LITE Kernel RDMA Support for Datacenter Applications

Shin-Yeh Tsai
Purdue University
tsai46@purdue.edu

Yiying Zhang
Purdue University
yiying@purdue.edu

ABSTRACT

Recently, there is an increasing interest in building datacenter applications with RDMA because of its low-latency, high-throughput, and low-CPU-utilization benefits. However, RDMA is not readily suitable for datacenter applications. It lacks a flexible, high-level abstraction; its performance does not scale; and it does not provide resource sharing or flexible protection. Because of these issues, it is difficult to build RDMA-based applications and to exploit RDMA's performance benefits.

To solve these issues, we built *LITE*, a *Local Indirection TiEr* for RDMA in the Linux kernel that virtualizes native RDMA into a flexible, high-level, easy-to-use abstraction and allows applications to safely share resources. Despite the widely-held belief that kernel bypassing is essential to RDMA's low-latency performance, we show that using a kernel-level indirection can achieve both flexibility *and* low-latency, scalable performance at the same time. To demonstrate the benefits of LITE, we developed several popular datacenter applications on LITE, including a graph engine, a MapReduce system, a Distributed Shared Memory system, and a distributed atomic logging system. These systems are easy to build and deliver good performance. For example, our implementation of PowerGraph uses only 20 lines of LITE code, while outperforming PowerGraph by 3.5× to 5.6×.

CCS CONCEPTS

• **Networks** → **Network design principles**; **Programming interfaces**; **Data center networks**; • **Software and its engineering** → **Operating systems**; **Message passing**;

KEYWORDS

RDMA, indirection, network stack, low-latency network

ACM Reference Format:
Shin-Yeh Tsai and Yiying Zhang. 2017. LITE Kernel RDMA Support for Datacenter Applications. In *Proceedings of SOSP '17, Shanghai, China, October 28, 2017,* 19 pages.
https://doi.org/10.1145/3132747.3132762

1 INTRODUCTION

Remote Direct Memory Access (*RDMA*) is the ability of directly accessing memory on a remote machine through network. RDMA provides low latency, high bandwidth, and low CPU utilization, and has been widely adopted in High Performance Computing environments for many years [32, 51, 55].

Because of datacenter applications' need for low-latency network communication and because of the more mature hardware support for RDMA [12, 17, 28, 41, 42, 54, 57, 60], there has been increasing interest from both academia and industry in recent years to build RDMA-based datacenter applications [6, 11, 19, 20, 37–39, 52, 58, 69, 70, 77, 78, 81, 82].

Although many design choices in RDMA suit a confined, single-purpose environment like HPC well, native RDMA (*i.e.*, unmodified RDMA hardware, driver, and default libraries) 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 [19, 20, 37–39, 58], significant amount of application adaptation [6, 70, 81], or changes in RDMA drivers [19, 39].

The second reason why native RDMA does not fit datacenter usages is because there is no software to manage or protect RDMA resources. 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.

The root cause of all the above issues is RDMA's design of letting applications directly access hardware RNICs. Although this design minimizes software overhead, the issues it causes will largely limit RDMA's adoption in datacenters.

We propose to *virtualize* 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.

We built *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 1 and 2 illustrate the architecture of native RDMA and LITE.

Our approach of *onloading* [68] functionalities into kernel is the opposite to RDMA and many other recent networking systems' [16, 40, 71] approach of offloading functionalities into hardware. 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 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 changes in 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 [38, 39], 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 [25], 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 [65] with 49 lines of LITE code, and it outperforms Hadoop [1] by 4.3× to 5.3×.

Overall, this paper makes the following key contributions:

- We identify three main issues of using native RDMA in datacenter environments and the root cause of them.
- We are the first to propose virtualizing RDMA with a generic kernel-level indirection layer for datacenter RDMA applications.
- We designed 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.
- We built the LITE system which solves all the three issues of native RDMA for datacenter applications. Datacenter applications can easily use LITE to perform low-latency network communication and distributed operations. RNICs can decrease its hardware complexity and on-RNIC memory by relying on LITE to manage and protect its resources.
- We developed four datacenter applications on LITE, evaluated their performance, and summarized our application programming experience.

Our implementation of LITE and LITE-based applications is publicly available at https://github.com/Wuklab/LITE.

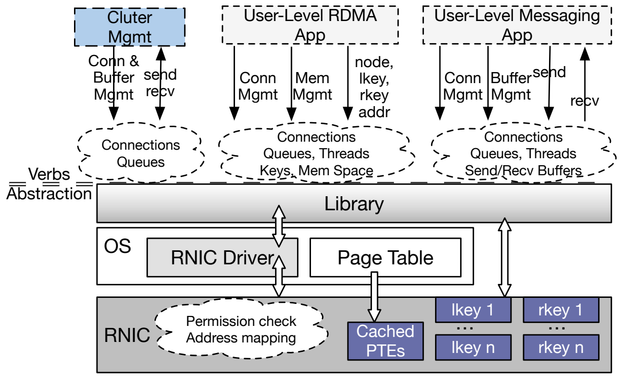

Figure 1: Traditional RDMA Stack.

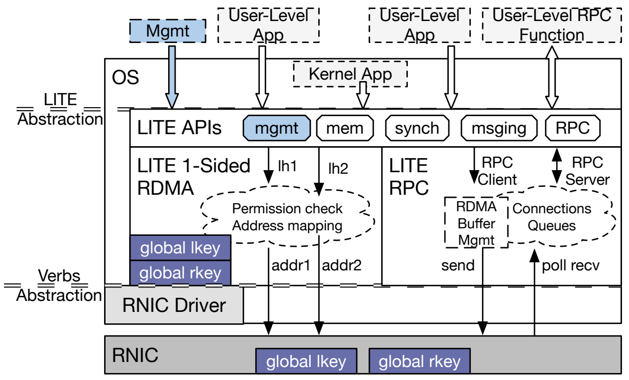

Figure 2: LITE Architecture.

2 BACKGROUND AND ISSUES OF RDMA

This section provides a brief background of RDMA, its usage in datacenters, and its limitations for datacenter applications. Figure 1 illustrates the architecture of native RDMA and how applications work with it.

2.1 Background on RDMA

RDMA allows user-space applications to directly read or write remote memory without kernel interference or memory copying. 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. Two-sided RDMA operations inform the remote node of a delivered message.

RDMA supports reliable and unreliable connections (*RC* and *UC*) and unreliable datagram (*UD*). 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.

RDMA communication is implemented using various types of *queues*. RC communication is established by building a pair of queues between two nodes, called a queue pair (*QP*). To perform a one-sided RDMA operation, an application process at a remote node needs to first register a *memory region* (*MR*), obtain a remote protection key (called *rkey*) for the MR, and convey the virtual address of the MR and its *rkey* to the local node. 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 RNIC. 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.

There are three implementations of RDMA: InfiniBand (IB) [34], RoCE [33], and iWARP [67]. IB is a switched network that is specifically designed for RDMA. RoCE and iWARP are two Ethernet-based technologies that also offer the RDMA Verbs interface. Since LITE builds on top of the Verbs interface, it is applicable to all these implementations of RDMA.

2.2 RDMA in Datacenter Applications

The past two decades have seen a growing usage of RDMA in HPC environments [32, 51, 55]. In recent years, there is an emerging trend in using RDMA in datacenter environments from both industry and academia [2, 7, 15, 27]. Recent RDMA-based applications include key-value store systems [19, 37–39, 58], DSM systems [19, 59], database and transactional systems [6, 11, 20, 78, 81], graph store system [70], consensus system [77], and distributed NVM systems [52, 69, 82].

We believe that the use of RDMA in datacenters will continue increasing in the future because of application needs and of hardware support. Many modern datacenter applications demand fast access to vast amount of data, most conveniently and efficiently as in-memory data. With memory on a single machine facing its wall [31], these applications can largely benefit from fast, direct access to remote memory [27, 59, 61]. At the same time, more hardware in datacenters are adding the support for direct RDMA accesses, such as NVMe over Fabrics [12, 57] and GPU Direct [17, 41, 54, 60].

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 discuss in §2.3, §2.4, and §2.5.

2.3 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 [38]. Applications have to explicitly manage various types of resources and go through several non-intuitive steps to perform an RDMA operation, as explained in §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 [19, 38, 39].

An API wrapper on top of the native RDMA interface such as Rsocket [30] 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 (§2.4), makes it hard to share resources across processes (§2.5), 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.

2.4 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 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 [21] 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 [53] 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 [49].

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 5), thrashing will happen and degrade performance. Unfortunately, most datacenter applications use large amount of memory. For example, performing PageRank [45] on a 1.3 GB dataset using GraphX [26] and Spark [80] will need 12 GB and 16 GB memory heaps respectively [27]. FaRM [19] 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 [22, 44, 75].

Finally, RNIC stores metadata for each QP in its memory. RDMA performance drops when the number of QPs increases [19], largely limiting the total number of nodes that can be connected through RC in an RDMA cluster. FaSST [39] 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 [68]. 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.

2.5 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 [19] 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 [46].

3 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.

3.1 Kernel-Level Indirection

Many of RDMA's issues discussed in §2 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 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 [19, 20, 37–39], 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.

3.3 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 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 (§4), and a stack for RPC functions and messaging based on

	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

Table 1: Major LITE APIs. *Only the APIs in bold have their counterparts in native Verbs.*

two-sided RDMA operations (§5). These parts share many resources, such as QPs, CQs, and LITE internal threads (§6).

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 (§2.5). Improvement to our current security model is possible, for example, by reducing the physical memory space LITE controls or utilizing techniques like MPK [14]. 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.

3.4 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 [5, 63].

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 (§2.4). 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 §4.2.

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 §4, 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 (§4.1).

Hide kernel cost. We design several techniques to *hide* system call overhead by moving most of the overhead off performance-critical paths (§5.2). Unlike previous lightweight system call solutions [73], 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. §4 and §5 describe in detail LITE's one-sided RDMA and RPC stacks. §4 also presents a new virtualized memory abstraction LITE uses. §6 discusses LITE's resource sharing and QoS mechanisms. §7 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 paper 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.

4 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

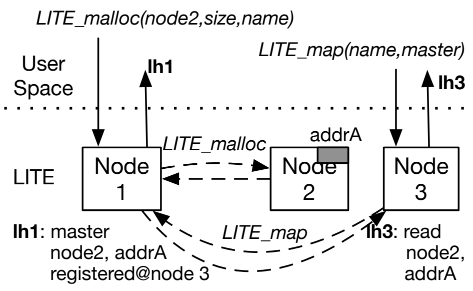

Figure 3: LITE *lh* Example. *Node1 allocates an LMR from node2 and is the master of it. Node3 maps the LMR with read permission by contacting Node1.*

at the local side, LITE can support more flexible, transparent, and efficient memory region management and access control.

4.1 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 [48, 66] 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, since we want to avoid any indirection at the remote node and retain RDMA's direct remote access. Figure 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.

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 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 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

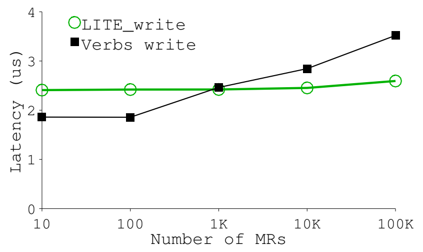

Figure 4: RDMA Write Latency against Num of (L)MRs. *Each (L)MR is 4 KB. Each write is 64B and its location is randomly chosen from all (L)MRs.*

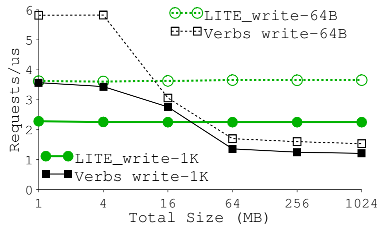

Figure 5: RDMA Write Throughput against (L)MR Size. *Each run uses one MR. Each RDMA operation randomly writes 64B or 1KB.*

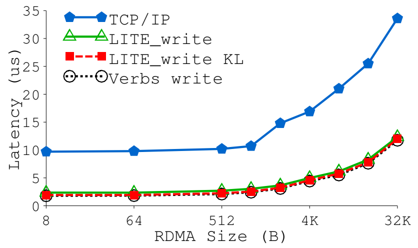

Figure 6: LITE and Native RDMA Write Latency. *KL denotes kernel-level invocation. Lines without KL denotes user-level.*

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 (§4.2).

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 one global *lkey* and *rkey* at the RNIC, LITE has no scalability issue with the amount of LMRs (§4.2).

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 [47, 56]. 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.

4.2 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 [19], 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 §2.4, while still delivering close-to-raw-RDMA performance when the scale is small. We expect datacenter environments to have large scale, making LITE a better performant choice than native RDMA.

Figure 4 presents the latency of *LT_write* and native RDMA write as the number of LMRs or MRs increases. Figure 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

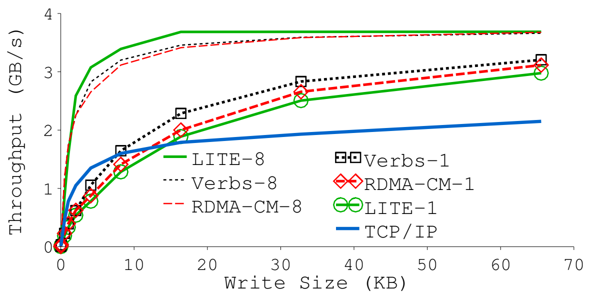

Figure 7: LITE RDMA Throughput. *The number at the end of each line label represents the level of parallelism in request issuing. TCP/IP uses 1 thread with tcp_bw test in qperf.*

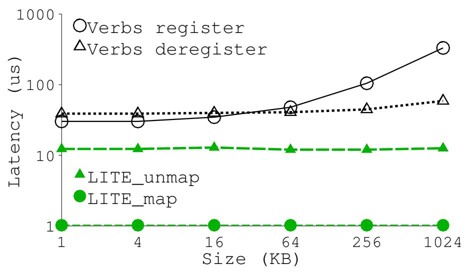

Figure 8: (De)Registering (L)MR Latency under LITE and native RDMA. *This LT_map targets to a local LMR.*

with both the number of LMRs and the total size of LMR and outperforms native RDMA when the scale is big.

Figure 6 and Figure 7 take a closer look at the latency and throughput comparison of LITE, native RDMA write, and TCP/IP. We use *qperf* [23] to measure the performance of TCP/IP on InfiniBand (via IPoIB). 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 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 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.

5 LITE RPC

The previous section describes LITE's memory abstraction and basic memory-like operations in Table 1. In addition to one-sided RDMA and memory-like operations, 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 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 [9]. Each RPC function is associated with an ID which multiple RPC clients can *bind* to and multiple RPC server threads can execute.

5.1 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 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 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

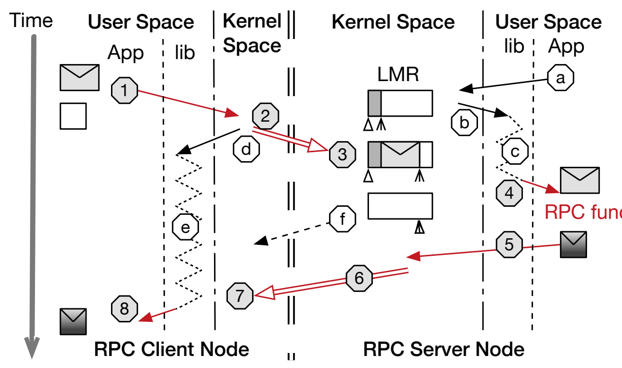

Figure 9: LITE RPC Mechanism. *Red arrows represent the performance critical path of LT_RPC.*

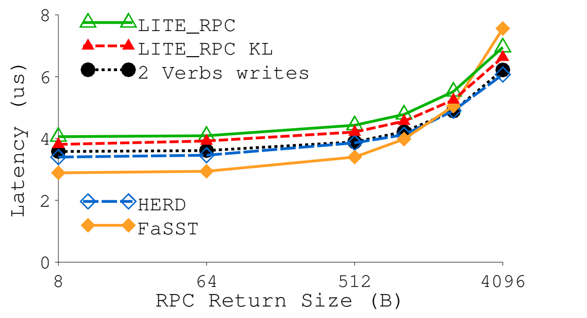

Figure 10: RPC Latency Comparison. *The RPC input size is always 8B. FaRM uses two RDMA writes to implement message passing primitive. Sender uses one RDMA write to write to a buffer, and receiver polls the buffer to get message. We use two RDMAwrites to emulate a RPC call in FaRM.*

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 timeout error to user. Thus, we can safely remove the check of sending states.

5.2 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 around 0.9 μs. 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 around 0.17 μs in our experiments.

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 [13, 73]. 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.

LITE minimizes CPU utilization when performing the above optimizations. Unlike previous solutions that require multiple kernel threads to reduce system call overhead [73] or require changes in system call interface [29, 73], 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.

5.3 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,

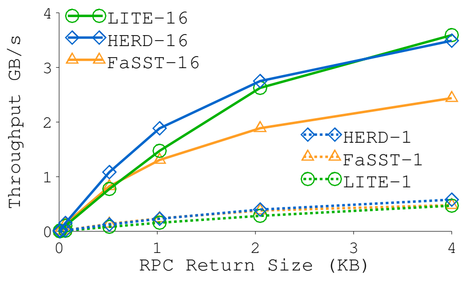

Figure 11: RPC Throughput. *RPC throughputs using 16 and 1 concurrent RPC clients and servers. RPC input size is always 8 bytes.*

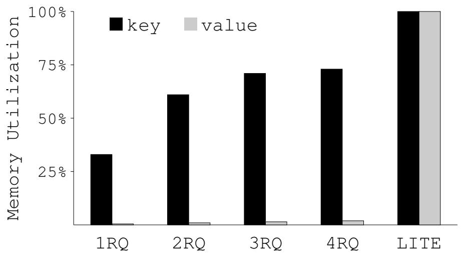

Figure 12: LITE RPC Memory Utilization. *The first 4 set of bars represent send-based RPC with different number of varying-sized RQs.*

and low CPU utilization. Figures 10 and 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 [19] 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.

Next, we compare LITE with two open-source RDMA-based RPC systems, HERD [38] and FaSST [39], 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 10). However, in practice, HERD's mechanism does not work for our purpose of serving datacenter applications, since it 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 memory space [72]. In comparison, with write-imm, LITE does not need receive buffers for any RPC data. Figure 12 presents the memory space utilization with LITE RPC and *send* under the Facebook key-value store distributions [3]. 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 [72]. 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.

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 [3]. 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 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.

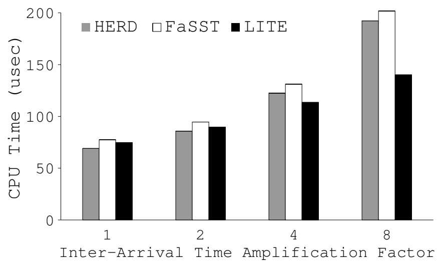

Figure 13: CPU Usage with Facebook Distribution. *Average CPU time per request when changing the Facebook distribution inter-arrival time with an amplification factor.*

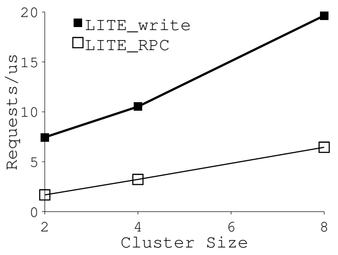

Figure 14: Scalability of LITE RDMA and RPC. *Each node runs 8 threads. LT_write writes 64B. LT_RPC sends 64B and gets 8B reply.*

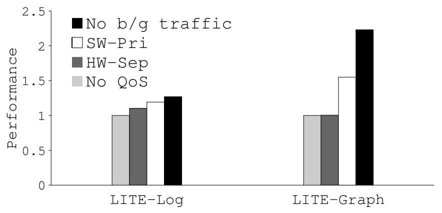

Figure 15: LITE QoS with Real Applications. *No b/g traffic represents running LITE-Graph and LITE-Log without low-priority background traffic. No QoS is used as a baseline of performance (higher is better).*

6 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.

6.1 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 $1{\le}K{\le}4$ 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 [19] shares QPs within an application and requires $2{\times}N{\times}T/q$ QPs, where q is a sharing factor. FaSST [39] 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 14 shows that LITE one-sided RDMA and *LT_RPC* both scale well with number of nodes.

6.2 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 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 (§8.3) and LITE-Log (§8.1), with high priority, and a low-priority background task of constantly writing data to four nodes. Figure 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.

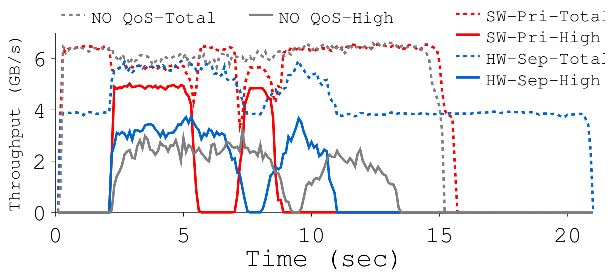

Figure 16: LITE QoS under Synthetic Workload. *20 low-priority threads start from time 0, each running 600K requests (5 doing 4 KB LT_write, 5 doing 8 KB LT_write, 5 doing 4 KB LT_read, and 5 doing 4 KB LT_read). After 2 seconds, 20 high-priority threads join the system, each running 200K requests (10 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.*

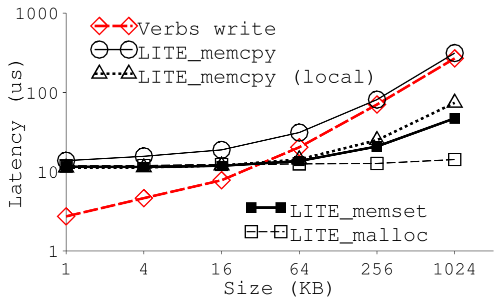

Figure 17: LITE Memory Operations Latency. *LT_memcpy (local) represents LT_memcpy between two LMRs on the same node. LT_memmove is the same as LT_memcpy.*

7 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 functionalities include memory-like operations that we did not cover in §4 (rest of the Memory part of Table 1) and synchronization operations (the Sync part of Table 1).

7.1 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 17 presents the latency of *LT_malloc*, *LT_memset*, *LT_memcpy*, and *LT_memmove* (Table 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.

7.2 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 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.

8 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.

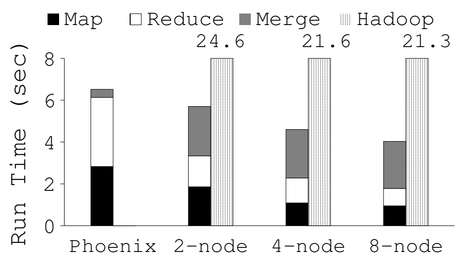

Figure 18: MapReduce Performance. *The left bars in the 2,4,8-node groups are LITE-MR.*

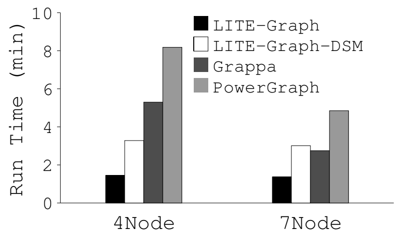

Figure 19: PageRank Performance. *Each node runs four threads.*

8.1 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.

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.

8.2 MapReduce

LITE-MR is our implementation of MapReduce [18] on LITE. We ported LITE-MR from Phoenix [65], a single-node multi-threaded implementation of MapReduce. We spread the original Phoenix mapper and reducer threads onto a set of worker nodes 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 18 presents the WordCount run time of the Wikimedia workloads [79] using Phoenix, LITE-MR, and Hadoop [1]. 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.

8.3 Graph Engine

We implemented a new graph engine, LITE-Graph, based on the design of PowerGraph [25]. 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 [25].

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 [45] on the Twitter dataset (1 M vertices, 1 B directed edges) [43] using LITE-Graph, PowerGraph, and Grappa [59]. Grappa is a DSM system that uses a customized IB-based network stack to aggregates network requests. PowerGraph uses IPoIB on InfiniBand. Figure 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.

8.4 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 [50, 83]. 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.

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

Figure 20: LITE Application Implementation Effort. **LITE-MR ports from the 3000-LOC Phoenix with 600 lines of change or addtion.*

In building these distributed protocols, we found the need of a multicast function [4, 7, 8]. 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 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.

8.5 Programming Experience

Overall, we find LITE very simple to use and it provides all the network functionalities that our applications need. Figure 20 lists the lines of code (LOC) and 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.

9 RELATED WORK

Several new RDMA-based systems were built in the past few years for datacenter environments. FaRM [19] is an RDMA-based distributed memory platform which inspired LITE. Its core communication stack uses RDMA read and write. On top of the basic communication stack, FaRM builds a distributed shared memory layer, a transaction system [20], 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 [58] and HERD [37] are two key-value store systems that implement customized RDMA stacks using RDMA read, write, or send. HERD-RPC [38] and FaSST [39] are two RDMA-based RPC implementations. FaSST's main goal is to scale with number of nodes. The Derecho project [4, 7, 8] 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 [6, 11, 20, 78, 81], distributed RDF store [70], DSM system [59], consensus system [77], and distributed NVM systems [52, 69, 82]. These systems all target a specific type of application and most of them builds 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.

There are several RDMA-based user-level libraries including the standard OFED library [62], rdma-cm [35], MVAPICH2 [32, 51], Rsockets [30], and Portals [10]. MVAPICH2 supports the MPI interface and is designed for the HPC environments. Rsockets implements a socket-like abstractions. Portals exposes an abstraction that is based on put and get operations. LITE's abstraction is designed for datacenter applications and is richer and more flexible than these libraries' abstractions. 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 IB, such as IPoIB, SDP [24], and SRP [74], to support traditional network and storage interfaces. They all have heavy performance overhead and do not offer the low-latency performance as LITE does.

Finally, there have been various efforts in user-level TCP/IP implementations such as mTCP [36], U-Net [76], IX [5], and Arrakis [63]. 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. 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 [64]. 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. Moreover, LITE is designed for RDMA and solves RDMA's issues in the datacenter environments.

10 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.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd Tim Harris for their tremendous feedback and comments, which have substantially improved the content and presentation of this paper. We also thank Felix Lin, T.N. Vijaykumar, Florentina Popovici, and Sanjay Rao for their suggestions to an early version of the paper.

REFERENCES

[1] 2011. Apache Hadoop. (2011). http://hadoop.apache.org/.

[2] 2016. Derecho project. (2016). https://github.com/Derecho-Project/derecho-unified.

[3] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. 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, UK.

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

[5] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. 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.

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

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

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

[9] Andrew D. Birrell and Bruce Jay Nelson. 1984. Implementing Remote Procedure Calls. *ACM Trans. Comput. Syst.* 2, 1 (1984), 39–59.

[10] Ron Brightwell, Bill Lawry, Arthur B. MacCabe, and Rolf Riesen. 2002. 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.

[11] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. 2016. Fast and General Distributed Transactions Using RDMA and HTM. In *Proceedings of the Eleventh European Conference on Computer Systems (EUROSYS '16)*. London, UK.

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

[13] Jonathan Corbet. 2011. On vsyscalls and the vDSO. (2011). https://lwn.net/Articles/446528/.

[14] Jonathan Corbet. 2015. Memory protection keys. (2015). https://lwn.net/Articles/643797/.

[15] Paolo Costa, Hitesh Ballani, Kaveh Razavi, and Ian Kash. 2015. 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, UK.

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

[17] Feras Daoud, Amir Watad, and Mark Silberstein. 2016. 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.

[18] Jeffrey Dean and Sanjay Ghemawat. 2004. 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.

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

[20] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. Monterey, CA, USA.

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

[22] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. 2014. 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.

[23] Johann George. 2009. qperf - Measure RDMA and IP performance. (2009). https://linux.die.net/man/1/qperf.

[24] Dror Goldenberg, Michael Kagan, Ran Ravid, and Michael S. Tsirkin. 2005. 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.

[25] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2010. 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.

[26] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. 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.

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

[28] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over Commodity Ethernet at Scale. In *Proceedings of the 2016 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '16)*. Florianopolis, Brazil.

[29] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. 2012. 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.

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

[31] Hewlett-Packard. 2010. 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.

[32] Wei Huang, Gopalakrishnan Santhanaraman, Hyun-Wook Jin, Qi Gao, and Dhabaleswar K. Panda. 2006. 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.

[33] InfiniBand Trade Association. 2014. RoCEv2 Architecture Specification. (2014). https://cw.infinibandta.org/document/dl/7781.

[34] InfiniBand Trade Association. 2015. InfiniBand Architecture Specification. (2015). https://cw.infinibandta.org/document/dl/7859.

[35] Intel. 2010. RDMA Communication Manager. (2010). https://linux.die.net/man/7/rdma_cm.

[36] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. 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.

[37] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. 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.

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

[39] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. 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.

[40] Antoine Kaufmann, Simon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. 2016. 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, Georgia, USA.

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

[42] Ashok Krishnamoorthy, Hiren Thacker, Ola Torudbakken, Shimon Muller, Arvind Srinivasan, Patrick Decker, Hans Opheim, John Cunningham, Ivan Shubin, Xuezhe Zheng, Marcelino Dignum, Kannan Raj, Eivind Rongved, and Raju Penumatcha. 2016. From Chip to Cloud: Optical Interconnects in Engineered Systems. *Journal of Lightwave Technology* 35, 15 (2016), 3103–3115.

[43] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. 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.

[44] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2016. 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.

[45] Page Lawrence, Brin Sergey, Rajeev Motwani, and Terry Winograd. 1998. *The PageRank Citation Ranking: Bringing Order to the Web.* Technical Report. Stanford University.

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

[47] Ilya Lesokhin, Haggai Eran, Shachar Raindel, Guy Shapiro, Sagi Grimberg, Liran Liss, Muli Ben-Yehuda, Nadav Amit, and Dan Tsafrir. 2017. 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.

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

[49] Feng Li, Sudipto Das, Manoj Syamala, and Vivek R. Narasayya. 2016. 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.

[50] Kai Li and Paul Hudak. 1989. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. Comput. Syst.* 7, 4 (1989), 321–359.

[51] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K. Panda. 2004. High Performance RDMA-based MPI Implementation over infiniBand. *Int. J. Parallel Program.* 32, 3 (2004), 167–198.

[52] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. 2017. Octopus: an RDMA-enabled Distributed Persistent Memory File System. In *2017 USENIX Annual Technical Conference (ATC '17)*. Santa Clara, CA, USA.

[53] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache Craftiness for Fast Multicore Key-value Storage. In *Proceedings of the 7th ACM European Conference on Computer Systems (EUROSYS '12)*. Bern, Switzerland.

[54] Mellanox Technologies. 2010. NVIDIA GPUDirect Technology - Accelerating GPU-based Systems. http://www.mellanox.com/pdf/whitepapers/TB_GPU_Direct.pdf. (2010).

[55] Mellanox Technologies. 2015. InfiniBand Now Connecting More than 50 Percent of the TOP500 Supercomputing List. http://tinyurl.com/zcz97fs. (2015).

[56] Frank Mietke, Robert Rex, Robert Baumgartl, Torsten Mehlan, Torsten Hoefler, and Wolfgang Rehm. 2006. Analysis of the Memory Registration Process in the Mellanox Infiniband Software Stack. In *Proceedings of the 12th International Conference on Parallel Processing (EUROPAR '06)*. Dresden, Germany.

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

[58] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using One-sided RDMA Reads to Build a Fast, CPU-efficient Key-value Store. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC'13)*. San Jose, CA, USA.

[59] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2015. Latency-Tolerant Software Distributed Shared Memory. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC '15)*. Santa Clara, CA, USA.

[60] NVIDIA. 2010. NVIDIA GPUDirect. https://developer.nvidia.com/gpudirect. (2010).

[61] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. 2011. Fast Crash Recovery in RAMCloud. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*. Cascais, Portugal.

[62] OpenFabrics Alliance. 2004. The OpenFabrics Enterprise Distribution. (2004). https://www.openfabrics.org.

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

[64] Jonas Pfefferle, Patrick Stuedi, Animesh Trivedi, Bernard Metzler, Ionnis Koltsidas, and Thomas R. Gross. 2015. 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.

[65] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. 2007. 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.

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

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

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

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

[70] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. 2016. 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.

[71] Pravin Shinde, Antoine Kaufmann, Timothy Roscoe, and Stefan Kaestle. 2013. 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.

[72] Galen M. Shipman, Ron Brightwell, Brian Barrett, Jeffrey M. Squyres, and Gil Bloch. 2007. 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.

[73] Livio Soares and Michael Stumm. 2010. 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.

[74] Technical Committee T10. 2002. SCSI RDMA Protocol. (July 2002). http://www.t10.org/drafts.htm#SCSI3_SRP.

[75] Madhusudhan Talluri, Shing Kong, Mark D. Hill, and David A. Patterson. 1992. Tradeoffs in Supporting Two Page Sizes. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA '92)*. Queensland, Australia.

[76] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. 1995. 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.

[77] Cheng Wang, Xusheng Chen, Jianyu Jiang, Ning Yi, and Heming Cui. 2017. APUS: Fast and Scalable PAXOS on RDMA. In *Proceedings of the 8th Annual Symposium on Cloud Computing (SOCC '17)*. Santa Clara, CA, USA.

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

[79] Wikimedia Foundation. 2015. Wikimedia Downloads. (2015). https://dumps.wikimedia.org/.

[80] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. 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.

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

[82] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. 2015. 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.

[83] Yuanyuan Zhou, Liviu Iftode, and Kai Li. 1996. 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.