits), since the header needs to be smaller than the size of an atomic RDMA operation. When the GC version of an entry increases beyond the maximum value, we will have to restart it from zero. With just the GC version number and our GC mechanism so far, CNs will have no way to tell if an entry matches its cached cursor version or has advanced by $2^8 = 256$ versions. To solve this rare issue without invalidation traffic to CNs, we use an epoch-based timeout mechanism. When the MS finds the GC version number of a data entry overflows, it puts the reclaimed entry into *OvflowList* and waits for T_e time before moving it to the *FreeList* that can be assigned to CNs. All CNs invalidate their own cursors after an inactive period of T_e (if during this time, the CN access the entity, it would have advanced the cursor already). To synchronize epoch time, the MS sends a message to CNs after T_e, and the MS can choose the value of T_e. Epoch message is the only communication the MS issues to CNs during GC.

Discussion.

The SepDS design offers four benefits. First, SepDS reads and writes are fast, with 1 RTT and 3 RTTs respectively when there is no contention. Even under contention, SepDS still achieves comparable performance as alternative systems. Achieving this low latency and guaranteeing atomic write and read committed is not easy and is achieved by the combination of four approaches: 1) ensuring the data path does not involve the MS, 2) reducing metadata communication to the MS and moving it off

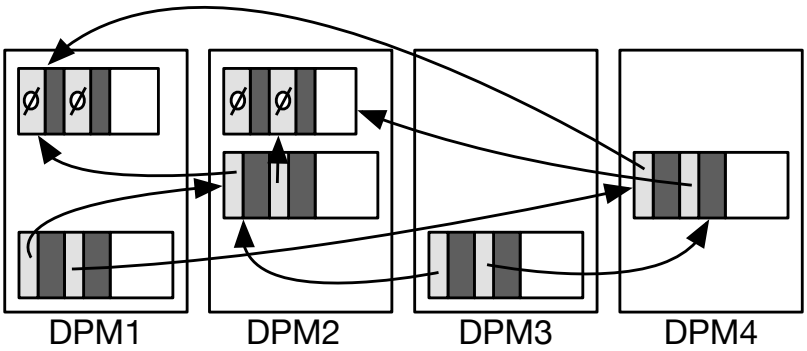


Figure 4.4.: **Replicated Data Entity.** *A replicated data entity on four DPMs. The replication factor is two.*

performance critical path, 3) ensuring no memory copy in the whole data path, and 4) leveraging the unique advantages of DPM to perform RDMA atomic operations.

Second, SepDS scales well with the number of CNs and DPMs, since its reads and writes are both lock free. Readers do not block writers or other readers and writers do not block readers. Concurrent writers to the same entity only contend for the short period of RDMA `c&s` operation. SepDS also minimizes the network traffic to MS and the processing load on MS to make MS scale well with number of CNs and data operations.

Third, we avoid *all* data movement or communication between the MS and DPMs during GC. To scale and support many CNs with few MSs, we avoid CN invalidation messages completely. The MS does not need to proactively send any other messages to CNs either. Essentially, the MS never *pushes* any messages to CNs. Rather, CNs *pull* information from the MS.

Finally, the SepDS data structure is flexible and can support load balancing very well. Different entries of a data entity do not need to be on the same DPM device. As we will see in Section 4.2.5 and Section 4.2.6, this flexible placement is the key to SepDS' load balancing and data replication needs.

However, SepDS also has its own limitation. It requires CNs to cache metadata. As we will see in Section 4.3, when CN's local metadata cache becomes small, SepDS's

performance drops. Thus, SepDS works the best when CNs have enough memory or when data accesses have good temporal locality.

4.2.5 Failure Handling

DPMs can fail independently from CNs. A DPM system needs to handle both the transient failure of a DPM (which can be rebooted) and a permanent failure of one. For the former, our three DPM systems guarantee crash consistency, *i.e.*, after reboot, the DPM can recover all its committed data. For the latter, we add the support for data replication across multiple DPMs to all the three data store systems. In addition, CentralDS and SepDS also need to handle the failure of the coordinator and the MS.

Recovery from Transient Failures

We now present how each system recovers from a single DPM's failure when it restarts. We assume that the rest of the system (*e.g.*, CNs, the coordinator, the MS) keeps alive. We will discuss the reliability of the coordinator and the MS in Section 4.2.5.

DirectDS. When recovering a DPM in DirectDS, we need to decide whether to use the data in the committed space or the un-committed space (*i.e.*, where the redo copy is). DirectDS validate the data in the committed space with its CRC. During recovery, we calculate the CRC of the committed space. If the CRC is correct, it indicates the committed space has the complete data. Otherwise, we copy the data from the redo copy to the committed space.

CentralDS. Handling the failure of a DPM in CentralDS is simple, as long as the coordinator stays alive. Since CentralDS performs out-of-place writes and the coordinator stores the state of all writes, we can simply use the information in the coordinator to know what writes have written their redo copies but haven't committed

yet and what writes have not written redo copies. For the former case, we advance to the redo copy, and for the latter, we use the original version.

SepDS. SepDS' recovery mechanism is also simple. If a DPM fails before a CN successfully links the new data it writes to the chain (indicating an un-committed write), the CN simply unsets lock bits (within a pointer) of the data entry (releasing the held lock) and discards the new write (by treating the space as unused).

Adding Redundancy

We now present how we add redundancy to DPM in all the three systems and how we handle coordinator and MS failures. With the user-specified degree of replication being N, our data store systems guarantee that data is still accessible after $N-1$ DPMs have failed.

DirectDS. In order to sustain DPM failure during a write, we need to replicate both the first write to the un-committed space (the redo copy) and the second write to the committed space. After getting the lock, a CN sends the new data to the un-committed space on N DPMs in parallel. Afterwards, it performs N read validation, also in parallel. Once read validation of all the copies succeeds, the CN writes the data to the committed space of the N DPMs in parallel and performs a parallel read validation afterwards.

CentralDS. To handle a replicated write RPC request, the coordinator writes multiple copies of the data to N DPMs in parallel and performs a parallel read validation of them. After the read validation, the coordinator updates its metadata to record the new locations of all these copies.

SepDS. We propose a new atomic replication mechanism designed for the SepDS data structure. The basic idea is to link each data entry version D_N to all the replicas of the next version (*e.g.*, D^a_{N+1}, D^b_{N+1}, D^c_{N+1} for three replicas) by placing pointers to all these replicas in the header of D_N. Figure 4.4 shows an example of replicated

data entry. With this all-way chaining, SepDS can always construct a valid chain as long as one copy of each version in an entry survives.

Each data entry has a primary copy and one or more secondary copies. To write a data entry D_{N+1} with R replicas to an entry whose current tail is D_N, a CN first writes all copies of D_{N+1} to R DPMs. In parallel, a CN performs a one-sided `c&s` to a bit, B_w, in the header of the primary copy of D_N to test if the entry is already in the middle of a replicated write. If not, the bit will be set, indicating that the entry is now under replicated write. All the writes and the `c&s` operation are sent out together to minimize latency.

After the CN receives the hardware acknowledgment of all the operations, it constructs a header that contains R pointers to the copies of D_{N+1} and writes it to all the copies of D_N. Once the new header is written to all copies of D_N, the system can recover D_{N+1} from crashes (up to $R-1$ concurrent DPM failure).

Backup coordinator and MS. To avoid the coordinator or the MS being the single point of failure in CentralDS and SepDS, we implement a mechanism to enabling one or more *backup* coordinator (MS), by having the primary coordinator (MS) replicate the metadata that cannot be reconstructed (*i.e.*, keys and locations of values) to the backup coordinator (MS) when changing these metadata.

4.2.6 Load Balancing

With the DPM model, a system will have a pool of DPMs. Thus, it is beneficial to balance the load to each of them.

With a centralized place to initiate all requests, it is easy for CentralDS to perform load balancing. The coordinator simply records the load to each DPM and directs new writes to the DPM with lighter load. When DPM is replicated, the coordinator can also balance read loads by selecting the replica that is on the DPM with lighter load.

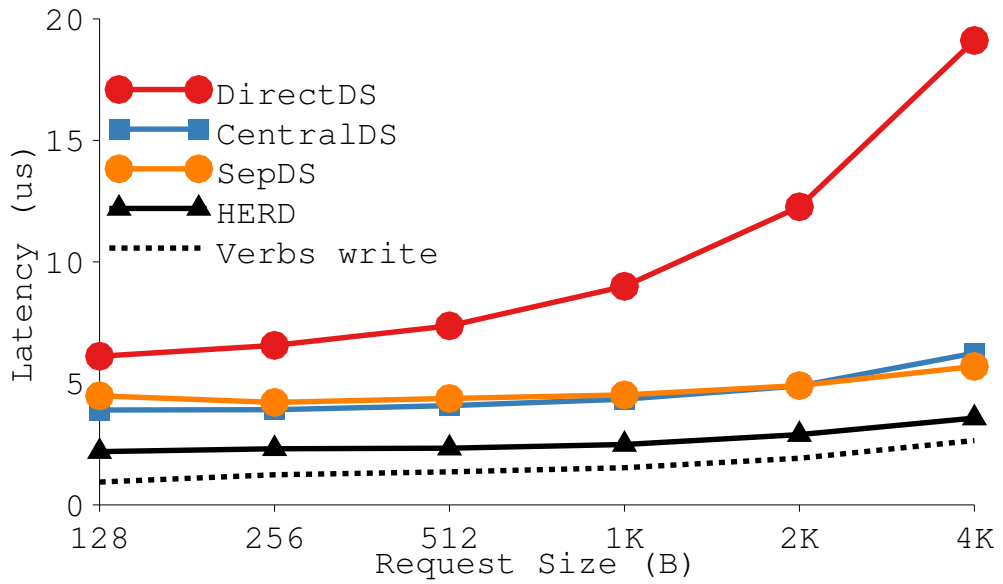


Figure 4.5.: **Write Latency**

We use a novel two-level approach to balance loads in SepDS: globally at MS and locally at each CN. Our global management leverages two features in SepDS: 1) MS assigns all new space to CNs; and 2) data entries of the same entity in SepDS can be on different DPMs. To reduce the load on a DPM, MS directs all new writes to other devices. At a local level, each CN internally balances the load to different DPMs. Each CN keeps one bucket per DPM to store free entries. It chooses buckets from different buckets for new writes according to its own load balancing needs.

However, balancing loads with the DPM-Direct architecture is hard, since there is no coordination across CNs.

4.3 Evaluation Results

This section presents the evaluation results of different DPM systems including DirectDS, CentralDS, and SepDS. All our experiments were carried out in a cluster of 14 machines, connected with a 100 Gbps Mellanox InfiniBand Switch. Each machine

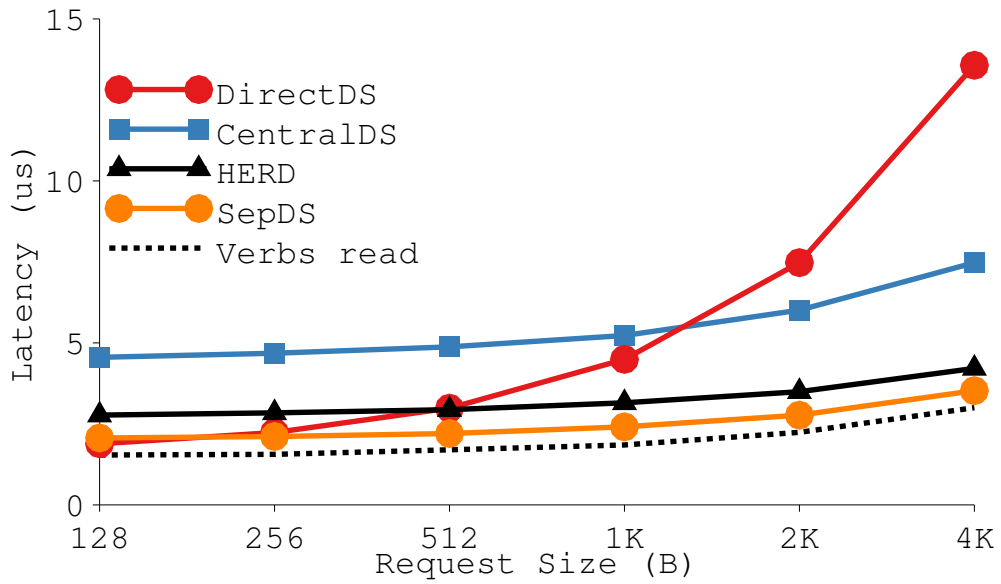


Figure 4.6.: **Read Latency**

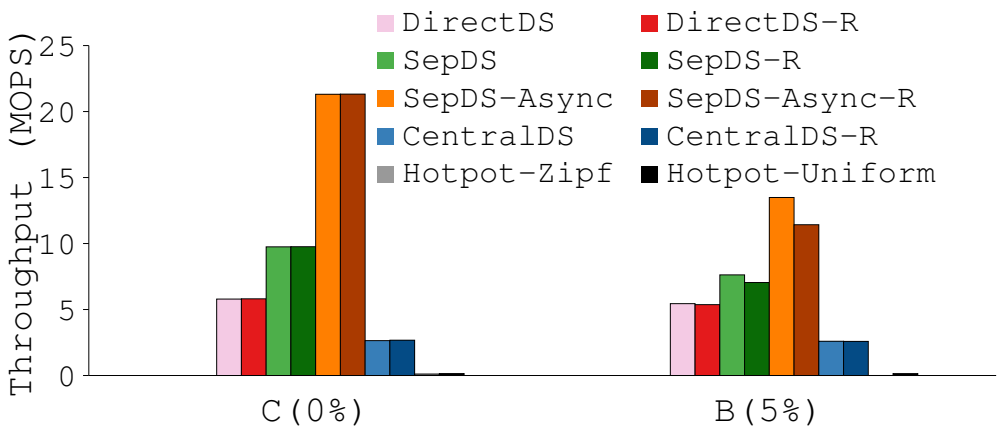


Figure 4.7.: **Throughput Comparison with YCSB** *Running YCSB on four CNs and four DPMs.*

is equipped with two Intel Xeon E5-2620 2.40GHz CPUs, 128 GB DRAM, and one 100 Gbps Mellanox ConnectX-4 NIC.

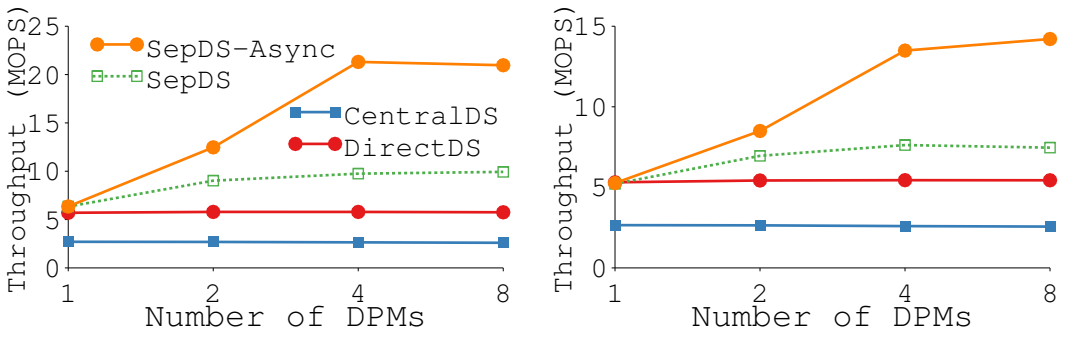
4.3.1 Micro-Benchmark Results

We first evaluate DPM data stores' read and write performance using a simple micro-benchmark. Figure 4.5 plots the average write latency with different request size. We use native RDMA one-sided write as the baseline; it only performs a write without any read validation and has the lowest latency. Among DPM systems, SepDS and CentralDS achieve the best write latency. DirectDS's write performance is the worst because of its 6-RTT write protocol. Its write performance also gets worse with larger request size because of the increased overhead of CRC calculation.

We also compare DPM systems with HERD [33], a two-sided RDMA-based in-memory key-value store. We use HERD's default configuration of using 12 busy polling receiving side's threads for all our experiments. HERD outperforms our DPM systems on write latency because it only does a write without read validation. Finally, we evaluate all the DPM systems' write performance without read validation (*i.e.*, treating DPM as volatile memory). We found each read validation to cost a constant of 1.5 μs overhead.

Read validation is just one way that works with current RDMA and PM hardware to ensure an PM write is persistent. We envision when PM is deployed in practice with RDMA, the supporting hardware will have another way of guarantee write persistence without the need of the extra RTT of read validation (*e.g.*, by simply changing the NIC to only send the ACK after data is written to the PM media). Because of this and to have a fair comparison with DRAM-based systems like HERD, we do not perform the read validation step in the rest of our evaluation.

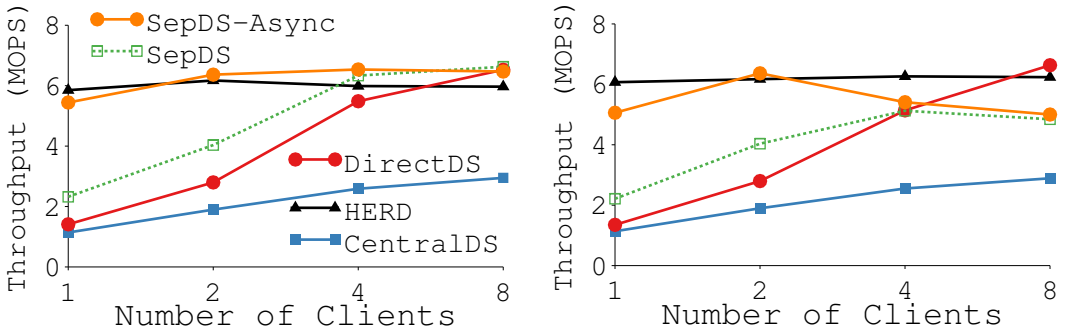
Figure 4.6 plots the read latency comparison. We again use native one-sided RDMA read as a baseline here. Overall, SepDS's performance is the best among DPM systems and is only slightly worse than native RDMA. DirectDS has very good read performance when request size is small. However, when request size increases, the overhead of CRC calculation dominates, largely hurting DirectDS's read performance. As expected, CentralDS's read performance is not good because of its 2-RTT read



(a) Workload C (0%) (b) Workload B (5%)

Figure 4.8.: **Scalability w.r.t. DPMs** *Running 4 CNs.*

protocol. HERD performs worse than SepDS because of it requires some extra CPU processing time for each read.



(a) Workload C (0%) (b) Workload B (5%)

Figure 4.9.: **Scalability w.r.t. CNs** *Running 1 DPMs.*

4.3.2 YCSB Results

We now present our evaluation results using the YCSB benchmark [184, 185]. We use a total of 100K key-value entries where the key size is 8 bytes and the value size is 1 KB. The accesses to keys follow the Zipf distribution. We use two workloads with different read-write ratios: read only (workload C) and 5% write (workload B), following common workload patterns in datacenters [170]. We compare the three DPM systems with HERD [33] and Hotpot [37], which serve as comparison points of the distributed

and remote memory model. We further evaluate the effect of CN asynchronously sending data access requests with the help of coroutines (SepDS-async), similar to the asynchronous mechanism implemented in FaSST [66] and DrTM+H [68]. HERD uses its customized asynchronous implementation and Hotpot does not support asynchronous interface.

Throughput Results

Basic performance. We first evaluate the performance of all our DPM systems under one configuration: 4 CNs and 4 DPMs, each CN running 8 application threads. Figure 4.7 shows the overall performance of DPM systems, replicated DPM systems (with degree of replication 2), and Hotpot.

SepDS performs the best among all systems regardless of read/write intensity, even under high contention (with Zipf distribution to keys). SepDS-async further improves SepDS's throughput by sending more requests at the same time. DirectDS performs well with workloads that are read intensive. DirectDS's read performance is not affected by contention, since it does not need to perform any lock. CentralDS's read performance is worse than DirectDS and SepDS because each read in CentralDS requires 2 RTTs and under contention the coordinator becomes the bottleneck.

The overall performance of Hotpot is orders of magnitude worse than all DPM data stores. The main reason is that each read and write in Hotpot involves a complex protocol that requires RPCs across multiple nodes. Hotpot's performance is especially poor with writes, since the distributed PM consistency protocol involves frequent invalidation of cached copies, especially under high write contention to the same data. Hotpot performs better when running workloads with uniform distribution (but still much worse than DPM systems). The Hotpot results are from its MRSW consistency level without replication and four servers in total, each running 8 application threads, a configuration reported by the Hotpot paper. Hotpot only sup-

ports the 40Gbps ConnectX-3 NIC, so its performance can also be partially impacted by the NIC compared to our ConnectX-4 environment.

Replication overhead. As expected, adding redundancy lowers the throughput of write operations in all data stores. Even though all systems issue the replication requests in parallel, they only use one thread to perform asynchronous RDMA read/write operations and doing so still has an overhead.

Scalability. Next, we evaluate the scalability of different DPM systems with respect to the number of CNs and the number of DPMs. Figure 4.8 shows the scalability of DPM data stores w.r.t. the number of DPMs (HERD only supports single memory node and we do not include it in this experiment). SepDS scales well with DPMs because CNs access DPMs directly for data accesses, having no scalability bottleneck. SepDS-async improves SepDS further by sending more asynchronous requests (it saturates the network full bandwidth beyond four DPM nodes. CentralDS has poor scalability because of the coordinator being the bottleneck that all requests have to go through. Surprisingly, DirectDS's scalability is also poor. Although CNs in DirectDS access DPMs directly, they need to calculate CRC for each read/write request. When the number of DPM nodes increases, CNs need to do more CRC calculation in the same amount of time and this computation overhead becomes a performance bottleneck.

Figure 4.9 shows the scalability of DPM data stores and HERD when varying the number of CNs with a single DPM. SepDS-async and HERD have the best (and similar) performance with workload C, effectively saturating the network full bandwidth. Both systems send asynchronous requests that can saturate network bandwidth and neither systems have any scalability bottlenecks. Under workload B, the performance of SepDS is slightly worse with more CNs because of the increased write contention. The rest three systems cannot saturate network bandwidth but scale well with more CNs.

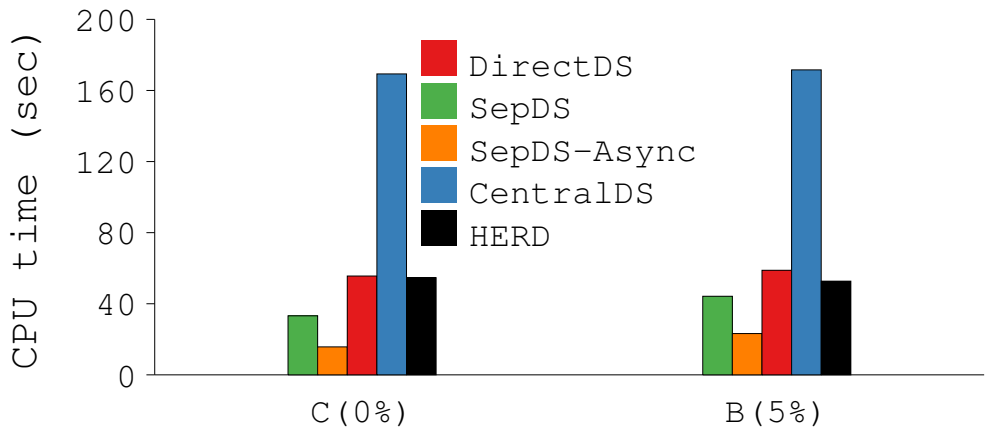


Figure 4.10.: **CPU Utilization**

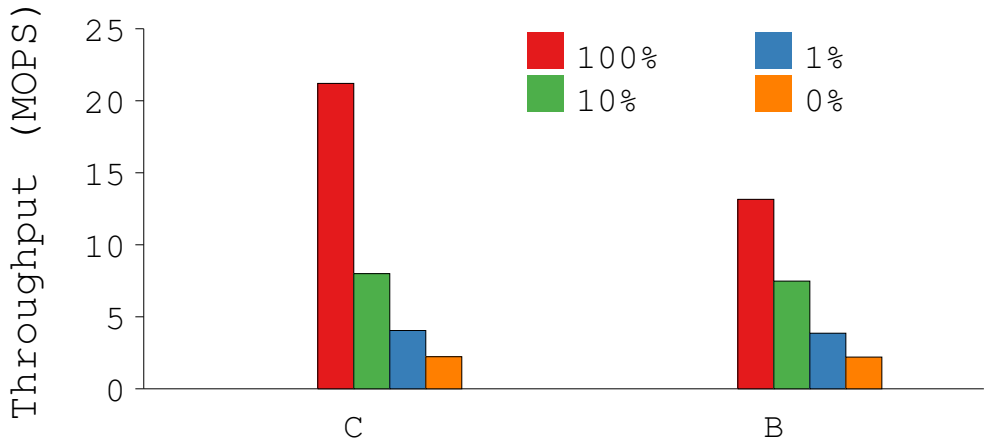


Figure 4.11.: **Effect of Metadata Cache in SepDS**

CPU Utilization and Cost

CPU utilization. We evaluate the CPU utilization of different data stores. Figure 4.10 plots the total CPU time to complete ten million requests. For read-intensive workload, DirectDS and SepDS use less CPU time than CentralDS and HERD because they perform one-sided RDMA directly from CNs to DPMs. SepDS-async reduces CPU time further because of its higher throughput performance. HERD also uses asynchronous requests but takes 2.3× longer total CPU time, because it uses many

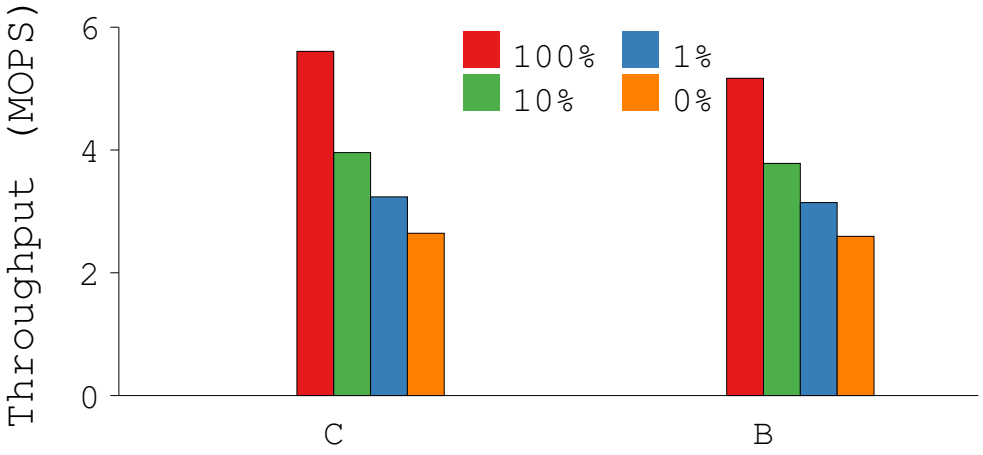


Figure 4.12.: **Effect of Data Cache in CentralDS**

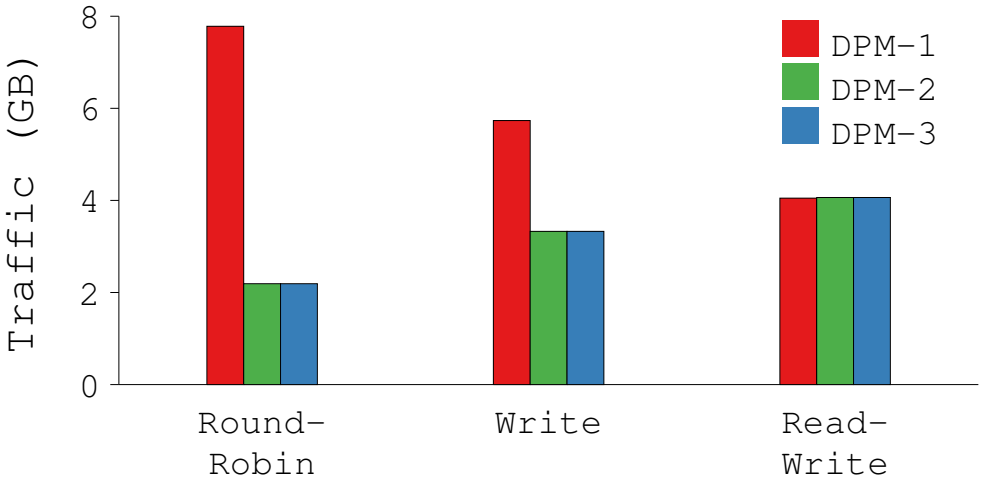


Figure 4.13.: **Load Balancing in SepDS**

busy-polling threads at its memory nodes to achieve good performance (12 threads by default). Note that HERD only supports one memory node, and we assume perfect scaling of HERD to estimate its upper bound of performance (lower bound of CPU time). CentralDS has high CPU utilization because the coordinator's CPU spends time on every request and the total time to finish the workloads with CentralDS is long.

CapEx and OpEx. Table 4.2 summarizes the cost to build different data stores with 8 CNs and 8 DPMs (for distributed PM, we use eight machines in total). We collected hardware and energy costs from online, and the hardware used in our calculation is the same or similar model as what's used in our experiments. The CapEx is calculated with the market price of datacenter-level server ($3800), CPU ($440 [209]), DRAM ($220 [210]), and NIC ($795 [211]). Because NVM is not yet available in the public market, we use DRAM cost for NVM. Distributed PM has the lowest CapEx because it can share PM and only needs eight machines in total. Remote PM requires 16 machines in total (8 for CNs and 8 for DPM nodes). The three DPM systems do not require full machines for DPM nodes and we only include PM and NIC costs for them.

The OpEx is calculated using the CPU time to complete one billion requests of workload B and CPU energy cost ($83.22 per year [212]). We do not include the energy to host NVM, since it is the same for all systems. Distributed PM has the highest OpEx because of its poor performance and high CPU utilization. SepDS has the lowest OpEx because of its low CPU utilization and overall good performance.

Client Caching Effect

Metadata caching effect. To evaluate the effect of different sizes of metadata cache at CNs in SepDS, we ran the same YCSB workloads and configuration as Figure 4.7 and plot the results in Figure 4.11. Here, we use the FIFO eviction policy (we also tested LRU and found it to similar or worse than FIFO). With smaller metadata cache, all workloads' performance drop because a CN has to get the metadata from the MS before accessing the data entry that does not have local metadata cache. With no metadata cache (0%), CNs need to get metadata from the MS before every request. However, under Zipf distribution, with just 10% metadata cache, SepDS can already achieve satisfying performance.

Data caching effect. We do not cache data at CNs because doing so would require coherence traffic, resulting in performance that is similar to distributed PM. However, it is possible to cache data at the coordinator with the DPM-Central architecture, because that is the only copy and does not need any coherence traffic. By caching hot data in a coordinator, the coordinator does not need to access DPMs to get data for every read which can reduce network traffic and improve performance. We built a FIFO data cache at the coordinator for CentralDS to analyze the effect of data caching. Figure 4.12 plots the throughput with different percentages of the data cache in a coordinator. With bigger data cache, the performance increases. However, the overall performance is still limited by network bandwidth. Overall, we found the effect of data caching to be small with CentralDS, but demands large amount of PM space at the coordinator.

Load Balancing

To evaluate the effect of SepDS's load balancing mechanism, we use a synthetic workload with six data entities, a, b, and $c1$ to $c4$. The workloads initially creates a (no replication) and b (with 3 replicas) and reads these two entities continuously. At a later time, it creates $c1$ to $c4$ (no replication) and keep updating them. One CN runs this synthetic workload on three DPMs. Figure 4.13 shows the total traffic to the three DPMs with different allocation and load-balancing policies. With a naive policy of assigning DPMs to new write requests in a round-robin fashion and reading from the first replica, write traffic spreads evenly across all DPMs but reads all go to DPM-1. With write load balancing, MS allocates free entries for new writes from the least accessed DPM. Doing so spreads write traffic more towards the lighter-loaded DPM-2 and DPM-3. With read load balancing, SepDS spreads read traffic across different replicas depending on the load of DPMs. As a result, the total loads across the three DPMs are completely balanced.

4.4 Conclusion

This chapter presents a low-cost way of building in-memory data stores, by removing all the computation needs at where data is hosted. We proposed three DPM architectures, built three atomic, crash-consistent, and reliable data stores on top of them, and performed extensive evaluation of these data stores. Our findings can guide future DPM system builders.

5 SECURITY IMPLICATIONS OF ONE-SIDED COMMUNICATION AND RDMA

The previous two chapters discussed our efforts in building better RDMA system stack and applications for datacenter environments. Our focus as well as focus of most other RDMA-related research have been on performance, scalability, and usability of RDMA, leaving one important aspect unexplored, *security*.

This chapter introduces the security vulnerabilities in one-sided communication. First, one-sided communication's feature of not having any software processing at the receiving side poses unique threats in distributed systems. For example, malicious clients can read or write to remote storage servers with one-sided network requests without being noticed, making it hard to account for errors. Attackers can also easily launch Denial of Service attacks using one-sided communication to swamp the network or the target machine's memory or storage resources.

Second, in order to bypass host processors, hardware devices that enable one-sided communication have to implement functionalities that are traditionally built in software. We identified several security issues in one-sided hardware, some caused by features added for better performance, some by the need to bypass host CPU, and some by ill-designed implementation.

To study the security issues in one-sided communication deeper, we focus on the most popular one-sided communication technology, RDMA, and introduces *Pythia*, a set of side-channel attacks that can be launched completely from the network through RDMA. With limited on-board SRAM, RDMA NICs cannot cache all the metadata such as page table entries and have to move metadata between their SRAM and the host machine's main memory through the PCIe bus. As a result, there exists a timing difference between RDMA operations that hit or miss SRAM. We demonstrated the

feasibility of exploiting the above vulnerability to launch side-channel attacks on RDMA-based systems.

Finally, we describe the opportunities to achieve user privacy by one-sided communication. One-side communications offer unique opportunities to help enhance security. Indeed, one-sided network communication is a *double-edged sword* in security.

The contributions of this security study are:

- Identify fundamental limitations in one-sided communication that can pose security threats.
- Discovery of new side channels in RDMA-based systems that leak client RDMA access patterns.
- Reverse engineering of the most widely used RDMA NIC hardware architecture, which can be leveraged in designing efficient side channels.
- Design, implementation, and evaluation of, Pythia, a set of RDMA-based side-channel attacks, which are fast, accurate, and can be launched solely from a separate machine across the network.
- Discussion of possible mitigations, most of which are uniquely applicable to RDMA systems.
- leverage one-sided communication to enhance the performance of an ORAM system.

5.1 Vulnerabilities in One-Sided Communication

While the processing needs at receiver nodes increase CPU utilization in two-sided communication, the receiver's processing software stack provides the means to implement various security defenses. In contrast, the lack of receiver involvement in

one-sided network communication poses several security threats in datacenter environments. A major threat model we envision is a cloud environment that provides in-memory data store services to cloud users. A victim accesses data that a server hosts by issuing one-sided network requests from a client machine. The attacker runs on another client machine and is another user of the cloud data store service (*i.e.*, the attacker has no access to the victim or the server machines).

This section presents four real cases of security issues in two different types: one-sided abstraction and one-sided hardware devices.

Case 1: threats to accountability. In distributed, multi-tenant datacenter environments, many parties can potentially cause errors (*e.g.*, node failures, data corruption, stealing information). It is important to *account* for errors when they happen. Accountability enables the pinpointing of the party responsible for a problem and allows other parties to be proven innocent [213]. In a threat model where a *server* hosts data that multiple *client*s can read and write, an attacker can be one of the clients that desire to write malicious data or read other clients' data without being noticed.

Under two-sided communication, the receiver handles all incoming network requests and can identify their senders, providing a means for accountability. However, it is fundamentally difficult for one-sided communication to be accountable, because CPU and software are completely bypassed at the receiving side. Attackers can exploit this vulnerability to perform one-sided network operations without being detected. For example, in RDMA-based in-memory storage systems that support one-sided writes [26, 67, 214], an attacker client can write malicious data to any locations in the store without being detected. One possible way benign clients can avoid reading unaccountable data is to authenticate the writer of the data with encryption keys. However, this mechanism needs a fair amount of computation, and the performance overhead is especially large considering one-sided networks' high speeds [56, 215].

Case 2: threats in denial of service. Without any processing at the receiving side, one-sided communication also makes it hard to throttle the speed of network requests

from any particular sender. This limitation can be exploited to launch different types of Denial of Service (DoS) attacks [216]. Our threat model here is a cloud environment where an attacker has one-sided network accesses to a server that provides some service (*e.g.*, an RDMA-based key-value store, GPU acceleration with GPUDirect) to multiple clients. An attacker can flood the network to the server with a large number of one-sided network requests without being detected.

Another potential DoS attack that is different from traditional network DoS attacks is to exhaust a server's hardware resources that are used to provide one-sided communication services. For example, RDMA NICs (*RNIC*s), devices that enable one-sided RDMA operations, store metadata for network connections and memory spaces in on-NIC SRAM so that RNICs can process incoming one-sided requests without involving the host machine. Attackers can fill the on-NIC SRAM by accessing many different memory spaces, forcing the receiver's RNIC to evict victims' metadata. With one-sided communication, lots of metadata are offloaded into hardware where the resource is really restricted in comparison with the resource in software in traditional two-sided communication. This makes the server more vulnerable to DoS attacks. To make it worse, the server cannot tell which client is the attacker, because the attacker's one-sided network traffic cannot be detected by the server.

Case 3: hardware-managed "keys" are not secret. RNICs use a pair of "keys", *lkey* and *rkey*, to protect the local and remote access of each *memory region* or *MR*. An RNIC of a machine generates and stores lkey and rkey (together with the memory address) at the time when a user process registers an MR on the machine. Applications running on other machines use the virtual memory address of a registered MR and its rkey to access it. When receiving an RDMA request, the RNIC checks its access permission using the rkey.

Surprisingly, we discovered that RNICs generate rkeys in a predictable, sequential pattern with Mellanox ConnectX-3, ConnectX-4, and ConnectX-5 RNICs, the three most popular generations of RNICs. Figure 5.1 plots the values of lkey/rkey of the first 5000 MRs that are registered at a host in the order of the registration time (lkey

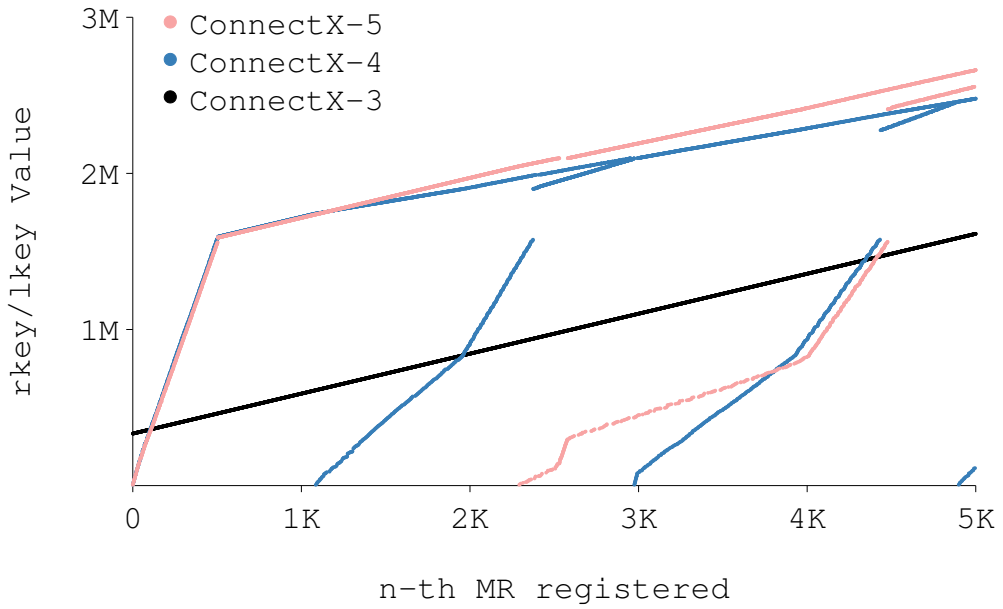


Figure 5.1.: **MR Key Value** *X axis represents the sequence of MRs as they are generated in time.*

and rkey have exactly the same values for all MRs). For all the three generations of RNICs, there are clear and easy-to-guess patterns in lkey/rkey values. Moreover, the first 500 MRs have sequentially increasing lkey/rkey values.

Most RDMA-based in-memory storage systems use a small number of large MRs [29, 214], making it even easier for attackers to guess MR keys. After guessing both the rkey and the memory address of an MR (the latter can be guessed in a similar way as traditional buffer-overflow attacks), attackers can gain full access to the MR, overwriting victims' memory content or stealing their data.

Case 4: exposing physical memory addresses to remote machines. Another case where RNIC design can threat security is a feature designed to improve performance. By default, user processes register MRs with RNICs using virtual memory addresses in their address spaces and RNICs store the virtual to physical memory address mapping in their SRAM. To eliminate the space and performance overhead of address mapping, Mellanox ConnectX-4 and later versions of Mellanox RNICs introduce a new feature that allows user processes to register MRs directly using

physical memory addresses [217]. Applications running on other machines can use these physical memory addresses to access the MRs.

Exposing physical memory addresses poses many new security threats, since knowing physical memory address layout is the basis of many attacks. For example, knowing physical memory addresses make both rowhammer [218–220] and throwhammer [221] attacks easier. Although registering MRs with physical address improves performance, we recommend applications builders to take caution when using this feature.

Discussion and defenses. Attacks that can successfully exploit the Case 1 and Case 2 rely on the assumption that their traffic cannot be sniffed by trusted parties. We believe this assumption to be reasonable because most packet sniffing tools require the *root* privilege to run and datacenter administrators usually prohibit packet sniffing. For example, packet sniffing is prohibited by RNIC by default and requires the RNIC administrator privilege to change the default configuration [222–224].

While many traditional defense mechanisms may not work or will break the one-sided communication pattern, we identified two possible directions for future defenses. First, at the sender, we can employ defense mechanisms to monitor or control network activities before network requests are sent out with the help of an intermediate layer like LITE [214] or with traffic dump tools [222, 223]. Second, at the receiver, we can enhance the hardware devices that handle one-sided requests with better security features, for example, by programming SmartNICs.

While the Case 3 and Case 4 all happen with Mellanox RNICs, we believe that similar issues can happen with other one-sided hardware and that there are deep-rooted reasons for why they happen. There are clear tradeoffs between including more security functionalities in hardware devices and the devices' performance and cost. For one-sided devices, vendors usually choose the latter over security, because what makes one-sided devices appealing is their superior performance and low cost (on saving CPU utilization). For example, the side channel threats in RDMA are a result of vendors choosing performance (caching hot metadata in on-board SRAM)

and cost saving (maximize the utilization of SRAM by not isolating SRAM space across applications) over security.

Mitigating security issues in one-sided hardware implementation is possible. For example, RNICs can use cryptographically generated keys as lkey and rkey (Case 3); they can isolate SRAM for different applications or introduce noise to disturb timing differences; and they can remove the exposure of physical memory addresses (Case 4). Certain one-sided hardware vendors have manufactured SmartNICs that supports one-sided network communication [56, 194]. Various defense mechanisms can be implemented on these SmartNICs (*e.g.*, encryption). Despite the promise of the above mitigations, it is still challenging but important to deliver security features with minimal impacts on performance, cost, and hardware complexity.

5.2 Pythia: Remote Oracles for the Masses

After studying the vulnerabilities in one-sided communication, in this section, we focus on RDMA, and introduce Pythia, a set of side-channel attacks in RDMA.

The need for RDMA NICs to bypass CPU and directly access memory result in them storing various metadata like page table entries in their on-board SRAM. When the SRAM is full, RNICs swap metadata to main memory across the PCIe bus. We exploit the resulting timing difference to establish side channels and demonstrate that these side channels can leak access patterns of victim nodes to other nodes.

The basic idea of Pythia is to issue RDMA network requests to the server to fill its RNIC SRAM, eventually evicting the metadata of the target data. Then the attacker reloads the target data with an RDMA request and based on the time it takes, predict if the victim has accessed the data.

Although the basic idea is similar to the Evict+Reload CPU cache side-channel attack [128], designing Pythia presents many new challenges. The first challenge is the difficulty in achieving good eviction performance. Existing CPU-cache based side-channel attacks leverage cache associativity to reduce the eviction set size, thereby

improving eviction performance. However, RNICs are vendor owned and are complete black boxes to public knowledge. To confront this challenge, we reverse engineered the memory architecture of the Mellanox ConnectX-4 RNIC [54], the type of RNIC that is used in all major datacenter RDMA deployment. We successfully discovered the internal architectural organization of RNIC SRAM and leverage this knowledge to achieve low-latency eviction.

The second challenge is in the reload and prediction process. Because of our environment of being in a shared datacenter network, the latency of an RDMA request can vary with different network state. The traditional approach of using a static threshold to differentiate cache hit from cache miss is not a good fit for our environment. We take an adaptive approach to dynamically train a hit/miss classifier based on RDMA access latency at the time of attack and use the trained classifier to statistically predict victim accesses [133, 225].

We evaluated Pythia in our lab environment and in a public cloud [226] with four different types of RNICs. Pythia completes one Evict+Reload cycle (across the network) in as low as 57 μs with 97% accuracy. The definition of accuracy throughout this chapter is the percentage of successful guesses over total guesses. Moreover, Pythia effectively hides its traces from the server and victims because it performs all its attack using RDMA operations from a separate machine.

We further built three variations of Pythia to attack a real RDMA-based system, the Apache Crail key-value store system [34, 35]. On a real application like Crail, it is more challenging to establish a strong side-channel attack because of limited application interface and noise coming from application performance overhead. After improving Pythia to accommodate these difficulties, we successfully launched a side-channel attack solely from a separate client machine using the unmodified Crail client interface. This attack is efficient and can accurately learn a victim's key-value pair access patterns.

Pythia is the first work that explores side-channel vulnerabilities in RDMA and exploits the vulnerabilities to launch attacks on RDMA-based datacenter systems.

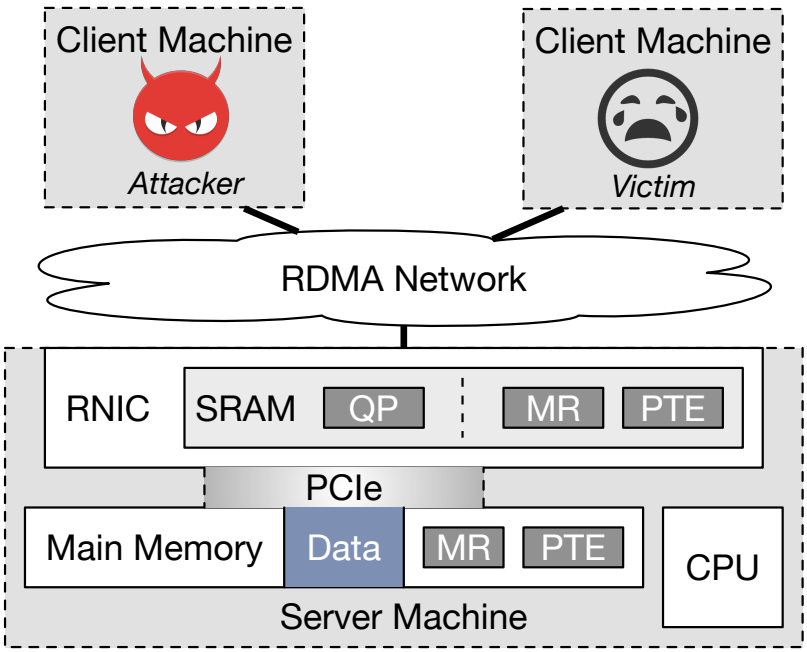


Figure 5.2.: **Attack Environment and RNIC Architecture.** *The attacker and the victim are both clients that can access data in the server machine's memory throuth RDMA.*

With today's datacenters all having robust defenses against direct sniffing or hijacking of network traffic, side channels are more feasible attack mechanisms and we believe that our work raises serious security concerns in a young but already widely-adopted network technology. We have responsibly disclosed the weaknesses to Mellanox and Crail.

5.2.1 Threat Model

In our attack, there are three parties: the server which hosts data in its main memory for other client machines to access (*e.g.*, an in-memory database or an in-memory key-value store), the victim who accesses the server's in-memory data through RDMA, and the attacker who tries to infer the victim's accesses and access patterns. The attacker and the victim are both normal clients that can access the data store service the server provides, and they run on separate machines. Following the threat mod-

els of related work that introduces and evaluates side channels, we assume that the attacker does not have direct control over the victim. As victim and attacker execute on different machines and communication to the server happens through the network, we assume that the attacker *cannot* observe the victim's network packets (as otherwise, the attacker could directly infer the accessed addresses and values as RDMA is currently not encrypted). This assumption is reasonable as sniffing victim's packets would require an attacker to have `root` access on either the victim's machine or the server's machine [222–224] or to launch man-in-the-middle attack to the network, both of which are well defended in cloud datacenters. Figure 5.2 illustrates this environment. We also assume that the server cannot *directly* observe memory accesses of either the victim or the attacker as both victim and attacker interact with the server through one-sided RDMA operations, not involving the server's CPU.

5.2.2 Side-Channel Attacks on RDMA

RDMA exposes node-internal memory to external hosts. Due to best practices of optimizing accesses and caching, current RDMA hardware is vulnerable to a variety of timing side channels. We present an overview of RDMA-based side channels, two basic attacks, refined attacks with our reverse engineered knowledge of RNIC internals, and evaluation results of the attacks. Unless otherwise stated, all our experiments use three machines, each equipped with a Mellanox ConnectX-4 100 Gbps network adapter [54], two Intel Xeon E5-2620 2.40GHz CPUs, and 128 GB main memory. They are connected with a Mellanox SB7700 100 Gbps InfiniBand switch [227]. One machine is used as the server that serves in-memory data through RDMA. The other two machines are the victim client machine and the attacker client machine, both of which can perform RDMA operations to access data on the server. In all the experiments in this section, the victim has a 50% chance of accessing the targeted data that the attacker tries to infer accesses on.

Attack Overview

The basic idea of our side-channel attacks is to exploit two weaknesses in RNIC: 1) the RNIC caches metadata in its SRAM, RDMA accesses whose metadata is not in SRAM must wait until that data is fetched from main memory, and 2) all RDMA accesses from all applications share the RNIC SRAM. As explained in Section 2.1, RNICs store three types of metadata in their SRAM: QP information, MR information, and PTEs. An RDMA access involves all three types of metadata: upon receiving a network request, an RNIC needs to locate which QP the request belongs to, which MR it falls into, and which page it is accessing. If any of these metadata is not in the RNIC SRAM, the RNIC will fetch it from the host memory, stalling the request until the required data arrives. By exploiting this timing difference, we can launch side-channel attacks to know which QP, which MR, and which PTE the victim has accessed.

Traditional CPU-cache-based side-channel attacks take three major forms: *prime*-based (*e.g.*, Prime+Probe [119–126]), *flush*-based (*e.g.*, Flush+Reload [129]), and *evict*-based (*e.g.*, Evict+Reload [128]). Because RNICs do not provide any interface to flush their SRAM, all flush-based side-channel attacks are incompatible with RDMA such as Flush+Reload [129] and Flush+Flush [228]. All the operations that are needed in prime-based and evict-based attacks can be implemented through RDMA network requests that an attacker performs over the network. Attackers can hide their traces during the attacks, since one-sided RDMA reads are oblivious to the server or other clients. Hardware performance counters [229] may help servers track DMA traffic, but it is challenging to associate traffic with RDMA accesses or attacks. Even if the server suspects that an attack is happening, it is still hard, in practice, to attribute an attack based on traffic counters.

For the rest of the chapter, we focus on RDMA-based Evict+Reload attacks. Prime+Probe attacks are also possible on RDMA and we briefly discuss them in Section 5.2.5.

Unique Advantages and Challenges

There are three unique advantages for attackers in RDMA systems. First, RDMA's one-sided communication pattern allows the attacker to hide her traces, since the receiving node is unaware of any one-sided accesses. Second, the RDMA network is much faster both in latency and in bandwidth than traditional datacenter networks. The latest generation of RDMA switches and RNICs can sustain 200 Gbps bandwidth and under $0.6\,\mu s$ latency [19]. RDMA's superior performance enables fine-grained, high-throughput, timing-based side-channel attacks over the network. Finally, RDMA's one-sided communication bypasses the sender's OS and does not involve CPU at the receiver, both of which help reduce disturbance to timing-based attacks.

At the same time, attacking the RNIC presents several novel challenges that no CPU-cache-based side-channel attack experiences. First, it is hard to discover efficient side channels in RNIC hardware. Unlike CPU caches, there is no public knowledge of how RNICs organize or use their SRAM. RNICs store different types of information in SRAM compared to a linear layout of CPU caches. Second, we set a strict threat model where attacks are launched from a separate machine that is different from the victim's machine and the server machine. This goal means that attacks have to be performed using RDMA network requests only. Finally, since our side channels are established over the network, noise in the network could potentially increase difficulties for timing-based attacks.

Basic Attack

Before presenting our side-channel attacks, we first discuss the type of victim information we choose as our attack target and the type of metadata we use to perform the eviction phase. Notice that these two dimensions are orthogonal and both have three options: QP, MR, and PTE.

Among these three types of information, knowing which QP the victim accesses leaks little information about the victim and usually is not useful in real attacks. MRs and PTEs can both leak more information. Using PTE as the attack target unit will reveal memory page (in virtual memory) accesses. All OSes use 4 KB as the default page size. The MR size is decided by the application that creates and registers it. For performance reasons [163], most RDMA-based applications choose to use large MRs. Thus, we choose PTE as the target of our attack. However, most of our ideas and techniques can be used to perform attacks that target MRs.

After choosing the attack target, we must decide what metadata to use to evict RNIC SRAM. To answer this question, we tried to evict SRAM using the three types of metadata and reload our targeted information (*i.e.*, PTE). We can successfully evict a PTE with PTEs, no matter whether or not the PTEs we use to evict belong to the same MR as the target PTE. We can also evict an MR with other MRs. We further find that when an MR is evicted, PTEs of all the pages belonging to this MR will also be evicted. But we can only use QPs to evict QP. This behavior implies that RNICs isolates the SRAM used for QPs and for MRs and PTEs. We present the evaluation results later this section. From this initial test, we discovered that we can evict a PTE by either evicting the MR it belongs to (using a large number of MRs) or by evicting the PTE directly using a large number of other PTEs.

Eviction by MRs

We now present our attack that evicts SRAM with MRs, $PythiaMR$. Algorithm 1 presents the pseudocode of $PythiaMR$.

Because the MR-based attack requires the attacker to use many MRs to evict the server's RNIC SRAM, the attacker requires access to a sufficient amount of MRs. If the number of MRs is restricted, the attacker may resort to a (hypothetical) MR gadget that allows her to register multiple MRs. One approach is to launch a process on the server that allows her to register multiple MRs (see Section 5.2.3 for details). Since RDMA provides the functionality of registering multiple MRs with the same

Algorithm 1: MR-based eviction

```
Input  : a target victim virtual memory address
Output:
if No access to sufficient amount of MRs then
    start process at server to create MR_set;
else
    foreach MR that the attacker has access to do
        if MR ≠ victim's MR then
            insert MR into MR_set;
        end
    end
end
foreach MR in MR_set do
    perform 8-byte RDMA-read to MR;
end
```

memory space, the attacker process at the server only needs to allocate a small memory space (of arbitrary size) and register it multiple times. This process then needs to send the *rkey*s corresponding to these registered MRs to the attacker running on a client machine.

In the eviction phase, the attacker performs one-sided RDMA reads from a client machine to the MRs it has access to at the server (except for the MR that the victim PTE is in). Since the server's RNIC needs to fetch and store metadata for each MR when the MR is accessed, its SRAM will eventually be filled with MR metadata that the attacker accessed.

Eviction by PTEs

Alternatively, we can use PTEs to establish a side channel. Compared to MR-based attacks, PTE-based attacks can often be performed entirely from a client machine. Algorithm 2 presents the pseudocode of our PTE-based attack.

To perform PTE-based eviction, the attacker issues one-sided RDMA reads to a sufficiently big memory space (1 GB to 4 GB for the RNICs we study). Most RDMA-

Algorithm 2: Naive PTE-based eviction

```
Input  : a target victim virtual memory address
Output:
VictimVPN ← victim_address >> 12;
generate eviction_set using VictimVPN;
foreach VPN in eviction_set do
    perform 8-byte RDMA-read to address VPN << 12;
end
```

based applications are services that provide in-memory data storage and use a large amount of memory, thus meeting the requirements of PTE-based attacks.

Accessing different memory pages will cause the RNIC to fetch PTEs to its SRAM. Because all accesses to the same memory page will hit the same PTE, we only need to perform one RDMA read (with the smallest RDMA operation size of 8 bytes) in every 4 KB virtual memory address range. To avoid loading the PTE that the victim accesses, we skip the memory addresses that are close to the victim address.

Reload and Predict

After the eviction phase (either by MRs or by PTEs), the attacker reloads the targeted victim data. If the reload time is smaller than a threshold, the attacker determines that the data has been accessed by the victim. Algorithm 3 presents the pseudocode of the reload and prediction process.

The threshold used at the reload time directly affects the result of an attack. Different from traditional CPU-cache side-channel attacks, our attacks are in a distributed environment where network status can vary with other workloads in the datacenter. Thus, instead of a fixed threshold, we adapt the threshold dynamically according to the network status. To adjust the threshold, the attacker periodically measures the latency of an RDMA operation which hits the RNIC SRAM and the latency of one that misses the SRAM. The threshold can be set as a value in the middle of these two latencies.

Algorithm 3: Reload and predict

Input : a target victim virtual memory address
Output: prediction of if the victim has accessed the target address

determine *Threshold* according to network status;

start timer;
RDMA-read *victim_address*;
end timer;
time ← *elapsed_time*;

if *time* ¡ *Threshold* **then**
  output *accessed*;
else
  output *not_accessed*;
end

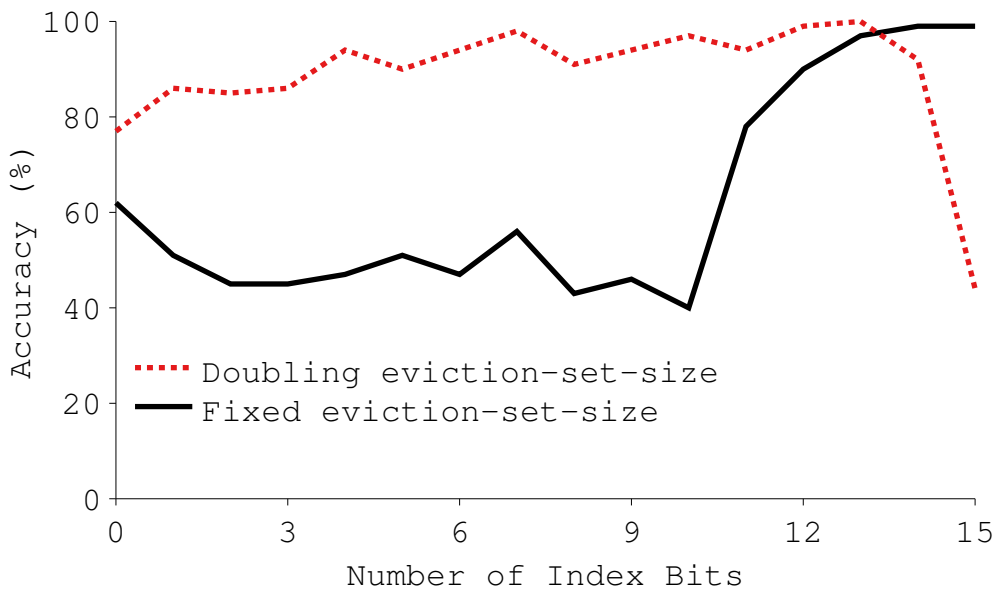


Figure 5.3.: **Effect of Number of Index Bits.**

In addition to using an average threshold, other more advanced methods can also be used to determine the reload result. In fact, we design a statistical method to determine the reload result for our real-application attacks, as will be presented in Section 5.2.3.

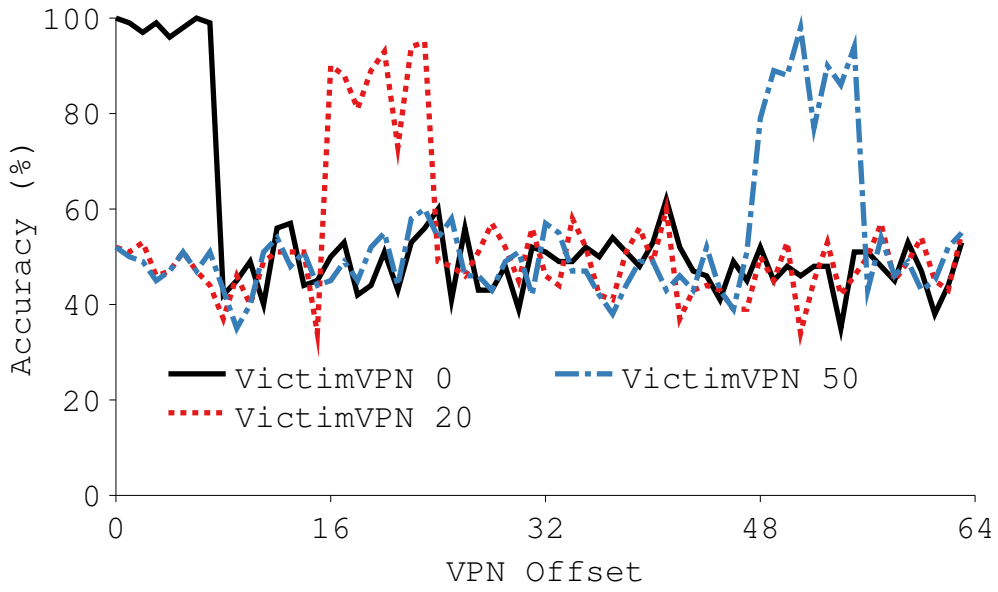


Figure 5.4.: **Effect of Eviction Set Offset.** *X axis represents the first VPN in an eviction set (i.e., the "offset" of an eviction set").*

Finding PTE Eviction Sets

We call the attack that uses the eviction phase presented in Algorithm 2 the basic PTE-based attack, or $PythiaPTE_{Basic}$. In order to reduce the time to perform eviction and improve the efficiency of $PythiaPTE_{Basic}$, we search for a smaller eviction set that achieves similar accuracy as $PythiaPTE_{Basic}$. Specifically, we perform a set of experiments to systematically reverse engineer the internal organization of RNIC SRAM and use our learned knowledge to construct a minimal PTE eviction set.

Reverse engineering RNIC SRAM organization is significantly harder than reverse engineering traditional CPU caches because there is no public knowledge of the internal organization of any RNIC. All we know is that the RNIC caches three types of metadata (PTEs, MRs, and QPs) in its SRAM. Moreover, reverse engineering the

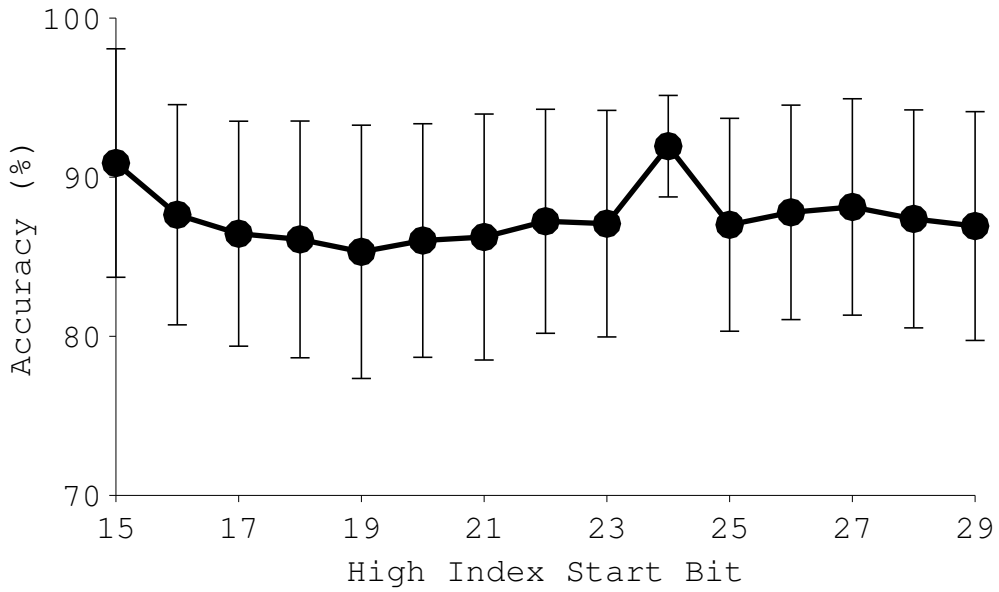


Figure 5.5.: **Effect of Secondary Index.** *Error bars show the standard deviation across 1000 VictimVPNs.*

RNIC involves network operations which add noise compared to a well-isolated CPU cache environment.

First attempt in finding index bits. We initially guess that RNICs organize their in-SRAM PTE caches as set-associative caches, similar to how CPUs organize their caches. To validate this guess, we assume that the PTE cache is organized as a fixed number of sets (*e.g.*, 2, 4, 8) and different number of bits (*e.g.*, 1, 2, 3) are used to calculate the index into these sets. Since PTEs are identified by virtual page number (*VPN*), we can ignore the lowest 12 bits (with page size being 4 KB). We then use the lowest K bits of *VPN*s to calculate the index into one of the 2^K cache sets. These K bits correspond to the 12th to the $(11 + K)$th bits of the full virtual memory address (we call the lowest bit the 0th bit and count upwards to higher bits).

We call the *VPN* of a victim memory address *VictimVPN*. The eviction set of a *VictimVPN* is formed by setting the same K bits as the *VictimVPN* and varying other bits in *VPN*s. To put it another way, for every 2^K pages, we pick one *VPN* to add to the eviction set. We keep adding distinct *VPN*s in this way until the number

Algorithm 4: Forming Eviction Set - Strawman

```
Input  : VictimVPN, eviction set size, num of index bits
Output: an eviction set targeting VictimVPN

eviction_set ← {};
mask = VictimVPN & (1 << num_index_bits − 1);

for i = 0 to evict_set_size do
    VPN ← i << num_index_bits + mask;
    if VPN ≠ VictimVPN then
        insert VPN into eviction_set;
    end
end

output eviction_set;
```

of *VPN*s in the set reaches an *eviction set size*. Algorithm 4 presents the pseudocode of how we form an eviction set. This is our straw-man approach and we call the PTE-based attack that uses this approach of forming eviction sets $PythiaPTE_{Straw}$.

Figure 5.3 plots the attack accuracy when we vary K from 0 to 15. We use two groups of attacks on *VictimVPN* 0. In the first group (the solid line), each eviction set has $2^7 = 128$ operations. The accuracy keeps improving until K is 13 and then flattens out. This result hints at the possibility of the RNIC using a set-associative cache with $2^{13} = 8192$ sets. It is because when K is smaller than 13, part of the operations in the eviction set will fall into a different cache set as the *VictimVPN*'s set, making the eviction set too small to evict the whole victim's cache set.

To verify this guess, we perform a second group of attacks (the dashed line). In this group, we double the eviction set size every time when we decrease K by 1. For example, we set the eviction set size to 128 when K is 13 and to 256 when K is 12. If the RNIC uses 8192 cache sets, then when K is 13, all of the eviction set will fall into the *VictimVPN*'s set, and when K is 12, half of the eviction set will fall into the *VictimVPN*'s set. These two attacks will then have the same effect in evicting the *VictimVPN*. Our results confirm this assumption. When K is less than 13, the attack accuracy is similar to when K is 13. However, when K is larger than 14, the

accuracy drops. This is because we only use 64 and 32 eviction set size when K is 14 and 15, and these eviction sets are not large enough to evict a whole cache set.

From this set of experiments, we suspect that virtual memory address bits 12 to 24 are used in calculating the PTE cache set index and that each PTE cache set has 128 entries (*i.e.*, a 128-way cache).

Discovering prefetching behavior. Our guess above uses 13 bits as the index into the PTE cache and assumes that the PTE cache has 8192 sets. If this guess is correct, then an eviction set whose (VPN%8192) is different from ($VictimVPN$%8192) should fall completely into a different cache set from the victim's. To verify this assumption, we perform another set of attacks to the $VictimVPN$ 0. In the nth attack, we construct its eviction set using VPNs where (VPN%8192 $= n$), and we change n from 0 to 8191.

Figure 5.4 plots the accuracy of the first 64 attacks (the rest of the attacks have the same pattern and we omit them from the figure). The black solid line shows the result of $VictimVPN$ 0. Surprisingly, not only the 0th attack (VPN%8192 $= 0$) has high accuracy, but also the 1st to the 7th attacks. To further understand this effect, we perform the same set of attacks on another two $VictimVPN$s, 20 and 50. With these $VictimVPN$s, the accuracy is high from 16th to 23rd attacks and 48th to 55th attacks respectively. These results imply that evicting any VPN within an eight-VPN range of ($VictimVPN - VictimVPN$%8) to ($VictimVPN - VictimVPN$%8 + 7) has the same effect.

We suspect that the RNIC prefetches eight VPNs at a time. This finding implies that instead of using 13 bits as cache index bits as in $PythiaPTE_{Straw}$, only the higher 10 bits (bits [15:24]) are used for index and the lower 3 bits (bits [12:14]) are used for prefetching. The RNIC cache thus only has $2^{10} = 1024$ sets.

Discovering secondary index. Our attack strategies so far work well ($> 90\%$ accuracy) with the $VictimVPN$s we tested (*e.g.*, 0, 20, 50). However, when we test the same attack strategy on some other $VictimVPN$s (*e.g.*, 6195, 30950), the accuracy can sometimes drop to $\sim$75%. A dropped accuracy means that our eviction set cannot

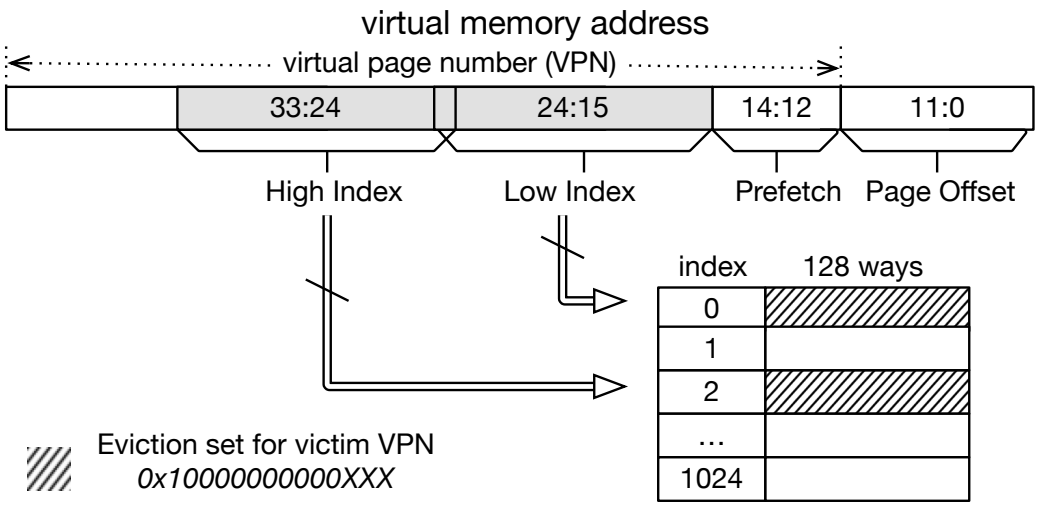


Figure 5.6.: **Reverse-Engineered PTE Cache Organization.**

evict the *VictimVPN*, *i.e.*, the *VictimVPN* is in another cache set whose index is not the same as the index calculated using bits [15:24].

We thus suspect that there exists another 10 bits that are used to calculate where out of the 1024 sets a *VictimVPN* can fall into. To answer this question, we use a moving window of 10 bits, from bits [15:24] to bits [29:48], in the victim's virtual memory address. We randomly pick 1000 *VictimVPN*s and attack each of them by forming an eviction set with an index calculated with the moving window of 10 bits and an index calculated with bits [15:24] (64 operations under each index). We use two indices in this experiment because our alternative experiment of using just the index calculated by the moving window does not yield good accuracy. Figure 5.5 plots the average attack accuracy across 1000 *VictimVPN*s and their standard deviation (in error bars). Bits [24:33] yields the best average accuracy and smallest deviation. Thus, we believe that these bits are used as a second index into the PTE cache. Note that when the moving window is bits [15:24], the accuracy deviation is high, indicating that using bits [15:24] alone is not good enough.

Complete algorithm. Figure 5.6 presents the final PTE cache architecture we speculate RNICs use based on our reverse engineering results. The PTE cache has

Algorithm 5: Forming Eviction Set - Full

```
Input  : VictimVPN, eviction set size
Output: an eviction set targeting the victim virtual memory address

eviction_set ← {};   prefetch_bits ← 3;   index_bits ← 10;
high_index_start ← 12;
low_index ← (VictimVPN >> prefetch_bits)&(1 << index_bits − 1);
mask_low ← low_index << prefetch_bits;
high_index ← (VictimVPN >> high_index_start)&(1 << index_bits − 1);
mask_high ← high_index << prefetch_bits;

for i = 0 to evict_set_size/2 do
    VPN ← i << (index_bits + prefetch_bits) + mask_low;
    if VPN ≠ VictimVPN then
        insert VPN into eviction_set;
    end
end

for i = 0 to evict_set_size/2 do
    VPN ← i << (index_bits + high_index_start) + mask_high;
    if VPN ≠ VictimVPN then
        insert VPN into eviction_set;
    end
end

output eviction_set;
```

1024 sets and 128 ways. A PTE can be cached at one of the two cache sets. Two groups of bits are used to calculate the index of these two cache sets. The first group is bits [15:24], and the second group is bits [24:33]. We call them *low index bits* and *high index bits*. From our observation, both the high and the low index bits can decide which cache set will be used to cache a PTE. A PTE will be cached in either the cache set calculated by the high index bits or the cache set indicated by the low index bits. Every time when a PTE is accessed, its neighboring PTEs will also be fetched to the same set and bits [12:14] determine the 8 PTEs that will be prefetched.

Based on this reverse-engineered architecture, we present the final PTE-based Evict+Reload attack, $PythiaPTE_{Full}$. We form half of the eviction set of a *VictimVPN* with *VPN*s that have the same low index bits as the *VictimVPN* and another

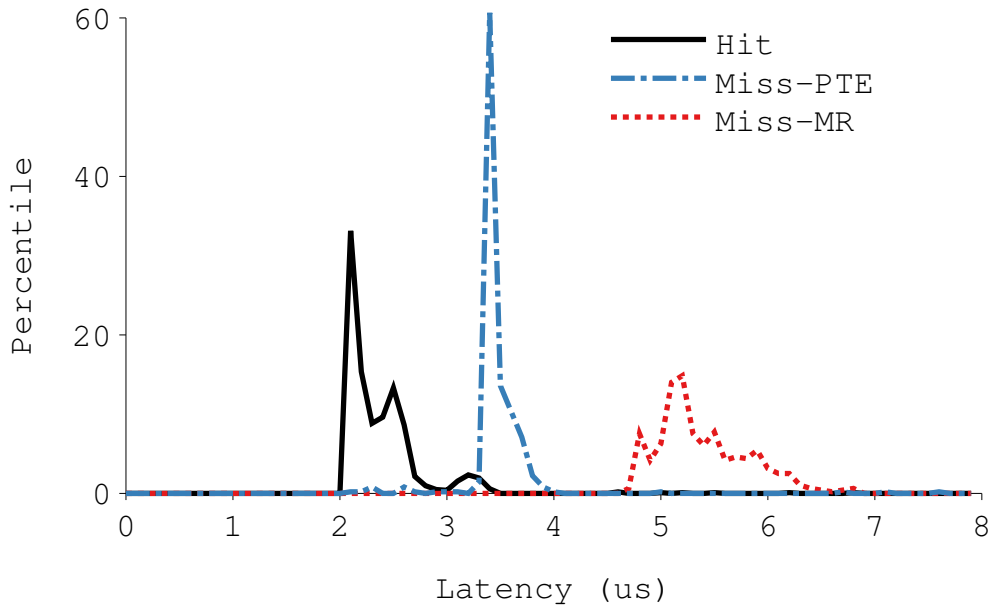


Figure 5.7.: **Timing Differences.** *Each line presents the timing differences of each case over 1000 trials.*

half with *VPN*s that have the same high index bits as the *VictimVPN*. Algorithm 5 presents the complete algorithm.

Evaluation Results

We now present our evaluation results with the attacks described above.

Isolated Environment

We performed a set of experiments with three machines as described in the beginning of this section in our lab environment without any other network traffic.

Timing differences. Our side channels are based on timing differences between consecutive loads of the same memory address. To measure miss latency, we evict the RNIC SRAM using either MRs or PTEs and then issue an RDMA operation. To measure hit latency, we simply repeatedly issue the same RDMA operation. The measurements in Figure 5.7 show a clear timing difference between hit and miss

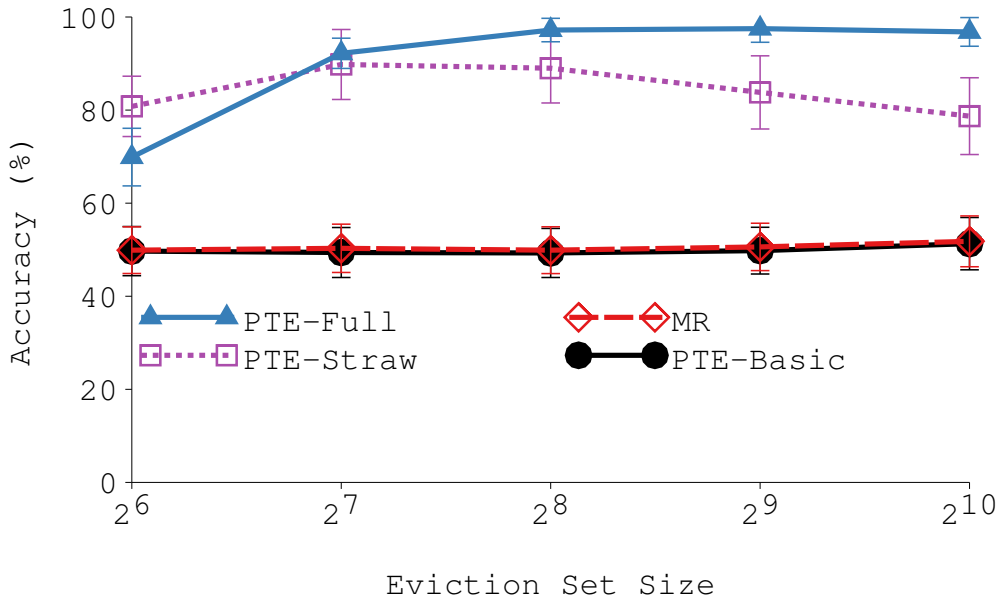


Figure 5.8.: **Accuracy of Attacks.** *Error bars show the standard deviation across 1000 VictimVPNs.*

latency. When we use MRs to evict SRAM, both the victim's MR and all the PTEs under this MR will be evicted. An RDMA operation afterwards will need to fetch both the PTE and the metadata for the MR containing the page from host main memory through the PCIe bus. On the other hand, using PTEs to evict will only evict the victim's PTE and reloading will only fetch the PTE. This explains why the miss latency of MR-based eviction is higher than that of PTE-based eviction.

Attack accuracy and latency. Figures 5.8 and 5.9 plot the accuracy and latency of four attack strategies: $PythiaMR$, $PythiaPTE_{Basic}$, $PythiaPTE_{Straw}$, and $PythiaPTE_{Full}$, as we change the eviction set size. As expected, with the same eviction set size, the time to perform these four attacks is similar, since they all use the same amount of RDMA operations. With bigger eviction sets, all attacks become slower.

$PythiaPTE_{Full}$'s accuracy is the highest: it can achieve 97% accuracy with only 57 μs per attack (when the eviction set size is 256). $PythiaMR$ and $PythiaPTE_{Basic}$

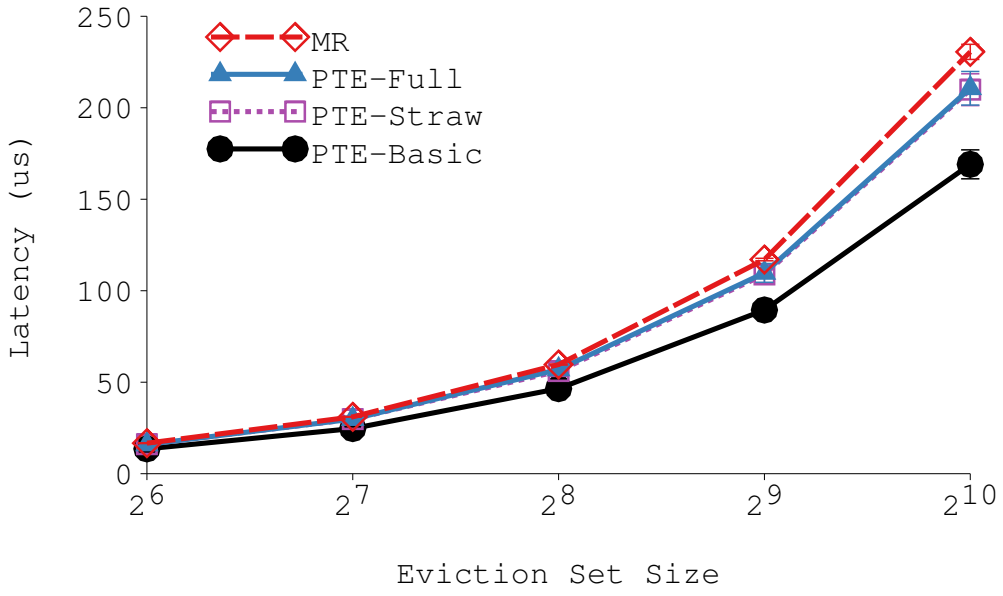


Figure 5.9.: **Latency of Attacks.**

have low accuracy, although we do observe $PythiaMR$'s accuracy improves significantly as the eviction set size increases ($PythiaMR$'s accuracy reaches 90% with 2^{15} eviction set size). This result demonstrates the benefit of using our reverse engineering findings.

Another observation is that the accuracy of $PythiaPTE_{Full}$ peaks when the eviction set size is 256 and remains the same when increasing the size further. This implies that the PTE cache has 128 ways, since we construct two cache sets with 256 entries in total.

Evaluation with different RNICs. All our experiments so far are performed with the Mellanox ConnectX-4 RNIC (most RDMA deployments in real datacenters use ConnectX-4 [18, 230]). We further validate our attacks on Mellanox ConnectX-5 [55] and ConnectX-3 [53] RNICs. ConnectX-5 is the latest generation of RNICs from Mellanox and ConnectX-3 is the previous generation of ConnectX-4.

Figure 5.10 plots the timing results of SRAM hits and misses (due to eviction by MRs and by PTEs) on ConnectX-5. ConnectX-5's performance is better than

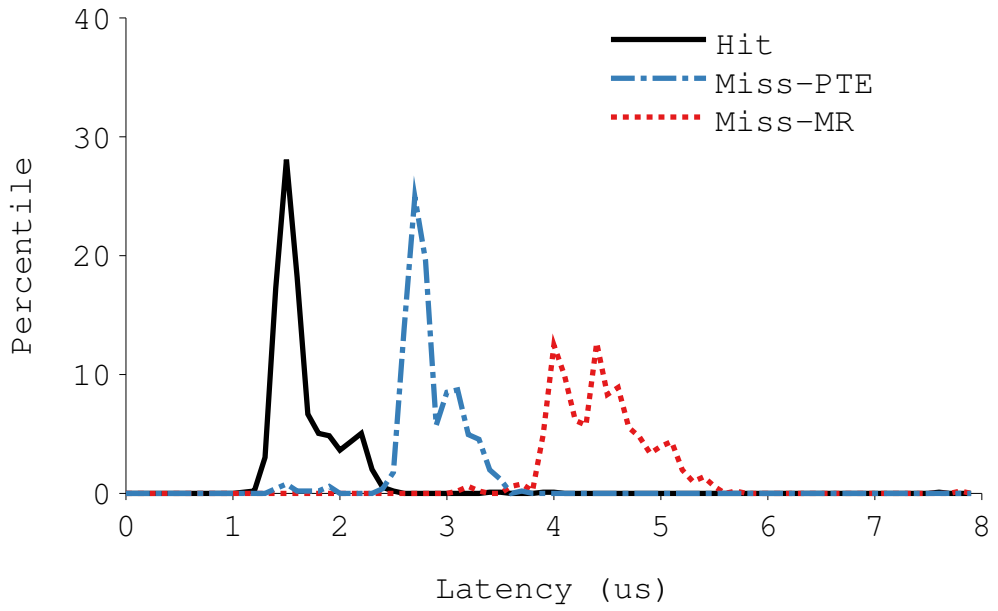


Figure 5.10.: **Timing Differences in ConnectX-5.** *Each line presents the timing differences of each case over 1000 trials.*

ConnectX-4 on all cases. A clear timing difference between misses and hits remains, and misses caused by MR-based eviction are slower than by PTE-based eviction. Figure 5.11 plots the accuracy of the four types of attacks. The accuracy results are similar to ConnectX-4. Attack latency is also similar to ConnectX-4 and we omit the latency figure. Thus, we can confirm that ConnectX-5 uses a similar SRAM architecture as ConnectX-4, and it has the same side channels as ConnectX-4. We can launch the same attacks on ConnectX-5 with high accuracy and low latency.

We then perform the same set of experiments on ConnectX-3, see Figure 5.12. The hit latency with ConnectX-3 is longer than ConnectX-4. As hardware evolves, its internal performance often improves, which can explain why hit latency improves over generations of Mellanox RNICs. Surprisingly, the miss latency due to MR-based eviction is shorter on ConnectX-3 than on ConnectX-4 and ConnectX-5. Misses in RNIC SRAM involve the RNIC fetching metadata from the host main memory. We suspect the reason why miss performance drops in newer generations is because RNICs add more metadata for each data entry in newer generations, requiring longer time

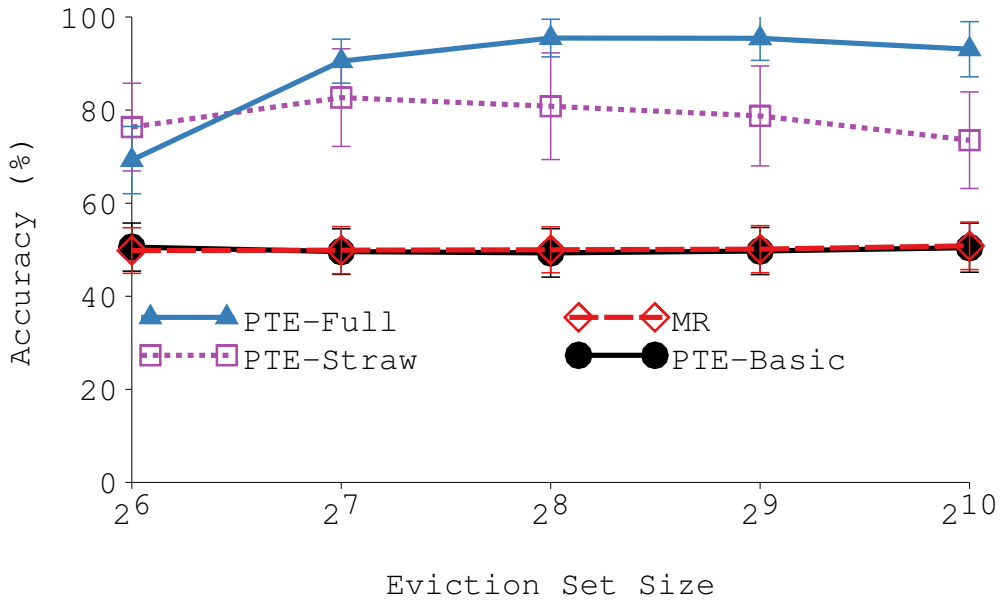


Figure 5.11.: **Accuracy of Attacks in ConnectX-5.** *Error bars show the standard dev of 1000 VictimVPNs.*

to fetch more metadata. As a result, the timing difference between miss and hit for ConnectX-3 is small.

Comparing ConnectX-3, ConnectX-4, and ConnectX-5, the three generations of RNICs from Mellanox, we found that as hardware RNICs evolve, their performance improves quickly, while the PCIe bus and host memory speed improve very slowly. As a result, the discrepancy between hit performance and miss performance becomes larger and we believe that this trend will continue in the future.

Public Cloud Environment

CloudLab [226] is a public cloud that has close to 15,000 cores distributed across three sites in the United States. We evaluated our attacks on a cluster that is connected with RoCE switches. Each machine in this cluster equips two Mellanox ConnectX-4 25 Gbps adapters. These RNICs are of the same product generation as our lab's ConnectX-4 RNICs, with the difference that our RNICs are 100 Gbps adapters. Both types of adapters can be configured for Ethernet (RoCE) and for In-

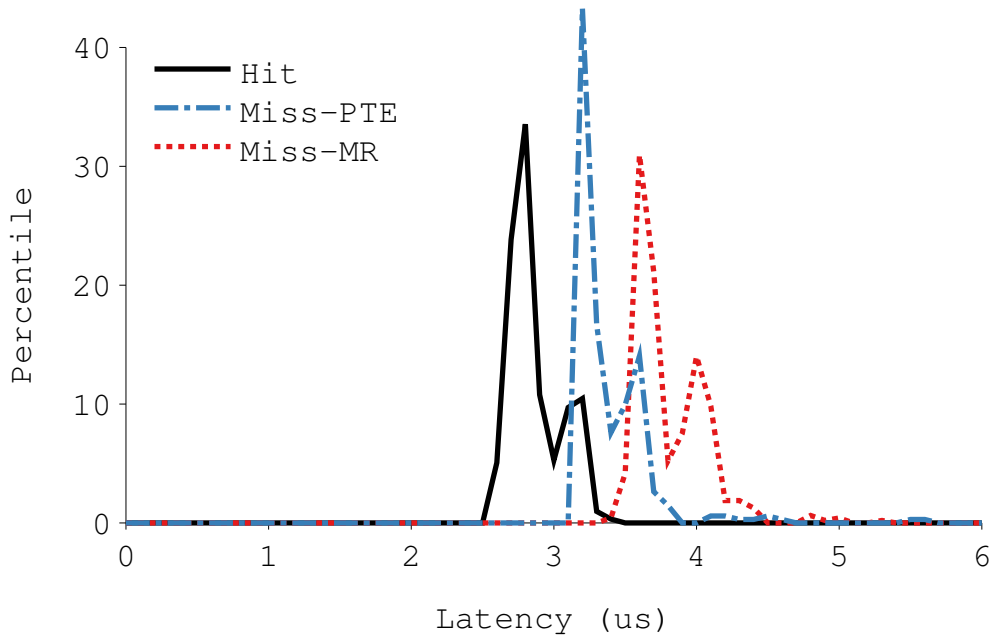


Figure 5.12.: **Timing Differences in ConnectX-3.** *Each line presents the timing differences of each case over 1000 trials.*

finiBand. We configure ours for InfiniBand, and CloudLab's are configured for RoCE. Apart from RNIC differences, CloudLab is used concurrently by many different users; it has a more complex, hierarchical network topology; and it uses a RoCE network instead of InfiniBand. At the time of our test, 129 out of 199 physical machines in the cluster were in use.

We repeat the same set of experiments in Figure 5.8. Similar to our lab's experiments, we use three machines, a server, a victim client, and an attacker client. Figure 5.13 plots the timing difference of RNIC SRAM hit and misses (due to MR-based eviction and PTE-based eviction). Similar to our isolated environment results, misses caused by MR-based eviction are slower than misses caused by PTE-based eviction, and both types of misses are slower than hits. In CloudLab's shared network and shared machine environments, the latencies of all accesses are longer than in our lab environment, but the timing differences are still clear.

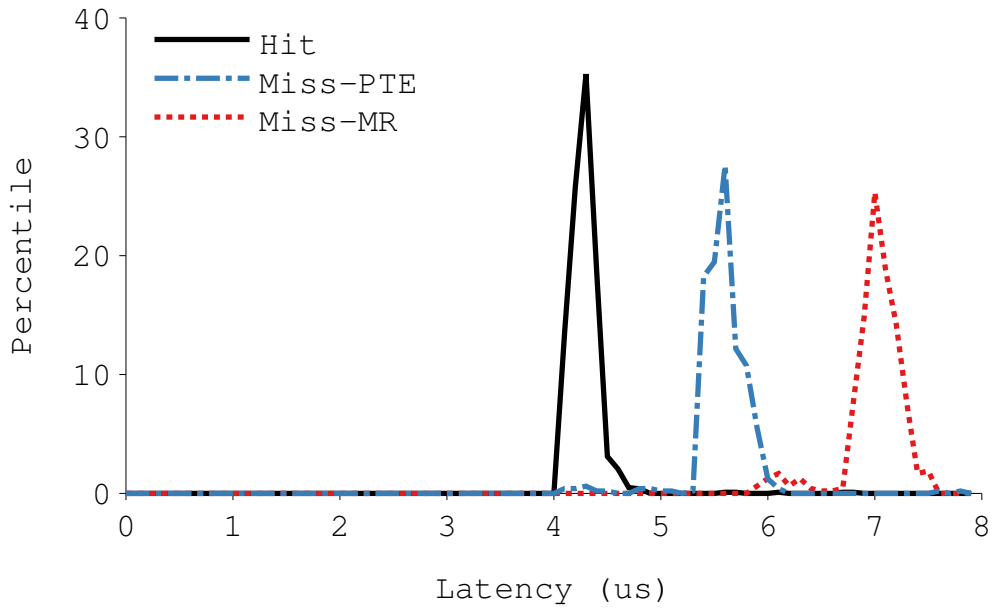


Figure 5.13.: **Timing Differences in CloudLab.** *Each line presents the timing differences of each case over 1000 trials.*

Figures 5.14 and 5.15 plot the accuracy and latency of the four types of attacks in CloudLab. Similar to the results in Figures 5.8 and 5.9, With the same eviction set size (and thus similar latency), $PythiaPTE_{Full}$ and $PythiaPTE_{Straw}$ have higher accuracy than $PythiaMR$ and $PythiaPTE_{Basic}$. However, these attacks have larger variation in accuracy compared to attacks in our lab's environment because of the more dynamic environment in CloudLab.

5.2.3 Attacking Real RDMA-Based Systems

To demonstrate the feasibility of launching side-channel attacks on real RDMA-based applications, we design and perform a set of attacks on *Crail* [34, 35], an open-source RDMA-based key-value store written in Java. A Crail system consists of several roles: a server which stores key-value pairs, a namenode which stores metadata and manages the control path, and clients which issue key-value pair *get*s and *set*s to the server via a Crail-provided API. We install each component on a separate machine

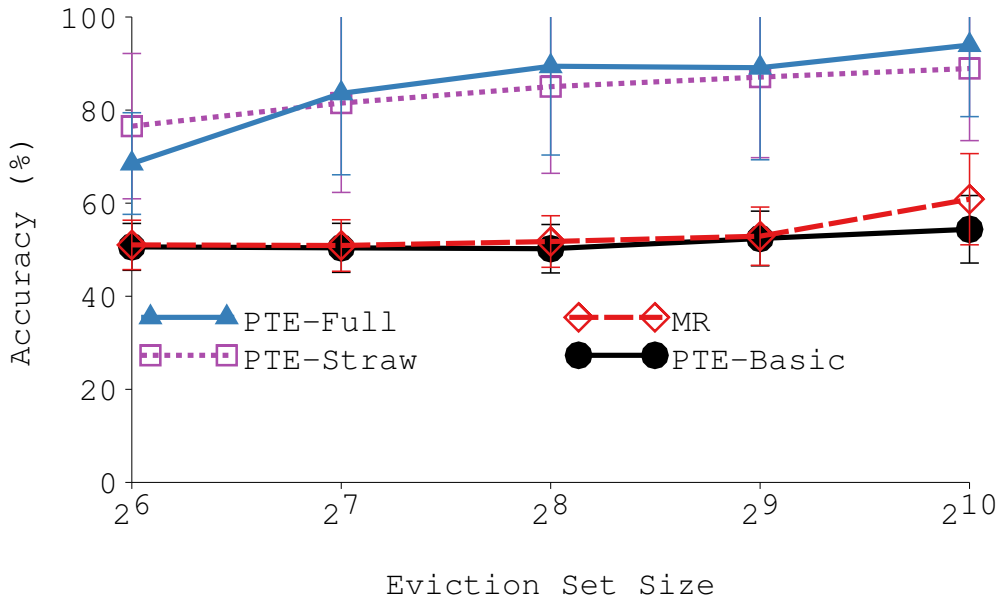


Figure 5.14.: **Accuracy of Attacks in CloudLab.** *Error bars show the standard deviation across 1000 VictimVPNs.*

and connect all of them with RDMA. This section presents our design and evaluation of attacks on Crail.

Attacks

Based on the attack primitives described in Section 5.2.2, we designed three attacks on Crail. All these attacks have the same goal: knowing whether or not the victim Crail client accesses a specific key-value pair.

MR-based attack ($PythiaCrail_{MR}$). Our first attack uses MR-based eviction as described in Algorithm 1. This attack requires three attacker processes. The first is a Crail client process (P_c). The second and the third processes run our attack code, with the second one running on the Crail server machine (P_s) and the third one running on any other machine (P_a) (it can be the same machine as the one where P_c runs). In the preparation phase, P_s registers a large number of MRs. In the eviction phase, P_a issues one-sided RDMA reads to these MRs. Finally, P_c performs a Crail

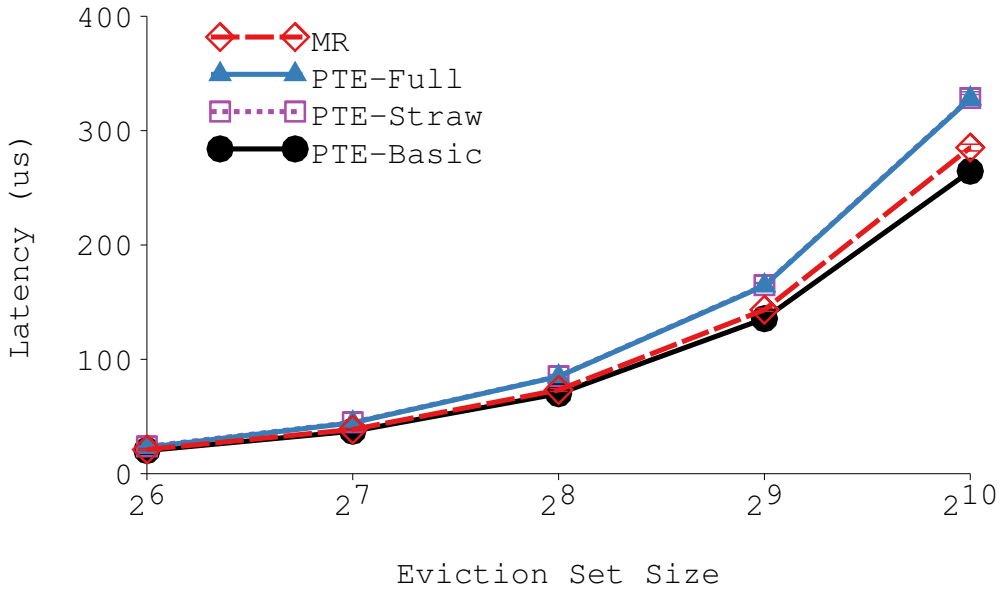


Figure 5.15.: **Latency of Attacks in CloudLab.**

get operation to reload the victim key-value pair. $PythiaCrail_{MR}$ requires P_s and P_a because MR-based attacks need to access many MRs but by default Crail only registers a small set of MRs.

PTE-based attack ($PythiaCrail_{PTE}$). The second attack uses PTE-based eviction. In this attack, we require three processes as in the MR-based attack: P_c, P_s, and P_a. In the preparation phase, we first use P_s to allocate a big chunk of memory and register it with an MR. In the eviction phase, P_a performs one-sided RDMA reads to different VPNs in the allocated memory space. Afterwards, P_c issues a Crail *get* request to the victim key-value pair.

Our reverse engineering results in Section 5.2.2 can be leveraged to reduce the eviction set size in $PythiaCrail_{PTE}$. However, to form the eviction set, we need to know the index of the SRAM cache set(s), which is calculated by the virtual memory address. Without modifying the source code of Crail which is written in Java, it is difficult to directly know the virtual memory address of a target key-value pair. Instead, we use a "learning" phase before launching the actual attack to determine

the eviction set to use for a target key-value pair. Specifically, we let P_c access the target key-value pair and let P_a try all 1024 different cache sets for eviction. After 1024 trials, we pick the cache set that yields the best accuracy as our attack eviction set.

Client-only attack ($PythiaCrail_{Client}$). Our last attack on Crail is launched exclusively from a regular Crail client process and requires no other privileges or resources. The attacker (as a normal Crail client) issues Crail *get* requests to different key-value pairs during the eviction phase. After the eviction phase, it performs a Crail *get* operation to the victim key-value pair.

Our initial design of $PythiaCrail_{Client}$ randomly picks key-value pairs to access during the eviction phase. However, we soon discovered two issues with this naive approach. First, it needs a large number of key-value pairs to effectively evict the target key-value pair. Doing so not only makes the attack slow but also requires the Crail system under attack to already be storing many key-value pairs. Second, we found that the Crail system becomes slower and unstable as the server processes more client requests. We suspect this to be caused by Crail's own (memory) management overhead. Unstable access latency makes our timing-based attack harder and prohibits an accurate prediction during the reload phase. We improve our initial design with the following optimization. We selectively choose a small set of key-value pairs as the eviction set. We make the assumption that key-value pairs are sequentially allocated in chunks of memory and pick the pairs that are likely to be in the same RNIC SRAM cache set as the victim key-value pair. After reducing the eviction set size, our attack runs very fast. Instead of continuously launching the attack in loops, we add some sleep time between eviction and reloading so that we do not issue too many Crail requests to make Crail unstable.

Probabilistic prediction. Under real workloads and noisy network environments, we found that a simple threshold as used in Section 5.2.2 cannot accurately determine if the victim has accessed the target data. Thus, we use a more dynamic and adaptive approach to predict the outcome of the attack. Similar to the approach used in

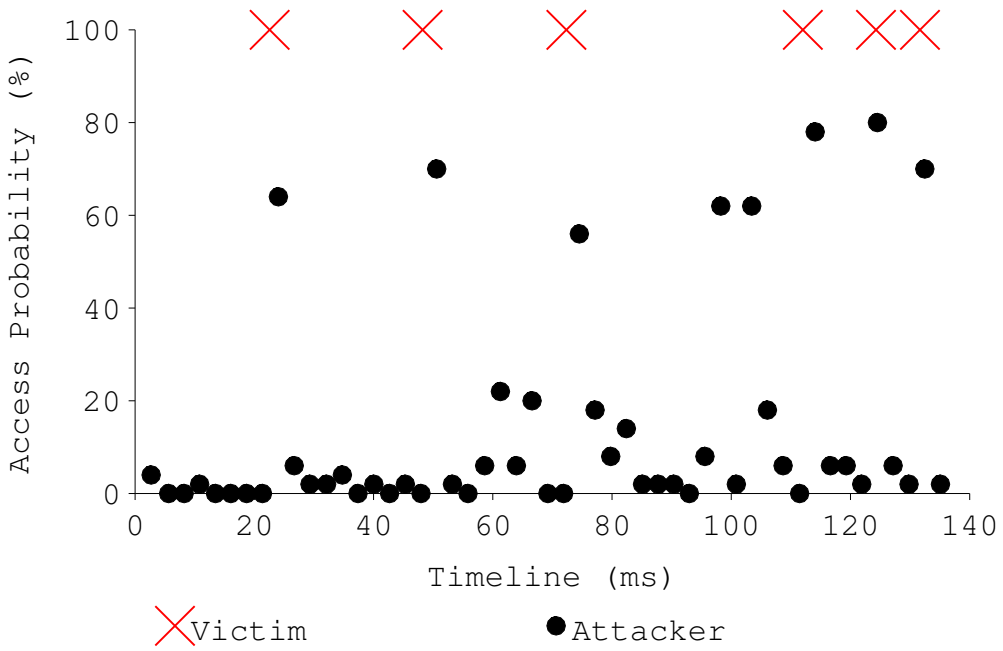


Figure 5.16.: $PythiaCrail_{MR}$

TLBleed [133], we perform a learning phase to train a classifier of operation latency with KNN [225] before the attack. We use the trained model to predict the probability of a reload latency implying a victim access (*i.e.*, a hit).

Results

We evaluated $PythiaCrail_{MR}$, $PythiaCrail_{PTE}$, and $PythiaCrail_{Client}$ using both controlled tests and workloads that model real datacenter key-value stores.

Controlled Test

We first compare the latency of a Crail client key-value pair *get* operation that hits RNIC SRAM, a client *get* that misses RNIC SRAM after the eviction phase in $PythiaCrail_{MR}$, in $PythiaCrail_{PTE}$, and in $PythiaCrail_{Client}$. In these controlled tests, the victim client has a 50% chance of accessing the targeted key-value pair that the attacker tries to infer accesses on. Figure 5.19 plots these four types of latencies,

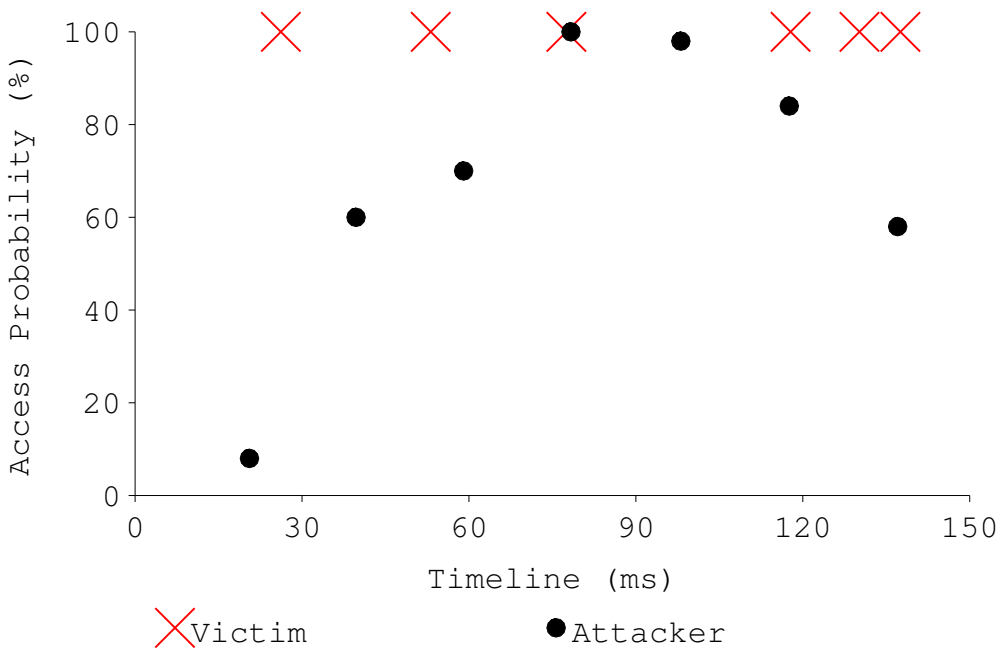


Figure 5.17.: $PythiaCrail_{PTE}$

each performing 1000 trials. All the three types of misses take longer than hits, with the timing difference of $PythiaCrail_{MR}$ the biggest and $PythiaCrail_{Client}$ the smallest. The timing difference implies that it is easiest to separate hits and misses with $PythiaCrail_{MR}$.

We launch the $PythiaCrail_{MR}$, and $PythiaCrail_{PTE}$, and $PythiaCrail_{Client}$ attacks by first performing their respective eviction phases. Next, we let victim access or not access the target key-value pair. Finally, we measure the time to reload the key-value pair and compare it with a threshold we determined from the timing difference testing phase. As expected, $PythiaCrail_{MR}$ gives the best accuracy. The accuracies of $PythiaCrail_{MR}$, $PythiaCrail_{PTE}$, and $PythiaCrail_{Client}$ are 96%, 85%, and 79% respectively, and the time to perform these attacks are $19\,ms$, $0.1\,ms$, and $0.3\,ms$.

Macro-benchmark Results

Workloads. To evaluate how our attacks perform with real datacenter key-value store workloads, we construct a macro-benchmark with the Yahoo! Cloud Serving

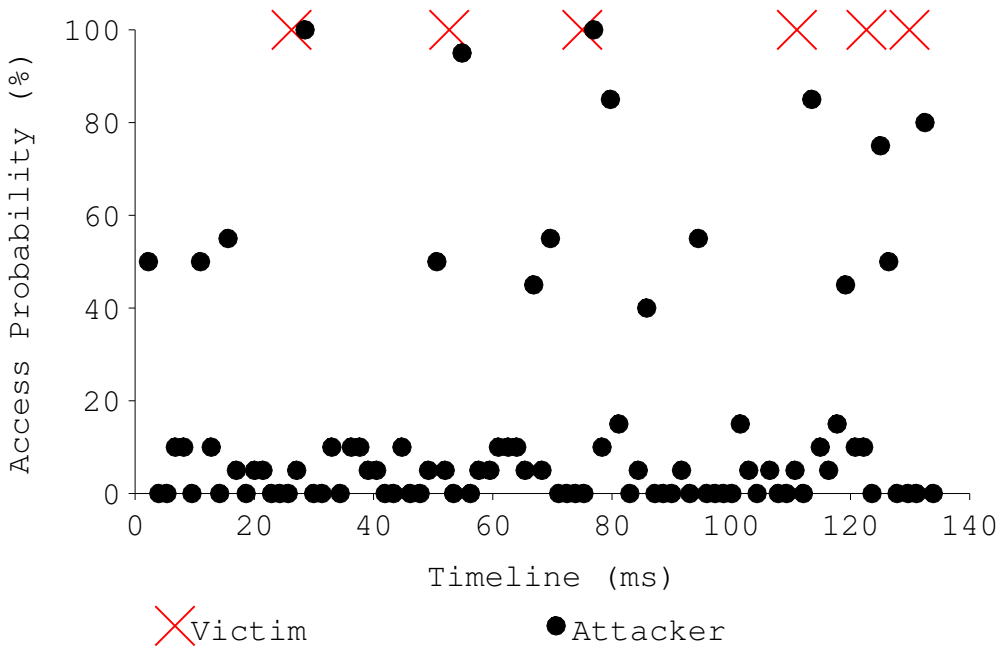


Figure 5.18.: $PythiaCrail_{Client}$

Benchmark (YCSB) [184] and statistics reported by Facebook in their production key-value store [170]. YCSB provides key-value *get/set* access pattern but no inter-arrival time between requests. Facebook provides the inter-arrival time of requests received at a server in its cluster, which includes requests from all the clients to this server. We set each key-value pair size to be 1 KB, the average key-value pair size reported by Facebook.

Attack environment setup. In this experiment, the victim (on a Crail client machine) executes our macro-benchmark to access key-value pairs on a Crail server machine using the Crail APIs. Since Facebook only provides aggregated request inter-arrival time across clients and does not reveal how many clients there are, we use one client machine to model the aggregated effect of all clients with the provided inter-arrival time. We run an attacker process as a normal Crail client on another machine. A fourth machine serves as the Crail namenode. While the victim process executes

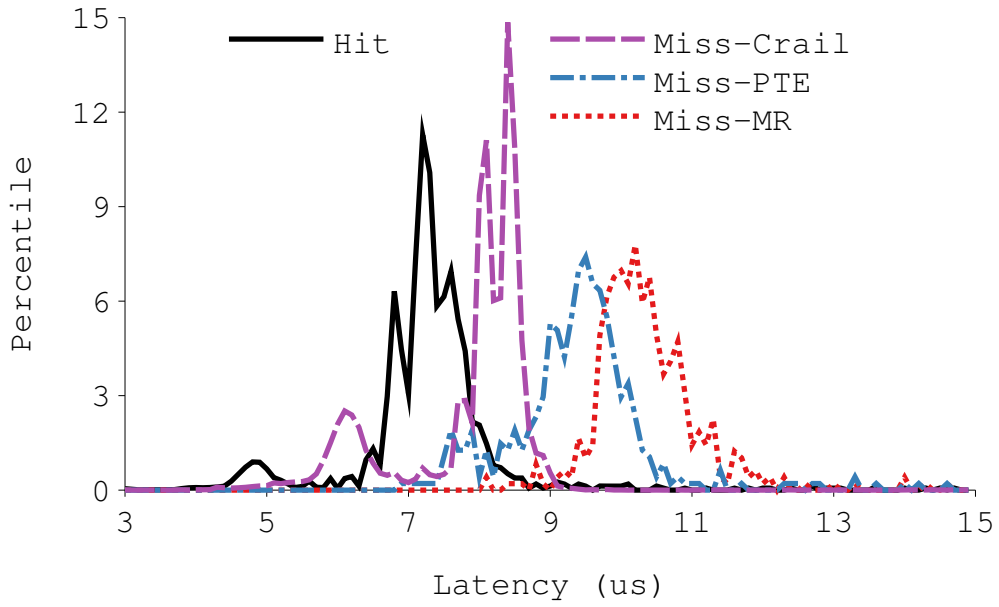


Figure 5.19.: **Timing Difference in Crail.** *Each line presents the timing differences of each case over 1000 trials.*

the macro-benchmark, we repeatedly perform $PythiaCrail_{Client}$, $PythiaCrail_{MR}$, or $PythiaCrail_{PTE}$ to detect if the victim accesses a target key-value pair.

Results. Figures 5.16, 5.17, and 5.18 present the timeline of the victim accessing the target key-value pair (red crosses) and the attacker's prediction (black dots with values as access probability). All three attacks can capture most if not all victim accesses. Among them, $PythiaCrail_{MR}$ is the worst in attack accuracy. This is because each attack in $PythiaCrail_{MR}$ takes $19\,ms$, which is much longer than the Facebook inter-arrival time. As a result, $PythiaCrail_{MR}$ misses victim accesses that happen more frequent than its attack length. Both $PythiaCrail_{PTE}$ and $PythiaCrail_{Client}$ run very fast and capture all victim accesses. In fact, these two attacks run so fast that we add a sleep time of $1\,ms$ between evict and reload to avoid issuing too many Crail requests and making Crail's performance unstable. Comparing $PythiaCrail_{PTE}$ and $PythiaCrail_{Client}$, $PythiaCrail_{PTE}$'s predictions are of low access probabilities and $PythiaCrail_{Client}$ has more predictions of around 50% access probabilities. The

attacker can set a threshold accordingly to determine the final set of victim accesses (*e.g.*, those with probabilities > 60% for $PythiaCrail_{Client}$).

Overall, we believe $PythiaCrail_{Client}$ to be the most effective attack, since it predicts victim accesses with high confidence and it requires the least amount of attacker resources: $PythiaCrail_{Client}$ can be launched exclusively from a separate client machine through the unmodified Crail client interface. If attackers can run modified Crail clients, they can launch more efficient side-channel attacks by forming the eviction set with known virtual addresses.

5.2.4 Mitigation Techniques

Defending against RDMA-based side-channel attacks is possible and feasible. We discuss both mitigations for current hardware as well as those for future hardware.

Huge virtual memory page or no virtual memory. PTE-based attacks are only possible when RNICs cache PTEs and when the attacker can form an effective eviction set. One way to prevent PTE-based attacks is to force all RDMA registrations and operations to directly use physical memory addresses. When physical memory addresses are used, RDMA does not need to access or cache PTEs, thereby preventing PTE-based attacks. Registering physical memory addresses is a privileged operation that RNICs allow the kernel [214] and privileged users to perform [217]. However, using physical memory addresses loses all the benefits of virtual memory and introduces new security concerns.

Another method to defend against PTE-based attacks is to use huge memory pages [29]. Using huge pages (*e.g.*, 1 GB pages) introduces two types of difficulties for attackers. First, the attacker can only guess victim accesses at coarse granularity (*e.g.*, 1 GB). Second, the attacker will need to have access to a huge memory space to form an eviction set with enough PTEs.

Isolate server's resource. Our experience with Crail demonstrates that attacking Crail is difficult when the attacker can only use Crail's interface without the access

to a large number of PTEs or MRs and without knowing Crail's data layout in the virtual memory address space. Our experiments show that for $PythiaCrail_{MR}$ and $PythiaCrail_{PTE}$ to work, an attacker needs to run a process, P_s, on the server machine. Otherwise, the attacker would not be able to launch those attacks (although $PythiaCrail_{Client}$ still works). Thus, a server that hosts RDMA service can prohibit normal users from running any processes to help defend against side-channel attacks. Various address randomization techniques can also complicate attacks.

Separate protection domains. When we disclosed the attacks in this chapter to Mellanox, the Mellanox engineers stated that separating *Protection Domain*s (*PD*s) between different clients and connections can potentially mitigate the attacks. We evaluated this mitigation by moving the attacker to a different PD and found that doing so mitigates Pythia attacks. Unfortunately, all existing RDMA applications that we are aware of [29, 35, 68] use only one PD for higher performance. Using multiple PDs results in low throughput and high latency overhead (15% throughput reduction and 21% latency overhead with 256 PDs in our experiments). We plan to further investigate both attack and defense mechanisms when separating PDs across clients.

Introduce noise. Our side channels are established on timing differences at the microsecond or sub-microsecond level. Attacking Crail running real workloads is more difficult than attacking raw RDMA accesses mainly because of Crail's non-deterministic performance overhead. Therefore, an effective countermeasure is to introduce random latency overhead at an RDMA-based application or in the data-center RDMA network, which, however, could impact application performance.

Detect and throttle attacker's network traffic. Our attackers can hide their attacks because one-sided RDMA operations are completely hidden from the receiver CPU (the server in our case). To detect these attacks, the server can deploy traffic sniffing tools to sniff all incoming RDMA network requests. If the sniffer detects heavy network activity from a client, it can raise a flag that this client may be

malicious. If it further detects an access pattern that matches eviction sets described in Section 5.2.2, this client is more likely to be an attacker. This defense comes with the same drawbacks of other heuristic-based defenses that an attacker may stay under the detection threshold.

A further countermeasure is to throttle the maximum bandwidth allowed at every client. If an attacker cannot issue enough operations to evict RNIC SRAM, its attack accuracy will drop significantly. However, throttling client bandwidth can hurt normal clients' performance.

Better hardware design. All existing RNICs share their SRAM across all users and across all connections. Because of this, an attacker can evict a victim's PTE and MR even when the attacker and the victim have different connections to the server. If RNICs can partition their SRAM to different isolated domains for different connections, then attackers can never evict victim's PTEs or MRs. However, isolation resources at hardware level will inevitably hurt performance and increase hardware complexity, which gives little incentive for RNIC vendors to change their hardware design.

5.2.5 Discussion

We now discuss the implications, impact, and limitations of Pythia, and some other attacks on RDMA that can be designed based on Pythia. In addition to discussing security vulnerabilities, this section also introduces unique opportunities to enhance security and privacy based on one-sided communication.

Attacking Real Applications

We demonstrated that it is feasible to launch Pythia attacks on Crail, a real RDMA-based system developed by the Crail team. $PythiaCrail_{Client}$, the attack

that is launched by performing Crail-provided client APIs only, can successfully infer victim's access patterns under real workloads.

We believe that Pythia can similarly attack other RDMA-based applications as well. Pythia only requires two features from an RDMA-based application: the application uses one-sided RDMA operations and allocates regular paged memory. Many applications meet these requirements, such as the NAM-DB RDMA-based in-memory database [26], the Pilaf RDMA-based key-value store [31], and the Wukong RDMA-based graph system [28]. Unfortunately, most of these systems are not available publicly.

Attack Limitations

Although our side-channel attacks are fast, accurate, and can be launched entirely from the network, they do have several limitations. First, the granularity of Pythia attacks (and therefore information leakage of accesses) is a memory page. Pythia currently cannot differentiate between two victim accesses that access the same target page. Second, Pythia can only predict if a data entity at the server has been accessed, but not which client machine(s) accessed it. Third, our MR-based attacks require access to a large number of MRs, and PTE-based attacks require access to large memory spaces. Finally, our attacks consume network bandwidth and can be detected by sniffing the network.

Other RDMA-Based Attacks

Pythia serves as a starting point for designing other types of RDMA-based attacks.

Prime+Probe. For example, similar to Pythia Evict+Reload attacks, it is possible to launch Prime+Probe attacks from the network by exploiting the MR or the PTE side channels.

Covert channel attack. The MR and the PTE side channels we established can also be used as covert channels. We implemented a naive covert-channel attack of

using one EVICT+RELOAD cycle to transmit one bit and it could reach a sending rate of 20 Kbps with $PythiaPTE_{Full}$.

5.3 Opportunity of One-Sided Communication

We have introduced the vulnerabilities in one-sided communication and shown side-channel threats with Pythia in this chapter. Pythia serves as a starting point for designing other types of attacks in one-sided communication. Although one-sided communication poses different threats to datacenter security, it provides one great opportunity to *enhance* security. Users can ensure their privacy by hiding their network activities from receiving servers using one-sided communication. This session discusses how we can leverage this opportunity to develop secure and fast data storage systems.

Environment and Threat Model. *ORAM*, or *oblivious* RAM, is a type of technology that makes access patterns "oblivious" by continuously re-encrypting and reshuffling data blocks at storage servers [231–239].

We adopt the same trust model as previous ORAM systems [240–243], where trusted clients access an untrusted storage service provider (*e.g.*, in-memory key-value store, NVMe-based storage). We also adopt the same level of "security" and "privacy" as existing ORAM solutions [238, 240–243], where the untrusted service provider should gain no information about client data or access patterns. Thus, hiding the content of data through encryption alone is not enough. In addition, no information should be leaked about: 1) which data is being accessed; 2) whether the access is a read or a write; 3) when the data was last accessed; 4) whether the same data is being accessed; or 5) the access pattern (sequential, random, etc).

One-Sided Oblivious RAM. Although proven to be cryptographically safe, existing ORAM-based systems have high performance overhead, making it too costly to be adopted in many datacenter environments. We now present our improved ORAM

system design. Our system is based on Path ORAM [241] but can significantly outperform Path ORAM.

We propose to leverage one-sided communication to hide data access information and to replace (costly) ORAM operations with one-sided operations, thereby improving the performance of ORAM. Although the basic idea is simple, one needs to take extra care when applying it to provide the same level of security guarantees as ORAM. For example, a receiver (the malicious service provider) cannot detect when a one-sided write happens. But it can take snapshots of data content periodically and compare two snapshots to detect modifications in between. It can then infer that a write happens by performing frequent snapshotting. Thus, one-sided writes can still leak information to the service provider. Because of this, we use the unmodified ORAM write protocol and only improve ORAM read performance.

Reading data using one-sided communication can be completely invisible to the service provider, even if the provider performs snapshotting. We use a simple technique to achieve this security goal. By default, we perform one-sided reads for client read requests but randomly choose a certain amount of client read requests (*e.g.*, $X\%$ of all read requests) to perform original ORAM operations. Meanwhile, performing a one-sided read is significantly faster than performing an ORAM operation, since the latter requires read and write of a whole path, while the former only performs a read to one object. Although the algorithmic complexity of the modified Path ORAM is still identical to the original Path ORAM, the performance of our improved ORAM read is significantly better than the original Path ORAM.

We implemented the original Path ORAM protocol and our modified read mechanism using RDMA and tested their performance with two network settings, a 40 Gbps InfiniBand network and a 100 Gbps InfiniBand network. Figure 5.20 presents our modified read performance when changing $X\%$ from 10% to 100% (under 100%, our system falls back to original Path ORAM). Here, we use a pure-read workload in the YCSB key-value store benchmark [184, 185], with 32,000 512-byte key-value pairs.

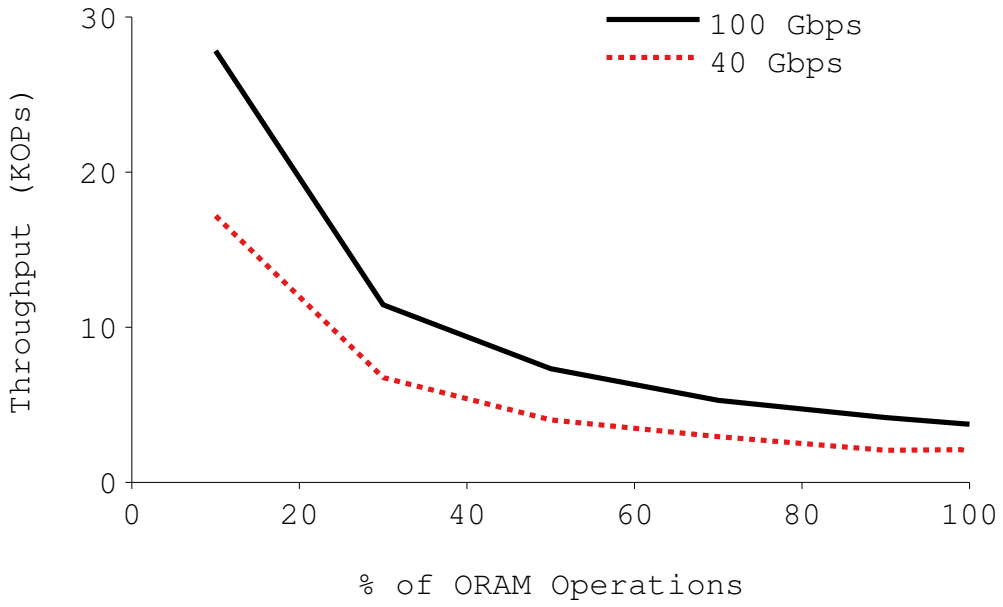


Figure 5.20.: **Read Performance of One-Sided ORAM** *X axis shows the amount of read operations that we use original ORAM operations to perform. The two lines show the performance when using a 100 Gbps and a 40 Gbps InfiniBand network.*

We encrypt all data with AES256. With 50% one-sided reads, we achieve ∼2× performance of pure Path ORAM.

Limitations and Discussion. One limitation of our one-sided ORAM solution is the requirement of prohibiting network packet sniffing. As described in Section 5.1, most systems enable this prohibition by default. However, if an attacker can bypass such prohibition mechanisms (*e.g.*, when it controls a physical machine), it can observe one-sided traffic. Thus, our threat model applies only to other cases, for example, when the attacker only owns a virtual machine.

One-sided NICs usually writes to memory through DMA. There are methods for a server to track (all) DMA activities (*e.g.*, with processor counter monitor [229]). However, it is still hard to pinpoint one-sided traffic, since other types of DMA activities may be happening at the same time. Even if a server can detect one-sided network traffic, it is still challenging to tell whom the sender is and which data is being accessed.

5.4 Conclusion

This chapter provides the first and initial look into the security aspect of one-sided communication and presents Pythia, the first set of side-channel attacks on RDMA-based systems. We demonstrate several vulnerabilities that are rooted in one-sided communication's basic design philosophies. We reverse engineer the internal data structures of current RDMA systems and leverage this information to improve our attack. Pythia can be launched completely from a normal client machine to steal access patterns of victims on other machines. We evaluate Pythia in laboratory settings to showcase the capabilities of the attack, on real software such as Crail, and on real data centers to show real-world impact. Pythia is fast, accurate, and can hide its trace from victims and the server.

Our findings can be a starting point to explore potential attacks and defenses for future researchers and practitioners. This chapter is a warning for future one-sided hardware vendors, software developers who want to use one-sided communication, and datacenters that have or plan to deploy one-sided network systems. We believe that these three parties should work together to achieve the security goals in datacenters while preserving one-sided communication's performance and cost benefits. One promising direction is to leverage programmable network devices like SmartNIC [56,194] and programmable switches [244–246] to implement security defenses in hardware.

Although adding security guarantees to existing one-sided communication technologies is not an easy job, we do not believe the future of one-sided communication to be diminishing. Moreover, one-sided communication provides a great opportunity to improve privacy, the security property that is increasingly important in today's cloud environments.

6 RELATED WORK

This chapter introduces various research efforts that are related to this dissertation. We first describe user-level TCP/IP implementations and user-level RDMA-based libraries. We then introduce literature on RDMA-based key-value stores, one- and two-sided primitives in RDMA-based systems, and several existing distributed PM systems. Finally, we close this chapter by introducing research efforts in side-channel attacks and the mitigations to side-channel attacks.

6.1 User-Level TCP/IP Implementation

There have been various efforts in user-level TCP/IP implementations such as mTCP [247], U-Net [248], IX [158], and Arrakis [157]. Moving TCP/IP stack from kernel to user space can reduce the performance cost of kernel crossings. However, resource sharing across user-level processes is inefficient and not safe. In fact, U-Net and IX rely on kernel to perform resource isolation. Hardware-enforced isolation mechanisms such as SR-IOV is one way to let user-level processes safely access hardware resources and has been used by systems like Arrakis to build low-latency user-level TCP/IP network stacks [157]. However, unlike TCP/IP, hardware isolation mechanisms limit the flexibility of RDMA, since these mechanisms require pre-allocating and pinning all memory of each application (or VM) for DMA [249]. A software layer in the kernel like LITE can allocate and map application memory on demand, *i.e.*, only when accessed. Software can also implement more flexible resource sharing and isolation policies. RDMA has different properties and performance implications from TCP/IP. Moreover, LITE is designed for RDMA and solves RDMA's issues in the datacenter environments.

6.2 RDMA-Based Libraries

There are several RDMA-based user-level libraries including the standard OFED library [250], rdma-cm [251], MVAPICH2 [58, 59], Rsockets [147], Portals [252], and eRPC [204]. MVAPICH2 supports the MPI interface and is designed for the HPC environments. Rsockets implements a socket-like abstraction. Portals [252] exposes an abstraction that is based on put and get operations. eRPC [204] is a general-purpose RPC library that is optimized for small messages, short duration RPC tasks, and congestion-free networks. LITE's abstraction is designed for datacenter applications and is richer and more flexible than the above libraries. Moreover, LITE uses a kernel-level indirection to solve native RDMA's issues in datacenters, which none of these existing libraries solve. There are also several kernel-level layers on top of InfiniBand, such as IPoIB [165], SDP [253], and SRP [254], to support traditional network and storage interfaces. They all have heavy performance overhead and do not offer the low-latency performance as LITE does.

6.3 RDMA-Based Key-Value Stores

Several new RDMA-based systems were built in the past few years for datacenter environments. FaRM [29] is an RDMA-based distributed memory platform which inspired LITE. Its core communication stack uses RDMA write. On top of the basic communication stack, FaRM builds a distributed shared memory layer, a transaction system [30], and a distributed hash table. LITE-DSM is also a distributed shared memory layer, but LITE is more generalized and flexible: it supports other types of remote memory usages and a write-imm-based RPC mechanism; it safely shares resources across applications; it does not require big pages or any driver changes.

Pilaf [31] and HERD [33] are two key-value store systems that implement customized RDMA stacks using RDMA read, write, or send. HERD-RPC [32] and FaSST [66] are two RDMA-based RPC implementations. FaSST's main goal is to scale with number of nodes. The Derecho project [61, 175, 176] is a set of efforts

to build a distributed computing environment. It uses a new RDMA-based multicast mechanism and a shared-state table to implement several consensus and membership protocols. There are also several RDMA-based database and transactional systems [26, 27, 30, 67, 116], distributed RDF store [28], DSM system [25], consensus system [71], and distributed NVM systems [36,37,73]. These systems all target a specific type of application and most of them build customized software to use RDMA. LITE is a generic, shared indirection layer that supports various datacenter applications. Building LITE has the unique challenge that its design decisions cannot be tailored to just one application.

6.4 One-Sided and Two-Sided Primitives in RDMA-Based Systems

Several in-memory key-value stores and transaction systems use a combination of one- and two-sided RDMA for their data path. FaRM use one-sided RDMA for reads and perform both two- and one-sided RDMA for replicated writes (depending on whether it is to the primary copy). Pilaf [31] and HERD [33] use two-sided RDMA for writes. HERD [33] and FaSST [66] use two-sided RDMA for reads too. Wei *et al.* compare one- and two-sided RDMA primitives and implemented a transaction system that uses one- and two-sided RDMA in different transaction phases [68]. Their group's previous work on RDMA-based transaction systems [67, 116, 255] also use a combination of one- and two-sided RDMA. KV-Direct [64] is an FPGA-based in-memory key-value store that uses the FPGA chip in a SmartNIC to manage metadata and data operations.

Another set of remote memory systems uses one-sided communication for the data plane but requires processing power for the control plane. NAM-DB [26, 27] is an RDMA-based in-memory database system that uses one-sided communication for both read and write. INFINISWAP [63] is an RDMA-based remote memory paging system. Remote regions [74] is a system that exposes remote memory as files that other host servers can access (through a file system interface). Although these systems

do not use two-sided communication for the data path, they all rely on processing power at remote nodes to run data management tasks. RAMCloud [92] is a remote key-value storage system that stores a full copy of all data in DRAM and backups in disks or SSDs. These systems all rely on local computation power at remote memory/storage servers to perform various online and recovery management services which differs from DPM model.

6.5 Distributed PM Systems

A set of recent research projects focus on building distributed PM systems, including Mojim [36], Hotpot [37], Octopus [73], and Orion [115]. Mojim [36] provides an efficient, RDMA-based, asynchronous replication mechanism for PM to make it more reliable and available. Hotpot [37] is a distributed shared persistent memory system, which integrates the idea of distributed shared memory and distributed storage systems to provide a global sharable PM space. Octopus [73] and Orion [115] are two PM-based distributed file systems that provide high performance, byte addressability, and reliability. These distributed PM systems all use the architecture in Figure 4.1(a). Building distributed PM systems with the DPM model presents a whole new set of challenges.

The Machine from HPE [22, 187, 189–192] is a system that relates to DPM most. The Machine organizes a rack by connecting a pool of SoCs to a pool of PMs through a specialized cache-coherent network layer [193]. This architecture is similar to DPM-Direct, but it relies on the special network to ensure coherence among concurrent data accesses. When moving to a general network, one needs to build more complex protocols to ensure coherence, which DirectDS explores. In addition, we also explore the DPM-Central and DPM-Sep architectures of building DPM, and our systems work with both normal machines and disaggregated systems.

6.6 Remote Side-Channel Attacks

This section introduces the research efforts in remote side-channel attacks and discusses how Pythia is different from all of them.

Several remote side-channel attacks exploit TCP sequence numbers to hijack connections [256–258]. Another line of work relies on traffic analysis to exploit sensitive information [259, 260]. Brumley *et al.* perform a timing-based attack on OpenSSL's ladder implementation to obtain the private key of a TLS server [261]. Cock *et al.* present an empirical study of remote timing channels on microkernel [262]. NetSpectre [263] presents an access-driven remote Evict+Reload cache attack. Zachary *et al.* combine CSS and JavaScript to remotely sniff victims' browsing patterns [264]. Different from all remote side-channel attacks, Pythia targets the RDMA network. Throwhammer [221] is another work using RDMA network, but it targets Rowhammer attack instead of exploiting side channels. Moreover, Pythia exploits RNIC hardware features to establish timing-based side channels, while previous remote side channels exploit network protocols or software features.

Various defense mechanisms have been proposed to combat CPU cache side-channel attacks in both hardware [265–269] and software [140, 270–273]. Unfortunately, none of the existing defense mechanisms can be directly applied to RDMA-based side-channel attacks. This dissertation introduces a set of new mitigations that target RDMA-based side-channel attacks.

7 CONCLUSION

RDMA is a technology designed for confined, single-tenant, small-scaled HPC environments. Because of its low-latency, high-throughput, and low-CPU-utilization benefits, datacenters have started to adopt RDMA in recent years. Unlike HPC environments, datacenters have a whole different set of requirements and features: applications in datacenters evolve fast and desire easy-to-use abstractions; the scale of a datacenter is much larger than an HPC cluster; datacenters are often multi-tenant where reliability, availability, and security are important considerations beyond performance.

Although RDMA works well in HPC environments, it falls short in datacenter environments. RDMA lacks a flexible, easy-to-use abstraction, it has various scalability limitations, it does not have good support for secure resource sharing, and there are security vulnerabilities in RDMA communication and RDMA hardware. Because of these issues, we believe that RDMA is not readily usable in datacenters.

We made initial efforts in improving RDMA's usability, scalability, safety, and security in this dissertation, making it a more suitable technology for datacenter environments. We presented LITE, a novel indirection layer to virtualize and manage RDMA for datacenter applications. LITE demonstrates that adding an indirection layer on RDMA not only improves its manageability, usability, and scalability, but also preserves most of its superior performance. We then presented DPM, a low-cost, fast, scalable, and reliable RDMA-based data store designed for datacenter read-most workloads. DPM uses persistent memory and removes processors at the memory to reduce owning and running cost. Finally, we discussed our efforts in exploring security implications of RDMA. We reverse-engineered state-of-the-art RDMA NICs, designed a set of side-channel attacks, and proposed mitigation techniques of these attacks.

This dissertation studies several design decisions in native RDMA, proposes a generic indirection layer for RDMA to make it more suitable for datacenter applica-

tions, presents a set of datacenter applications and systems that we ported or built on (improved) RDMA, and studies an important and unexplored factor, security, of RDMA. All together, this dissertation significantly advances state-of-the-art RDMA technologies and outlines a set of promising directions to incorporate RDMA in datacenters.

This chapter discusses the set of lessons we learned and future directions where our work can possibly be extended.

7.1 Lessons Learned and Discussion

Although RDMA has been adopted in the HPC environments for almost two decades, it is still a young and immature technology in datacenter environments. When we started working on this dissertation, we were faced by many challenges in moving RDMA to datacenter environments. Looking back, we have learned a few lessons while working on this dissertation.

7.1.1 Moving from HPC to Datacenters

A technology that works well in one environment may not work for another.

In the early days of exploring solutions to integrate RDMA in datacenter environments (both by us and by others), we found that many of the major challenges rooted from the fact that RDMA was designed for the HPC environment, which is very different from datacenters. HPC environments are usually single-tenant and single-purpose, and they usually adopt specialized hardware and fine-tuned, customized software stack. In contrast, datacenters are multi-tenant, general-purpose environments that usually adopt commodity hardware and needs to support many types of software applications. Moreover, the scale of a datacenter is much larger than a supercomputer. Moving RDMA to datacenter environments thus meet unforeseen challenges in usability, scalability, and security.

To solve these challenges, this dissertation explores solutions that preserve original RDMA architecture and add indirection layers on top. We have demonstrated the effectiveness and benefits of this approach. Yet, other approaches may also work, for example, by changing the RDMA protocol, software, and hardware implementation.

In general, it is clear that when moving a technology from one environment to another, one should not expect it to work without any changes. A bigger and more challenging question is how and where to change the technology. This dissertation explores one direction in the design space, and we encourage future researchers and practitioners to explore more design choices of integrating RDMA in datacenters.

7.1.2 Hardware Offloading

Scalability and security are key challenges in hardware offloading.

The main approach RDMA adopts to achieve many of its benefits such as CPU and kernel bypassing is offloading network and management stacks to the hardware (*i.e.*, RNICs). This dissertation reveals that although hardware offloading delivers excellent performance, it raises flexibility, scalability, and security issues. We demonstrate that by carefully choosing appropriate functionalities and metadata to implement in software, we can solve most of the flexibility, scalability, and security issues with minimal performance impact.

Recent trends in several other fields also move towards offloading more software functionalities to hardware, ranging from offloaded network stacks [274, 275] and smartNIC [64, 143, 276] to smart SSDs [277–279]. As illustrated in this dissertation, hardware is fundamentally less flexible and harder to program than software; the amount of resources in specialized hardware is also a lot less abundant than general-purpose computers. When evaluating what functionalities and metadata to offload to hardware, one should not only consider performance but also scalability, extensibility, and security.

7.1.3 One-Sided vs. Two-Sided Communication

Abstraction drives usage cases; implementation largely decides performance and cost.

RDMA supports both one-sided and two-sided communication. There has been a huge debate on the choice of one-sided and two-sided RDMA in recent years. We have explored various tradeoffs between these two RDMA communication methods in this dissertation, ranging from performance and scalability to monetary cost and security. We share our final thoughts on them below.

Abstraction and usages. From programmers' (applications') view, there is a fundamental difference between these two communication patterns. With two-sided communication, both sides are active parties that can perform actions; with one-sided communication, one side treats the other side as a passive, non-acting party. Thus, we believe that the best use case and the best match for two-sided communication is an environment where both sides of the communication are *compute* and the best match for one-sided communication is an environment where one side is *compute* and the other side is *data*. In a sense, two-sided communication follows traditional *messaging* usages, and one-sided communication is more like data/memory accesses (with a major difference being RDMA not providing coherence across different copies of data, unlike CPU memory bus).

Although all traditional network and messaging usages are applicable to two-sided communication, it is less clear what are the best way to use one-sided communication. Here, we give a set of recommendations. Compute servers can use one-sided communication to store and access data at remote memory or storage. These data can be local memory that is swapped out to a remote server [63, 74], disaggregated compute nodes' memory stored in disaggregated memory nodes [38, 183], or key-value or relational data in disaggregated nodes or regular server [26, 27]. What makes it challenging to use one-sided communication for some of these environments are 1) the possible need for processing data at remote side, and 2) the possible need for ensuring coherence when there are multiple writers to remote memory. We identify two promising solutions for these challenges. For 1), the processing can be moved

to the sender (compute) side, be changed into functionalities that can be performed asynchronously by remote side's processor after data has been transmitted, or be implemented in hardware [64]. For 2), Chapter 4 demonstrated several ways to support concurrent writers.

Implementation. Under the hood, there can be different ways to implement one-sided and two-sided communication. For example, although both RDMA and traditional TCP supports two-sided communication, their implementation differs dramatically. RDMA NICs perform most of the network functionalities and directly read/write to pre-registered space in main memory and then notify host CPU, which then performs the rest of the two-sided communication functionalities. With traditional TCP, host CPU performs most of the network stack functionalities.

On the other hand, the difference between one-sided and two-sided communication can simply be viewed as the location where network stacks are implemented. For one-sided communication, the network stacks are entirely implemented in hardware (ASIC-, FPGA-, SoC-based NICs), while two-sided communication stacks are implemented entirely or in part in host CPU. The difference then becomes specialized vs. general-purpose processing units.

What we have learned in this dissertation is that different implementation can result in different performance, scalability, and security tradeoffs. As past wisdom goes [280], systems designers should keep abstraction separate from implementation and consider functionality, performance, and reliability (security) when building a new system or porting a mature technology to a new environment.

7.2 Concluding Remarks

When migrating a technology that is mature for one environment to another, there can be many unforeseeable challenges. RDMA is one such technology. This dissertation explores many different challenges of integrating RDMA into datacenters and presents our solutions to them. We hope that our experiences and insights would

not only help future datacenter RDMA developments but also shed light on the migration of other technologies to datacenters.

REFERENCES

[1] Tejas Karmarkar. Availability of linux rdma on microsoft azure. https://azure.microsoft.com/en-us/blog/azure-linux-rdma-hpc-available, 2015.

[2] Alibaba Cloud. Super computing cluster. https://www.alibabacloud.com/product/scc, 2018.

[3] 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, CA, USA, April 2012.

[4] Apache Hadoop, 2011. http://hadoop.apache.org/.

[5] Google Brain Team. TensorFlow, 2015. https://www.tensorflow.org/.

[6] 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)*, Vancouver, BC, Canada, October 2012.

[7] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. A distributed graph engine for web scale rdf data. In *Proceedings of the 39th international conference on Very Large Data Bases (VLDB '13)*, Trento, Italy, February 2013.

[8] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*, New York, NY, USA, June 2013.

[9] Amazon. Aws lambda, 2014. https://aws.amazon.com/lambda/.

[10] Google. Google cloud functions, 2018. https://cloud.google.com/functions/.

[11] 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)*, Savannah, GA, October 2016.

[12] Sangjin Han, Norbert Egi, Aurojit Panda, Sylvia Ratnasamy, Guangyu Shi, and Scott Shenker. Network support for resource disaggregation in next-generation datacenters. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks (HotNets '13)*, College Park, ML, USA, November 2013.

[13] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. Putting the "micro" back in microservice. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '18)*, Boston, MA, USA, July 2018.

[14] Akshitha Sriraman and Thomas F. Wenisch. utune: Auto-tuned threading for oldi microservices. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '18)*, Carlsbad, CA, USA, October 2018.

[15] 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.

[16] Kestutis Patiejunas and Amisha Jaiswal. Facebooks disaggregated storage and compute for map/reduce. In *Data @Scale (Scale '16)*, Seattle, WA, USA, June 2016.

[17] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference (SIGCOMM '10)*, New Delhi, India, August 2010.

[18] 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.

[19] Mellanox. Mellanox ConnectX-6 VPI Card, 2019. `http://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX-6_VPI_Card.pdf`.

[20] Chris Mellor. Ai servers will need much more memory. and you know who's going to be there? yep, big daddy micron, 2018. `https://www.theregister.co.uk/2018/05/23/micron_analysts_day/`.

[21] Peng Sun, Yonggang Wen, Ta Nguyen Binh Duong, and Xiaokui Xiao. Graphh: High performance big graph analytics in small clusters. In *2017 IEEE International Conference on Cluster Computing (CLUSTER '17)*, Honolulu, HI, USA, September 2017.

[22] Mijung Kim, Jun Li, Haris Volos, Manish Marwah, Alexander Ulanov, Kimberly Keeton, Joseph Tucek, Lucy Cherkasova, Le Xu, and Pradeep Fernando. Sparkle: Optimizing spark for large memory machines and analytics. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*, Santa Clara, CA, USA, September 2017.

[23] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*, Carlsbad, CA, USA, October 2018.

[24] Mingjie Tang, Yongyang Yu, Qutaibah M. Malluhi, Mourad Ouzzani, and Walid G. Aref. Locationspark: A distributed in-memory data management system for big spatial data. *Proceedings of the VLDB Endowment*, 9(13):1565–1568, 2016.

[25] 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)*, Santa Clara, CA, USA, July 2015.

[26] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. The End of a Myth: Distributed Transactions Can Scale. *Proceedings of the VLDB Endowment*, 10(6):685–696, 2017.

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

[28] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. Fast and Concurrent RDF Queries with RDMA-Based Distributed Graph Exploration. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, Savanah, GA, USA, November 2016.

[29] 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.

[30] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*, Monterey, CA, USA, October 2015.

[31] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference (ATC '13)*, San Jose, CA, USA, June 2013.

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

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

[34] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Nikolas Ioannou, and Ioannis Koltsidas. Crail: A high-performance i/o architecture for distributed data processing. *IEEE Bulletin of the Technical Committee on Data Engineering*, 40:40–52, March 2017. Special Issue on Distributed Data Management with RDMA.

[35] Apache. Crail: High-performance distributed data store. https://crail.incubator.apache.org/, 2018.

[36] 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.

[37] 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.

[38] 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, USA, October 2018.

[39] InfiniBand Trade Association. InfiniBand Architecture Specification, 2015. https://cw.infinibandta.org/document/dl/7859.

[40] InfiniBand Trade Association. InfiniBand Architecture Volume 1 Architecture Specification, Release 1.3. https://cw.infinibandta.org/document/dl/7859, March 2015.

[41] RDMA Consortium. iWARP, Protocol of RDMA over IP Networks, 2009. http://www.rdmaconsortium.org/.

[42] InfiniBand Trade Association. RoCEv2 Architecture Specification, 2014. https://cw.infinibandta.org/document/dl/7781.

[43] InfiniBand Trade Association. InfiniBand Architecture Annex A 16: RoCEv2. https://cw.infinibandta.org/document/dl/7148, September 2014.

[44] Recio, R., Metzler, B., Culley, P., Hilland, J., and D. Garcia. A Remote Direct Memory Access Protocol Specification, 2007. https://tools.ietf.org/html/rfc5040.

[45] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. Freeflow: Software-based virtual RDMA networking for containerized clouds. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*, Boston, MA, USA, February 2019.

[46] 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)*, Budapest, Hungary, August 2018.

[47] 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)*, London, United Kingdom, August 2015.

[48] Cisco. Cisco nexus 3000 series nx-os qos configuration guide. https://www.cisco.com/c/en/us/td/docs/switches/datacenter/nexus3000/sw/qos/7x/b_3k_QoS_Config_7x/b_3k_QoS_Config_7x_chapter_011.html, 2019.

[49] Dell. Dell poweredge fn i/o aggregator configuration guide. https://topics-cdn.dell.com/pdf/poweredge-fx2_reference-guide8_en-us.pdf, 2015.

[50] Mellanox. Mellanox sn3000 series. http://www.mellanox.com/related-docs/prod_eth_switches/BR_SN3000_Series.pdf, 2019.

[51] Mellanox. Mellanox mcx345a-bcpn connectx-3 pro. https://store.mellanox.com/products/mellanox-mcx345a-bcpn-connectx-3-pro-en-network-interface-card-for-ocp-40gbe-single-port-qsfp-pcie3-0-x8-no-bracket-rohs-r6.html, 2019.

[52] Intel. Intel ethernet converged network adapter xl710 10/40 gbe. https://www.intel.com/content/www/us/en/ethernet-products/converged-network-adapters/ethernet-xl710-brief.html, 2019.

[53] Mellanox. Mellanox ConnectX-3 VPI Card, 2017. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX-3_Pro_Card_VPI.pdf.

[54] Mellanox. Mellanox ConnectX-4 VPI Card, 2018. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX-4_VPI_Card.pdf.

[55] Mellanox. Mellanox ConnectX-5 VPI Card, 2018. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX-5_VPI_Card.pdf.

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

[57] Mellanox Technologies. InfiniBand Now Connecting More than 50 Percent of the TOP500 Supercomputing List. https://www.rdmag.com/news/2015/07/infiniband-now-connecting-more-50-percent-top500-supercomputing-list, 2015.

[58] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K. Panda. High Performance RDMA-based MPI Implementation over infiniBand. *International Journal of Parallel Programming*, 32(3):167–198, 2004.

[59] Wei Huang, Gopalakrishnan Santhanaraman, Hyun-Wook Jin, Qi Gao, and Dhabaleswar K. Panda. Design of High Performance MVAPICH2: MPI2 over InfiniBand. In *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID '06)*, Rio de Janeiro, Brazil, May 2006.

[60] Paolo Costa, Hitesh Ballani, Kaveh Razavi, and Ian Kash. R2c2: A network stack for rack-scale computers. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*, London, United Kingdom, August 2015.

[61] Ken Birman. A real-time cloud for the internet of things, 2016. Keynote talk at MesosCon North America '16.

[62] Derecho project, 2016. https://github.com/Derecho-Project/derecho-unified.

[63] 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.

[64] 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.

[65] Christopher Mitchell, Kate Montgomery, Lamont Nelson, Siddhartha Sen, and Jinyang Li. Balancing cpu and network in the cell distributed b-tree store. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference (ATC '16)*, Denver, CO, USA, June 2016.

[66] 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, USA, 2016.

[67] 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, United Kingdom, April 2016.

[68] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing rdma-enabled distributed transactions: Hybrid is better! In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*, Carlsbad, CA, USA, October 2018.

[69] Ming Wu, Fan Yang, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and Lidong Zhou. Gram: Scaling graph computation to the trillions. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC '15)*, Kohala Coast, HI, USA, August 2015.

[70] Rajarshi Biswas, Xiaoyi Lu, and Dhabaleswar Panda. Accelerating tensorflow with adaptive rdma-based grpc. In *25th IEEE International Conference on High Performance Computing, Data, and Analytics (HiPC '18)*, Bengaluru, India, December 2018.

[71] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. Apus: Fast and scalable paxos on rdma. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*, Santa Clara, CA, USA, September 2017.

[72] Marius Poke and Torsten Hoefler. Dare: High-performance state machine replication on rdma networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '15)*, Portland, OR, USA, June 2015.

[73] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: an rdma-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference (ATC '17)*, Santa Clara, CA, USA, July 2017.

[74] 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)*, Boston, MA, USA, July 2018.

[75] Hewlett-Packard. Memory Technology Evolution: An Overview of System Memory Technologies the 9th edition, 2010. `http://h20565.www2.hpe.com/hpsc/doc/public/display?sp4ts.oid=348553&docId=emr_na-c00256987`.

[76] 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)*, Cascais, Portugal, October 2011.

[77] Cisco, EMC, and Intel. The Performance Impact of NVMe and NVMe over Fabrics, 2014. `http://www.snia.org/sites/default/files/NVMe_Webcast_Slides_Final.1.pdf`.

[78] Dave Minturn. NVM Express Over Fabrics. In *11th Annual OpenFabrics International OFS Developers' Workshop*, Monterey, CA, USA, March 2015.

[79] Mellanox Technologies. NVIDIA GPUDirect Technology - Accelerating GPU-based Systems. `http://www.mellanox.com/pdf/whitepapers/TB_GPU_Direct.pdf`, 2010.

[80] NVIDIA. NVIDIA GPUDirect. `https://developer.nvidia.com/gpudirect`, 2010.

[81] Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, Emmett Witchel, and Mark Silberstein. GPUnet: Networking Abstractions for GPU Programs. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, USA, October 2014.

[82] Feras Daoud, Amir Watad, and Mark Silberstein. GPUrdma: GPU-side Library for High Performance Networking from GPU Kernels. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS '16)*, Kyoto, Japan, June 2016.

[83] 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, MA, USA, 1996.

[84] 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, WA, USA, November 1996.

[85] Songnian Zhou, Michael Stumm, Kai Li, and David Wortman. Heterogeneous Distributed Shared Memory. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):540554, September 1992.

[86] Michael Stumm and Songnian Zhou. Algorithms Implementing Distributed Shared Memory. *IEEE Computer*, 23(5):5464, May 1990.

[87] 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.

[88] 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)*, Litchfield Park, AZ, USA, December 1989.

[89] 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.

[90] Brad Fitzpatrick. Distributed Caching with Memcached. *Linux Journal*, 2004(124):5, 2004.

[91] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *In Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*, Lombard, IL, USA, April 2013.

[92] 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.

[93] Intel Corporation - Product and Performance Information. Intel Non-Volatile Memory 3D XPoint. `http://www.intel.com/content/www/us/en/architecture-and-technology/non-volatile-memory.html?wapkw=3d+xpoint`, 2018.

[94] M. Hosomi, H. Yamagishi, T. Yamamoto, K. Bessho, Y. Higo, K. Yamane, H. Yamada, M. Shoji, H. Hachino, C. Fukumoto, H. Nagao, and H. Kano. 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.

[95] 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.

[96] 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.

[97] 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.

[98] Micron Technology Inc. P8P Parallel Phase Change Memory (PCM). http://www.micron.com/~/media/Documents/Products/Data\%20Sheet/PCM/p8p_parallel_pcm_ds.pdf.

[99] 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 '10)*, Saint-Malo, France, June 2010.

[100] 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.

[101] J Joshua Yang, Dmitri B Strukov, and Duncan R Stewart. Memristive devices for computing. *Nature nanotechnology*, 8(1):13–24, 2013.

[102] E. Kltrsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu. Evaluating stt-ram as an energy-efficient main memory alternative. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '13)*, Austin, TX, USA, April 2013.

[103] Jim Handy. Understanding the intel/micron 3dxpoint memory. https://www.snia.org/sites/default/files/SDC15_presentations/persistant_mem/JimHandy_Understanding_the-Intel.pdf, 2015.

[104] Dongliang Xue, Chao Li, Linpeng Huang, Chentao Wu, and Tianyou Li. Adaptive memory fusion: Towards transparent, agile integration of persistent memory. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA '18)*, Vienna, Austria, February 2018.

[105] 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)*, Newport Beach, CA, USA, March 2011.

[106] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*, Newport Beach, CA, USA, March 2011.

[107] 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, MN, USA, June 2014.

[108] 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)*, Belgrade, Serbia, April 2017.

[109] 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)*, Atlanta, GA, USA, April 2016.

[110] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*, Big Sky, MT, USA, October 2009.

[111] 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.

[112] 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)*, Seattle, WA, USA, Nov 2011.

[113] 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)*, London, United Kingdom, April 2016.

[114] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing dram footprint with nvm in facebook. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*, Porto, Portugal, April 2018.

[115] Jian Yang, Joseph Izraelevitz, and Steven Swanson. Orion: A distributed file system for non-volatile main memory and rdma-capable networks. In *17th USENIX Conference on File and Storage Technologies (FAST '19)*, Boston, MA, USA, February 2019.

[116] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Efficient In-Memory Transactional Processing Using HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*, Monterey, CA, USA, October 2015.

[117] S. Chen, R. Wang, X. Wang, and K. Zhang. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *2010 IEEE Symposium on Security and Privacy (SP '10)*, Oakland, CA, USA, May 2010.

[118] YongBin Zhou and DengGuo Feng. Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing, 2005. `http://eprint.iacr.org/2005/388`.

[119] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. Oblivm: A programming framework for secure computation. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*, San Jose, CA, USA, May 2015.

[120] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of aes. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA '06)*, San Jose, CA, USA, February 2006.

[121] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on aes, and countermeasures. *Journal of Cryptology*, 23(1):37–71, January 2010.

[122] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*, Raleigh, NC, USA, October 2012.

[123] Onur Aciiçmez. Yet another microarchitectural attack: Exploiting i-cache. In *Proceedings of the 2007 ACM Workshop on Computer Security Architecture (CSAW '07)*, Fairfax, VA, USA, November 2007.

[124] Onur Aciiçmez and Werner Schindler. A vulnerability in rsa implementations due to instruction cache analysis and its demonstration on openssl. In *Proceedings of the 2008 The Cryptopgraphers' Track at the RSA Conference on Topics in Cryptology (CT-RSA '08)*, San Francisco, CA, USA, April 2008.

[125] Onur Aciiçmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. In *Proceedings of the 12th International Conference on Cryptographic Hardware and Embedded Systems (CHES '10)*, Santa Barbara, CA, USA, August 2010.

[126] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *Proceedings of the 2Nd ACM Symposium on Information, Computer and Communications Security (ASIACCS '07)*, Singapore, March 2007.

[127] Stephan van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Malicious management unit: Why stopping cache attacks in software is harder than you think. In *Proceedings of the 27th USENIX Conference on Security Symposium (SEC '18)*, Baltimore, MD, USA, August 2018.

[128] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *Proceedings of the 24th USENIX Conference on Security Symposium (SEC '15)*, Washington, D.C., USA, August 2015.

[129] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium (SEC '14)*, San Diego, CA, USA, August 2014.

[130] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *Proceedings of the 27th USENIX Conference on Security Symposium (SEC '18)*, Baltimore, MD, USA, August 2018.

[131] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael

Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP '19)*, San Francisco, CA, USA, May 2019.

[132] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Conference on Security Symposium (SEC '18)*, Baltimore, MD, USA, August 2018.

[133] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *Proceedings of the 27th USENIX Conference on Security Symposium (SEC '18)*, Baltimore, MD, USA, August 2018.

[134] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. Smotherspectre: exploiting speculative execution through port contention. `https://arxiv.org/abs/1903.01843`, 2018.

[135] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*, Scottsdale, Arizona, USA, November 2014.

[136] Luis M. Vaquero, Luis Rodero-Merino, and Daniel Morán. Locking the sky: A survey on iaas cloud security. *Computing*, 91(1):93–118, 2011.

[137] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. Homealone: Co-residency detection in the cloud via side-channel analysis. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy (SP '11)*, Oakland, CA, USA, May 2011.

[138] Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. Side channels in cloud services: Deduplication in cloud storage. *IEEE Security and Privacy*, 8(6):40–47, November 2010.

[139] Zhichao Hua, Dong Du, Yubin Xia, Haibo Chen, and Binyu Zang. Epti: Efficient defence against meltdown attack for unpatched vms. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (ATC '18)*, Boston, MA, USA, July 2018.

[140] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA '16)*, Barcelona, Spain, March 2016.

[141] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy (SP '09)*, Oakland, CA, USA, May 2009.

[142] Stephen M. Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John K. Ousterhout. It's Time for Low Latency. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems (HotOS '11)*, Napa, CA, USA, May 2011.

[143] 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)*, Atlanta, GA, USA, April 2016.

[144] Alexandras Daglis, Dmitrii Ustiugov, Stanko Novaković, Edouard Bugnion, Babak Falsafi, and Boris Grot. Sabres: Atomic object reads for in-memory rack-scale computing. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '16)*, Taipei, Taiwan, October 2016.

[145] Pravin Shinde, Antoine Kaufmann, Timothy Roscoe, and Stefan Kaestle. We need to talk about nics. In *Presented as part of the 14th Workshop on Hot Topics in Operating Systems (HotOS '13)*, Santa Ana Pueblo, NM, USA, May 2013.

[146] 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)*, Washington, DC, USA, February 2007.

[147] Sean Hefty. Rsockets. In *2012 OpenFabrics International Workshop*, Monterey, CA, USA, March 2012.

[148] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems (EUROSYS '12)*, Bern, Switzerland, April 2012.

[149] Feng Li, Sudipto Das, Manoj Syamala, and Vivek R. Narasayya. Accelerating relational databases by leveraging remote memory and rdma. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*, San Francisco, CA, USA, June 2016.

[150] Page Lawrence, Brin Sergey, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford University, 1998.

[151] 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 Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, USA, October 2014.

[152] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. Coordinated and efficient huge page management with ingens. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*, Savannah, GA, USA, November 2016.

[153] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. Large pages may be harmful on numa systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (ATC '14)*, Philadelphia, PA, USA, June 2014.

[154] Madhusudhan Talluri, Shing Kong, Mark D. Hill, and David A. Patterson. Tradeoffs in supporting two page sizes. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA '92)*, Queensland, Australia, May 1992.

[155] Manhee Lee, Eun Jung Kim, and Mazin Yousif. Security Enhancement in InfiniBand Architecture. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS '05)*, Washington, DC, USA, April 2005.

[156] Jonathan Corbet. Memory protection keys, 2015. `https://lwn.net/Articles/643797/`.

[157] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, USA, October 2014.

[158] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, USA, October 2014.

[159] Livio Soares and Michael Stumm. Flexsc: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, BC, Canada, October 2010.

[160] Richard Rashid, Daniel Julin, Douglas Orr, Richard Sanzi, Robert Baron, Alessandro Forin, David Golub, and Michael Jones. Mach: a system software kernel. In *Thirty-Fourth IEEE Computer Society International Conference: Intellectual Leverage (COMPCON '89)*, San Francisco, CA, USA, February 1989.

[161] Henry M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, USA, 1984.

[162] Ilya Lesokhin, Haggai Eran, Shachar Raindel, Guy Shapiro, Sagi Grimberg, Liran Liss, Muli Ben-Yehuda, Nadav Amit, and Dan Tsafrir. Page Fault Support for Network Controllers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*, Xi'an, China, April 2017.

[163] Frank Mietke, Robert Rex, Robert Baumgartl, Torsten Mehlan, Torsten Hoefler, and Wolfgang Rehm. Analysis of the memory registration process in the mellanox infiniband software stack. In *Proceedings of the 12th International Conference on Parallel Processing (Euro-Par '06)*, Dresden, Germany, September 2006.

[164] Johann George. qperf - measure rdma and ip performance, 2009. `https://linux.die.net/man/1/qperf`.

[165] J. Chu and V. Kashyap. Transmission of IP over InfiniBand (IPoIB), 2006. `https://tools.ietf.org/html/rfc4391`.

[166] Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, 1984.

[167] Jonathan Corbet. On vsyscalls and the vDSO, 2011. `https://lwn.net/Articles/446528/`.

[168] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. MegaPipe: A New Programming Interface for Scalable Network I/O. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI '12)*, Hollywood, CA, USA, October 2012.

[169] Galen M. Shipman, Ron Brightwell, Brian Barrett, Jeffrey M. Squyres, and Gil Bloch. Investigations on infiniband: Efficient network buffer utilization at scale. In *Proceedings of the 14th European Conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface (PVM/MPI '07)*, Paris, France, September 2007.

[170] 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.

[171] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, CA, USA, December 2004.

[172] Wikimedia Foundation. Wikimedia Downloads, 2015. `https://dumps.wikimedia.org/`.

[173] 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, NC, USA, April 2010.

[174] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, 1989.

[175] Ken Birman, Jonathan Behrens, Sagar Jha, Matthew Milano, Edward Tremel, and Robbert Van Renesse. Groups, Subgroups and Auto-Sharding in Derecho: A Customizable RDMA Framework for Highly Available Cloud Services. Technical report, Cornel University, 2016.

[176] Jonathan Behrens, Ken Birman, Sagar Jha, Matthew Milano, Edward Tremel, Eugene Bagdasaryan, Theo Gkountouvas, Weijia Song, and Robbert Van Renesse. Derecho: Group Communication at the Speed of Light. Technical report, Cornel University, 2016.

[177] redislabs. Redis. `https://redis.io/`, 2009.

[178] Michaela Blott, Kimon Karras, Ling Liu, Kees Vissers, Jeremia Bär, and Zsolt István. Achieving 10gbps line-rate key-value stores with fpgas. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '13)*, San Jose, CA, USA, June 2013.

[179] Michaela Blott, Ling Liu, Kimon Karras, and Kees Vissers. Scaling out to a single-node 80gbps memcached server with 40terabytes of memory. In *Proceedings of the 7th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage '15)*, Santa Clara, CA, USA, July 2015.

[180] Intel. Intel optane technology. `https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html`, 2019.

[181] Intel Corporation - Product and Performance Information. Reimagining the data center memory and storage hierarchy. `https://newsroom.intel.com/editorials/re-architecting-data-center-memory-storage-hierarchy/`, 2019.

[182] Sudarsun Kannan, Ada Gavrilovska, and Karsten Schwan. pvm: Persistent virtual memory for efficient capacity scaling and object storage. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*, London, United Kingdom, April 2016.

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

[184] 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)*, Indianapolis, IN, USA, June 2010.

[185] Jinglei Ren. Ycsb-c. `https://github.com/basicthinker/YCSB-C`, 2015.

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

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

[188] Krste Asanovi. 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).

[189] 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.

[190] 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.

[191] Hewlett Packard Labs. The machine. `https://www.labs.hpe.com/the-machine`, 2019.

[192] Kimberly Keeton . Memory driven computing. https://www.youtube.com/watch?v=eSP9euiV4-M, 2017.

[193] Gen-Z Consortium. Gen-z overview, 2016. https://genzconsortium.org/wp-content/uploads/2018/05/Gen-Z-Overview-V1.pdf.

[194] Mellanox. Innova-2 flex open programmable smartnic. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_Innova-2_Flex.pdf, 2018.

[195] 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 *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation (NSDI '18)*, Renton, WA, USA, April 2018.

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

[197] Open Coherent Accelerator Processor Interface, 2018. https://opencapi.org/.

[198] SNIA, Chet Douglas. Rdma with pmem, 2015. https://www.snia.org/sites/default/files/SDC15_presentations/persistant_mem/ChetDouglas_RDMA_with_PM.pdf.

[199] PCI Express. Pci express base specification revision 4.0 version 0.3, 2014.

[200] Paul Grun and Stephen Bates and Rob Davis. Persistent memory over fabrics. https://www.snia.org/sites/default/files/PM-Summit/2018/presentations/05_PM_Summit_Grun_PM_%20Final_Post_CORRECTED.pdf, 2018.

[201] Dan Tang, Yungang Bao, Weiwu Hu, and Mingyu Chen. Dma cache: Using on-chip storage to architecturally separate i/o data from cpu data for improving i/o performance. In *The Sixteenth International Symposium on High-Performance Computer Architecture (HPCA '10)*, Bangalore, India, Jan 2010.

[202] Mellanox Technologies. Rdma aware networks programming user manual. http://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf, 2015.

[203] Cloud Native Computing Foundation. grpc. https://grpc.io/, 2019.

[204] 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.

[205] Tao Chen and G. Edward Suh. Efficient data supply for hardware accelerators with prefetching and access/execute decoupling. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*, Taipei, Taiwan, October 2016.

[206] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proceedings of the VLDB Endowment*, 10(7):781–792, March 2017.

[207] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communication of the ACM*, 33(6):668–676, June 1990.

[208] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '00)*, Cambridge, MA, USA, November 2000.

[209] Intel. Intel xeon processor e5-2620 price. `https://www.intel.com/content/www/us/en/products/processors/xeon/e5-processors/e5-2620-v4.html`, 2019.

[210] Memory.net. Micron 32gb ddr4 price. `https://memory.net/product/mta36asf4g72pz-2g6-micron-1x-32gb-ddr4-2666-rdimm-pc4-21300v-r-dual-rank-x4-module/`, 2019.

[211] Mellanox. Mellanox connectx-4 adapter price. `https://store.mellanox.com/products/mellanox-mcx455a-ecat-connectx-4-vpi-adapter-card-edr-ib-and-100gbe-single-port-qsfp28-pcie3-0-x16-rohs-r6.html`, 2019.

[212] CPUBoss. Intel xeon e5-2620 review. `http://cpuboss.com/cpu/Intel-Xeon-E5-2620`, 2012.

[213] Raluca Ada Popa, Jacob R. Lorch, David Molnar, Helen J. Wang, and Li Zhuang. Enabling security in cloud storage slas with cloudproof. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference (ATC '11)*, Portland, OR, USA, June 2011.

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

[215] Mellanox. Connectx-6 en card. `http://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX-6_EN_Card.pdf`, 2019.

[216] Allyn Romanow, Jeffrey Mogul, Tom Talpey, and S. Bailey. Remote Direct Memory Access (RDMA) over IP Problem Statement. `https://tools.ietf.org/html/rfc4297`, 2005.

[217] Mellanox. Physical address memory region. `https://community.mellanox.com/s/article/physical-address-memory-region`, 2018.

[218] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation. In *Proceedings of the 25th USENIX Conference on Security Symposium (SEC '16)*, Austin, TX, USA, August 2016.

[219] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A remote software-induced fault attack in javascript. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '16)*, San Sebastián, Spain, July 2016.

[220] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *Proceeding of the 41st Annual International Symposium on Computer Architecture (ISCA '14)*, Minneapolis, MN, USA, June 2014.

[221] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Throwhammer: Rowhammer attacks over the network and defenses. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC '18)*, Boston, MA, USA, July 2018.

[222] Mellanox Technologies. Mellanox OFED for Linux User Manual. `http://www.mellanox.com/related-docs/prod_software/Mellanox_OFED_Linux_User_Manual_v3.1-1.0.0.pdf`.

[223] Mellanox. RDMA/RoCE Solutions. `https://community.mellanox.com/s/article/rdma-roce-solutions`, 2018.

[224] The Tcpdump Group. tcpdump - Dump Traffic on A Network. `https://www.tcpdump.org/manpages/tcpdump.1.html`, 2018.

[225] Thomas. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, September 1967.

[226] Robert Ricci, Eric Eide, and the CloudLab Team. Introducing cloudlab: Scientific infrastructure for advancing cloud architectures and applications. *The USENIX Magazine*, 39(6):36–38, December 2014.

[227] Mellanox. Mellanox InfiniBand EDR 100Gb/s Switch, 2018. `http://www.mellanox.com/related-docs/prod_ib_switch_systems/pb_sb7700.pdf`.

[228] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+flush: A fast and stealthy cache attack. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '16)*, San Sebastián, Spain, July 2016.

[229] Intel. Intel performance counter monitor. `https://software.intel.com/en-us/articles/intel-performance-counter-monitor`, 2012.

[230] Mellanox. Mellanox Network Adapters for 25G RoCE Ethernet Cloud Deployed in Alibaba, 2017. `http://www.mellanox.com/page/press_release_item?id=1964`.

[231] O. Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing (STOC '87)*, New York, NY, USA, May 1987.

[232] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *Journal of ACM*, 43(3):431–473, May 1996.

[233] Benny Pinkas and Tzachy Reinman. Oblivious ram revisited. In *Proceedings of the 30th Annual Conference on Advances in Cryptology (CRYPTO '10)*, Santa Barbara, CA, USA, August 2010.

[234] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious ram simulation. In *Proceedings of the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms (SPDA '12)*, Kyoto, Japan, January 2012.

[235] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious ram with o((logn)3) worst-case cost. In *Proceedings of the 17th International Conference on The Theory and Application of Cryptology and Information Security (ASIACRYPT '11)*, Seoul, South Korea, December 2011.

[236] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious ram and a new balancing scheme. In *Proceedings of the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '12)*, Kyoto, Japan, January 2012.

[237] Emil Stefanov, Elaine Shi, and Dawn Song. Towards practical oblivious RAM. In *Proceedings of the Network and Distributed System Security Symposium (NDSS '12)*, San Diego, CA, USA, February 2012.

[238] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: An extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer and communications Security (CCS '13)*, Berlin, Germany, November 2013.

[239] Xiao Shaun Wang, Yan Huang, T-H. Hubert Chan, Abhi Shelat, and Elaine Shi. Scoram: Oblivious ram for secure computation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*, Scottsdale, AZ, USA, November 2014.

[240] Cetin Sahin, Victor Zakhary, Amr El Abbadi, Huijia Lin, and Stefano Tessaro. Taostore: Overcoming asynchronicity in oblivious data storage. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP '16)*, San Jose, CA, USA, May 2016.

[241] Emil Stefanov and Elaine Shi. Oblivistore: High performance oblivious cloud storage. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP '13)*, San Francisco, CA, USA, May 2013.

[242] Peter Williams, Radu Sion, and Alin Tomescu. Privatefs: A parallel oblivious file system. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*, Raleigh, North Carolina, USA, October 2012.

[243] Jacob R. Lorch, Bryan Parno, James Mickens, Mariana Raykova, and Joshua Schiffman. Shroud: Ensuring private access to large-scale data in the data center. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST '13)*, San Jose, CA, USA, February 2013.

[244] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. Incbricks: Toward in-network computation with an in-network

cache. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*, Xi'an, China, April 2017.

[245] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, Shanghai, China, October 2017.

[246] Jianzhe Liang, Jun Bi, Yu Zhou, and Cheng Zhang. In-band network function telemetry. In *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos (SIGCOMM '18)*, Budapest, Hungary, August 2018.

[247] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mtcp: a highly scalable user-level TCP stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*, Seattle, WA, USA, April 2014.

[248] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-net: A user-level network interface for parallel and distributed computing. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*, Copper Mountain, CO, USA, December 1995.

[249] Jonas Pfefferle, Patrick Stuedi, Animesh Trivedi, Bernard Metzler, Ionnis Koltsidas, and Thomas R. Gross. A hybrid i/o virtualization framework for rdma-capable network interfaces. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '15)*, Istanbul, Turkey, July 2015.

[250] OpenFabrics Alliance. The OpenFabrics Enterprise Distribution, 2004. `https://www.openfabrics.org`.

[251] Intel. RDMA Communication Manager, 2010. `https://linux.die.net/man/7/rdma_cm`.

[252] Ron Brightwell, Bill Lawry, Arthur B. MacCabe, and Rolf Riesen. Portals 3.0: Protocol building blocks for low overhead communication. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS '02)*, Washington, DC, USA, April 2002.

[253] Dror Goldenberg, Michael Kagan, Ran Ravid, and Michael S. Tsirkin. Transparently achieving superior socket performance using zero copy socket direct protocol over 20gb/s infiniband links. In *2005 IEEE International Conference on Cluster Computing*, Burlington, MA, USA, Septemter 2005.

[254] Technical Committee T10. Scsi rdma protocol, July 2002. `http://www.t10.org/drafts.htm\#SCSI3_SRP`.

[255] Xingda Wei, Sijie Shen, Rong Chen, and Haibo Chen. Replication-driven live reconfiguration for fast distributed transaction processing. In *2017 USENIX Annual Technical Conference (ATC '17)*, Santa Clara, CA, USA, July 2017.

[256] Zhiyun Qian, Z. Morley Mao, and Yinglian Xie. Collaborative tcp sequence number inference attack: How to crack sequence number under a second. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*, Raleigh, NC, USA, October 2012.

[257] Zhiyun Qian and Z. Morley Mao. Off-path tcp sequence number inference attack - how firewall middleboxes reduce security. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP '12)*, San Francisco, CA, USA, May 2012.

[258] Yue Cao, Zhiyun Qian, Zhongjie Wang, Tuan Dao, Srikanth V. Krishnamurthy, and Lisa M. Marvel. Off-path tcp exploits: Global rate limit considered dangerous. In *Proceedings of the 25th USENIX Conference on Security Symposium (SEC '16)*, Austin, TX, USA, August 2016.

[259] Rob Jansen, Marc Juarez, Rafa Galvez, Tariq Elahi, and Claudia Diaz. Inside job: Applying traffic analysis to measure tor from within. In *25th Annual Network and Distributed System Security Symposium (NDSS '18)*, San Diego, CA, USA, February 2018.

[260] Xun Gong, Nikita Borisov, Negar Kiyavash, and Nabil Schear. Website detection using remote traffic analysis. In *Privacy Enhancing Technologies - 12th International Symposium (PETS '12)*, Vigo, Spain, July 2012.

[261] Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In *Proceedings of the 16th European Conference on Research in Computer Security (ESORICS '11)*, Leuven, Belgium, September 2011.

[262] David Cock, Qian Ge, Toby Murray, and Gernot Heiser. The last mile: An empirical study of timing channels on sel4. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*, Scottsdale, AZ, USA, November 2014.

[263] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. Netspectre: Read arbitrary memory over network, 2018. `http://arxiv.org/abs/1807.10535`.

[264] Zachary Weinberg, Eric Y. Chen, Pavithra Ramesh Jayaraman, and Collin Jackson. I still know what you visited last summer: Leaking browsing history via user interaction and side channel attacks. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy (SP '11)*, Oakland, CA, USA, May 2011.

[265] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting SGX enclaves from practical side-channel attacks. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (ATC '18)*, Boston, MA, USA, July 2018.

[266] Intel. Improving Real-Time Performance by Utilizing Cache Allocation Technology , 2015. `https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf`.

[267] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization*, 8(4):35:1–35:21, January 2012.

[268] Zhenghong Wang and Ruby B. Lee. Covert and side channels due to processor architecture. In *Proceedings of the 22Nd Annual Computer Security Applications Conference (ACSAC '06)*, Miami Beach, FL, USA, December 2006.

[269] Zhenghong Wang and Ruby B. Lee. A novel cache architecture with enhanced performance and security. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '41)*, Lake Como, Italy, November 2008.

[270] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud. In *Proceedings of the 21st USENIX Conference on Security Symposium (SEC '12)*, Bellevue, WA, USA, August 2012.

[271] Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSNW '11)*, Hong Kong, China, June 2011.

[272] Ziqiao Zhou, Michael K. Reiter, and Yinqian Zhang. A software approach to defeating side channels in last-level caches. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*, Vienna, Austria, October 2016.

[273] Yinqian Zhang and Michael K. Reiter. Dúppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *Proceedings of the 2013 ACM SIGSAC conference on Computer and communications Security (CCS '13)*, Berlin, Germany, November 2013.

[274] Chelsio. TCP Offload Engine, 2019. `https://www.chelsio.com/nic/tcp-offload-engine/`.

[275] 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)*, Florianopolis, Brazil, August 2016.

[276] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. Softnic: A software nic to augment hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.

[277] Joel Coburn, Trevor Bunker, Meir Schwarz, Rajesh Gupta, and Steven Swanson. From aries to mars: Transaction support for next-generation, solid-state drives. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*, SOSP '13, Farminton, PA, USA, November 2013.

[278] 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 Conference on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, USA, October 2014.

[279] Hung-Wei Tseng, Qianchen Zhao, Yuxiao Zhou, Mark Gahagan, and Steven Swanson. Morpheus: Creating application objects efficiently for heterogeneous computing. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*, Seoul, Republic of Korea, June 2016.

[280] Butler W. Lampson. Hints for computer system design. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles (SOSP '83)*, Bretton Woods, NH, USA, October 1983.

VITA

Shin-Yeh Tsai was born in Kaohsiung, Taiwan. He received bachelor degree in Computer Science at National Sun Yat-Sen University, Kaohsiung, Taiwan in 2009, and master degree in Computer and Communication Engineering in National Cheng Kung University, Tainan, Taiwan in 2012. In 2013 Fall, he joined Department of Computer Science at Purdue University to pursue a Ph.D. degree. His research interests span operating systems, distributed systems, and datacenter networking, with a current focus on RDMA-based datacenter networks and remote/distributed memory systems.