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
High-speed rack-scale network is strongly demanded

- Compute/storage is faster
  - Non-volatile Memory
  - GPU/TPU
- Ultra-low latency
  - 3-5 us\(^1\)
- High throughput
  - Frequent interactions

RDMA
Hardware Offloading

\(^1\) Gao, et al. (OSDI ‘16)
Why is current RDMA insufficient?

RDMA Verbs

RDMA Protocol

RNIC

PCIe

Applications

Local Host

Remote Host

RDMA Verbs

RDMA Protocol

RNIC

Net

PCIe

Applications

At least 1.6us delay for one RTT

• Inevitable protocol translation overhead
  • Conversion between PCIe and Net Packets with different MTU size

• Complex in-NIC resource management
  • Limited NIC cache for RDMA connection context and memory mapping table
Why is current RDMA insufficient?

- Inevitable protocol translation overhead
- Conversion between PCIe and Net Packets with different MTU size
- Complex in-NIC resource management
- Limited NIC cache for RDMA connection context and memory mapping table

How to get rid of the protocol translation overhead and complex in-NIC resource management?

At least 1.6us delay for one RTT
New opportunities with high-speed PCIe interconnect

Advanced PCIe interconnect

- Provide super fast cross-machine accesses directly over lossless PCIe Fabric
- E.g., CXL*, Gen-Z, and PCIe NTB (Non-Transparent Bridge)

✅ Inherently eliminate protocol translation
✅ Bypass complex in-NIC management

* CXL: Compute eXpress Link
New opportunities with high-speed PCIe interconnect

**Advanced PCIe interconnect**

- Ultra-low latency
  - ~500ns one-way latency with PCIe NTB

PCIe NTB: 2.3~5.6X speedup than RDMA

- High bandwidth
  - Can match evolving PCIe bandwidth

- Cache-coherent remote memory access

<table>
<thead>
<tr>
<th>PCIe Generation</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCIe 3.0 x16</td>
<td>128 Gbps</td>
</tr>
<tr>
<td>PCIe 4.0 x16</td>
<td>256 Gbps</td>
</tr>
<tr>
<td>PCIe 5.0 x16</td>
<td>512 Gbps</td>
</tr>
<tr>
<td>PCIe 6.0 x16</td>
<td>1024 Gbps</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
New opportunities with high-speed PCIe interconnect

Advanced PCIe interconnect

- Ultra-low latency
  - ~500ns one-way latency with PCIe NTB

- High bandwidth
  - Can match evolving PCIe bandwidth

- Cache-coherent remote memory access

Rethink the design of **ultra-low latency** and compatible rack-scale communication with advanced PCIe Interconnect

<table>
<thead>
<tr>
<th>PCIe Generation</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCIe 3.0 x16</td>
<td>128 Gbps</td>
</tr>
<tr>
<td>PCIe 4.0 x16</td>
<td>256 Gbps</td>
</tr>
<tr>
<td>PCIe 5.0 x16</td>
<td>512 Gbps</td>
</tr>
<tr>
<td>PCIe 6.0 x16</td>
<td>1024 Gbps</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Vision: PCIe Interconnect for High-speed Rack-scale Network

Enable ultra-low latency and lightweight PCIe interconnect* capabilities for rack-scale communication.

The in-rack network architecture with PCIe NTB fabric

* We use PCIe NTB in this paper.
Strawman solution: native PCIe Non-Transparent Bridge

The native PCIe NTB lacks transparency support due to the low-level interfaces.

1. Control Path
   - Setup global NTB Context
   - Post memory-mapped IO (MMIO) requests

2. Data Path
   - NTB endpoints process MMIO requests using address routing
   - NTB App directly accesses memory
Our Work: NTssocks with three key challenges

- A lightweight end-host network stack over PCIe Interconnect that achieves transparency while preserving high performance.

- However, this is hard in general due to the following three challenges:

  C1: Mismatch in communication abstraction
  C2: How to enable scalable NTB dataplane?
  C3: How to ensure performance isolation?
Agenda

• Motivation

• NT Sokcs Design

• Implementation and Evaluation

• Summary
We focus on user-space PCIe NTB to bypass the kernel's complexity.

* NTB App 1
* NTB App 2

**Host**

- NTB App 1
  - Socket API
  - NTSocks
  - User-level NTB* library
  - PCIe NTB Endpoint

- NTB App 2
  - Socket API
  - NTSocks
  - User-level NTB* library
  - PCIe NTB Endpoint

**Runtime Library**

- NTSocks Library (libnts)

**Applications**

- NTSocks Monitor (NTM)
- NTSocks Proxy (NTP)

**For Control Plane**

- NTM

**For Data Plane**

- NTP
Challenge #1: Mismatch in communication abstractions

- **Expect**: NTB APP
  - ?
  - NTB library
  - NTB command
  - PCIe NTB Endpoint

- **In Fact**: Socket CTX
  - connect(int sockfd, struct sockaddr* addr, ...)

- **Lack of connection abstraction**

- **Attempt**: Reuse kernel socket management → Inefficient
Socket-like Connection Abstraction in User Level

**Idea:** Leverage global user-space management for virtual socket, vIP and vPort

- **NTB App**
  - Socket CTX
  - `connect(int sockfd, struct sockaddr* addr, ...)`

- **NTSocks library** *(i.e., libnts)*
  - Mapping
  - vSocket Mgr
  - vIP/vPort Mgr

- **NTSocks**
  - User-level NTB library
  - Verbs API
  - NTB command
  - PCIe NTB Endpoint

- **NTSocks Mgr**
  - Monitor
  - NTB CTX

- **Monitor** is running as an independent control-plane component.
- Bridge the **control-path** semantic gap between Socket CTX and NTB CTX!
Performant Data Path at the same time

**Idea:** Transparent Zero Copy support

- Write function: `write(int sockfd, const void *BUF-1, ...)`
- BUF-1 and S-BUF-1 can be remapped into the same physical memory region

- NTBsocks Proxy is running as an independent lock-free data plane component
  - Just like a "micro-kernel"

**Diagram:**
- NTB App
  - Socket CTX
  - BUF-1
- NTBs socks Proxy
  - NTB CTX
  - S-BUF-1
- Verbs API
- User-level NTB library
  - NTB command
  - PCIe NTB Endpoint
- MEM
  - NTB mapped remote memory
  - S-BUF
Challenge #2: Enable Scalable PCIe NTB Dataplane

How to balance limited NTB resource sharing and data plane scalability?
Strawman Approach for NTB Resource Sharing

Attempt: Organize whole NTB Mem into one globally shared ringbuffer using lock
→ Inefficient due to sacrificing dataplane scalability

Key Observation: The performance of a single CPU core is hard to keep up with modern PCIe Bandwidth (>= PCIe 3x16).
Core-Driven Partition Abstraction for Scalable Data Path

**Idea:** Divide NTB Mem to multiple core-driven parallel units – **Partitions.**

- **NTB App**
  - NTStream Proxy
    - Global NTB Mem
  - Verbs API
  - User-level NTB library
  - NTB command
  - PCIe NTB Endpoint

- **App-1**, **App-2**, **App-3**, **App-4**
  - Partition-1
  - Partition-2

1. Per-connection SHM queue
2. Lock-free proactive forwarding

- Each Partition is responsible for a group of connections
  - Lock-free NTB resource sharing while preserving multi-core scalability of dataplane

Ringbuffer over remote NTB memory
Challenge #3: Ensure Performance Isolation

NTB App

NTSocks Proxy

Partition

Verbs API

User-level NTB library

NTB command

PCIe NTB Endpoint

NTB App-1

Big Flow

NTB App-2

Small Flow

TX pointer

tail

Partition-1

Head-of-Line (HoL) Blocking Issue due to coarse-grained forwarding

Ringbuffer over remote NTB memory

Shared memory queue
**Intra-Partition Performance Isolation**

**Idea:** Per-flow message slicing + fine-grained proactive forwarding

This mechanism helps mitigate intra-partition HoL blocking issue
Inter-Partition Connection Scheduling

**Idea:** Isolate bandwidth-sensitive and latency-sensitive flows into different Partitions

- **NTB App**
  - NTSox Proxy
    - Partition
    - Verbs API
    - User-level NTB library
      - NTB command
        - PCIe NTB Endpoint

- **NTB App-1**
  - Big Flow
    - BW-sensitive
    - Partition-1

- **NTB App-2**
  - Small Flow
    - Latency-sensitive
    - Partition-2

Enable better inter-partition load balance and performance isolation
More Optimizations for Performance in the Paper

• NTB Ringbuffer with efficient NTB verbs
• Receiver-Driven Flow Control
• Thread model
• Data packet batch forwarding
• Runtime NTSocks implementation in a tightly-coupled manner
• ......

Please refer to our paper 😄
Agenda

• Motivation

• NTSoックス Design

• Implementation and Evaluation

• Summary
Implementation and Experimental Setup

<table>
<thead>
<tr>
<th>Components</th>
<th>Lines in C</th>
</tr>
</thead>
<tbody>
<tr>
<td>NTSock Library (i.e., libnts)</td>
<td>3700</td>
</tr>
<tr>
<td>NTSock Proxy (i.e., NTP)</td>
<td>2500</td>
</tr>
<tr>
<td>NTSock Monitor (i.e., NTM)</td>
<td>4100</td>
</tr>
<tr>
<td>NTSock Common Utils</td>
<td>4000</td>
</tr>
<tr>
<td>In Total</td>
<td>14300</td>
</tr>
</tbody>
</table>

- Build NTSock on DPDK NTB Poll Mode Driver (PMD)
- Testbed setup
  - Two Intel Xeon Gold 5218 32-core CPUs, 64 GB RAM, PCIe GEN 3x16
  - 80Gbps PCIe NTB reference adapter by Intel (experimental platform)
  - Mellanox ConnectX-5 NICs (100Gbps)
Does NTSox support ultra-low latency?

NTSox achieves dramatically better latency by up to 20.4x and 2.3x than Linux TCP and RDMA socket, respectively.

![Graph showing latency comparison]

Ping-Pong micro-benchmark
Does NT Socks Support Scalability and Performance Isolation?

NT Socks achieves better multi-core scalability

NT Socks eliminates Head-of-Line blocking issue

Impact of intra-Partition isolation (message slicing) on NT Socks

Multi-thread scalability
Do Applications Benefit from NTSoxks?

Key-Value Store:
- No code modification with NTSoxks

End-to-end median latency of key-value stores with YCSB workloads

NTSoxks reduces latency by up to 24.5x and 1.6x, compared to TCP Redis and RDMA respectively.
• Disaggregation today requires high-speed rack-scale communication.
• Existing solutions are insufficient due to the inevitable protocol translation overhead and in-NIC resource management.
• Ultra-low latency PCIe Interconnect has great potential without protocol translation.

• NTSoxks enables rack-scale applications to benefit from ultra-low latency PCIe Interconnect.
  • Socket-like compatible connection abstraction
  • Partition abstraction for scalable data plane
  • Hierarchical performance isolation mechanism
• Outperforms state-of-the-art solutions.

Yibo Huang    ybhuang.cs@gmail.com

github.com/NTSoxks/ntsoxks