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

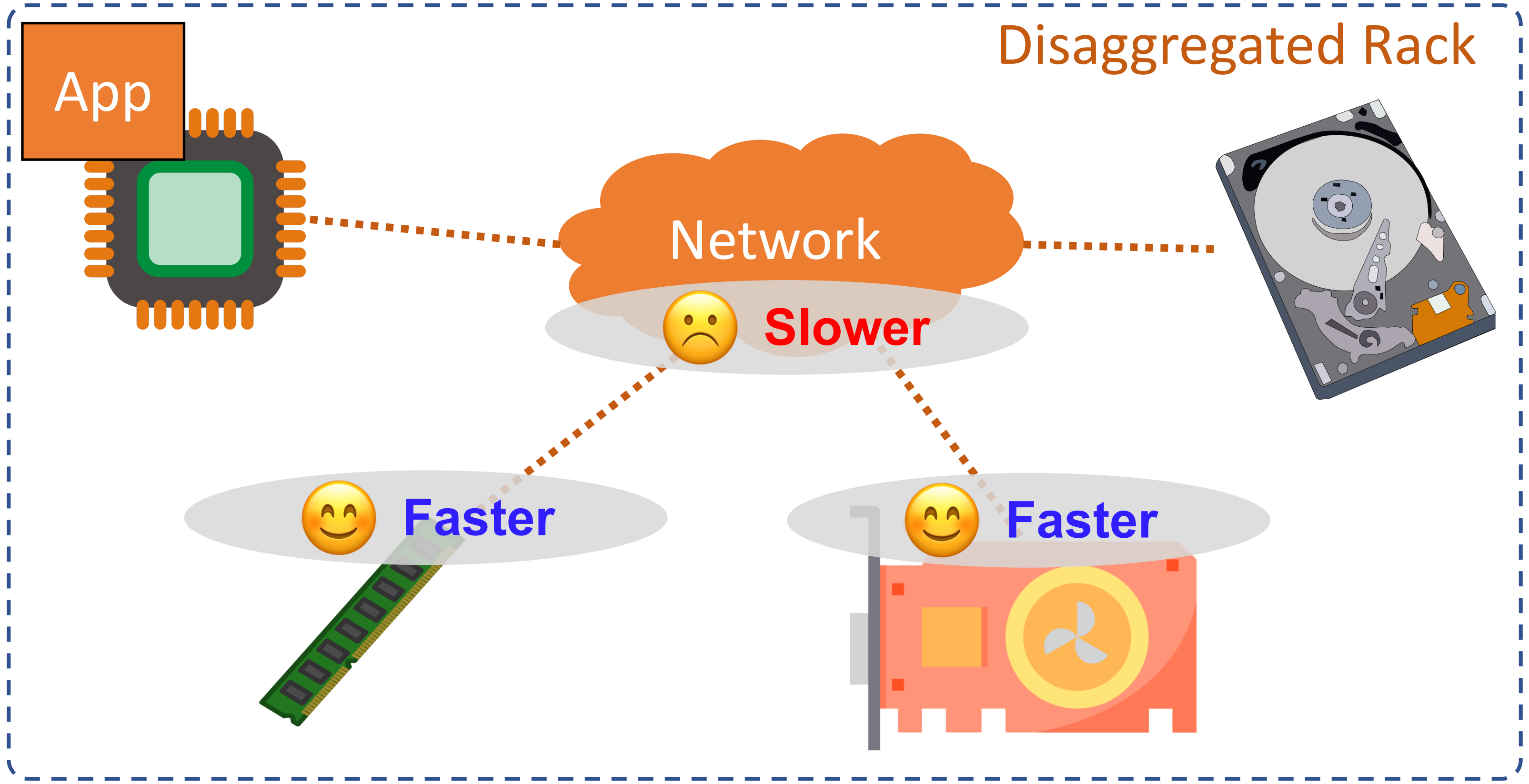
Yibo Huang, Yukai Huang, Ming Yan, Jiayu Hu, Cunming Liang, Yang Xu, Wenxiong Zou, Yiming Zhang, Rui Zhang, Chunpu Huang, Jie Wu

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

⇩

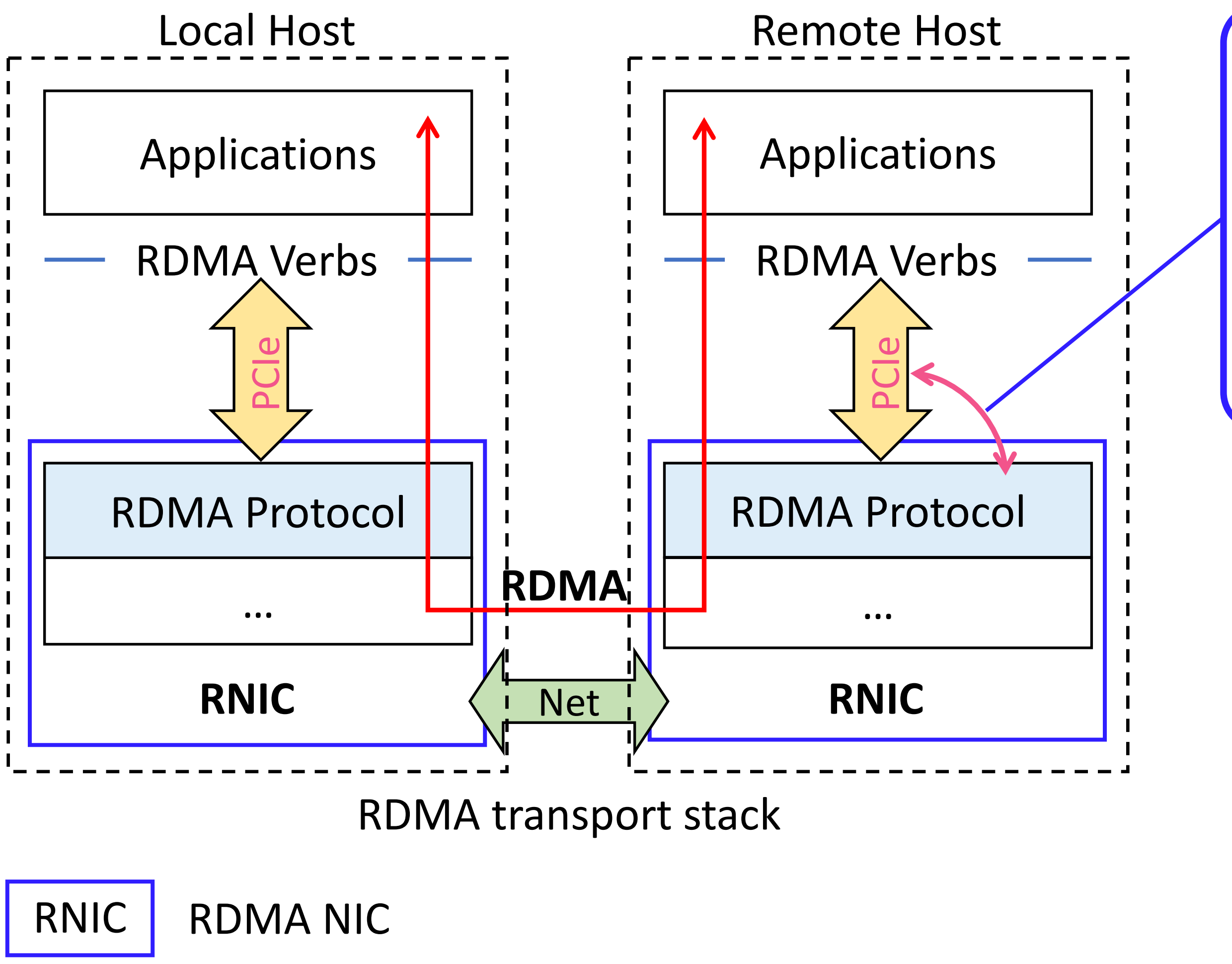
RDMA
Hardware Offloading



The network lags behind

[1] Gao, et al. (OSDI '16)

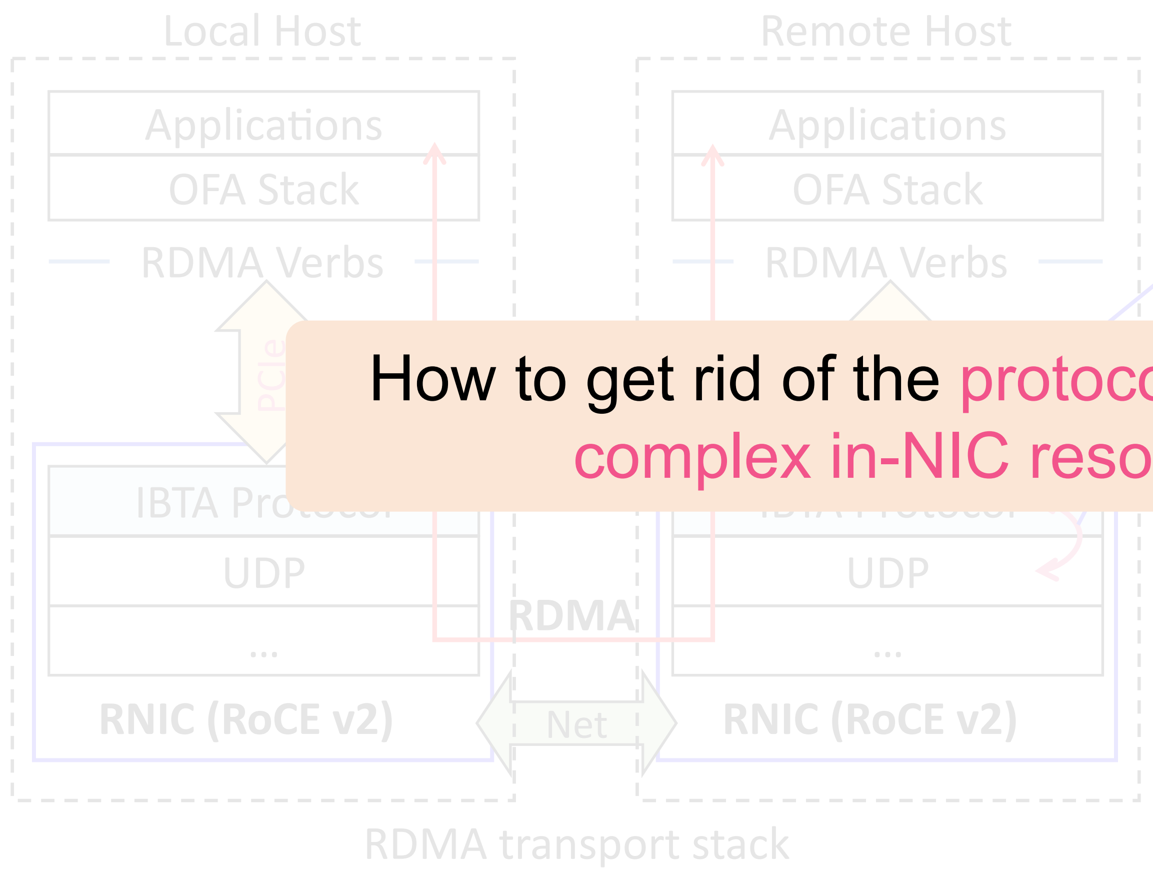
Why is current RDMA insufficient?



- 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

At least 1.6us delay for one RTT

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 … ing table[1]

At least 1.6us delay for one RTT

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

RNIC RDMA NIC

[1] eRPC (NSDI '19)

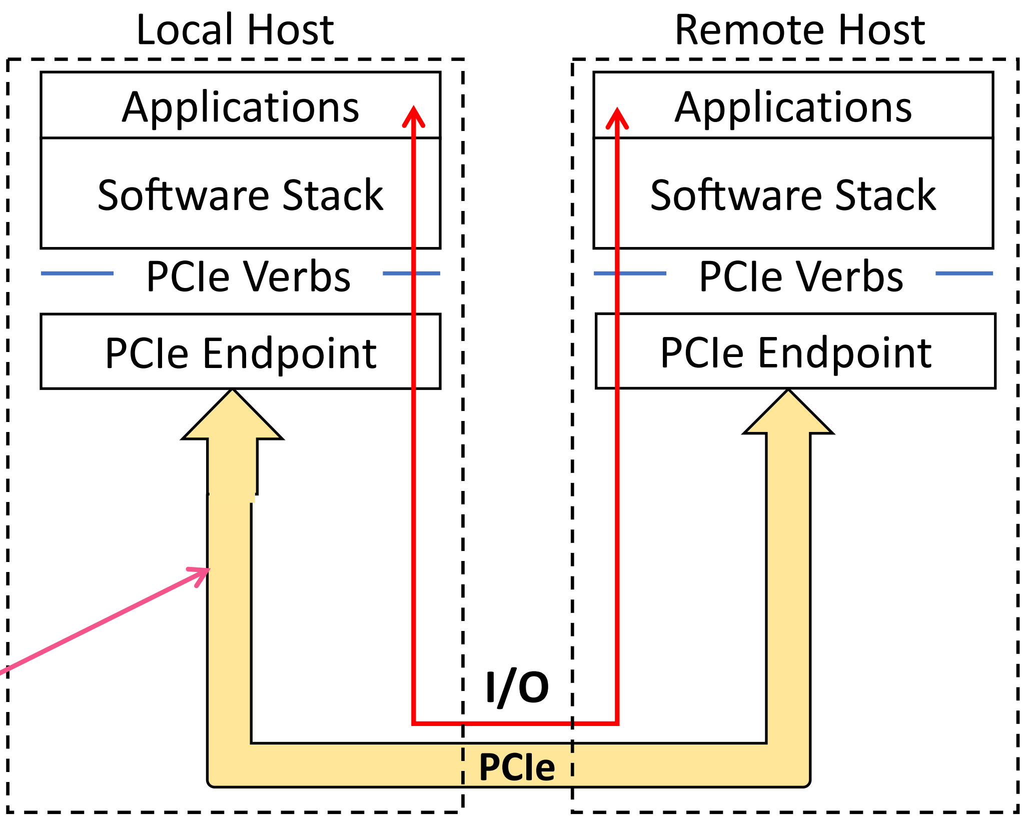
New opportunities with high-speed PCIe interconnect

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



PCIe interconnect transport stack

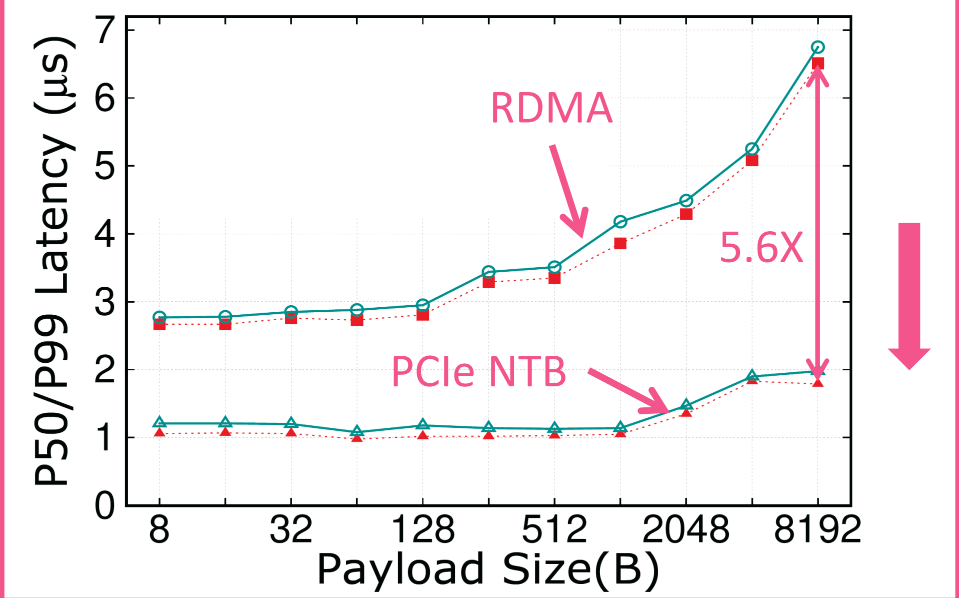
* CXL: Compute eXpress Link

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

PCIe Generation	Bandwidth
PCIe 3.0 x16	128 Gbps
PCIe 4.0 x16	256 Gbps
PCIe 5.0 x16	512 Gbps
PCIe 6.0 x16	1024 Gbps
…	…

- Cache-coherent remote memory access

New opportunities with high-speed PCIe interconnect

Advanced PCIe interconnect

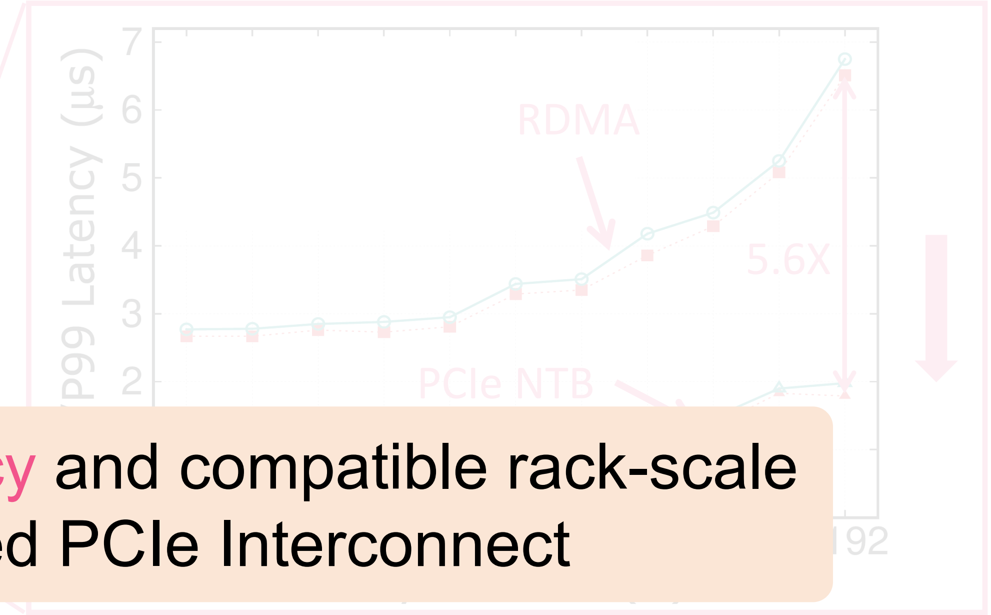
- Ultra-low latency
  - ~*500ns* one-way latency with PCIe NTB

PCI

- High bandwidth

Can match evolving PCIe bandwidth

- Cache-coherent remote memory access

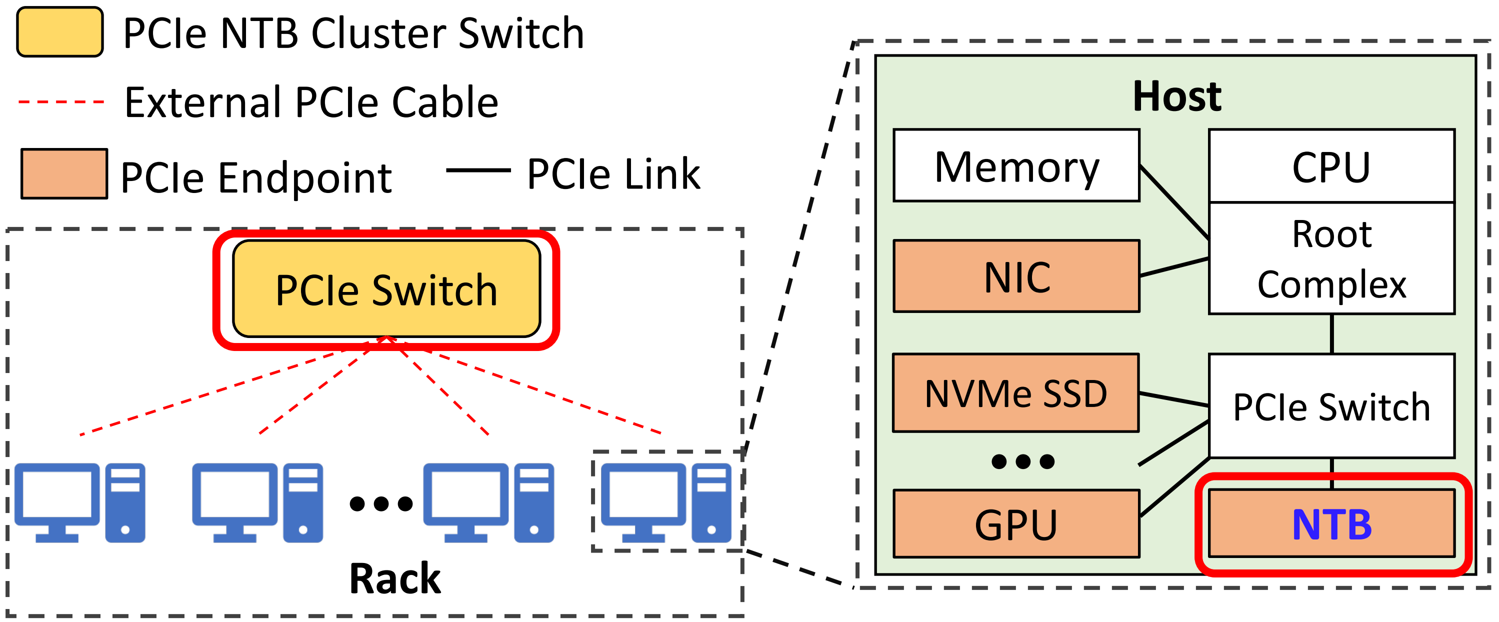


PCIe Generation	Bandwidth
PCIe 3.0 x16	128 Gbps
PCIe 4.0 x16	256 Gbps
PCIe 5.0 x16	512 Gbps
PCIe 6.0 x16	1024 Gbps
...	...

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

Vision: PCIe Interconnect for High-speed Rack-scale Network

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

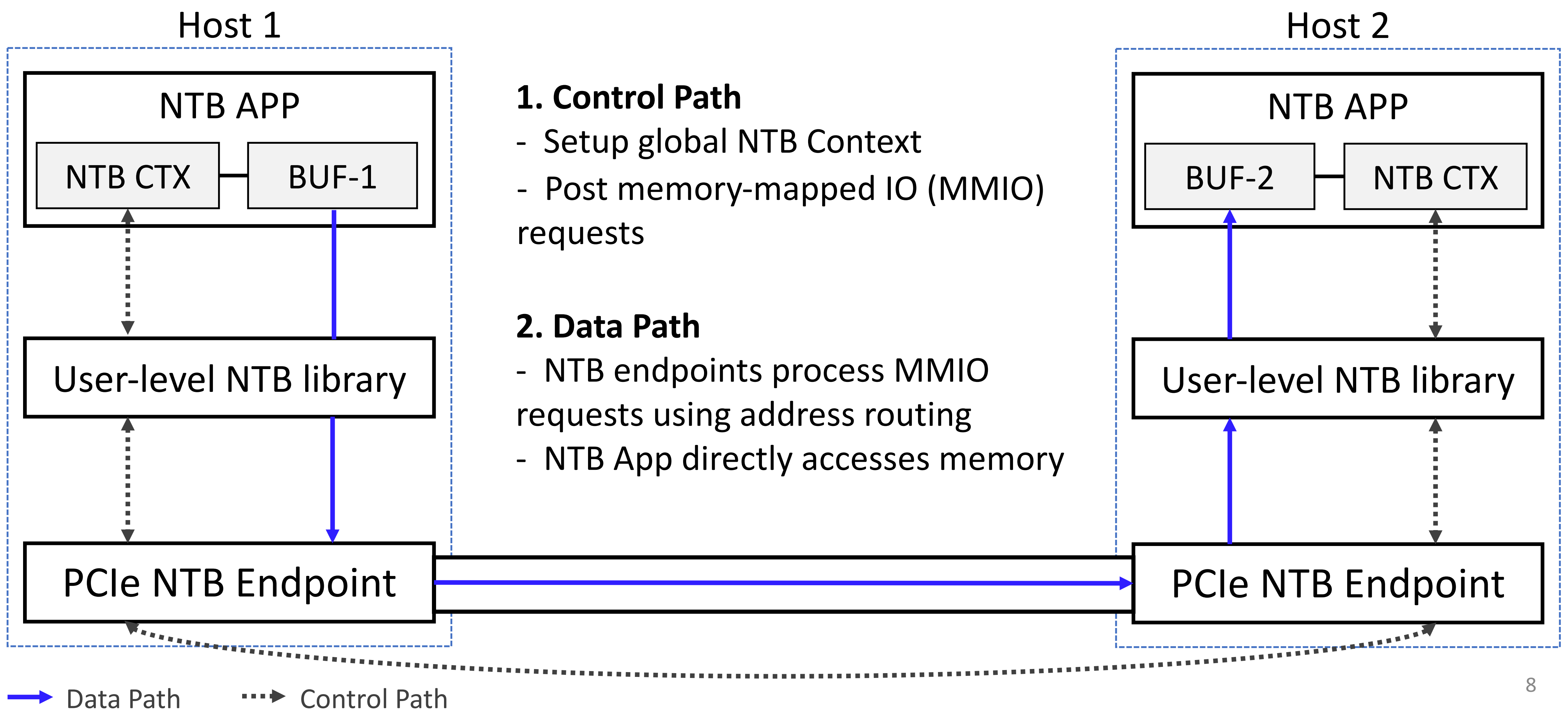


The in-rack network architecture with PCIe NTB fabric

* 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.

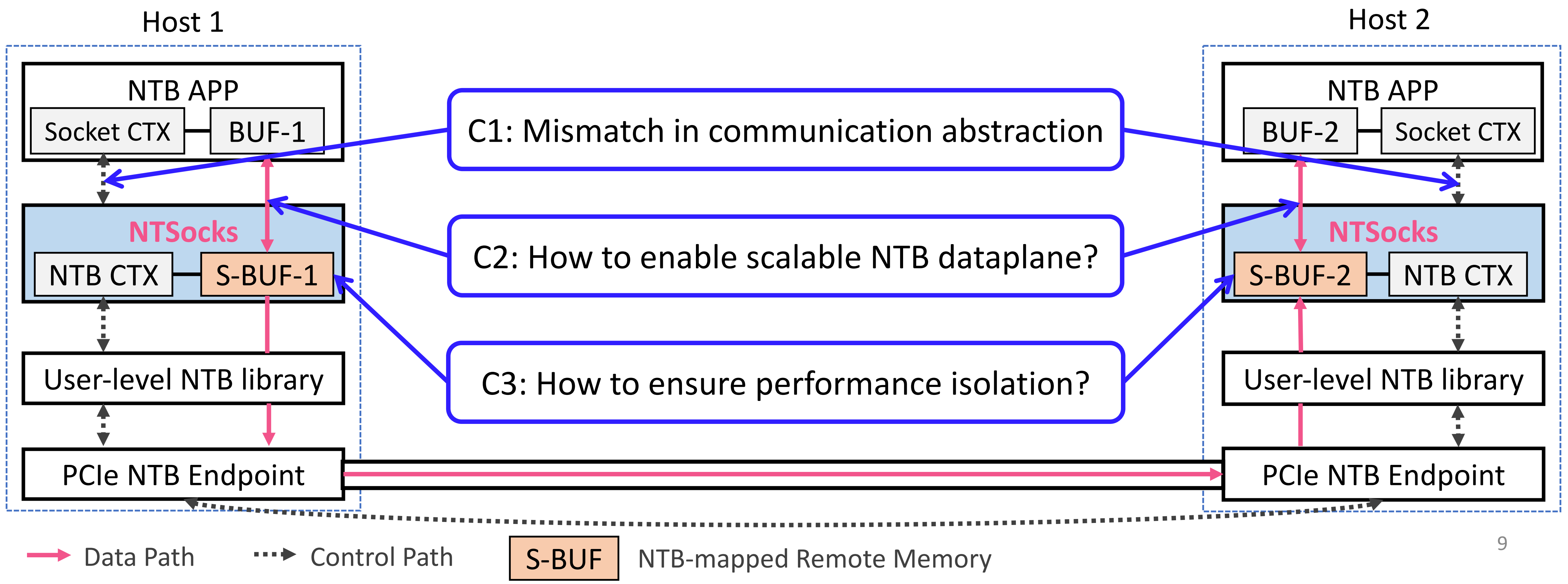


1. Control Path
- Setup global NTB Context
- Post memory-mapped IO (MMIO) requests

2. Data Path
- NTB endpoints process MMIO requests using address routing
- NTB App directly accesses memory

Our Work: NTSocks with three key challenges

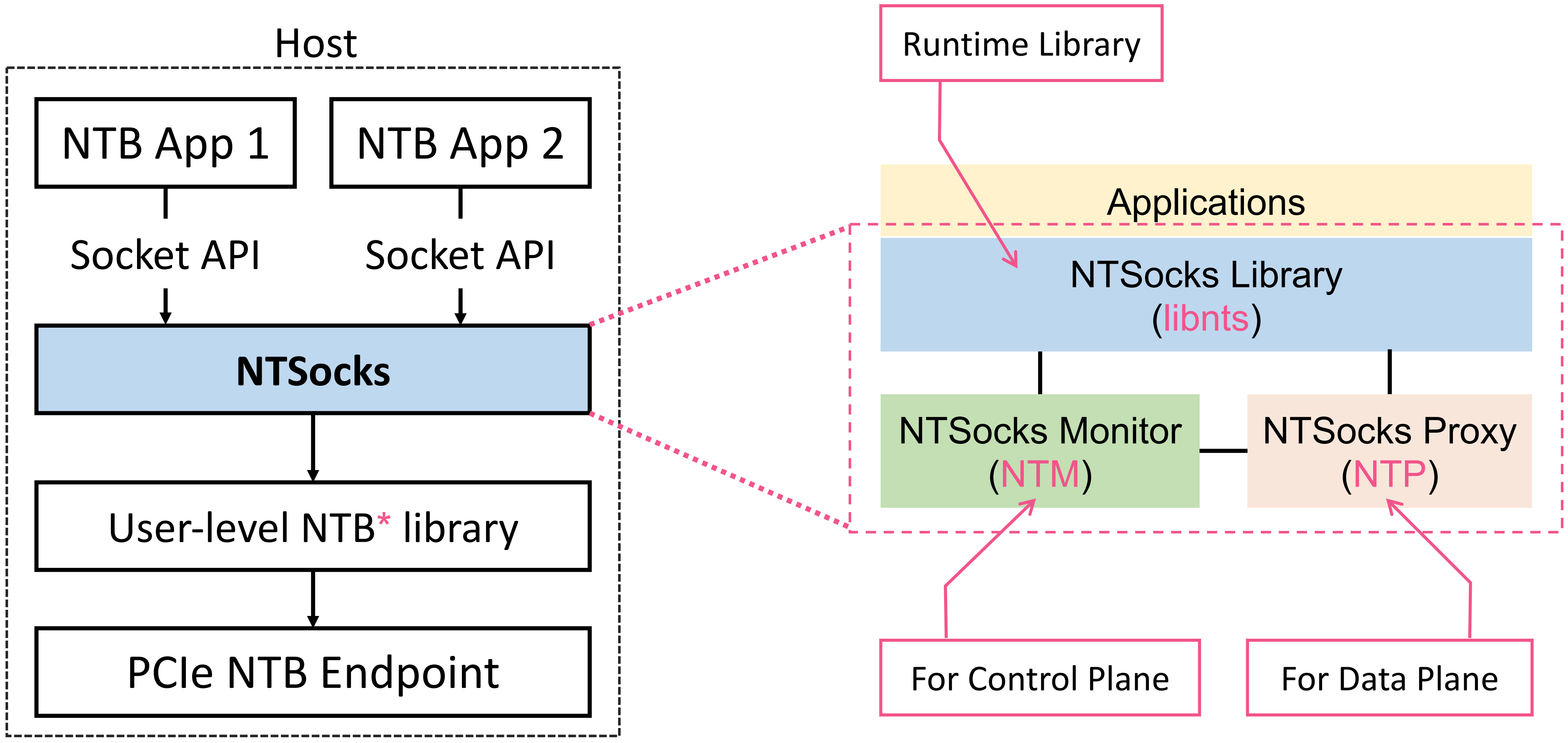
- A lightweight end-host network stack over PCIe Interconnect that achieves transparency while preserving high performance.
- However, this is hard in general due to the following three challenges:



Agenda

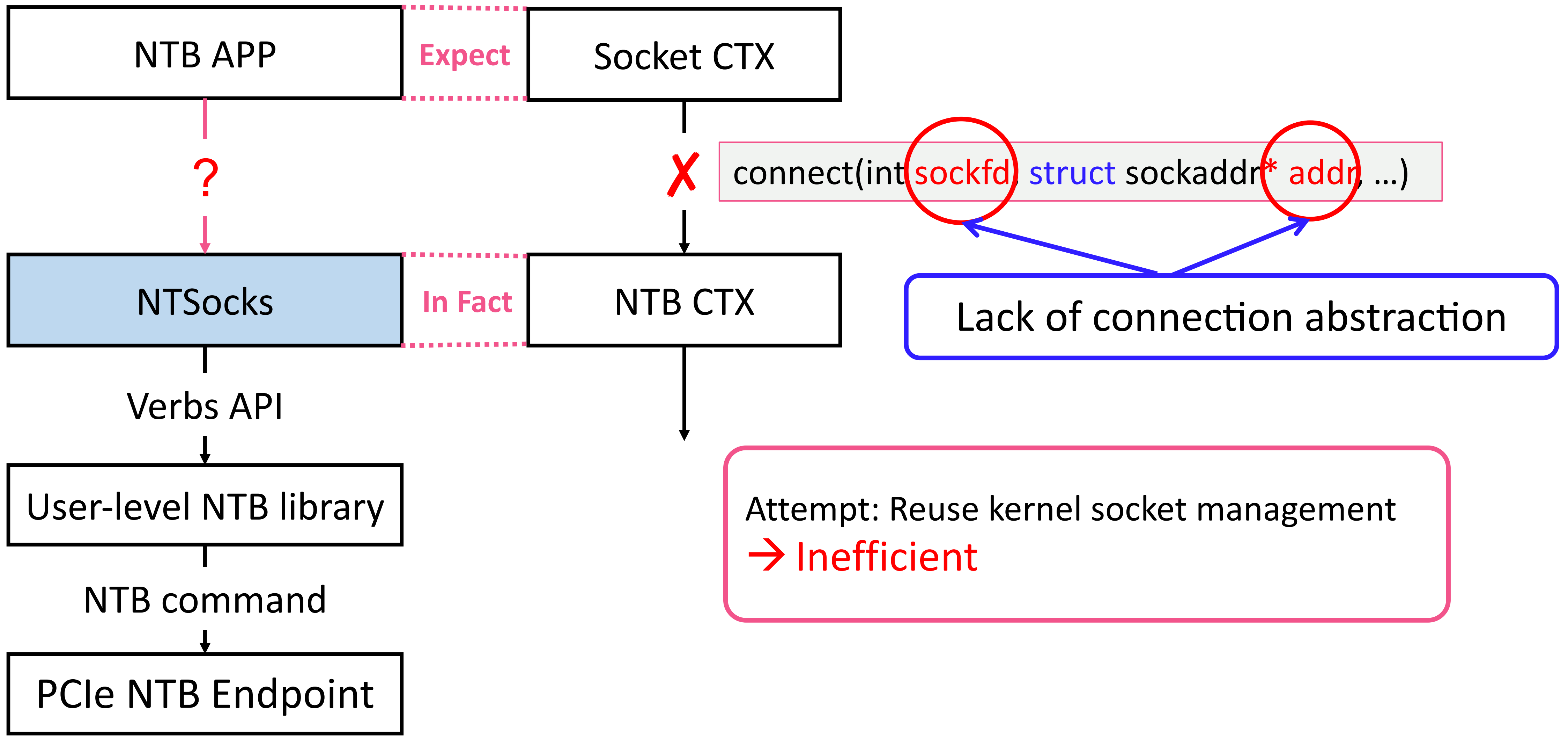
- Motivation
- NTSocks Design
- Implementation and Evaluation
- Summary

NTSocks Architecture Overview



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

Challenge #1: Mismatch in communication abstractions



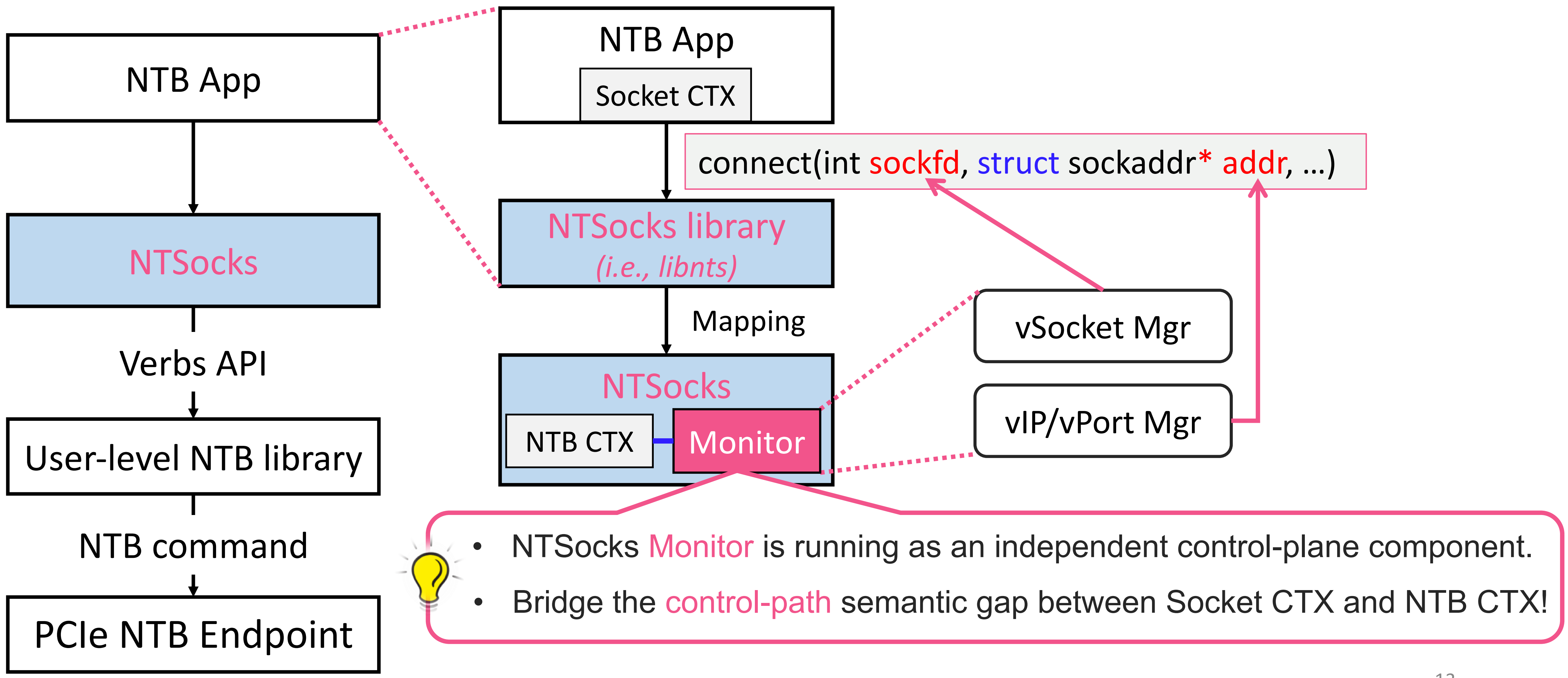
connect(int sockfd, struct sockaddr* addr, …)

Lack of connection abstraction

Attempt: Reuse kernel socket management
→ Inefficient

Socket-like Connection Abstraction in User Level

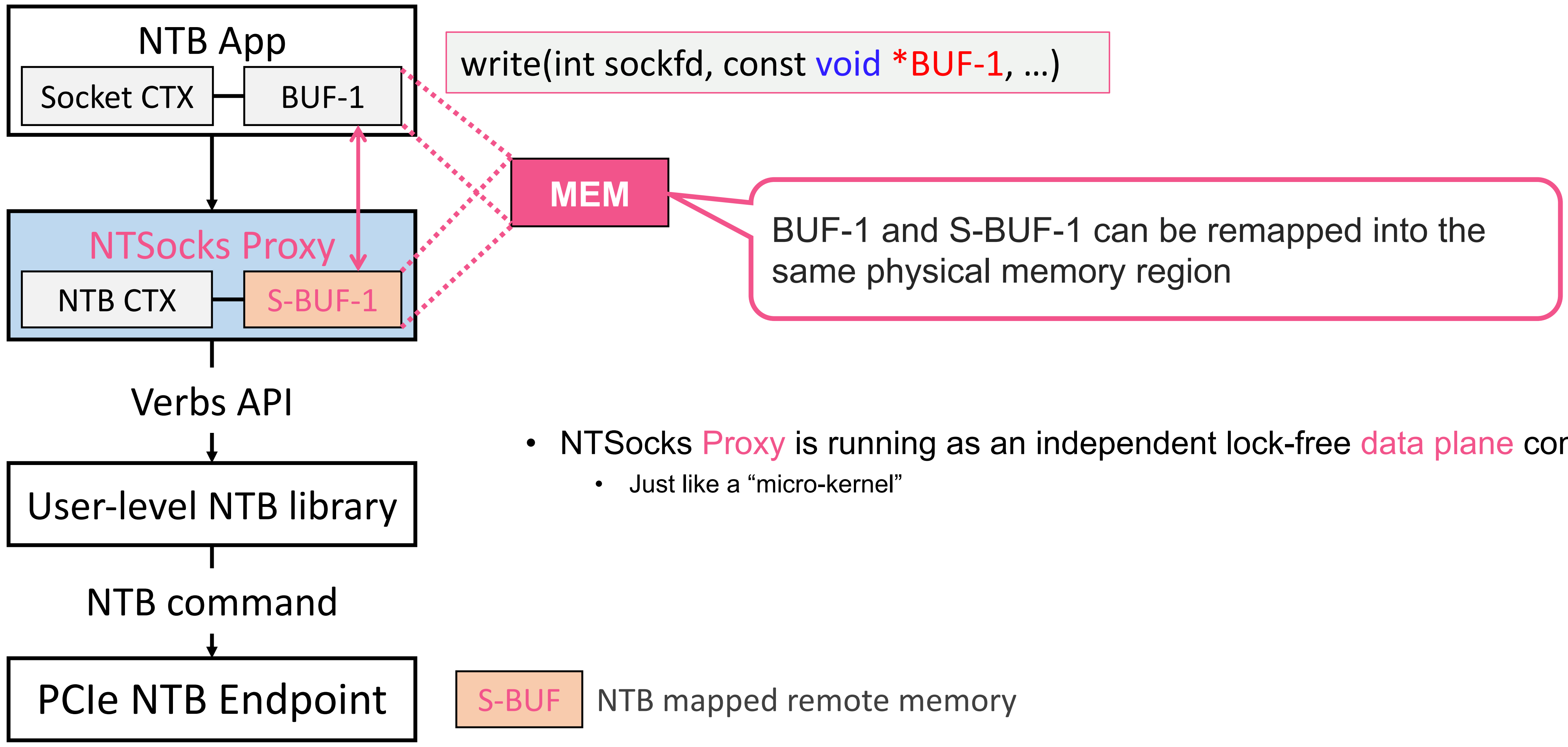
Idea: Leverage global user-space management for virtual socket, vIP and vPort



- NTSocks Monitor is running as an independent control-plane component.
- Bridge the control-path semantic gap between Socket CTX and NTB CTX!

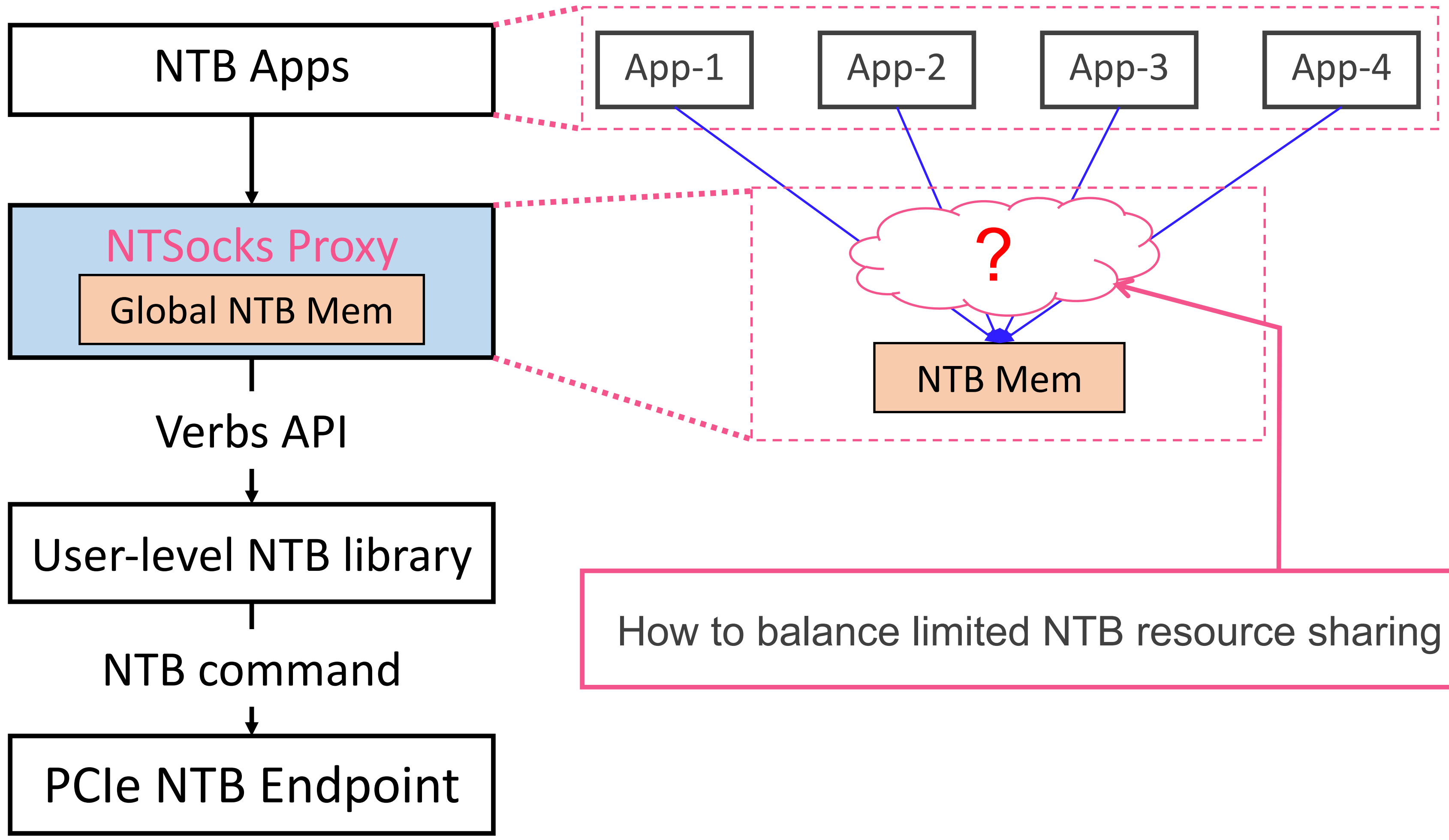
Performant Data Path at the same time

Idea: Transparent Zero Copy support



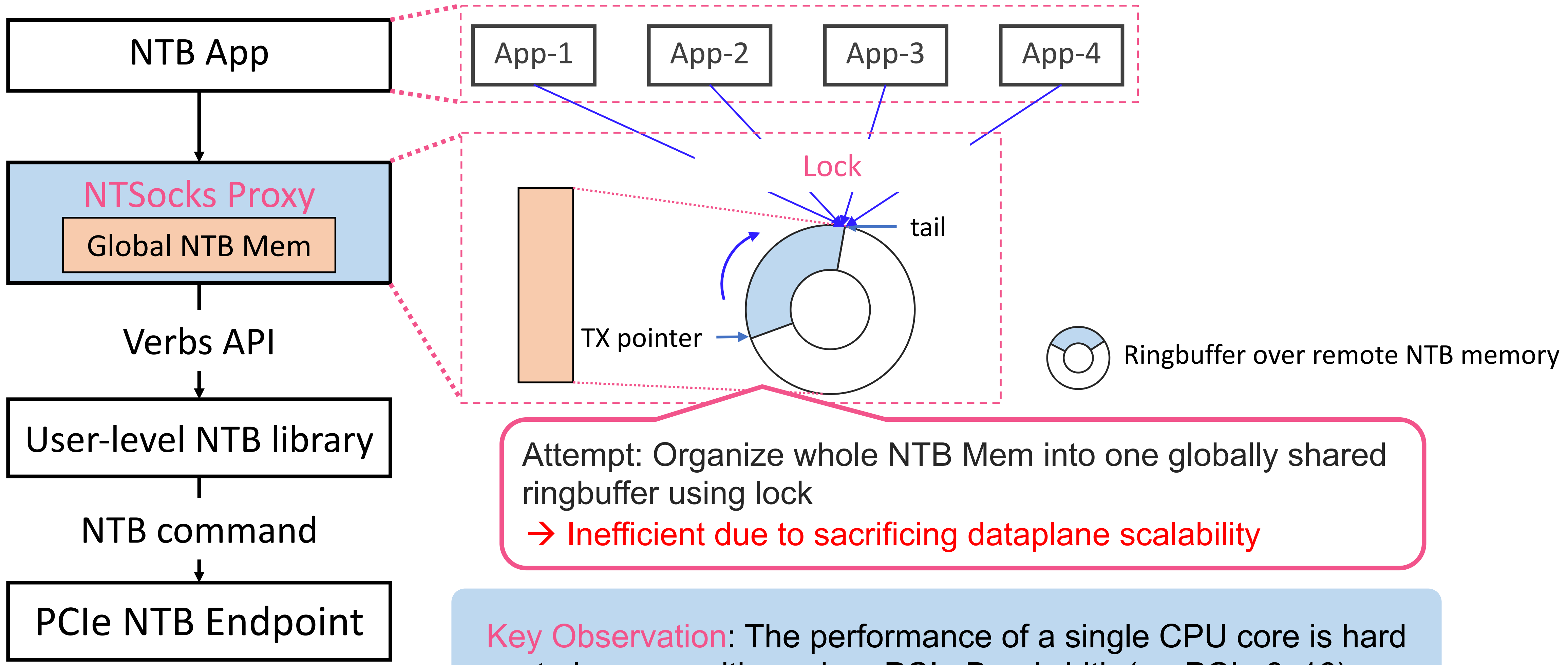
- NTSocks Proxy is running as an independent lock-free data plane component
  - Just like a "micro-kernel"

Challenge #2: Enable Scalable PCIe NTB Dataplane



How to balance limited NTB resource sharing and data plane scalability?

Strawman Approach for NTB Resource Sharing

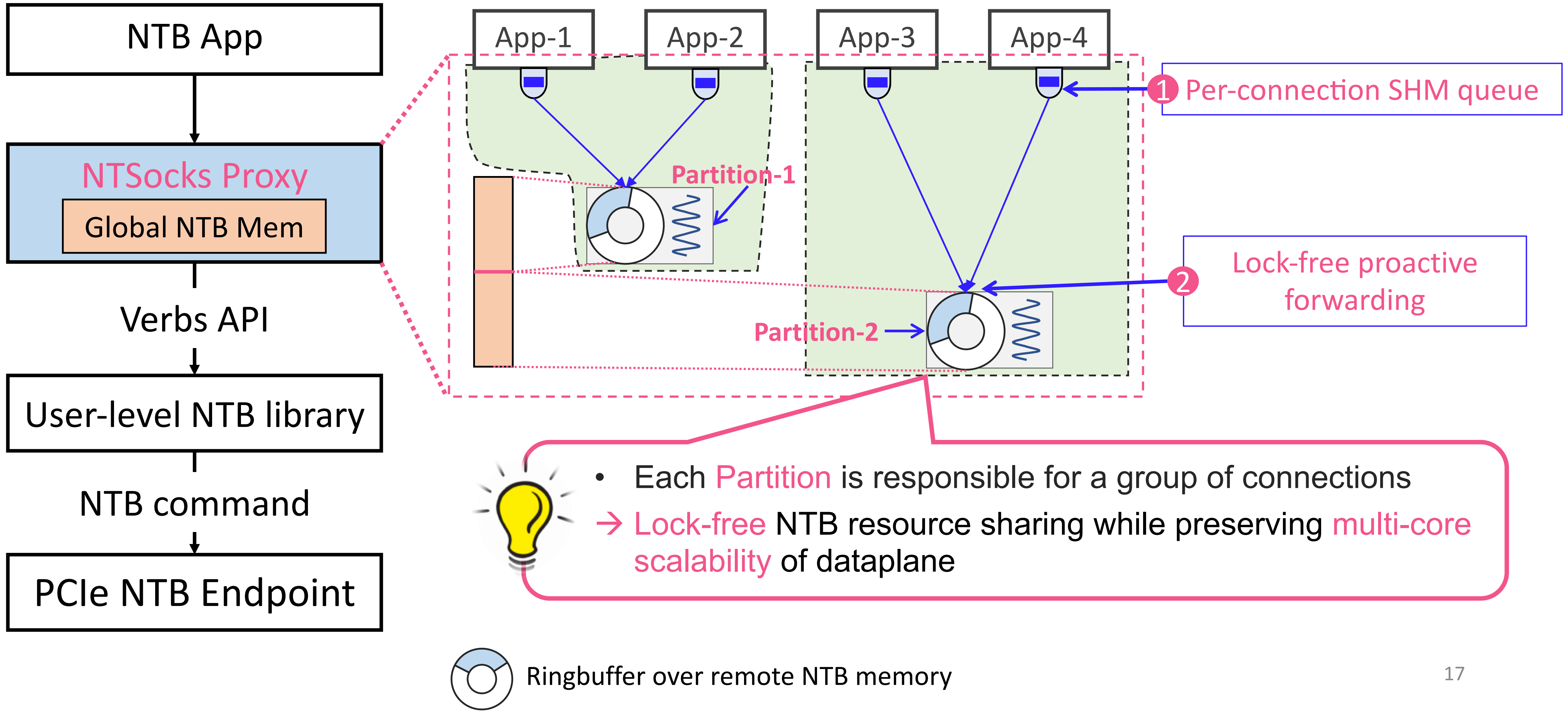


Attempt: Organize whole NTB Mem into one globally shared ringbuffer using lock
→ Inefficient due to sacrificing dataplane scalability

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

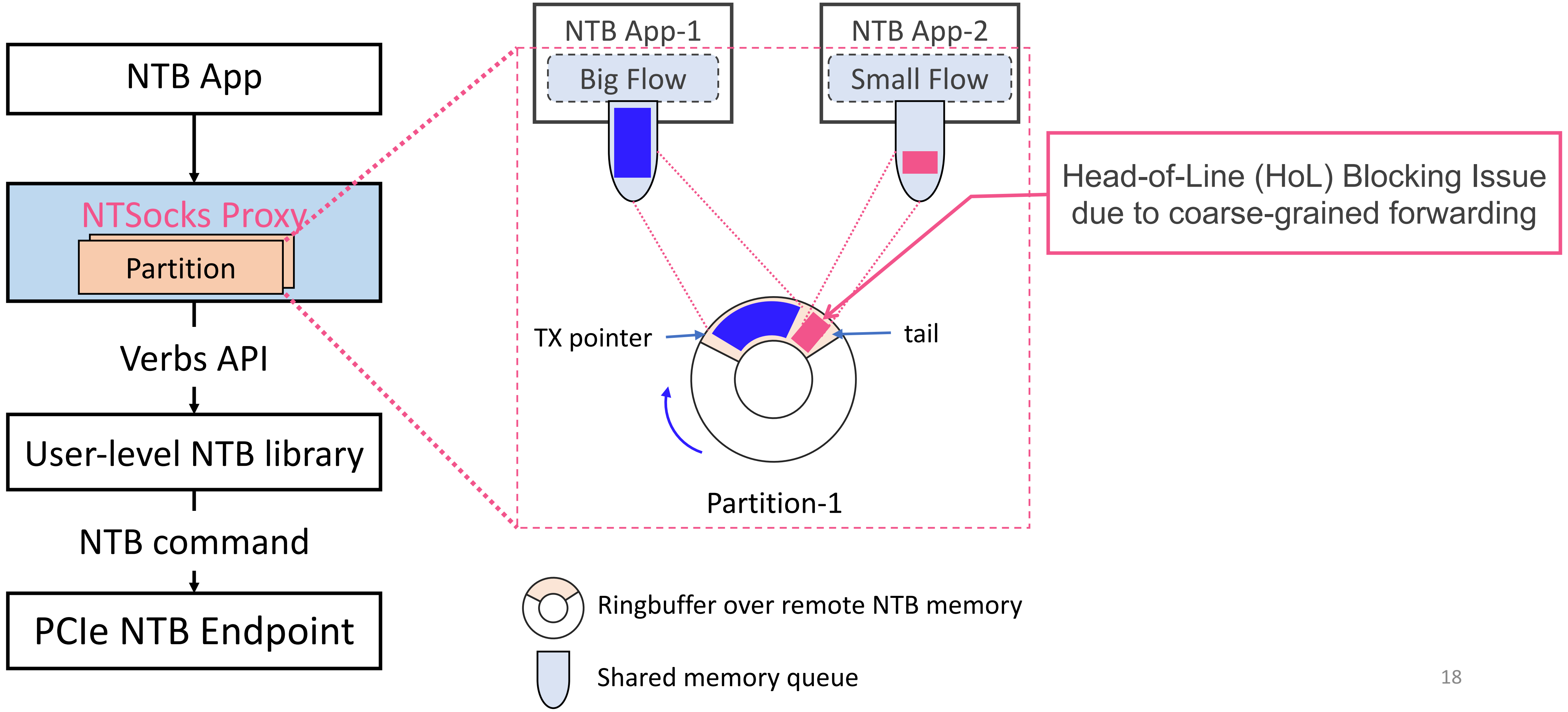
Idea: Divide NTB Mem to multiple core-driven parallel units – Partitions.



- Each Partition is responsible for a group of connections

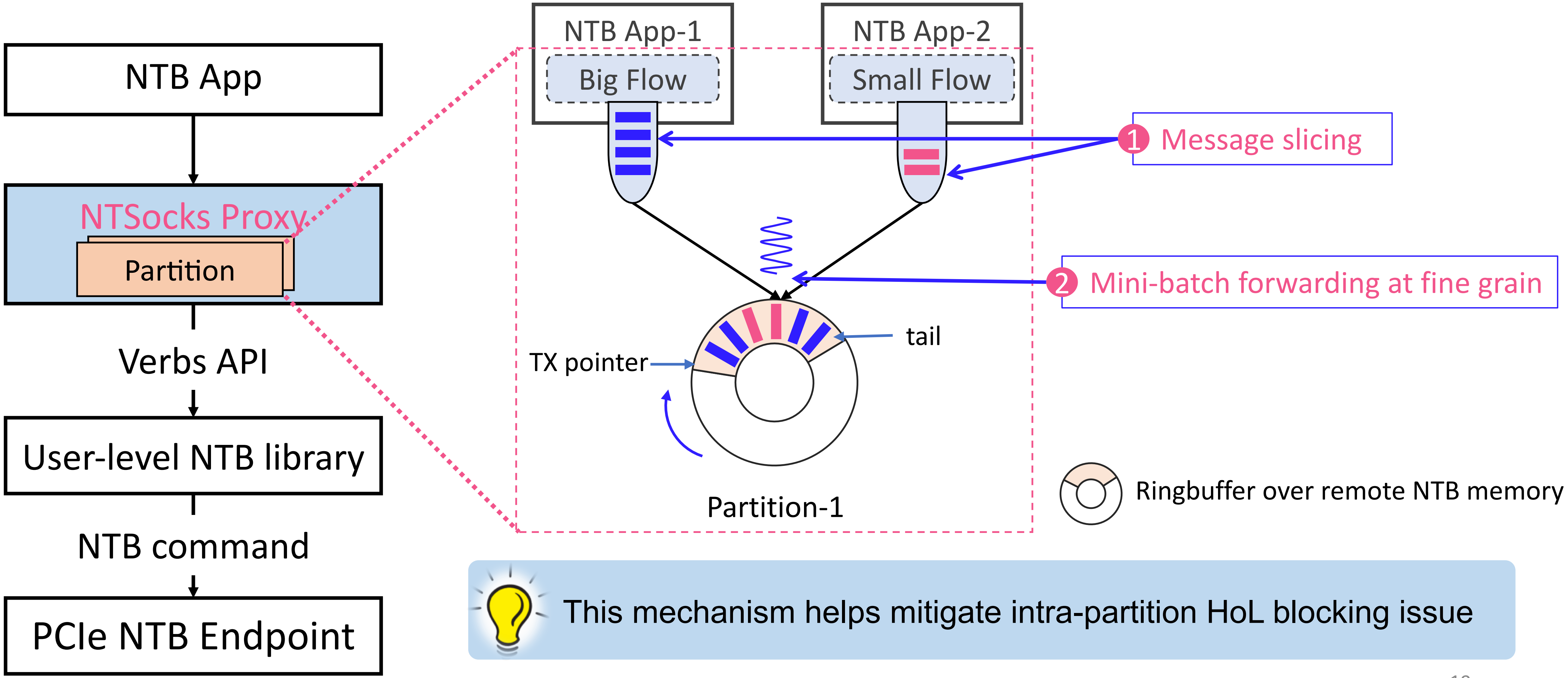
→ Lock-free NTB resource sharing while preserving multi-core scalability of dataplane

Challenge #3: Ensure Performance Isolation



Intra-Partition Performance Isolation

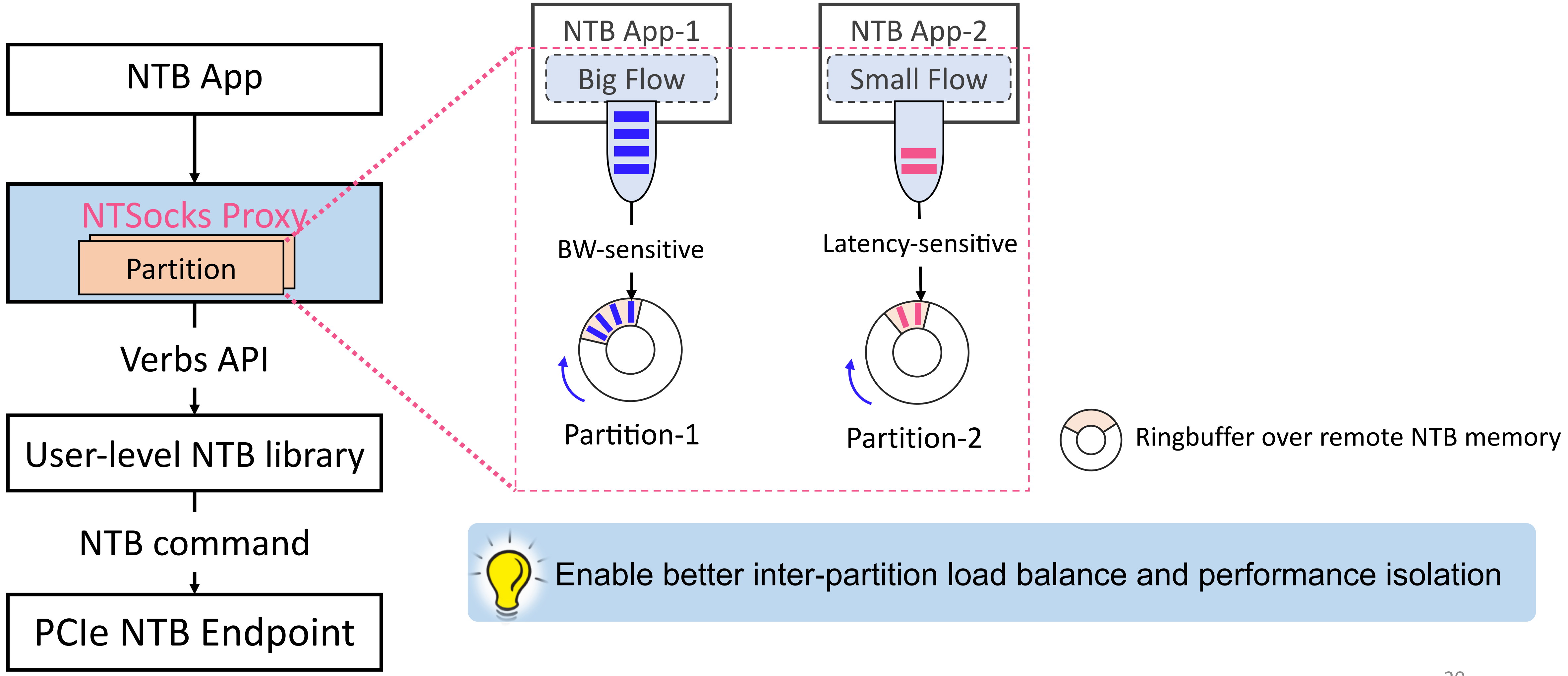
Idea: Per-flow message slicing + fine-grained proactive forwarding



This mechanism helps mitigate intra-partition HoL blocking issue

Inter-Partition Connection Scheduling

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



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
- ……

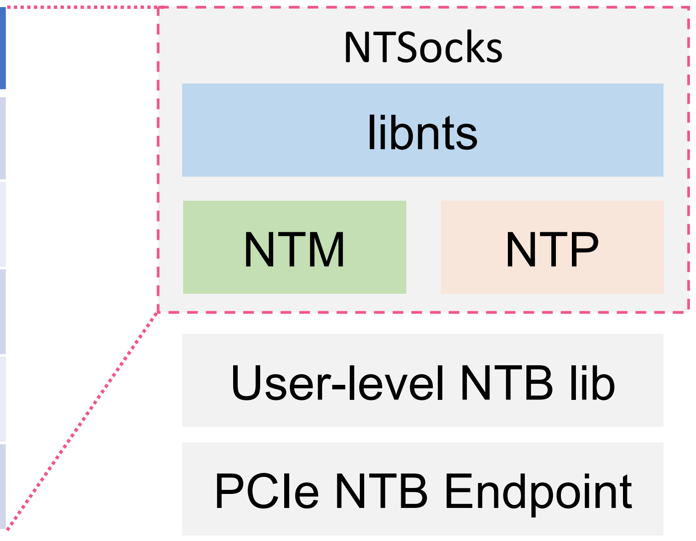
Please refer to our paper 😄

Agenda

- Motivation
- NTSocks Design
- Implementation and Evaluation
- Summary

Implementation and Experimental Setup

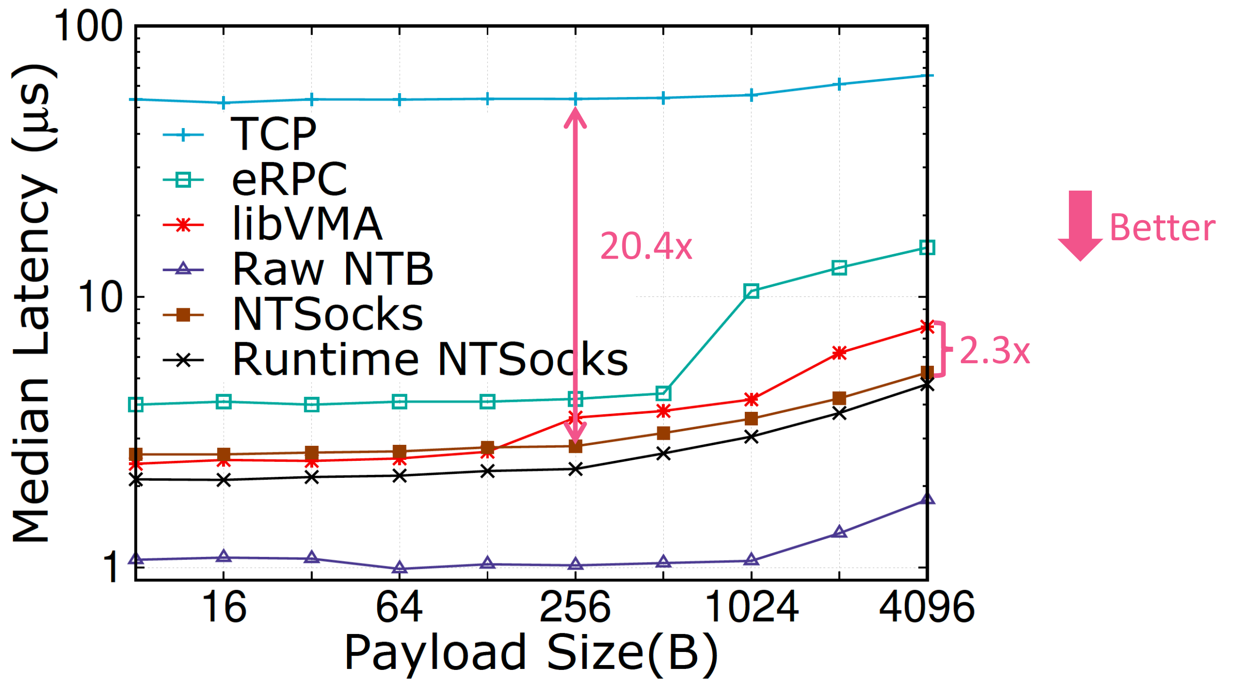
Components	Lines in C
NTSocks Library (i.e., *libnts*)	**3700**
NTSocks Proxy (i.e., *NTP*)	**2500**
NTSocks Monitor (i.e., *NTM*)	**4100**
NTSocks Common Utils	**4000**
In Total	**14300**



- Build NTSocks 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 NTSocks support ultra-low latency?

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

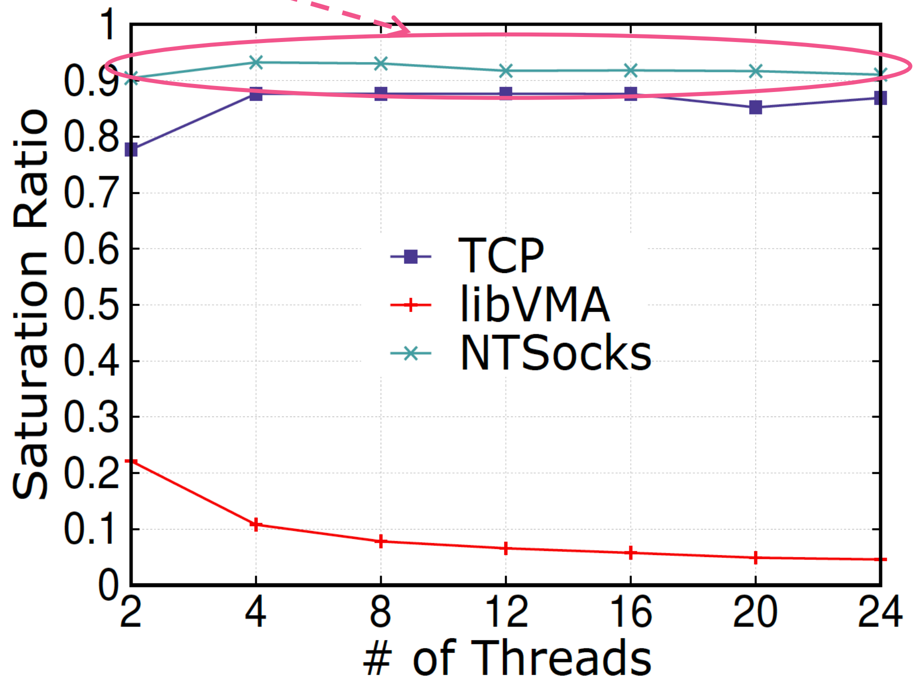


Ping-Pong micro-benchmark

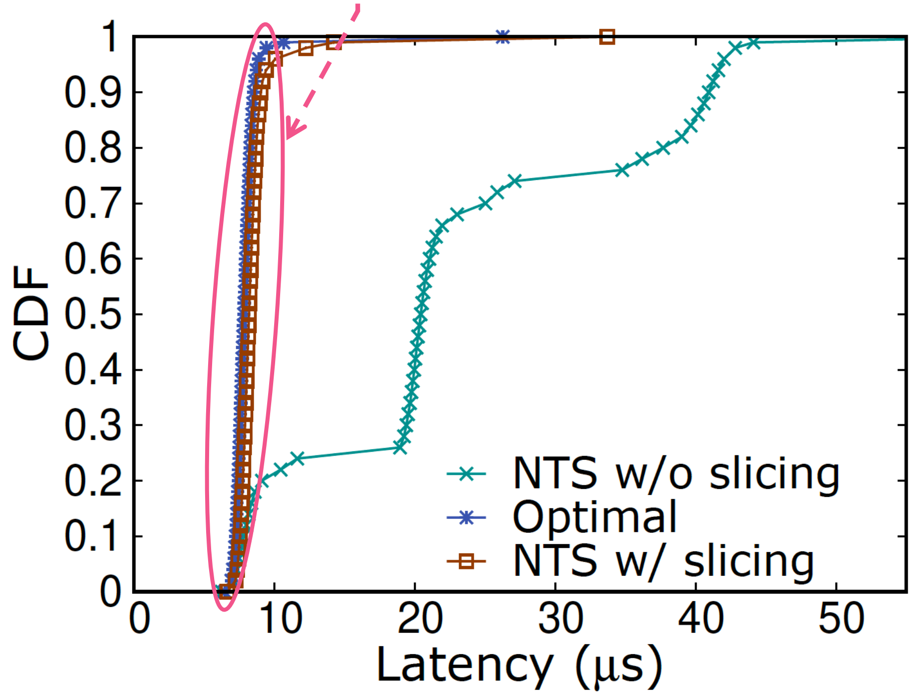
Does NTSocks Support Scalability and Performance Isolation?

NTSocks achieves better multi-core scalability

NTSocks eliminates Head-of-Line blocking issue



Multi-thread scalability



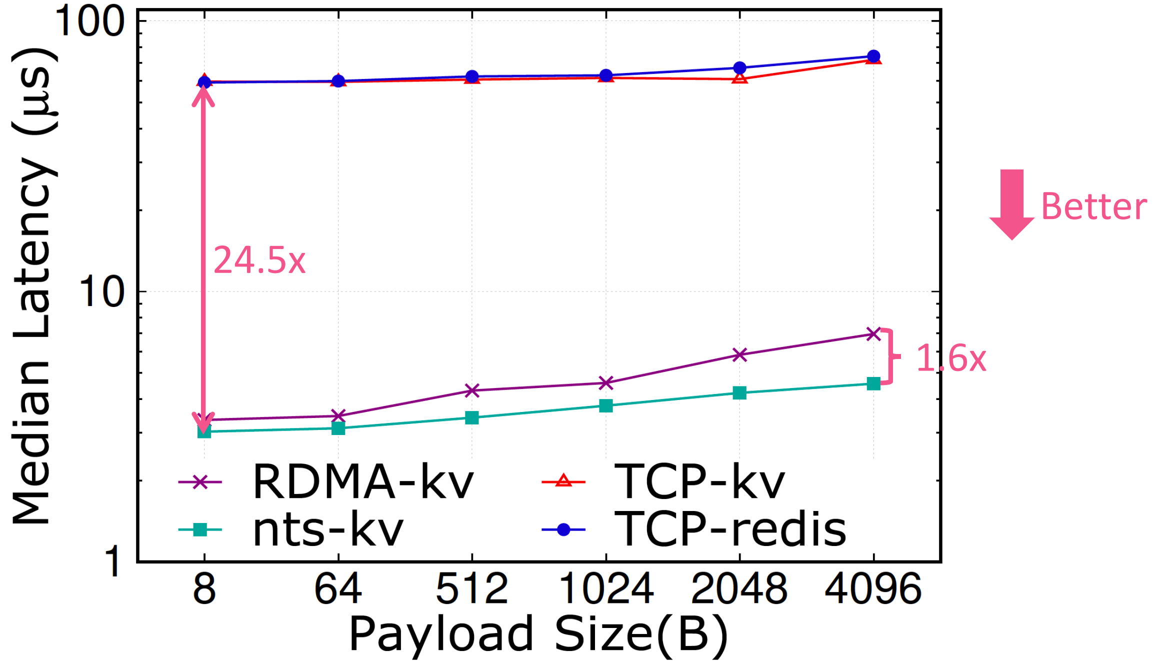
Impact of intra-Partition isolation (message slicing) on NTSocks

Do Applications Benefit from NTSocks?

Good Compatibility

Key-Value Store:

- No code modification with NTSocks



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

NTSocks reduces latency by up to 24.5x and 1.6x, compared to TCP Redis and RDMA respectively

Summary

- 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.

- NTSocks 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.

github.com/NTSocks/ntsocks

Yibo Huang ybhuang.cs@gmail.com