An Ultra-Low Latency and Compatible PCIe Interconnect for Rack-scale Communication

Yibo Huang★, Yukai Huang★, Ming Yan★, Jiayu Hu†, Cunming Liang†, Yang Xu★, Wenxiong Zou★, Yiming Zhang★, Rui Zhang★, Chunpu Huang★, Jie Wu★*
★Fudan University, China †Intel, China

ABSTRACT

Emerging network-attached resource disaggregation architecture requires ultra-low latency rack-scale communication. However, current hardware offloading (e.g., RDMA) and user-space (e.g., mTCP) communication schemes still rely on heavily layered protocol stacks which requires the translation between PCIe bus and network protocol, or complex connection/memory resource management within RNICs, inevitably bringing latency overhead.

We argue that PCIe Non-Transparent Bridge (NTB) is a superior high-speed in-rack network technology to interconnect PCIe-attached machines or devices with the same PCIe fabric since no translation is needed between PCIe and network protocol. We present **NTSocks**, the first user-space in-rack interconnect over PCIe fabric which virtualizes native NTB into high-level network functionalities for rack-scale systems with software-hardware co-design. NTSocks provides (1) compatibility with a fast socket-like abstraction, (2) multi-thread scalability using a core-driven dataplane model, and (3) fair and efficient resource sharing with a multi-tenant isolation mechanism. Even though PCIe NTB is originally designed for device communication across PCIe domains, NTSocks shows a flexible user-level indirection with performance close to bare-metal NTB while providing common network stack features. In the evaluations with latency-sensitive Key-Value Store, NTSocks achieves better latency by up to 24.5× and 1.58× than kernel and RDMA socket, respectively.

CCS CONCEPTS

• **Networks → Data center networks**.

KEYWORDS

Rack-Scale Communication, PCIe Interconnect, Disaggregation, PCIe Non-Transparent Bridging, High-Speed Networks

ACM Reference Format:
Yibo Huang★, Yukai Huang★, Ming Yan★, Jiayu Hu†, Cunming Liang†, Yang Xu★, Wenxiong Zou★, Yiming Zhang★, Rui Zhang★, Chunpu Huang★, Jie Wu★. 2022. An Ultra-Low Latency and Compatible PCIe Interconnect for Rack-scale Communication. In *The 18th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '22), December 6–9, 2022, Roma, Italy*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3555050.3569128

*Jie Wu is the corresponding author.

CoNEXT '22, December 6–9, 2022, Roma, Italy

ACM ISBN 978-1-4503-9508-3/22/12…$15.00
https://doi.org/10.1145/3555050.3569128

1 INTRODUCTION

Emerging resource disaggregation architecture highlights the demand for ultra-low latency and high throughput rack-scale communication behind datacenters for high-quality online services and real-time analysis [56, 21, 19, 25, 59, 23]. The lower the latency, the better [39, 20]. There is a growing trend that, high-density modern computing (e.g., TPU [3]) and storage (e.g., Non-Volatile Memory [62]) hardware are deployed and disaggregated in a rack [19], termed as rack-scale computers [66], which shift the potential bottleneck from computation to network [65, 40]. The latency requirements of rack-scale networks need to be maintained in the range of 3-5us [19]. However, many Ethernet-based studies on rack-scale networks [9, 36] are dependent on either workload which limits generalization, or a centralized controller which compromises performance.

Recent efforts therefore proposed hardware offloading like remote direct memory access (RDMA) [24] or user-space IO like DPDK [30] to significantly reduce latency by kernel-bypassing techniques. Yet, these approaches still rely on heavily layered protocol stacks, and the translation overheads between protocol layers are inevitable. For example, while RDMA over Converged Ethernet (RoCE) [24] allows the network card to directly access the memory of remote machines, one-way data movement in Figure 1(a) requires at least four translations among PCIe bus, RDMA protocol (i.e., IBTA protocol) and UDP. Thus, it is hard to avoid latency overhead. Meanwhile, the Ethernet-based access to PCIe-attached peripherals across machines in a rack further increases the frequency of datapath translations. Additionally, protocol stack hardware offloading schemes also require complex connection/memory resource management within RNICs, which further adds communication overhead. For example, RDMA exploits limited RNIC memory to cache physical-virtual memory mapping tables and connection contexts, resulting in higher tail latency under cache miss [33].

Can we get rid of protocol translation overhead and complex in-NIC resource management for the rack-scale networks to further reduce the communication latency? We argue that using PCIe fabric as the high-speed networking technology is an ideal choice since no translation between PCIe and network protocol is required and complex in-NIC resource management is bypassed (as shown in Figure 1(b)). Specifically, PCIe non-transparent bridge (NTB) as a special device makes it possible to interconnect independent machines to the same PCIe fabric like a bridge [43, 53], although PCIe was originally used to connect various PCIe devices to a CPU-centric computer system [46]. Due to the well-known inefficiency

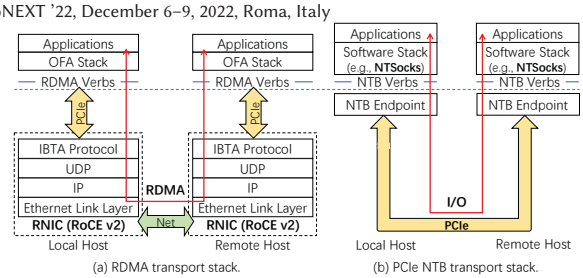


Figure 1: PCIe fabric avoids the translation between PCIe protocol and network protocol, compared to RDMA.

(e.g., high system call overhead) [30, 37] of the kernel-space implementation (e.g., *ntb_hw_intel*) [6], this paper focuses only on user-space NTB (e.g., *DPDK polling mode driver*) [51] to bypass the kernel's complexity. By mapping the shared memory space between independent systems via reliable transaction layer protocol (TLP), NTB allows direct access to the memory of remote machines with extremely low latency (e.g., 2.3~5.6× speedups than RDMA in Figure 3) and high bandwidth (close to the PCIe bandwidth limit).

However, since PCIe NTB is originally designed for device communication (e.g., device sharing [43]) across independent PCIe domains, there are three critical challenges when it is actually used for rack-scale systems.

First, there is an **abstraction mismatch** between the interfaces provided by native NTB (i.e., user-space NTB) and that desired by applications. Unlike Ethernet-based TCP/IP, native NTB lacks general-purpose network function abstraction. It is challenging to craft expected abstractions to *provide compatibility while preserving the performance benefits of native NTB.*

Second, native NTB **lacks a scalable dataplane** for concurrent workloads. This is because the original drivers default to have exclusive control over an NTB device [43]. This calls for a new dataplane design to *efficiently multiplex NTB transport with multi-core scalability.*

Third, the current NTB architecture **lacks performance isolation** between applications. As applications vary in traffic patterns, NTB resource multiplexing is certainly not immune to Head-Of-Line (HOL) blocking and load imbalance between different multiplexing units.

In this paper, we present *NTSocks*, the first user-space in-rack interconnect over PCIe fabric that provides *ultra-low latency, compatible, scalable, and isolated network functionalities for rack-scale systems.* The core of NTSocks is *an indirection proxy* (i.e., data plane) running on each machine to virtualize NTB transport resources (e.g., shared remote memory) within the same PCIe fabric in a lightweight manner, and *a separate software monitor* (i.e., control plane) to enable an Ethernet-like control path. The separate architecture at end host removes the control path (e.g., address, routing) from the data path (e.g., data traffic) to preserve NTB performance benefits.

Based on this architecture, we further propose three key approaches to better balance multiple targets, as follows:

- *Generic* and *high-performance* socket-like abstraction with common network functions by using: i) lock-free ringbuffers over NTB memory via *Remote Write* primitive (§ 4.1), ii) transparent zero-copy support (§ 4.5), iii) and adaptive receiver-driven flow control for preventing message overflow (§ 4.4).
- Dataplane *core-partition model* (§ 4.2), which allocates cores for each partition on demand (e.g., one core for multiple partitions), to trade-off between *multi-core scalability* and *CPU efficiency*. *Partition* is a new abstraction for scalability on multi-core machines that divides the limited NTB-enabled shared memory into multiple parallel units, and each unit (i.e., *a partition*) is *core-driven* and multiplexed by a set of connections.
- *Hierarchical performance isolation mechanism* on top of *partition* (§ 4.3) with: i) a per-connection message slicing for eliminating head-of-line (HOL) blocking among *intra-partition* connections, and ii) *inter-partition* load balancing at connection granularity which uses a round-robin connection distribution based on workloads (e.g., number of per-*partition* active connections).

By building NTSocks on DPDK NTB Polling Mode Driver (PMD), we find that NTSocks outperforms the prior state-of-the-art network stacks [33, 44] with acceptable overhead while realizing high scalability and isolation (§ 6). For example, NTSocks achieves dramatically better latency by up to 20.4× and 2.3×, and lower tail latency by up to 22.7× and 2.6× than Linux TCP and *libVMA* [44], respectively. We further port typical *Key-Value Store (KVS)*, *Nginx* and *Apache benchmarking tool (ab)* to NTSocks with little or even no modification on them. By benchmarking *KVS* with various YCSB workloads [7], NTSocks achieves better latency by up to 24.5× and 1.58×, compared to TCP Redis and RDMA respectively. For Nginx, NTSocks outperforms Linux TCP by up to 6.7×. We will open source NTSocks at https://github.com/NTSocks/.

2 BACKGROUND AND MOTIVATION

2.1 PCIe Non-Transparent Bridge

The PCIe Non-Transparent Bridge (NTB) is a special type of PCIe bridge device that can connect multiple separate computer systems to the same PCIe fabric like a "bridge". PCIe NTB maps the memory address of the remote host to the address space of the local NTB device through the address translation of the reliable PCIe transaction layer protocol on the hardware and maps the memory address of the local NTB device through memory-mapped I/O (MMIO) to the memory space of the local host. Thus, PCIe NTB allows local application processes to directly access the far memory without the CPU involvement of the remote host, providing ultra-low latency and high throughput close to PCIe bandwidth. The PCIe interconnect is originally used to connect various PCIe devices to a CPU-centric computer system, and maps the device memory and host memory to the same memory address space[35]. And PCIe NTB makes it possible to enable high-performance inter-host communication based on PCIe interconnection. From the perspective of hardware availability, PCIe NTB can be embedded in the CPU processor [57, 63], or can be implemented in the PCIe switch chip [58, 2], allowing pluggable NTB adapters and external cables to setup PCIe interconnection. From the perspective of topology, a PCIe cluster switch integrated with a PCIe NTB chip can interconnect dozens of machines at high speed [43], which is suitable for relatively flat and simple intra-rack communication. Typical network topology with PCIe NTB deployment is shown in Figure 2.

Emerging DPDK NTB PMD (Poll-Mode Driver) [51, 46] exploits several optimizations for performance improvement. For example, Intel DPDK reduces the number of PCIe transactions and increases PCIe bandwidth by disabling interrupts, adopting *write combining* [28], and polling write-back descriptors in host memory [46, 5].

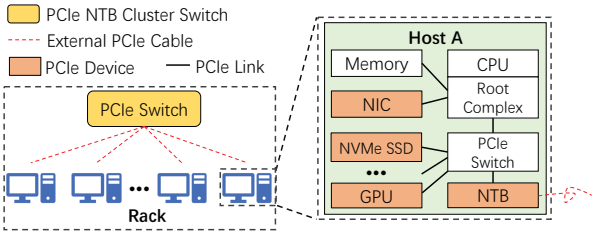


Figure 2: The in-rack network with PCIe NTB fabric.

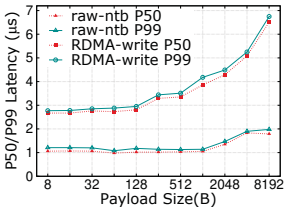


Figure 3: User-space PCIe NTB can achieve better latency than RDMA (Mellanox ConnectX-5 RNIC).

Userspace NTB PMD also leverages Intel Data Direct I/O Technology (DDIO) to accelerate the data movement between the NTB device and CPU cache, which delivers lower latency and higher bandwidth [29, 15]. We focus on poll-enabled userspace NTB in this paper.

2.2 Rack-Scale Network

With the emerging network-attached resource disaggregation architecture [56, 21, 19, 25, 59, 23], rack-scale networks behind datacenters are expected to provide ultra-low latency and high throughput for high-quality services and applications [12, 47, 66]. To cope with the increasing scale-out demand, putting high-density modern hardware into a rack is an inevitable trend [66]. This is evidenced by recent rack-scale computers (e.g., Intel RSA [27], TPU Pods [3], FireBox [1]) or storage (e.g., Facebook Lightning [50], EMC's DSSD [14], Decibel [45]). Recent works [66, 61] further leverage programmable switches to enhance rack-scale systems. The above innovations pose a challenge for faster and more efficient rack-scale communication [11, 36, 40, 9].

Consequently, recent efforts propose hardware offloading like RDMA [24, 20, 13] or user-space IO like DPDK [30, 17] for latency reduction in a kernel-bypassing manner. For example, Pangu [20] introduces production-level intra-podset lossless RDMA to accelerate multi-role oriented cloud storage systems. mTCP [30] provides a user-level TCP/IP library OS with high performance DPDK packet IO over Ethernet devices.

However, these approaches still rely on heavily layered protocol stacks, introducing inevitable translation overhead between protocol layers for rack-scale networks, especially the translation between the PCIe and network protocol. For example, one-way data movement over RoCE requires at least two translations between PCIe bus and RDMA IBTA protocol, and two translations between UDP and IBTA protocol. This unavoidably results in non-negligible latency overhead. Besides, accessing PCIe-attached peripherals in a rack via Ethernet further increases the number of protocol translations and requires the CPU involvement of the remote host. Meanwhile, complex in-NIC resource management required by hardware offloading schemes further adds communication latency. So, how to eliminate the above protocol translation and bypass complex in-NIC resource management for in-rack networks is essential to achieve lower latency.

2.3 PCIe Interconnect for Rack-Scale Network

Opportunity: PCIe NTB. Since no translation between PCIe protocol and network protocol is required, NTB-enabled lightweight PCIe fabric can provide ultra-low latency and high throughput close to PCIe bandwidth, and is an ideal high-speed intra-rack network technology compared to RDMA. Based on PCIe protocol, PCIe NTB fabric naturally does not require translation between PCIe and network protocol. The PCIe protocol consists of a physical layer, data link layer, and transaction layer from the bottom up, which is a data communication based on the Transaction Layer Packet (TLP). By providing reliable transmission and QoS (Quality of Service, QoS) mechanisms through PCIe transaction layer and data link layer protocol, PCIe NTB can ensure that the one-sided read and write operations of each data packet are transactional [42]. Figure 3 shows that PCIe NTB can achieve 2.3~5.6× latency speedup than RDMA, and the one-way transmission delay of PCIe NTB can even reach about *500 ns*. The main latency overhead of PCIe NTB comes from the TLP-based address translation and routing between different PCIe domains within the PCIe NTB chips [49].

Challenges: Although providing ultra-low latency, *native PCIe NTB lacks compatible, scalable, and isolated network functionalities.* Since PCIe NTB was originally designed for device communication across PCIe domains, there are three critical challenges when PCIe NTB is directly used for rack-scale applications, as follows:

Challenge 1: Mismatch in Communication Abstractions. A critical reason why NTB is not friendly and compatible is the communication abstraction mismatch between the native NTB stack and what applications desire. Developers expect an easy-to-use connection-level network function abstraction (like TCP socket or RDMA Queue Pair) to focus on implementing application logic. However, as native NTB has a default assumption that the NTB endpoint device is exclusively controlled by the device driver to achieve secure device sharing between machines [43], it adopts a single-user-oriented programming model and lacks a multi-connection-oriented abstraction, which needs to be tightly coupled with upper-layer applications.

For example, on the control plane, through complex register-based negotiation between NTB endpoints, PCIe NTB exchanges shared memory address metadata mapped by both ends. On the data plane, PCIe NTB only supports a single application process to operate the mapped remote memory and does not support asynchronous event-driven communication like POSIX *epoll*. Complex NTB operations without any connection-friendly interfaces make it difficult to optimize applications, which requires developers to modify applications by carefully choosing different NTB operation options and configuration parameters. Recent work [51, 32] provides a relatively low-level queue design, which is far from compatible connection-oriented abstraction.

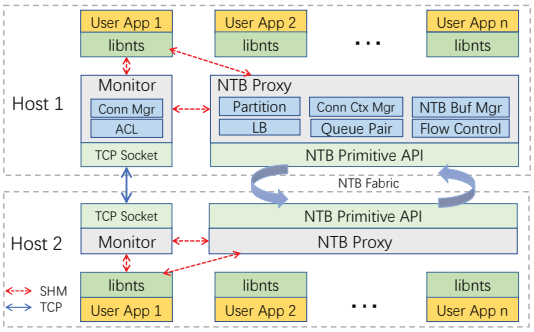


Figure 4: NTSocks architecture.

Challenge 2: Lack of Scalable NTB Dataplane. Native NTB does not provide any dataplane-oriented multi-core scalable mechanisms for efficient NTB transport across various applications. Rack-scale systems usually require the coexistence and combination of multiple applications. However, due to the aforementioned exclusive assumption, user-space NTB only allows intra-process transport sharing with an exclusive tightly coupled style, which has limited sharing scope. Meanwhile, it is also challenging to efficiently utilize NTB memory resources among multiple applications.

Challenge 3: Lack of Performance Isolation. Current NTB architecture lacks multi-tenant performance isolation with efficient NTB resource sharing across applications from a global perspective. Without global management, an NTB-based program can easily lead to the malicious occupation of large amounts of NTB memory, which likely impacts the QoS of other NTB-enabled processes.

As different applications have different traffic patterns, naive NTB queue sharing inevitably introduces performance interference like HOL blocking and load imbalance among multiplexing units, which cannot guarantee fairness across applications. For instance, latency-sensitive flows dominated by small messages can likely be blocked by bandwidth-sensitive flows dominated by large messages, termed HOL blocking. In addition, the load imbalance between different multiplexing units is also challenging when facing concurrent workloads.

3 OVERVIEW OF NTSOCKS

3.1 NTSocks Architecture

To address the challenges, we design NTSocks, a user-space rack-level network architecture over native PCIe NTB with efficient resource sharing. The core of NTSocks is to introduce a user-level indirection at the end host with a software-hardware co-design which transforms the native NTB stack into high-level abstraction and bridges the semantic gap between them.

Figure 4 shows the NTSocks architecture. NTSocks has three main components – user-level network library (*libnts*), dataplane runtime proxy *NTB Proxy (NTP)*, and NTSocks control plane *Monitor (NTM)*. These three components are interconnected through the inter-process SHM channel.

Libnts, located inside the application process, provides socket-like generic APIs for user-level applications with no (or negligible) modification. It provides compatible and high-performance communication abstractions via three methods: a connection-level abstraction that contains per-socket lock-free TX/RX SHM queues

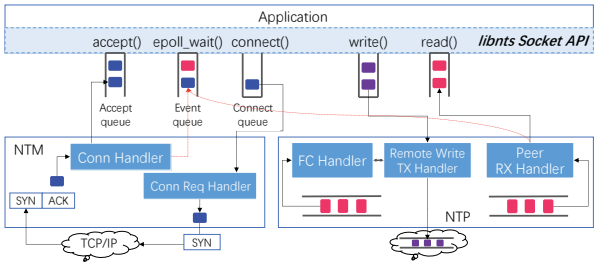


Figure 5: A NTSocks-enabled data transfer example.

and pools (§ 4.1), a packet-level module that performs message slicing and merging for fairness (§ 4.3), and an NTB memory sharing model that optimizes memory copy between application buffers, TX/RX SHM and NTB buffers and provides three interface modes of *origin-nts*, *shm-nts*, and *direct-nts* for different cases (§ 4.5). *Libnts* coordinates with *NTP* to realize fast data-path operations like *read()/write()*.

NTP runs a single instance on each host and works with all *libnts*-based applications on the same host to provide NTB transport networking from a global perspective. As a dataplane runtime, *NTP* performs efficient, scalable, and isolated NTB resource sharing via several modules: a connection manager (*Conn Ctx Mgr*) that efficiently caches the connection contexts (e.g., TX/RX SHM *Queue Pairs* (QP)) and packet routing under the guidance of *NTM* (§ 4.1), an NTB buffer managing module (*NTB Buf Mgr*) that organizes limited NTB memory as lock-free ringbuffers and provides enqueue and dequeue operations with native NTB primitives for efficiency (§ 4.1), a *Partition* abstraction that binds dedicated core-tied TX/RX worker threads to each NTB ringbuffer and performs packet forwarding of a group of connections for multicore scalability (§ 4.2), a load balancing module (*LB*) that realizes an inter-partition connection distribution for performance isolation (§ 4.3), and a receiver-driven adaptive *Flow Control* that enables reliable connection-level transmission (§ 4.4). *NTP* implements the dataplane resource policies (e.g., QoS) by controlling the TX/RX SHM queues between applications and *NTP*.

NTM also runs as a single instance on each host and performs control-plane actions for all *libnts*-based applications. It talks to peer-side *NTM* with TCP socket to exchange link resource metadata (e.g., link status, link selection and connection establishment (handshake) ,and disconnection (wave). It globally manages user-level port allocation for each socket. By treating PCIe NTB fabric as first-order in-rack communication, *NTM* leverages PCIe NTB message registers [51] to negotiate NTB metadata such as Partition mode and the number of Partitions. If the PCIe link is not available, in-rack communication will fall back to the TCP link with the assistance of NTM. *NTM* also works with *libnts* and *NTP* via SHM to complete handshake (e.g.,*connect(), accept()*) or wave (e.g., *shutdown(), close()*) by creating or destroying connection-level entries like SHM QP.

We discuss the thread model of NTSocks in detail in § 5.1.

4 NTSOCKS DESIGN

We have proposed overall NTSocks architecture with microkernel-style global resource management in the previous section. In this section, we demonstrate critical designs on the NTSocks data path

(i.e., *libnts* and *NTP*) to address design trade-offs for compatibility, multi-threaded scalability, and performance isolation.

4.1 Generic Communication Abstraction

To solve the issue of mismatched communication abstractions, we aim to realize general-purpose programming abstractions while maintaining high performance close to bare-meta NTB. Therefore, we combine multiple datapath optimizations to design a flexible socket-like non-intrusive interface (Figure 5).

4.1.1 Connection-level Abstraction.

Since *NTSocks* intercepts every Linux socket call via *libnts*, translates, and forwards them to NTB devices via *NTP*, it is essential to have an efficient SHM-based connection-level abstraction between *libnts* and *NTP* that provides compatibility while preserving high NTB performance. This section presents the lock-free design of such connection-level abstraction (i.e., per-connection TX/RX SHM Queues and Pools).

Lock-free Connection-oriented TX/RX SHM Queue Pair. When establishing a connection, a unique corresponding TX/RX SHM queue pair between *NTP* and *libnts* will be created. As a significant design principle is to avoid the use of locks on the data path, basic *push()*/*pop()* operations on the SHM IO queues are implemented in a lock-free manner. For throughput-sensitive scenarios, we also realize lock-free batch operations (e.g., *bulk_push()*, *bulk_pop()*) that reduce the number of atomic operations on SHM queues to improve throughput. Users can customize the *batch size* depending on the application's demands.

Per-connection TX/RX SHM Pools. The naive implementation of one SHM IO queue is to copy data from the input buffer to the SHM buffer when *enqueue* and copy data from the SHM buffer to the output buffer when *dequeue*. This brings intolerable performance reduction because of the frequent *push()*/*pop()* operations between *libnts* and *NTP*. To this end, we realize lock-free TX/RX SHM pools for each connection to remove redundant data copies. Specifically, when *enqueue*, the input buffer is allocated from the SHM pool, and the corresponding offset index is pushed to the SHM queue. When *dequeue*, the poped offset index is mapped to the corresponding SHM buffer as the output buffer. After the forwarding process is completed, the allocated SHM buffer will be recycled to the SHM Pool.

4.1.2 Shared Lock-free NTB Ringbuffer.

NTB Ringbuffer with efficient NTB verbs. PCIe NTB allows the local CPU to directly access the mapped remote memory with one-sided `Remote Write` and `Remote Read` verbs. We observe that PCIe `Remote Read` is much more expensive than `Remote Write` since `read`/`write` remote system's memory is through the PCI bus [51]. We also find that *write-combining* achieves higher throughput than *write-back*, since write-combining combines multiple TLPs (Transaction Layer Packets) into one PCIe transaction and reduces the number of PCIe memory write transactions [51, 42]. Thus, the above observations of NTB verbs motivate us to leverage *Remote Write with write-combining* and *Local Read* to design a lock-free ringbuffer over NTB memory, to preserve the ideal ultra-low latency and high throughput of data forwarding in *NTP*.

Mode	50% RTT	99% RTT	Request Rate
SP2C	2.66 µs	2.88 µs	5.8 Mrps
SP1C	3.96 µs	7.29 µs	6.6 Mrps

Table 1: NTSocks' 32-Byte message performance with SP2C and SP1C Partition mode. Note that *Mrps* indicates million requests per second.

We organize the limited NTB memory into a ringbuffer with 64-bit *write_index*/*read_index* where *Remote Write* and *Local Read* are exploited to perform push-like TX and pop-like RX operations, respectively. To be more specific, for the TX operation of localhost on the remote NTB ringbuffer, the *TX worker thread* in *NTP* forwards the message from *per-connection TX SHM queue* to the remote NTB buffer via *Remote Write* and updates local *write_index*. For the RX operation of localhost on the local NTB ringbuffer, the *RX worker thread* in *NTP* polls the header metadata *msg_len* of the element pointed by the current *read_index* to find the next message, forwards the next message to the corresponding *RX SHM queue*, and updates the local *read_index*.

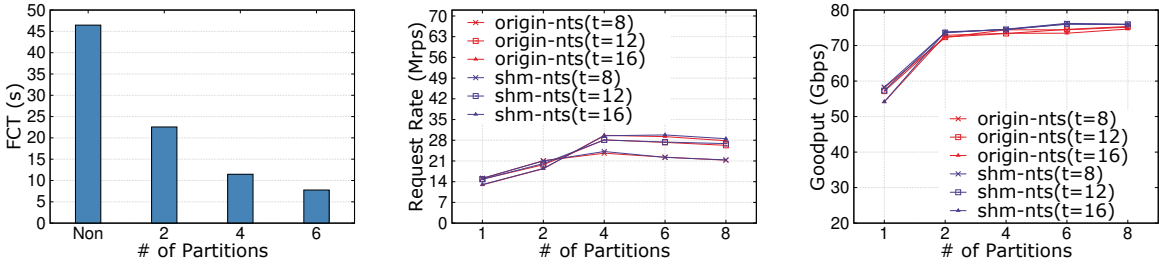
To tell whether the remote NTB ringbuffer is full, the sender exploits a local shadow *read_index* to track the remote *read_index*. Specifically, before the sender performs message TX each time, it uses the local next *write_index* and shadow *read_index* to estimate the proportion *pending_rate* of pending messages. Once the *pending_rate* is greater than 1/2, the sender will mark the metadata of the next message to request synchronization of the remote *read_index*. For the receiver, whenever finding the above mark in the next message, it will actively update the local latest *read_index* to peer-side shadow *read_index* via *Remote Write*.

To keep the consistency between per-message payload and metadata, the *TX worker* in *NTP* writes the payload before metadata as the per-message metadata is 8 bytes and an 8-byte *MOV* instruction behind NTB *Remote Write* is atomic.

4.2 Partition Abstraction

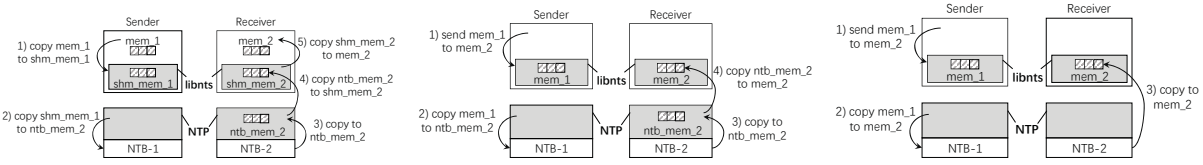
Realizing scalable NTB resource sharing is crucial because of the limited NTB memory and the end of *Dennard Scaling*. However, our initial design of the NTSocks dataplane was to organize the entire NTB memory into a globally unique shared ringbuffer. We design a scalability verification experiment between two NTB-connected servers: The sender concurrently sends 12 fixed-size pressure flows to the receiver, and we calculate the average flow completion time (FCT) based on the measured all FCTs. Figure 6(a) shows that the initial design with *Non-Partition* cannot scale well.

To unlock the multicore scale-out performance of the dataplane, NTSocks further employs *Partition* abstraction. The core of *Partition* is that the limited NTB memory is split into multiple parallel units, and each unit is bound to dedicated TX/RX CPU cores to be responsible for forwarding packets for a group of connections. The result in Figure 6(a) shows that as the number of partitions increases, the average FCT decreases nearly linearly until the number of Partitions in NTSocks increases to 6. In addition, to verify the impact of the number of Partitions on the request rate in small message cases and the application throughput (i.e., *Goodput*) in



(a) Flow completion time. (b) Request rate for 32-Byte messages. (c) Throughput for 32KB messages.

Figure 6: With the *Partition* mechanism, NTSocks can achieve high data-plane parallelism under concurrent CPU core-bound flows. Note that *t indicates the number of core-bound threads.*



(a) The execution flow of *origin-nts*. (b) The execution flow of *shm-nts*. (c) The execution flow of *direct-nts*.

Figure 7: By optimizing memory copy between application buffers, RX/TX SHM and NTB buffers, three data transfer modes (i.e., *origin-nts, shm-nts* and *direct-nts*) are designed to meet the needs of various scenarios. Different modes have different execution flows when sending *mem_1* to *mem_2*.

large message cases, we adjust the number of Partitions and concurrently send 32-Byte and 32KB messages from the sender to the receiver, respectively. Request rate and application throughput are measured in a real-time manner. The results in Figure 6(b) and 6(c) demonstrate that NTSocks with proper *Partition* settings can easily achieve saturated request rate and throughput along with better multi-core scalability under concurrent load.

Improve CPU efficiency. The above *Partition* design needs to use two dedicated cores as TX/RX workers respectively, called Single Partition with 2 Cores (*SP2C*). Although *SP2C* mode friendly ensures ultra-low latency (*<5us*) and tail latency for latency-sensitive small-message cases, it sacrifices CPU efficiency. So, we further propose a Single Partition with 1 Core (*SP1C*) mode. *SP1C* mode only requires one dedicated core to alternately do TX/RX operations. We evaluate the performance of the two modes under concurrent pressure flows. Table 1 shows that compared to *SP2C*, *SP1C* mode introduces acceptable latency overhead in 50% and 99% tail latency. This is because the dataplane worker thread in *SP1C* mode needs to switch between data packet TX and RX tasks. Table 1 also shows that *SP1C* mode even achieves a higher small-message request rate than *SP2C* due to better CPU locality. We further find that, just like SP2C, *SP1C* mode can also saturate the PCIe NTB bandwidth as the message size increases. In practice, users can choose *SP2C* for latency-sensitive cases while *SP1C* for throughput-sensitive cases. Without the special declaration, the paper uses *SP2C* mode by default.

4.3 Performance Isolation

Next, for performance isolation across applications, we leverage a message slicing and merging design for *Intra-Partition* fairness while a *Round-Robin* connection distribution strategy for *Inter-Partition* load balancing.

Intra-partition fairness. A group of connections shares the same *Partition* and corresponding *TX/RX workers*, which means that latency-sensitive flows with small messages are likely blocked by BW-sensitive flows with large messages. To this end, the *TX worker* is required to forward messages from the per-connection *TX SHM queue* with a limited batch size during each round of polling. Meanwhile, a message slicing and merging are essential to split a large message into a set of fixed-size (*mtu size*) packets, which can achieve fairness between hybrid small-message and large-message flows. The question is, where does *NTSocks* perform the slicing/merging? To prevent the message slicing/merging from affecting the TX/RX efficiency of *NTP*, we place it within the *libnts*-enabled application thread. Furthermore, we extensively verify and analyze the fairness between small and large flows reusing the same *Partition* in the experiments in § 6.4. And it is proved that the head-of-line blocking issue within one *Partition* can be effectively eliminated by using connection-level message slicing mechanism.

Inter-partition load balancing. With multiple *Partitions* in *NTSocks*, which *Partition* should we distribute the incoming new connection to? Load imbalance between *Partitions* will impact the overall quality of service (QoS) and reduce the efficiency of NTB transport. Closely coordinated with intra-partition message slicing and merging, a *Round-Robin* connection distribution can efficiently balance inter-partition load.

4.4 Receiver-Driven Flow Control

To make connection-level transmission more reliable, *NTSocks* employs *receiver-driven adaptive flow control*. This aims to avoid the overflow of the per-connection RX SHM queue. Although NTB is a lossless fabric, we cannot guarantee that the connection-level SHM receive queue is lossless. During *NTP* on the receiver side receives/distributes data packets, once the available length of SHM receive queue triggers the congestion threshold, the congestion signal will be generated and transmitted from receiver to sender

through one dedicated control ringbuffer based on NTB memory. Then sender can adaptively adjust the send rate on the *TX SHM queue* based on the current congestion degree.

4.5 Zero Copy

From a high-level perspective, the SHM buffer employed by non-intrusive *libnts* (called *origin-nts*) acts as an indirection layer between application buffers and NTB buffer. To meet the needs of various scenarios, we realize three modes of *origin-nts*, *shm-nts*, and *direct-nts* by optimizing memory copy between application buffers, SHM buffers, and NTB buffers. In particular, *origin-nts* realizes non-intrusive interfaces, *direct-nts* intrusively provides zero-copy support, and *shm-nts* is a compromise between them. The detailed execution flows of a data transfer over different modes are shown in Figure 7.

5 NTSOCKS IMPLEMENTATION

We implement NTSocks in C language, including three components: *NTP*, *libnts* and *NTM*. Based on the DPDK NTB polling mode driver, we implement data plane component *NTP* with about 2500 lines of C code (LoCs). The runtime library *libnts* can be directly linked to the application program to provide all POSIX socket-related function calls, and its logic implementation uses about 3700 LoCs. The control plane *NTM* runs on each machine as a daemon process, stores user-defined control plane information, and provides IP port allocation, access control, fault migration, and TCP fallback capabilities. It takes about 4100 LoCs to implement *NTM*. The above three components all rely on a customized general function library *libnts-utils* implemented by about 4000 LoCs, which provides functions such as inter-process SHM communication protocol, SHM pool, hash function, and so on. NTSocks currently supports 64-bit X86 architecture, adapting to other platforms which needn't change lots of code. The entire NTSocks system runs as a user-mode process in the Linux system environment without any changes.

5.1 Thread Model Implementation

Figure 8 shows how *libnts*, *NTM*, and *NTP* interact with each other. There are several critical SHM queues and threads used to realize efficient coordination between the above three components. With runtime library *libnts*, applications realize the control plane and data plane operations with *NTM* and *NTP*, respectively.

For control plane, *NTM* exploits *Local Req Thread* and *Ctrl Queue* to handle *libnts*' requests, e.g., socket allocation *socket()*, port binding *bind()*, backlog initialization *listen()*, active handshake *connect()*, and socket recycling *close()*. *Local Req Thread* also performs custom security policies like an access control list (ACL). *Peer Req Thread* within *NTM* mainly listens to remote connection requests (*SYN*), instructs *NTP* to initialize the corresponding connection context and TX/RX SHM queues (*IO Queue*), and then pushes the client socket to *Accept Queue* where *libnts* calls *accept()* to obtain the socket.

For the data plane, *NTP* contains three critical threads: *TX*, *RX*, and *FC Thread*. For data TX, the application invokes *libnts*' *write()* to push data in *App Buffer* to per-socket *TX IO Queue*. *TX Thread* polls all connections in a round-robin manner, pops packets from per-connection *TX IO Queue*, adds routing data to the per-packet

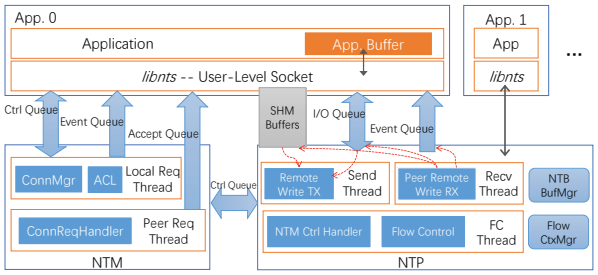


Figure 8: NTSocks thread model.

header, and pushes the packets to remote NTB ringbuffers via native NTB primitives. If the *POLLOUT* event is marked by the connection, a *writable* event will be generated into the *Event Queue* and notify the corresponding socket. For data RX, *RX Thread* pops packets from local NTB ringbuffers by poll, queries the mapped connection context by parsing packets, and pushes the packets to per-socket *RX IO Queue*. If the *POLLIN* event is marked by the connection, a *readable* event will be generated into the *Event Queue* and notify the socket to call *read()*. *FC Thread* also provides a reliable connection-oriented flow control with an NTB control ringbuffer. Figure 5 visually shows the above transfer process.

5.2 Data Path Optimization

Zero Copy. The core of zero copy in NTSocks is to expose the remote NTB buffer to application processes through one *VFIO* driver [51]. There are two key steps required to allow local applications to access the NTB space of the opposite end: First, the application process needs to obtain the *VFIO* file description (FD) of the local NTB device. Second, the application process maps the device-bound physical memory address to the local virtual memory address by calling `mmap()` with the above VFIO FD. The offset and size of remapped memory are determined by the `vfio_region_info` structure of the local NTB device. *NTP* can directly send the obtained FD to the application process through one `Unix socket`.

Data packet batch forwarding. On the critical data path, each dequeue/enqueue operation on the SHM queue pair per connection requires an atomic operation to ensure transaction consistency. NTSocks supports batch forwarding of packets on the queue to improve message throughput. One batch of enqueueing or dequeuing N packets can reduce atomic operations from N to 1. Socket *write()* in *libnts* enqueues the sliced packets into the corresponding TX SHM queue with a specified batch size in the data TX direction. When the CPU core of the corresponding partition within NTP polls the TX SHM queue of the connection, batch-size packets are dequeued with one atomic operation and forwarded to the NTB ringbuffer. Data TX direction has a similar process.

Runtime NTSocks. With microkernel-style NTSocks, one RTT introduces about 500ns IPC latency. In pursuit of the ultimate ultra-low latency, we also support a tightly coupled implementation, termed *Runtime NTSocks*, where the NTP component is tightly integrated into libnts-based application processes to eliminate the IPC overhead. We verify the latency reduction brought by *Runtime NTSocks* in § 6.2.1.

Communication between intra-host application processes. The communication between application processes in the traditional host is usually based on the loop-back traffic transmission of the Ethernet card. It will occupy a large amount of local host PCIe bus bandwidth in the case of high concurrent internal-host traffic load, thereby reducing the peak throughput of PCIe-based communication between PCIe NTB-based hosts [31]. Therefore, NTSocks uses SHM channels instead of Ethernet loopback traffic for intra-host communication to eliminate the performance degradation caused by the above PCIe interference.

6 EVALUATION

This section mainly evaluates the performance and overhead of NTSocks. It is intended to answer the following questions:

- How much performance gain can NTSocks achieve compared to other network protocol stack socket systems? How does the internal mechanism of NTSocks affect the observed performance? (§ 6.2)
- Can NTSocks achieve multi-core performance scalability compared to typical network stacks? (§ 6.3)
- Can NTSocks ensure multi-tenant performance isolation? (§ 6.4)
- Can the communication benefits of NTSocks improve the end-to-end performance of real-world application systems? (§ 6.5)

6.1 Methodology

Testbed setup. The testbed mainly runs on PCIe NTB and RoCE networks. Each machine has two 32-core Intel Xeon Gold 5218 CPUs, 64 GB of RAM, PCIe GEN 3×16 hardware (including a PCIe NTB adapter), and 100Gbps Mellanox CX-5 single-port RNIC. We use the 80Gbps PCIe NTB reference adapter by Intel which is built with Microsemi Switchtec PM853x PFX PCIe GEN-3 chipset [26]. The OS is Ubuntu 18.04.5 LTS with Linux 5.1.3. Unless otherwise specified, NTSocks applies Single Partition with 2 Cores *(SP2C)* mode by default.

The testbed, deployed with PCIe NTB, consists of two servers that are equipped with PCIe NTB adapters and interconnected with a back-to-back topology. Since PCIe NTB is a point-to-point communication, the one-hop latency with the PCIe switch on the data plane could be negligible [43, 42]. So, the latency overhead of one-hop PCIe topology based on PCIe NTB switches is almost the same as that of a back-to-back PCIe connection.

Schemes compared. The comparison of NTSocks, Raw NTB and compatible RDMA sockets (such as *LibVMA* [44]) shows that NTSocks works best. It is verified that NTSocks can provide a virtualized PCIe NTB network for multiple upper-layer applications with minimal performance loss. We also compare the performance of NTSocks with the widely deployed Linux kernel socket (Linux Socket) and eRPC.

6.2 Baseline Micro-benchmark

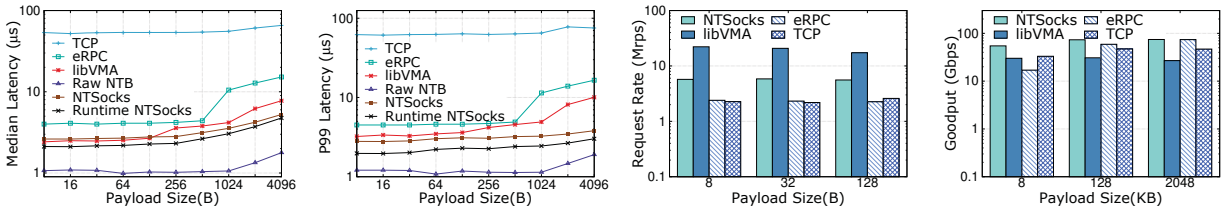
We focus on two basic performance metrics for micro-benchmark: latency and throughput. For communications that support network socket abstraction (i.e., *NTSocks*, *libVMA* and *Linux TCP*), we implement a unified benchmark platform. The platform starts to collect runtime latency/throughput and aggregate them after a sufficient number of warm-up data transmissions. Non-intrusive communication library with POSIX socket support can be easily used to run the benchmark by loading the dynamic link library through the system variable *LD_PRELOAD*. We use *ib_write_lat* and *ib_write_bw* provided by *perftest* to test the latency and throughput of one-side RDMA Write with poll mode. We implement the benchmark tool *ntb-bench-tool* based on raw NTB poll-based transmission, which imitates the evaluation process for one-side RDMA in *perftest*. We will show inter-host communication performance in this section.

6.2.1 Latency. We build a data transmission program similar to *Ping-Pong* to measure and record the round-trip transmission latency of different message sizes (i.e., 8B~4KB). We focus on median latency and 99% tail latency (P99 latency). To ensure the reliability of the results, we measure the latency 100000 times for each message size.

NTSocks achieves ultra-low communication latency. Figure 9(a) and 9(b) show median latency and P99 latency. Due to the inter-process communication (IPC) and memory copy between *libnts* and *NTP*, the translation between sockets and NTB primitives result in additional latency of fewer than 1.7µs in small messages (<=256 bytes) and of about 2.8µs in larger messages (>256 bytes and <=4KB). The median latency of eRPC is 1.66~3.44× that of NTSocks, and NTSocks achieve better P99 latency than eRPC by up to 3.49×. NTSocks achieves better median and P99 latency by up to 2.3× and 2.6× than RDMA socket *libVMA*, and both of them are non-intrusive network socket libraries. In addition, the latency of NTSocks is an order of magnitude lower than that of Linux Socket, because the kernel-mode protocol stack packet processing is inefficient [30, 37], which is not friendly to latency. Noted that median and P99 tail latency have similar behaviors and root causes. Due to eliminating 4 inter-process communications between App and NTP in one round trip, the latency of Runtime NTSocks is about 500ns lower than that of NTSocks.

6.2.2 Throughput. The throughput performance is reflected in the round-trip request rate for small messages (i.e., *Request Rate*) and the application-level throughput for large messages (i.e., *Goodput*). For the request rate, we use the Ping-Pong data transmission program with 10,000 outstanding requests (*Outstanding Requests*) to measure the amount of processed message per second (i.e., *Million Requests Per Second, (Mrps)*). For the application throughput under large message cases, we use the one-way bandwidth of data transmission from the client to the server. To ensure the stability of the measurement, the throughput benchmark of each message size runs for 20 seconds.

NTsocks achieves high and stable throughput. Figure 9(c) and 9(d) show the *Request Rate* of small messages and the *Goodput* of large messages. In the case of a small message request rate, NTSocks is about 2~3× higher than Linux TCP and eRPC. This is because the protocol stack that NTSocks relies on is more lightweight, which eliminates the translation overhead between the PCIe protocol and the network protocol, and bypasses the system kernel. Because RDMA socket *libVMA* uses small message batch processing to greatly increase throughput [37, 44], *libVMA* has a higher request rate than NTSocks. However, the small-message batch processing optimization of *libVMA* hardly brings throughput gains with gradually increasing message sizes, and the overhead is mainly caused by



(a) Median latency. (b) P99 latency. (c) Request rate for small message. (d) Goodput for large message.

Figure 9: Performance comparison between NTSocks and other typical communication stacks.

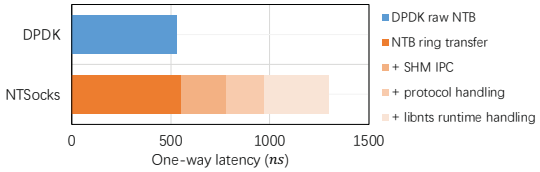


Figure 10: The breakdown of one-way latency (8B payload size) for NTSocks (*origin-nts*).

memory copy and lock contention introduced by the translation of sockets to RDMA primitives, so *libVMA* has a higher small message request rate but lower application-level throughput (i.e., `Gootput`) under large message cases than NTSocks.

Regarding large message throughput, we find that the saturated bandwidth of NTSocks after convergence is only 74.3Gbps. The reason is that the NTB adapter we use is an experimental platform provided by the manufacturer. The design specification of the built-in non-transparent bridge chip particles is 80Gbps. But the design specifications of the released commercial NTB chips are aligned with PCIe bandwidth. We believe that NTSocks can easily approach the theoretical value of PCIe bandwidth under commercial NTB adapters. Despite this, NTSocks has higher bandwidth and reaches saturation earlier than eRPC. When the payload size is 8 KB, the bandwidth of NTSocks is even 3.22× larger than that of eRPC.

Although users usually do not want small messages to saturate the bandwidth and pay more attention to the latency [34], when the message size is small, such as 2 KB, NTSocks can still achieve throughput of more than half of saturated bandwidth. In this case, the limitation of throughput is just using only one partition of *NTP*, and more partitions can be added to solve the throughput bottleneck.

6.2.3 One-way latency breakdown. In order to profile the latency overhead in detail between NTSocks and raw NTB, we implement *echo* benchmark programs and measure the one-way transmission latency by dividing the collected round-trip echo latency by 2. Compared with raw NTB, the overhead of NTSocks mainly includes 4 parts: (1) NTB ring buffer transmission latency in the *NTP* partition, (2) lightweight protocol handling overhead in *NTP*, (3) SHM-based inter-process communication overhead between *NTP* and *libnts* application handling, (4) *libnts* runtime handling overhead. We directly use PCIe NTB WRITE primitive to build an *echo* program over the DPDK PMD driver for raw NTB, while using *origin-nts* to build an *echo* program for NTSocks.

The overhead of the NTSocks components is acceptable compared to raw NTB. Figure 10 shows the breakdown of one-way latency. Compared with the raw DPDK NTB, NTSocks partitioned NTB ring buffer transmission only adds a negligible latency overhead, which mainly comes from the overhead of maintaining the *write_index* and *read_index*. However, the internal protocol handling overhead of *NTP*, the inter-process communication between *NTP* and *libnts*, and the runtime handling of *libnts* do introduce some latency overhead. Protocol handling overhead mainly includes the encapsulation and analysis of the header of the data packet. The inter-process communication overhead is mainly reflected in the atomic operation of entering and leaving the per-connection SHM queues and the inherent synchronous transmission overhead of the SHM. The runtime handling overhead of *libnts* mainly includes allocating SHM from the SHM pool and copying data from the application buffer to the allocated SHM during *write()*, reclaiming SHM and copying data from SHM to the Application buffer during *read()*. Although the above overheads are acceptable, we believe that these overheads can be further reduced in the future by offloading the *NTP* component into hardware like the ARM cores of Mellanox SmartNIC.

6.3 Multiple Thread Scalability

Multi-thread performance scalability is a significant advantage for NTSocks. This mainly benefits from the Partition mechanism and lock-free data path in NTSocks to unlock the power of multi-core scale-out. To verify the multi-core scalability of NTSocks, we use multiple concurrent clients bound to the threads to send 128KB message requests to the same server, and a background thread is used to aggregate the total throughput in a real-time manner according to the throughput of each connection on the server side. The NTSocks data plane sets 4 partitions for this benchmark. Since the PCIe NTB adapter in the experimental environment is an 80Gbps design specification and the RoCEv2 ConnectX-5 network card is 100Gbps, we normalize the aggregate throughput by defining bandwidth *saturation rate* (i.e., measured aggregate throughput divided by hardware bandwidth capacity) to unify performance metrics and make an intuitive comparison.

As shown in Figure 11, regarding the aggregate bandwidth saturation rate under various concurrent loads, NTSocks is the best (i.e., more than 90%), followed by Linux TCP (i.e., more than 78%), and RDMA socket *libVMA* is the worst (i.e., only up to 22%). The aggregate throughput of NTSocks is very stable with the increasing number of concurrent threads, and the bandwidth saturation rate is between 90.4% and 93.6%. Due to lock contention on shared NIC queues [37, 44], the aggregate throughput of *libVMA* shows

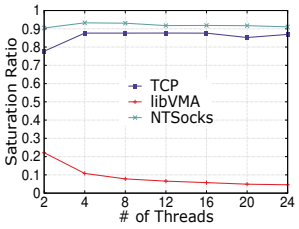


Figure 11: Multiple thread scalability.

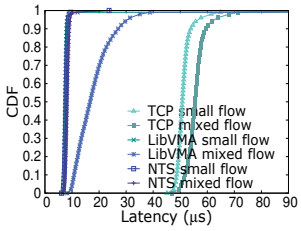


Figure 12: Performance isolation results between NTSocks and kernel TCP.

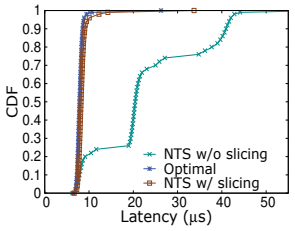


Figure 13: NTSocks with message slicing can eliminate the HOL issue.

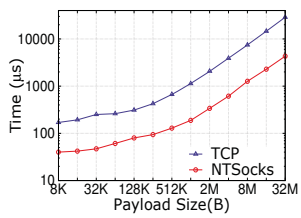
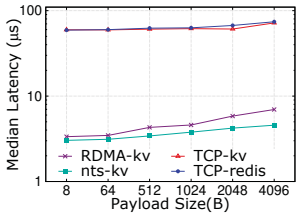
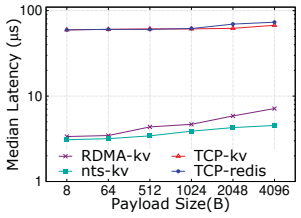


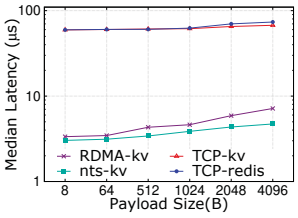
Figure 14: The end-to-end file transfer performance of the Nginx HTTP server.



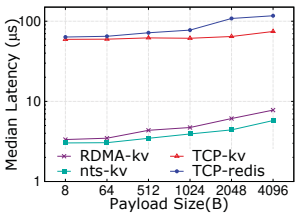
(a) YCSB-R100



(b) YCSB-R95I5



(c) YCSB-R90U10



(d) YCSB-R50U50

Figure 15: End-to-end median latency of key-value stores with various YCSB workloads.

a nearly linear decrease and then tends to be stable as the number of concurrent threads increases. For example, compared to 2 threads, the throughput of *libVMA* with 4 threads is reduced to *1/2*. Meanwhile, the bandwidth saturation rate of Linux Socket is between 77.6% and 87.6%. Although Linux Socket has improved throughput by using TCP Segmentation Offload (TSO) and TCP window scaling optimizations, its bandwidth saturation rate is still lower than NTSocks. This is mainly due to the high overhead of system calls and multiple memory copies induced by the heavily layered protocol stack in Linux TCP [30].

6.4 Performance Isolation

NTSocks ensures the isolated performance of multiple applications or tenants with different traffic patterns by co-designing inter/intra-partition hierarchical performance isolation mechanism. To evaluate the effect of multi-tenant performance isolation in NTSocks, we use the cumulative distribution of the transmission latency of a small flow that continuously sends 2KB messages as the observation target, while using one large flow that continuously sends 64KB messages as the background traffic. Regarding inter-partition fairness in NTSocks, we run the small and large flows in different partitions and compare the performance isolation effect of NTSocks with Linux TCP and libVMA. Regarding intra-partition performance isolation, we particularly run the small and large flows in the same partition.

NTSocks achieves the best inter-partition fairness between large and small flows, and has a one-order-of-magnitude lower latency jitters than Linux TCP in Figure 12. NTSocks achieves almost the same latency distribution in the case of small flow with background traffic and no background traffic. The latency gap between the two is about 0.36µs (0.072~0.535µs), and the P99 tail latency gap is only 0.53µs. Linux TCP shows the worst performance isolation effect, and the latency gap in the two experimental scenarios is about 4.83µs (0.759~7.216µs), and the P99 tail latency gap is as high as 7.216µs. This is mainly due to the well-known inefficiency of kernel TCP protocol stack in the data path. Meanwhile, Figure 13 shows that **NTSocks with message slicing mechanism achieves near-optimal intra-partition performance isolation between small and large flows**. NTSocks without slicing encounters dramatically higher tail latency due to HOL blocking issue. These gains benefit from the hierarchical isolation mechanism proposed by NTSocks(§ 6.3), which eliminates the head-of-line (HOL) blocking of NTB link multiplexing through the inter/intra-partition co-design, and achieves better load balancing among partitions.

6.5 Real-world Applications

In this section, we evaluate the performance improvement of real-world applications, to demonstrate the high performance, flexibility, and ease of use for NTSocks. We build a Key-Value Store (*KVS*) and Nginx with NTSocks, representative small-message, and large-message cases.

6.5.1 Key-value Store. Key-Value Store is a typical time-sensitive application scenario that focuses on small messages. We apply the prevalent MurmurHash algorithm to build a general key-value store system based on the POSIX socket, which supports high-performance `Read` (R), `Update` (U), `Insert` (I), and `delete` operations. We use *libnts* and *libVMA* to implement code non-intrusive NTSocks (*nts-kv*) and RDMA socket (*RDMA-kv*) support in a dynamic link library by Linux environment variable *LD_PRELOAD*. We evaluate *KVS* using YCSB workloads [7] including `R100`, `R95I5`, `R90U10`, and `R50U50`.

NTsocks achieves the best performance improvement for Key-Value Store. Figure 15 shows the median latency with increasing key-value size under different YCSB workloads. We find that both NTSocks (*nts-kv*) and *libVMA* (*RDMA-kv*) have an order

of magnitude better end-to-end latency than Linux TCP. Meanwhile, even if *libVMA* has used the performance advantages of RDMA to accelerate key-value storage, NTSocks still achieves a 57.6% lower median latency than *libVMA*. NTSocks is better than the RDMA socket *libVMA* library and has powerful generalization ability even under different traffic patterns. Particularly, Figure 15(a) shows that compared to *libVMA*, NTSocks has a 10.4%∼52.7% lower median latency under the YCSB `R100` workload. In Figure 15(b), NTSocks achieves 8.7%∼57.6% lower median latency than *libVMA* under YCSB `R95I5` workload. Also, Figure 15(c) and 15(d) demonstrate that NTSocks achieves 10.3%∼51.7% and 10.2%∼38.4% lower median latency than *libVMA* under YCSB `R90U10` and `R50U50` workloads, respectively. This is because NTSocks minimizes the translation overhead from connection-level socket abstraction to memory-semantic PCIe NTB primitives by employing multiple lock-free and efficient data path designs (e.g., shared lock-free NTB ringbuffer and Partition abstraction).

6.5.2 Nginx HTTP Server and ab. Nginx is an open-source and high-performance event-driven HTTP server. We use *direct-nts* to replace the corresponding interface of Nginx that achieves zero-copy and low-overhead. Only 150 LoCs are required to build Nginx with NTSocks *direct-nts*, which also demonstrates the compatibility and flexibility of NTSocks' zero-copy interfaces. We further employ the well-known Apache benchmarking tool [18] to generate the HTTP file requests. LibVMA does not work with unmodified Nginx due to fork. We compare the end-to-end Nginx HTTP file transfer performance with NTSocks and TCP.

Due to the zero-copy design of NTSocks, the file transfer latency of Linux TCP is 3.88∼6.69× that of NTSocks when the payload size is between 8KB and 32MB in Figure 14. As the file size increases, the memory copy overhead increases, and the file transmission latency gap between NTSocks and Linux TCP gradually increases. When the payload size is 32MB, Nginx over NTSocks has 24.5ms less latency than TCP. **In a word, NTSocks also brings significant performance gains in large-message cases**.

7 DISCUSSION

Abstraction for rack-scale communication. Although memory semantic is a candidate abstraction for rack-scale systems, NTSocks focuses on socket abstraction for two reasons. First, socket abstraction is the most widely used communication primitive for modern rack-scale applications. Much existing work aims at improving compatibility and socket performance by building Socket abstraction over user-space network stack (e.g., f-stack [4], Seastar [55] and mTCP [30]) or commodity RDMA hardware (e.g., libVMA [44] and SocksDirect [37]). Second, regarding memory-semantic use cases (e.g., distributed shared memory [61]), PCIe NTB inherently supports memory-semantic communication, so it is easier to extend NTSocks to enable lightweight unified memory address space across multiple hosts. We leave it to future work.

Inter-rack communication. NTSocks aims to utilize ultra-low latency PCIe interconnect to accelerate intra-rack communication scenarios where the entire applications runs in a rack. Alternatively, inter-rack communication scenarios might split the entire application traffic into inter-rack and intra-rack traffic. In order to handle such scenarios, NTSocks would also need to be augmented to scale beyond a rack by potentially exploiting the hybrid Ethernet/PCIe switch [60]. Meanwhile, we found that PCIe NTB memory can be registered by RDMA, which would provide an inter-rack fast path for hybrid RDMA/PCIe deployment.

SmartNIC-offloaded data path. NTSocks is currently just an experimental proof of exploring to use ultra-low latency PCIe interconnect for in-rack networking. With the popularity of SmartNICs [16, 52, 8], software CPU overhead in NTSocks can be eliminated by offloading the data plane components (e.g., Partition scheduling and flow control within *NTP*) to cost-effective ARM cores. We will explore it in future work.

8 RELATED WORK

Advanced PCIe and network systems. Modern advanced PCIe interconnect techniques like PCIe CXL [10], Gen-Z [54] and NTB [43, 53] demonstrate great opportunities for disaggregation-oriented rack-scale communication due to its hardware properties of ultra-low latency, high bandwidth, cache coherency, and remote memory access. With these benefits, network systems design using PCIe interconnect is an important topic. Servers in Aquila [21] use PCIe to connect to Pods for their NIC functionality. RASHPA [41] implements a complex PCIe-based network framework using FPGA and Linux-based software stack, which primarily focuses efficient data acquisition system and cannot saturate the bandwidth like NTSocks does. Although Marlin [60], P-Socket [64] and NTSocks enjoy similar motivations and all propose a PCIe-based Socket abstraction for application compatibility, both Marlin and P-Socket lack the considerations of multi-tenancy performance isolation and multi-core scalability required by modern rack-scale systems. Furthermore, they are based on kernel-space PCIe-based design, which inevitably introduces system calls and interrupts overhead. In comparison, NTSocks focuses on a full user-space PCIe interconnect design for compatible, scalable and isolated rack-scale communication.

PCIe-enabled device sharing and disaggregation. Several works use advanced PCIe interconnect to facilitate resource sharing and disaggregation [42, 22, 38]. SmartIO [42] leverages PCIe NTB for efficient remote IO device sharing at the device driver layer for typical virtualization scenarios, which has a different motivation than NTSocks. Further, SmartIO lacks considerations for compatibility and dataplane multi-core scalability which is essential for general rack-scale systems. Huaicheng et al. [38] propose a PCIe CXL-based memory disaggregation with memory semantics for public clouds to resolve DRAM inefficiency induced by platform-level memory stranding while preserving high performance. In comparison, in addition to supporting memory semantics inherently enabled by PCIe NTB, NTSocks focuses on socket abstraction for wider generality. So, NTSocks is orthogonal and complementary to these works.

Rack-scale communication. As online services require strict Service-Level Objective (SLO), many work on rack-level communication behind data center networks has been studied in academia and industry [66, 9, 47, 48, 40, 45, 11, 36, 59]. These studies achieve microsecond-level latency and million-level throughput by designing network routing and congestion control algorithms [9, 11] or programmable switches [66, 45]. NTSocks is orthogonal to these works because it creatively uses high-speed PCIe NTB interconnect

to provide rack-scale systems with an easy-to-use, fast, scalable, and isolated network architecture. Compared with the works of protocol stack hardware offloading, NTSocks reduces translation between protocol layers by providing a lightweight protocol stack, and further reduces network transmission latency.

9 CONCLUSION

In this paper, we present *NTSocks*, an ultra-low latency and lightweight network stack over advanced PCIe fabric for disaggregation-oriented rack-scale communication. It employs a microkernel-style architecture to enable compatibility, scalability, isolation, and efficient PCIe resource sharing required by rack-scale communication. NTSocks demonstrates that an efficient user-level indirection can achieve close-to-bare-metal NTB performance while addressing three critical challenges when applying advanced PCIe connectivity for rack-scale networks. Evaluations with real-world applications show that *NTSocks* dramatically outperforms the art-of-the-state network stacks.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful suggestions and Chen Sun for shepherding the paper. Also, we would like to sincerely thank Chen Tian and Ang Chen for their valuable comments.

REFERENCES

[1] Krste Asanović. 2014. Firebox: a hardware building block for 2020 warehouse-scale computers.
[2] Broadcom. 2011. Pex8733, pci express gen 3 switch, 32 lanes, 18 ports. https://docs.broadcom.com/docs/12351852. (2011).
[3] Google Cloud. 2018. Tpu pods. https://cloud.google.com/tpu/. (2018).
[4] Tencent Cloud. 2019. High-performance network framework based on dpdk. http://f-stack.org/. (2019).
[5] DPDK Community. 2020. Data plane development kit. https://www.dpdk.org/. (2020).
[6] Linux Kernel Community. 2020. Ntb drivers in linux kernel. https://www.kernel.org/doc/Documentation/ntb.txt. (2020).
[7] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving sys- tems with ycsb. In *Proceedings of the First ACM Symposium on Cloud Computing*, 143–145.
[8] NVIDIA Corporation. 2022. Bluefield smartnic. https://www.nvidia.com/en-us/networking/products/data-processing-unit/. (2022).
[9] Paolo Costa, Hitesh Ballani, Kaveh Razavi, and Ian Kash. 2015. R2c2: a network stack for rack-scale computers. *ACM SIGCOMM Computer Communication Review*, 45, 4, 551–564.
[10] CXL. 2020. Compute express link: the breakthrough cpu-to-device interconnect. https://www.computeexpresslink.org/. (2020).
[11] Alexandros Daglis, Stanko Novaković, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2015. Manycore network interfaces for in-memory rack-scale computing. *ACM SIGARCH Computer Architecture News*, 43, 3S, 567–579.
[12] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Communications of the ACM*, 56, 2, 74–80.
[13] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. Farm: fast remote memory. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, 401–414.
[14] EMC. 2016. Dssd d5. https://www.emc.com/enus/storage/flash/dssd/dssd-d5/index.htm. (2016).
[15] Alireza Farshin, Amir Roozbeh, Gerald Q Maguire Jr, and Dejan Kostić. 2020. Reexamining direct cache access to optimize i/o intensive applications for multi-hundred-gigabit networks. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, 673–689.
[16] Daniel Firestone et al. 2018. Azure accelerated networking: smartnics in the public cloud. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, 51–66.
[17] Linux Foundation. 2020. What is the vector packet processor (vpp). https://fd.io/docs/vpp/master/. (2020).
[18] The Apache Software Foundation. 2020. Ab - apache http server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html. (2020).
[19] Peter X Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network requirements for resource disaggregation. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 249–264.
[20] Yixiao Gao et al. 2021. When cloud storage meets {rdma}. In *18th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 21)*, 519–533.
[21] Dan Gibson et al. 2022. Aquila: a unified, low-latency fabric for datacenter networks. In *19th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 22)*, 1249–1266.
[22] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. 2022. Direct access,{high-performance} memory disaggregation with {directcxl}. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 287–294.
[23] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. 2017. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 649–667.
[24] 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 SIGCOMM Conference*, 202–215.
[25] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. 2022. Clio: a hardware-software co-designed disaggregated memory system. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 417–433.
[26] Microchip Technology Inc. 2019. Microchip switchtec pm853x. https://ww1.microchip.com/downloads/en/DeviceDoc/00002849.pdf. (2019).
[27] Intel. 2017. Intel rack scale design. https://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html. (2017).
[28] Intel. 2020. Intel® 64 and ia-32 architectures optimization reference manual. https://software.intel.com/content/www/us/en/develop/down-load/intel-64-and-ia-32-architectures-optimization-reference-manual.html. (2020).
[29] Intel. 2020. Intel® data direct i/o technology. https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html. (2020).
[30] 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)*, 489–502.
[31] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 2020. A unified architecture for accelerating distributed {dnn} training in heterogeneous gpu/cpu clusters. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, 463–479.
[32] Wu Jingjing and Maslekar Omkar. 2019. Dpdk pmd for ntb. https://static.sched.com/hosted_files/dpdkna2019/35/DKPMDforPCleNon-TransparentBridge.pptx. Intel, (2019).
[33] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter rpcs can be general and fast. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, 1–16.
[34] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. 2019. Freeflow: software-based virtual {rdma} networking for containerized clouds. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, 113–126.
[35] Yohei Kuga, Ryo Nakamura, Takeshi Matsuya, and Yuji Sekiya. 2020. Nettlp: a development platform for pcie devices in software interacting with hardware. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, 141–155.
[36] Sergey Legtchenko, Nicholas Chen, Daniel Cletheroe, Antony Rowstron, Hugh Williams, and Xiaohan Zhao. 2016. Xfabric: a reconfigurable in-rack network for rack-scale computers. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, 15–29.
[37] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. 2019. Socksdirect: datacenter sockets can be fast and compatible. In *Proceedings of the ACM Special Interest Group on Data Communication*, 90–103.
[38] Huaicheng Li et al. 2022. First-generation memory disaggregation for cloud platforms. *arXiv preprint arXiv:2203.00241*.
[39] Yuliang Li et al. 2019. Hpcc: high precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*, 44–58.
[40] Liang Luo, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. 2018. Parameter hub: a rack-scale parameter server for distributed deep neural network training. In *Proceedings of the ACM Symposium on Cloud Computing*, 41–54.
[41] Wassim Mansour, Pablo Fajardo, Nicolas Janvier, et al. 2017. High performance rdma-based daq platform over pcie routable network. *ICALEPCS, Barcelona, Spain*, 8–13.

[42] Jonas Markussen, Lars Bjørlykke Kristiansen, Pål Halvorsen, Halvor Kielland-Gyrud, Håkon Kvale Stensland, and Carsten Griwodz. 2021. Smartio: zero-overhead device sharing through pcie networking. *ACM Transactions on Computer Systems (TOCS)*, 38, 1-2, 1–78.

[43] Jonas Markussen, Lars Bjørlykke Kristiansen, Håkon Kvale Stensland, Friedrich Seifert, Carsten Griwodz, and Pål Halvorsen. 2018. Flexible device sharing in pcie clusters using device lending. In *Proceedings of the 47th International Conference on Parallel Processing Companion*, 1–10.

[44] Mellanox. 2019. Messaging accelerator (vma). Available at https://github.com/mellanox/libvma. (2019).

[45] Mihir Nanavati, Jake Wires, and Andrew Warfield. 2017. Decibel: isolation and sharing in disaggregated rack-scale storage. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, 17–33.

[46] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W Moore. 2018. Understanding pcie performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 327–341.

[47] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2016. The case for rackout: scalable data serving using rack-scale systems. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, 182–195.

[48] Stanko Novakovic, Alexandros Daglis, Dmitrii Ustiugov, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2019. Mitigating load imbalance in distributed data serving with rack-scale memory pooling. *ACM Transactions on Computer Systems (TOCS)*, 36, 2, 1–37.

[49] 2014. Pci express® base specification revision 4.0 version 0.3. https://xdevs.com/doc/Standards/PCI/PCI_Express_Base_4.0_Rev0.3_February19-2014.pdf. (2014).

[50] C. PETERSEN. 2016. Introducing lightning: a flexiblenvme jbof. https://code.facebook.com/posts/989638804458007/introducinglightning-a-flexible-nvme-jbof/. (Mar. 2016).

[51] DPDK Project. 2020. Ntb rawdev driver. https://doc.dpdk.org/guides/rawdevs/ntb.html. (2020).

[52] Yiming Qiu, Jiarong Xing, Kuo-Feng Hsu, Qiao Kang, Ming Liu, Srinivas Narayana, and Ang Chen. 2021. Automated smartnic offloading insights for network functions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 772–787.

[53] Jack Regula. 2004. Using non-transparent bridging in pci express systems. *PLX Technology, Inc*, 31.

[54] Holly Schroth. 2019. Are you ready for gen z in the workplace? *California Management Review*, 61, 3, 5–18.

[55] ScyllaDB. 2019. Seastar: high-performance server-side application framework. http://seastar.io/. (2019).

[56] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. Legoos: a disseminated, distributed {os} for hardware resource disaggregation. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 69–87.

[57] Mark J Sullivan. 2010. Intel xeon processor c5500/c3500 series non-transparent bridge. *Technology@ Intel Magazine*.

[58] PLX Technologies. 2005. Multi-host system and intelligent i/o design with pci express. https://lwn.net/Articles/672752/. (2005).

[59] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. 2020. Disaggregating persistent memory and controlling them remotely: an exploration of passive disaggregated key-value stores. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, 33–48.

[60] Cheng-Chun Tu, Chao-tang Lee, and Tzi-cker Chiueh. 2014. Marlin: a memory-based rack area network. In *2014 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE, 125–135.

[61] Qing Wang, Youyou Lu, Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu. 2021. Concordia: distributed shared memory with in-network cache coherence. In *19th {USENIX} Conference on File and Storage Technologies ({FAST} 21)*, 277–292.

[62] Xingda Wei, Xiating Xie, Rong Chen, Haibo Chen, and Binyu Zang. 2021. Characterizing and optimizing remote persistent memory with rdma and nvm. In *2021 {USENIX} Annual Technical Conference ({USENIX}{ATC} 21)*, 523–536.

[63] Xiangliang Yu. 2016. Ntb: add support for amd pci-express non-transparent bridge. https://lwn.net/Articles/672752/. (2016).

[64] Liuhang Zhang, Rui Hou, Sally A McKee, Jianbo Dong, and Lixin Zhang. 2016. P-socket: optimizing a communication library for a pcie-based intra-rack interconnect. In *Proceedings of the ACM International Conference on Computing Frontiers*, 145–153.

[65] Xiantao Zhang, Xiao Zheng, Zhi Wang, Hang Yang, Yibin Shen, and Xin Long. 2020. High-density multi-tenant bare-metal cloud. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 483–495.

[66] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. 2020. Racksched: a microsecond-scale scheduler for rack-scale computers. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, 1225–1240.