Advanced Computer Networks

# Endhost Network Stack in Data Center Networks (I)

https://pages.cs.wisc.edu/~mgliu/CS740/F25/index.html

**Ming Liu**
**mgliu@cs.wisc.edu**

# Outline

- Last lecture
  - Transport in Data Center Networks (III)

- Today
  - Endhost Network Stack in Data Center Networks (I)

- Announcements
  - In-class Exam 11/20/2025

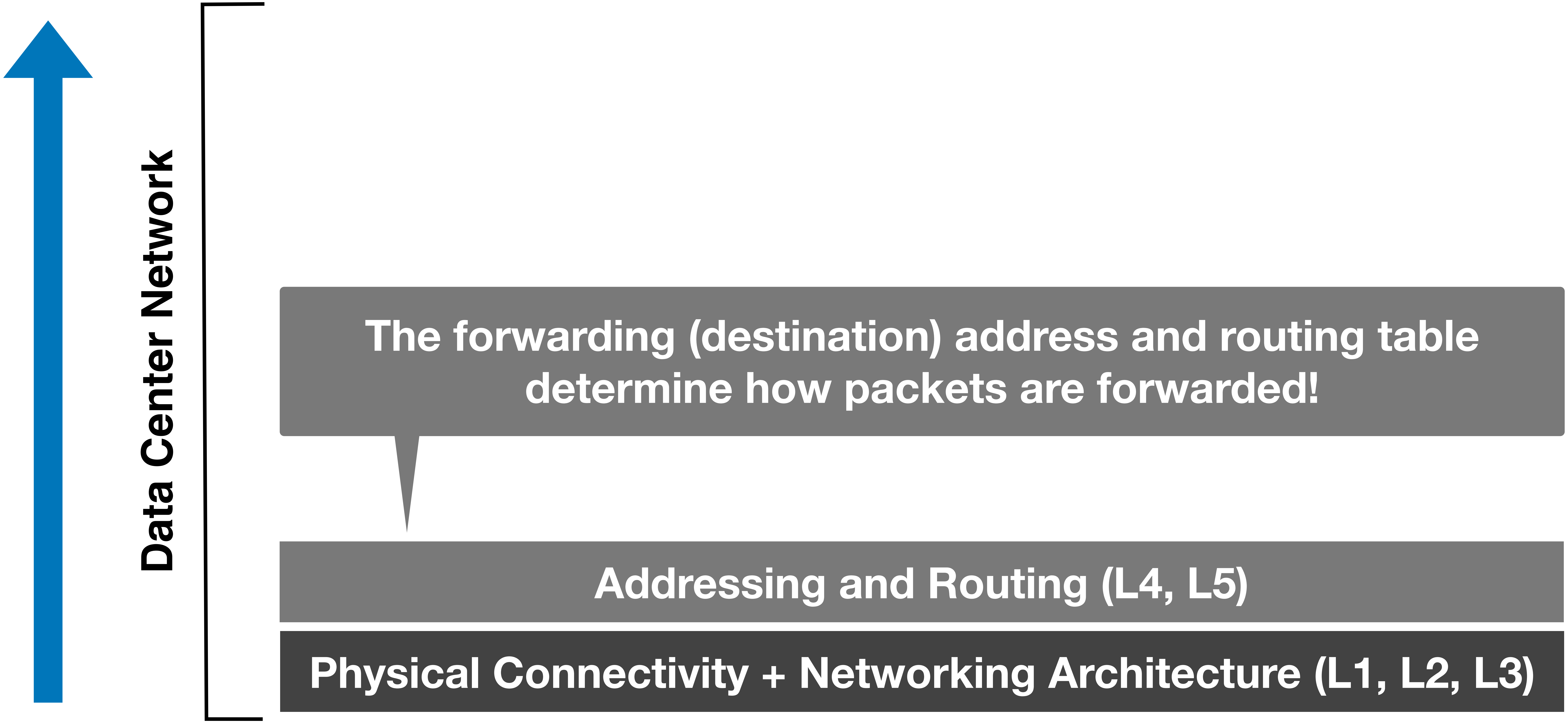
# Where we are?

Data Center Network

Multiple communication paths exist when accessing and traversing data center networks!

Physical Connectivity + Networking Architecture (L1, L2, L3)

# Where we are?



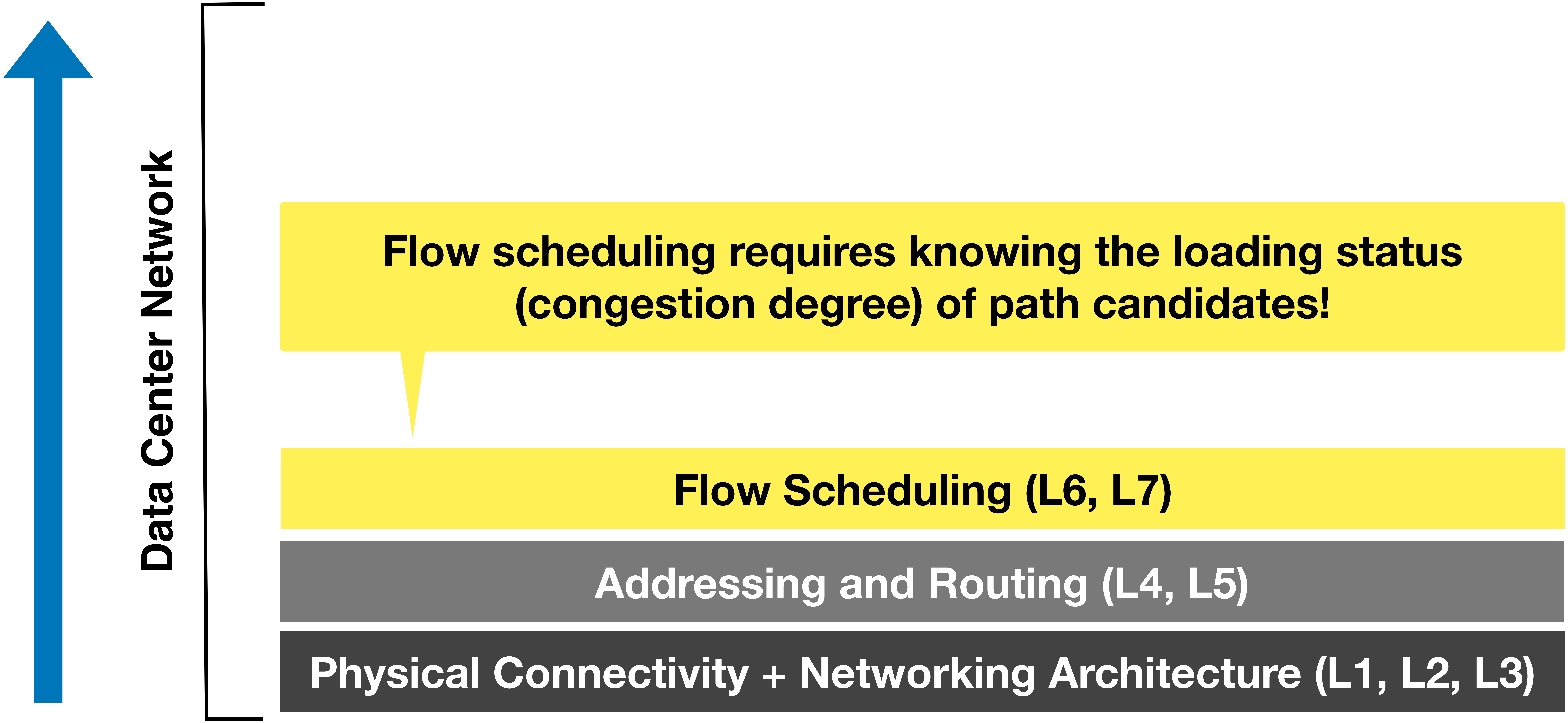
Data Center Network

The forwarding (destination) address and routing table determine how packets are forwarded!

Addressing and Routing (L4, L5)

Physical Connectivity + Networking Architecture (L1, L2, L3)

# Where we are?



Data Center Network

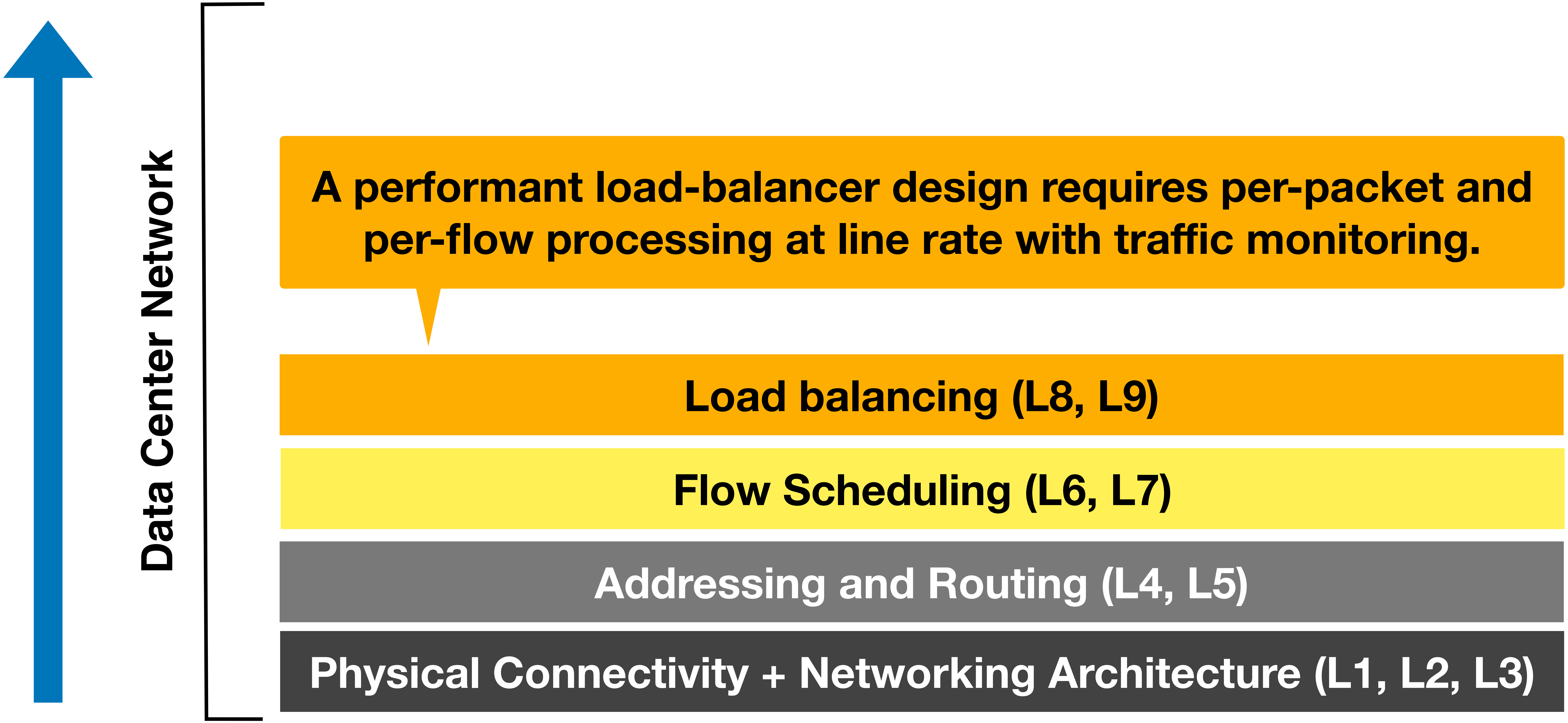
Flow scheduling requires knowing the loading status (congestion degree) of path candidates!

Flow Scheduling (L6, L7)

Addressing and Routing (L4, L5)

Physical Connectivity + Networking Architecture (L1, L2, L3)

# Where we are?



Data Center Network

A performant load-balancer design requires per-packet and per-flow processing at line rate with traffic monitoring.

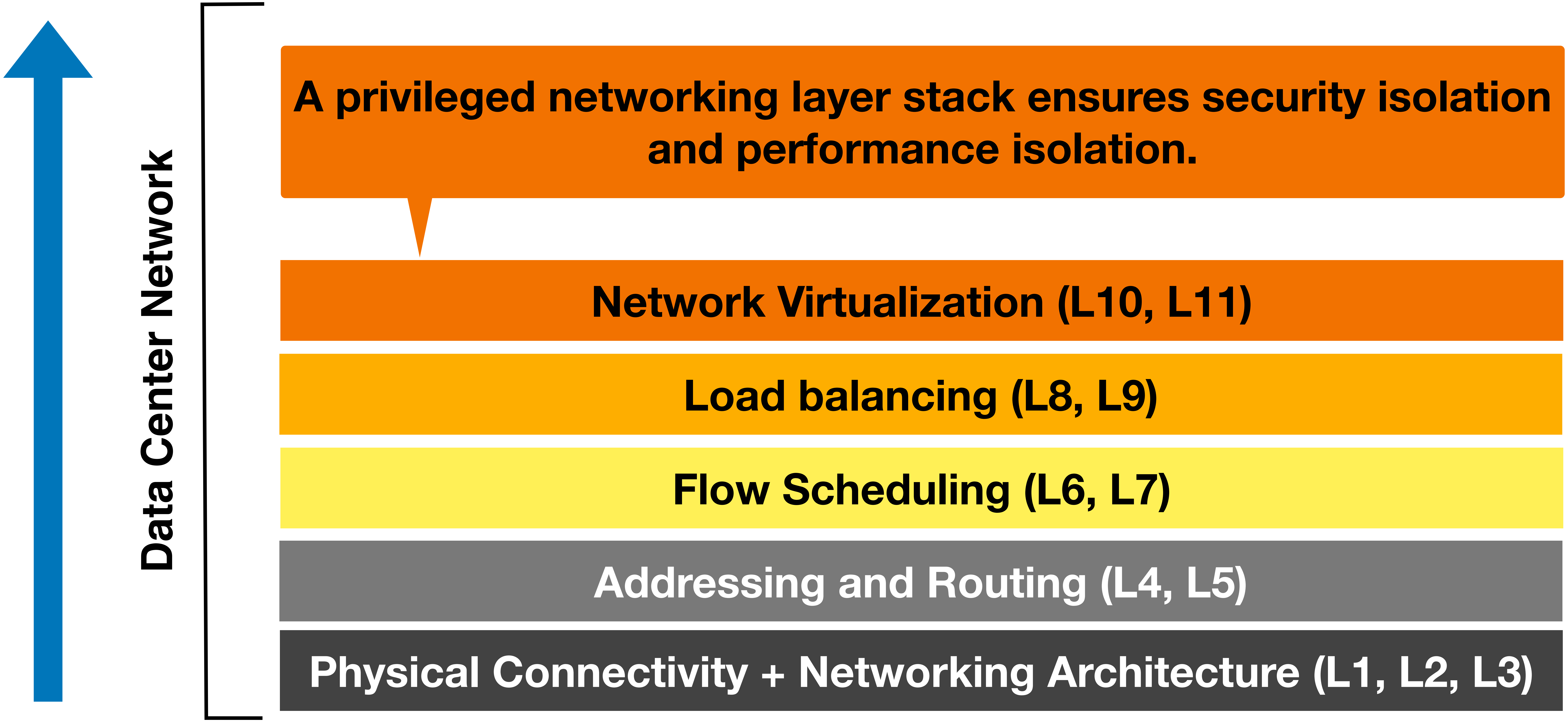
Load balancing (L8, L9)

Flow Scheduling (L6, L7)

Addressing and Routing (L4, L5)

Physical Connectivity + Networking Architecture (L1, L2, L3)

# Where we are?



Data Center Network

A privileged networking layer stack ensures security isolation and performance isolation.

Network Virtualization (L10, L11)

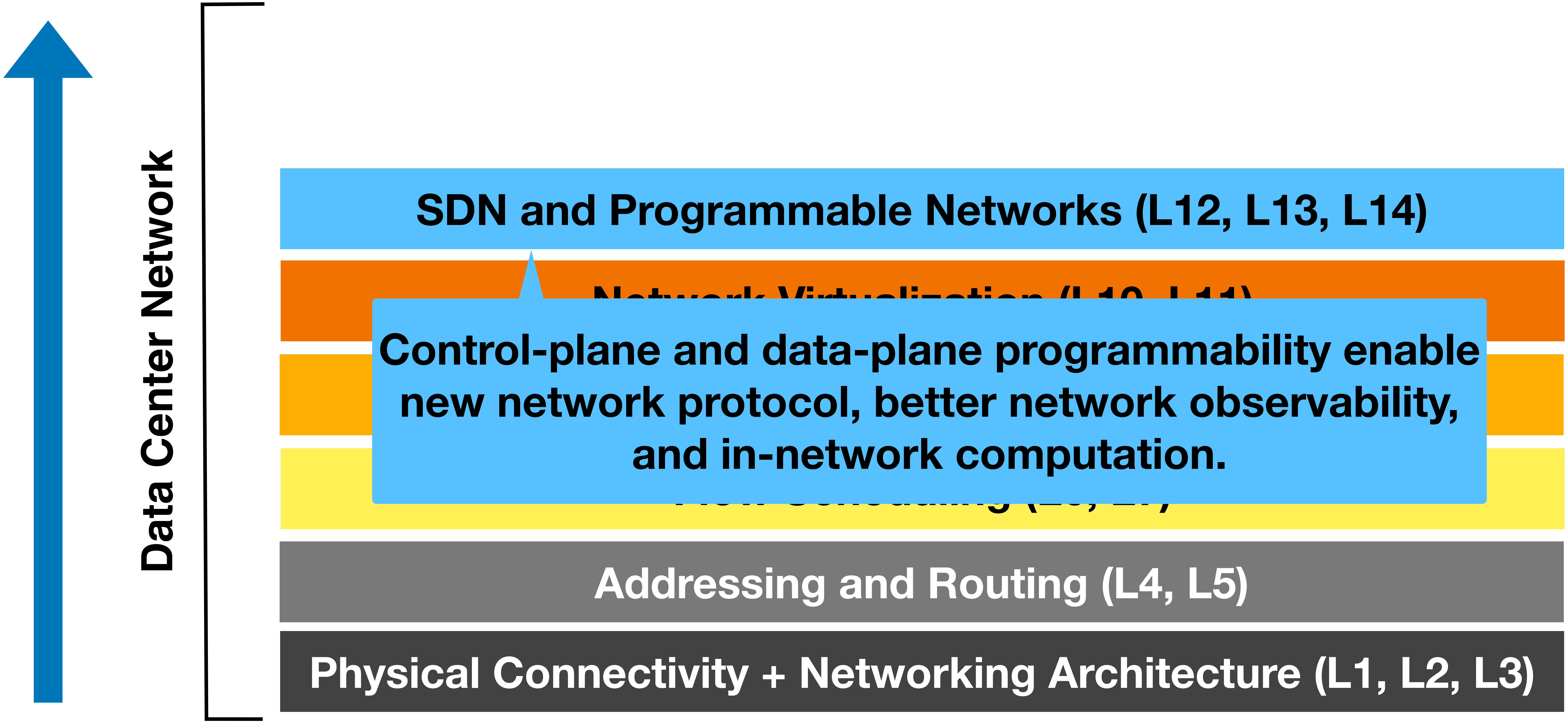
Load balancing (L8, L9)

Flow Scheduling (L6, L7)

Addressing and Routing (L4, L5)

Physical Connectivity + Networking Architecture (L1, L2, L3)

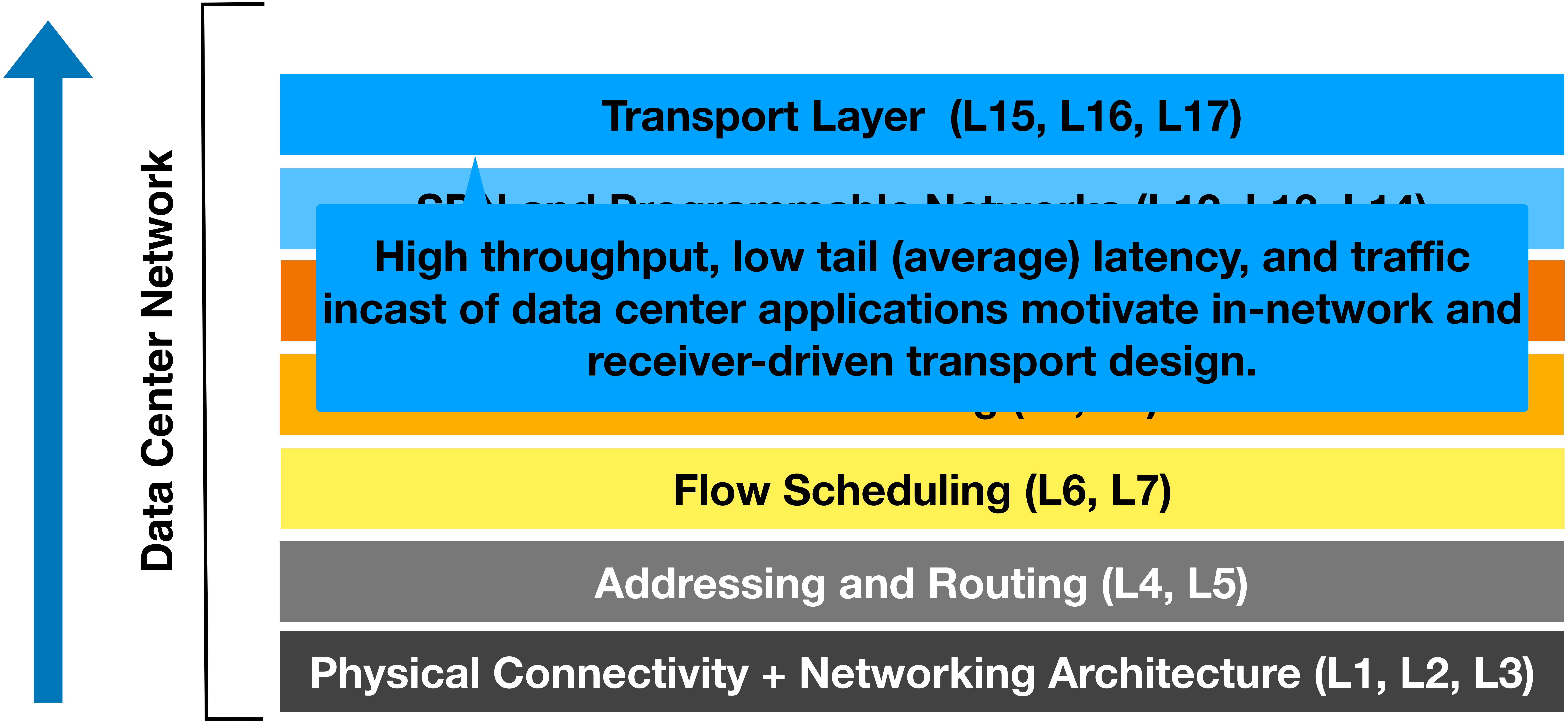
# Where we are?



Data Center Network
SDN and Programmable Networks (L12, L13, L14)
Network Virtualization (L10, L11)
Flow Scheduling (L6, L7)
Addressing and Routing (L4, L5)
Physical Connectivity + Networking Architecture (L1, L2, L3)

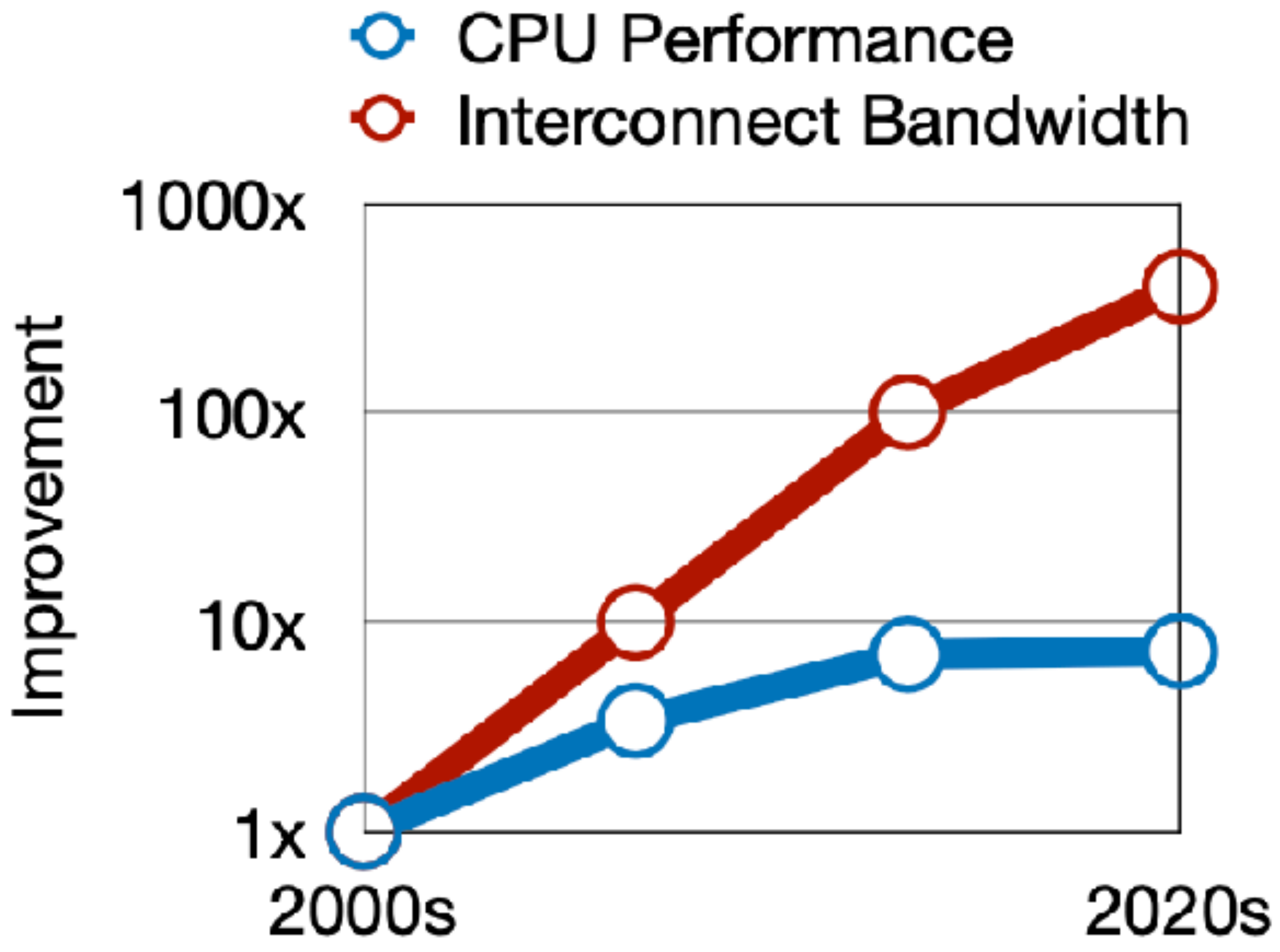
Control-plane and data-plane programmability enable new network protocol, better network observability, and in-network computation.

# Where we are?


Data Center Network
Transport Layer (L15, L16, L17)
SDN and Programmable Networks (L12, L13, L14)
Flow Scheduling (L6, L7)
Addressing and Routing (L4, L5)
Physical Connectivity + Networking Architecture (L1, L2, L3)

High throughput, low tail (average) latency, and traffic incast of data center applications motivate in-network and receiver-driven transport design.

# Problem: The software networking stack becomes the bottleneck under hardware bandwidth scaling!

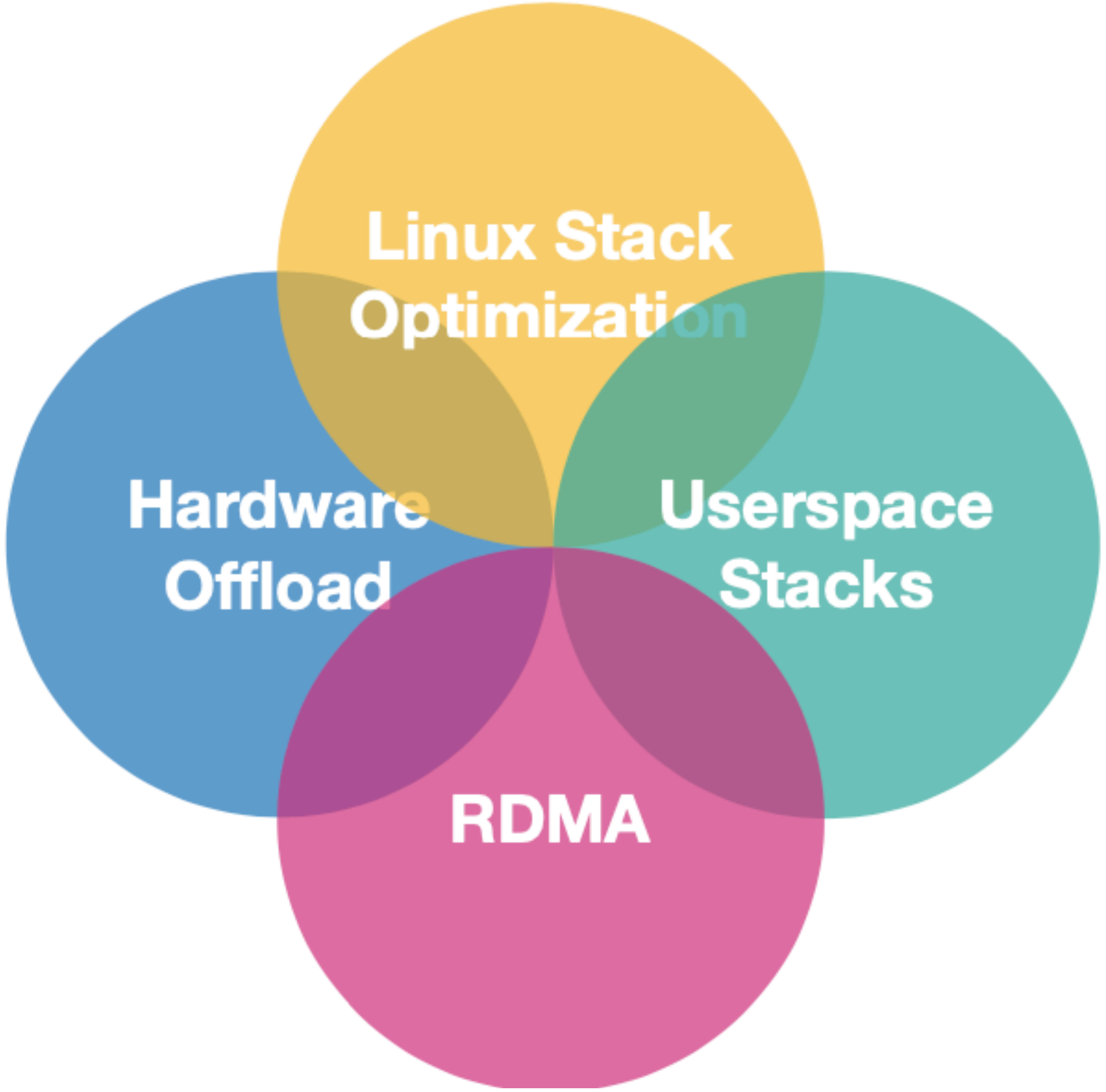


Exponentially Increasing Bandwidths.

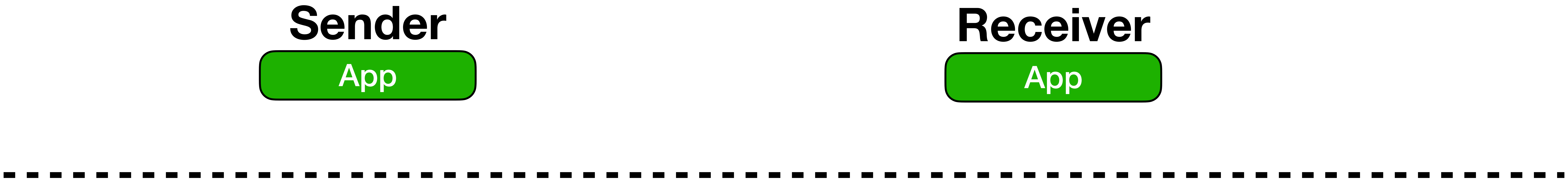
Slowdown of Moore's Law.

# Prior Solutions

- Lack of systematic understanding



# Linux Network Stack Data Path Walk-Through

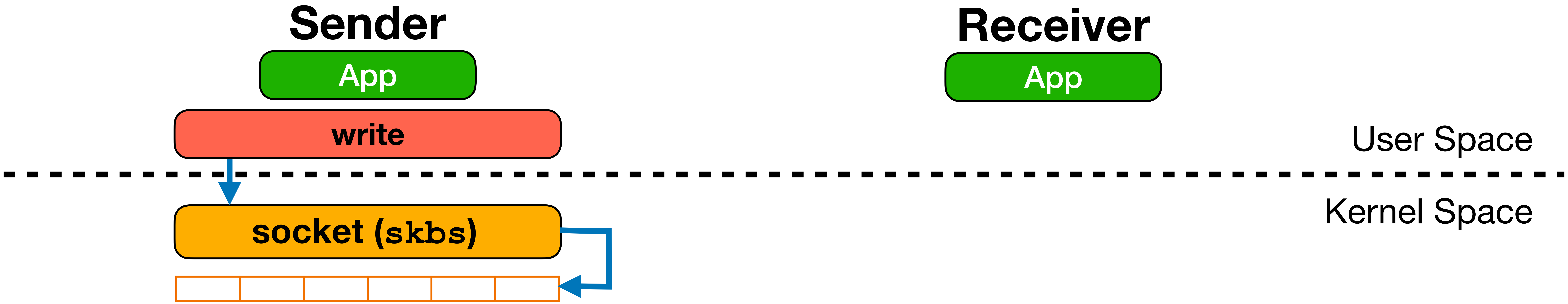


# Linux Network Stack Data Path Walk-Through

Sender
App
write
Receiver
App

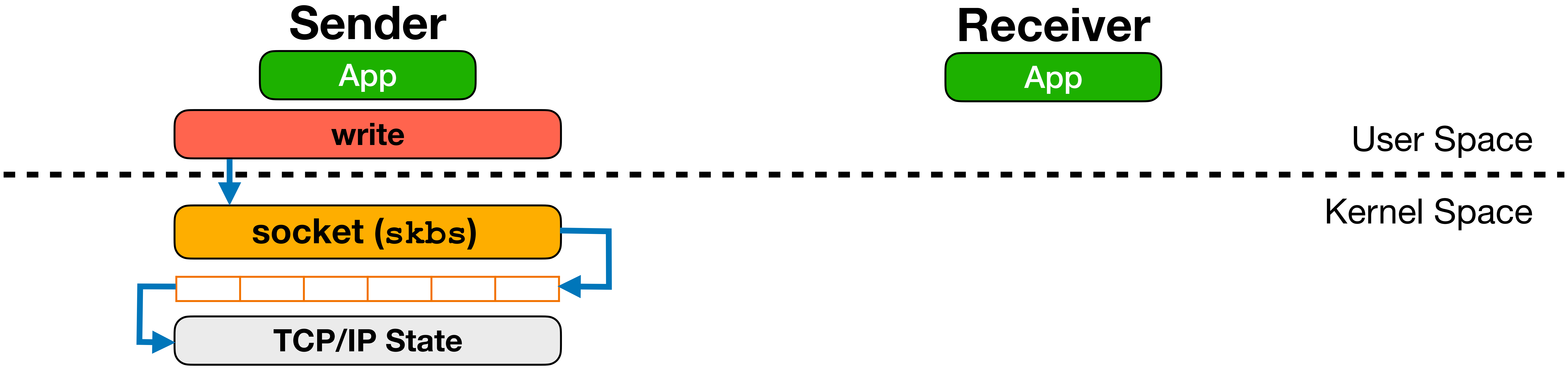
- Sender #1: Apps execute a `write` system call

# Linux Network Stack Data Path Walk-Through



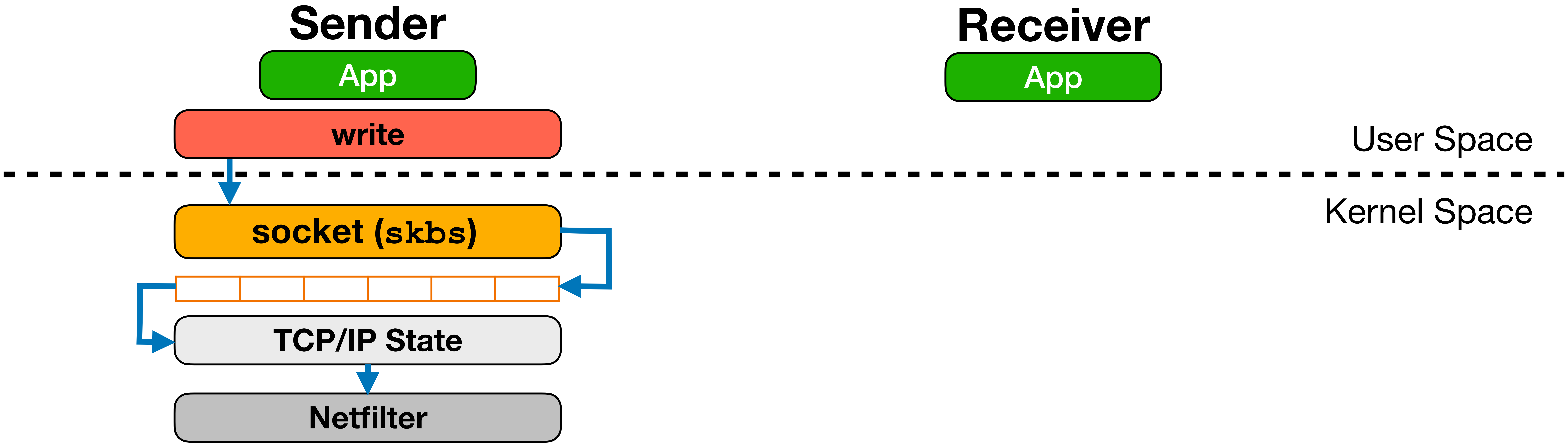
- Sender #2: The kernel initializes socket buffers

# Linux Network Stack Data Path Walk-Through



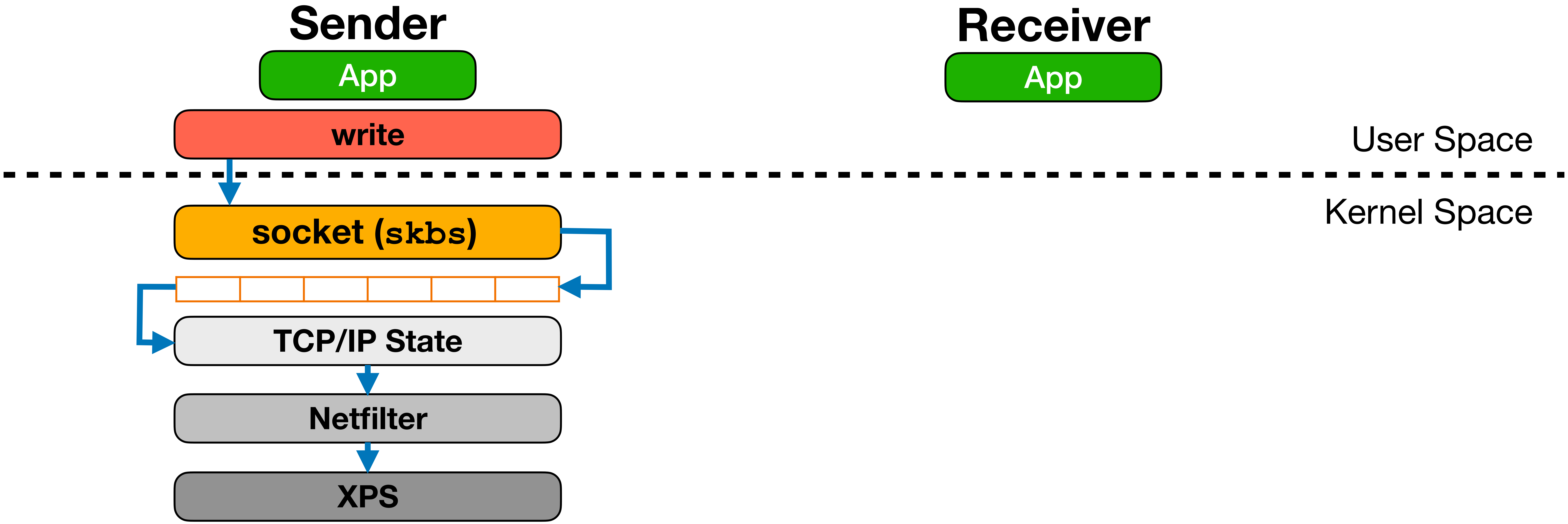
- Sender #3: `skbs` are processed by the TCP/IP layer

# Linux Network Stack Data Path Walk-Through



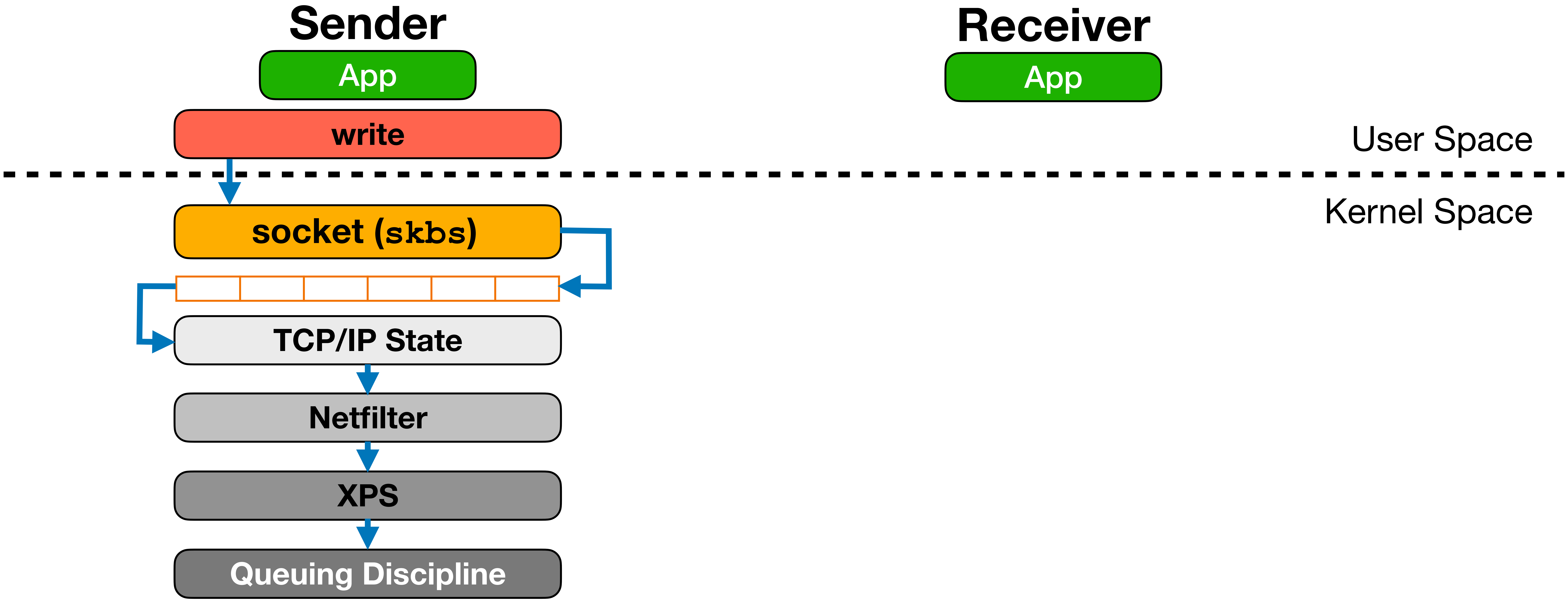
- Sender #4: Packets (`skbs`) are processed by customized networking functions
- Netfilter: a framework provided by the Linux kernel, allowing registering callback functions for packet handling

# Linux Network Stack Data Path Walk-Through



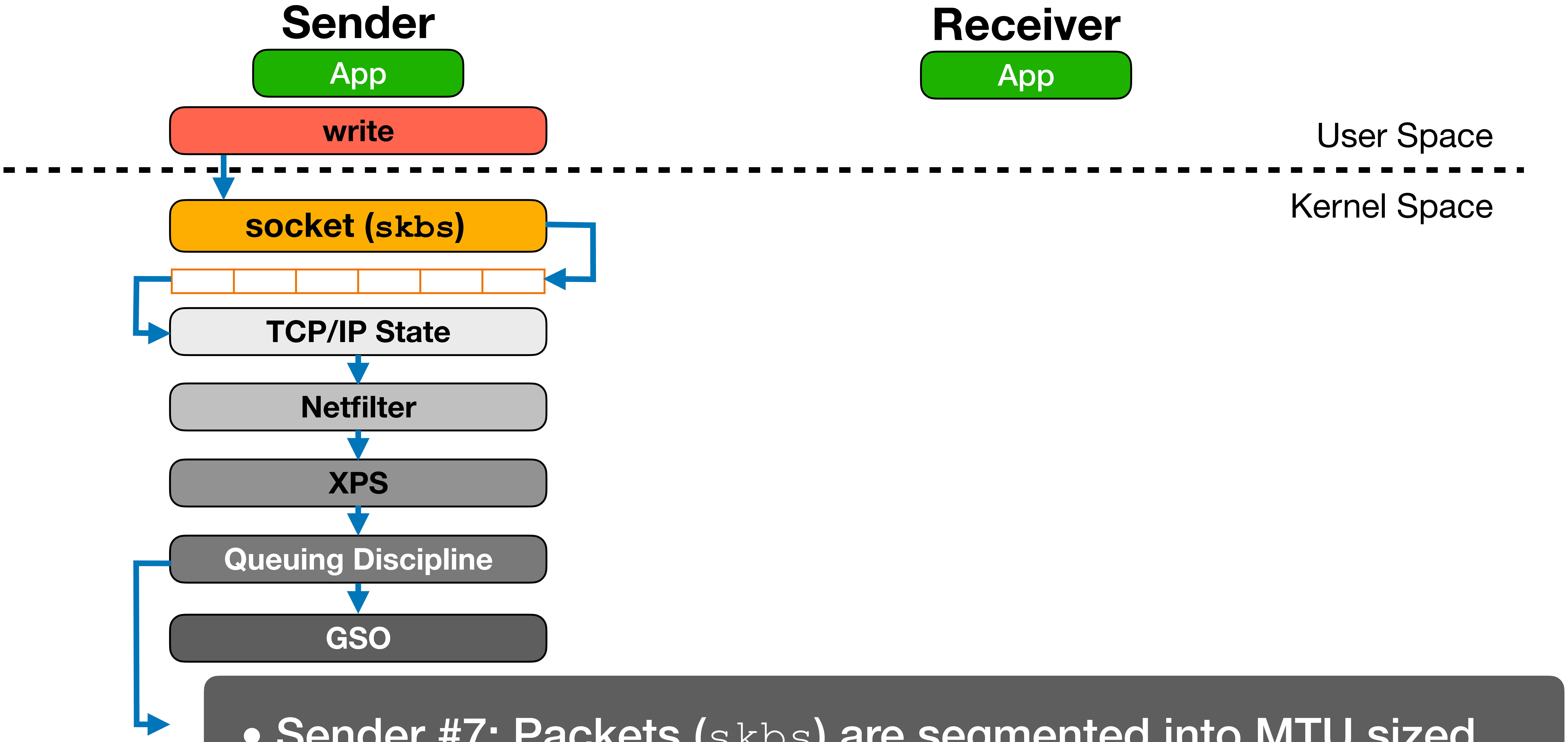
- Sender #5: Packets (`skbs`) are orchestrated by transmit-side traffic steering (XPS)
- Two approaches: using CPUs map or receive queue map

# Linux Network Stack Data Path Walk-Through



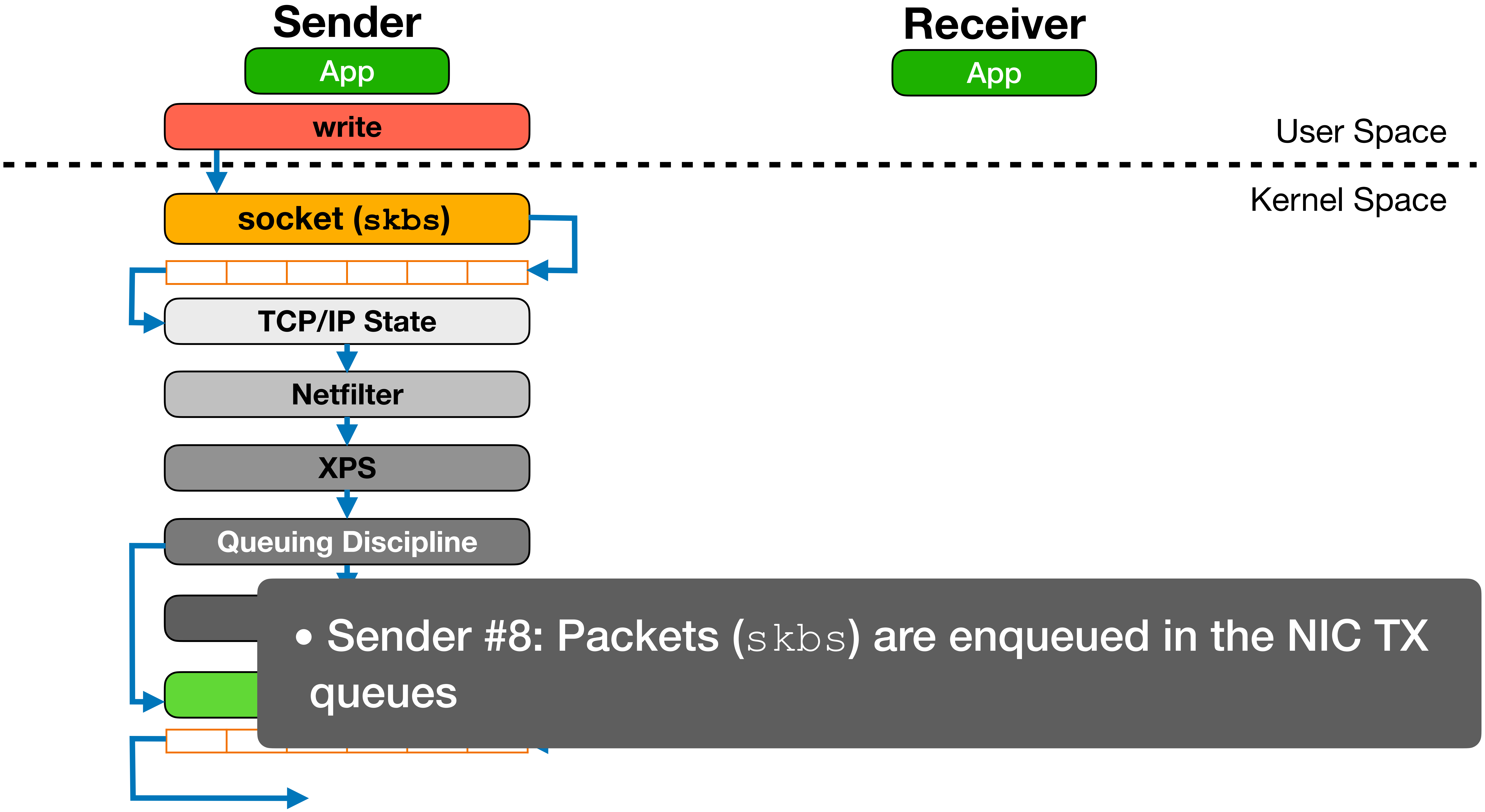
- Sender #6: Packets (`skbs`) are shaped via `qdisc`
- Rate limiting, FIFO, priority

# Linux Network Stack Data Path Walk-Through



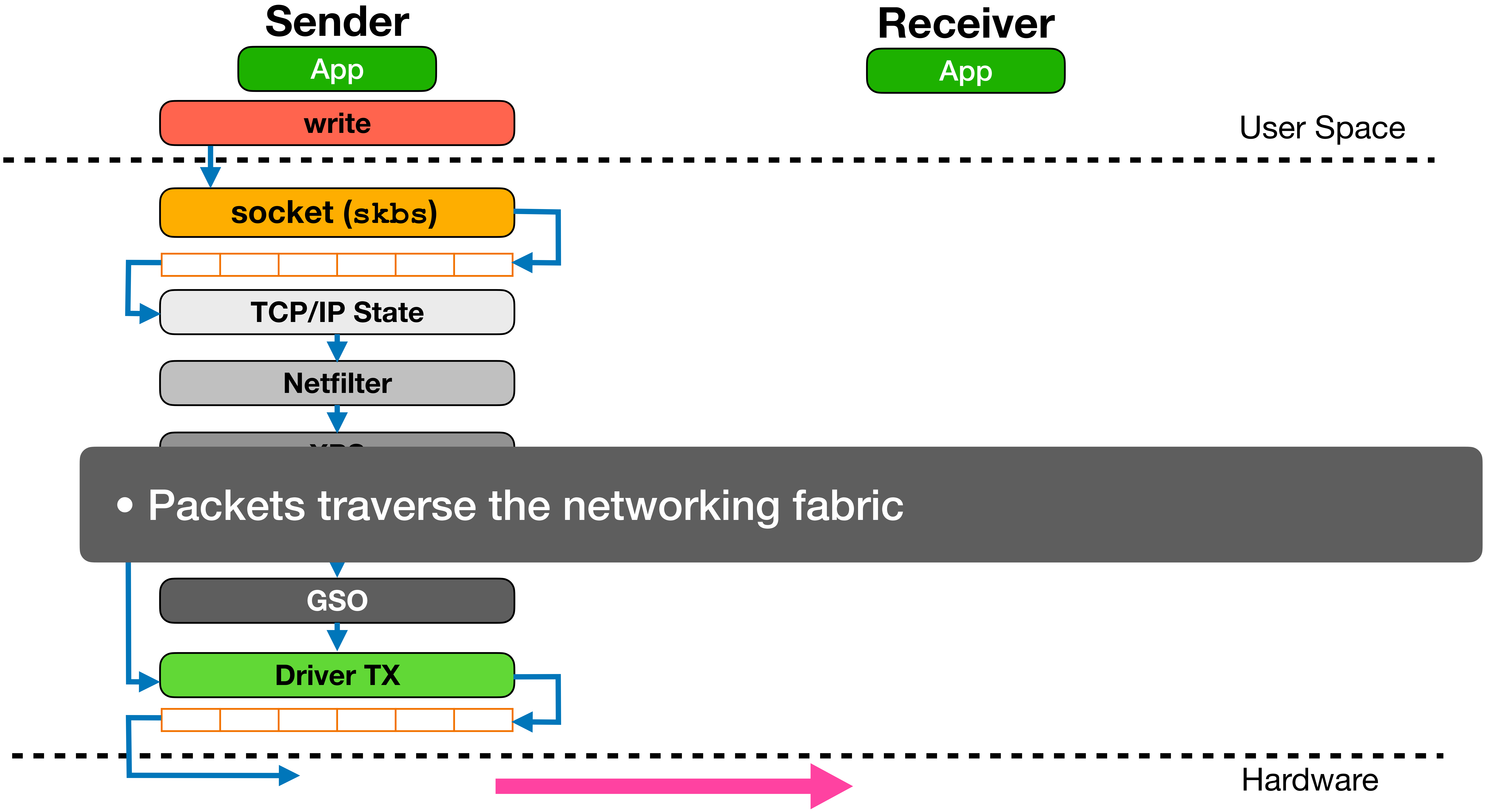
- Sender #7: Packets (skbs) are segmented into MTU sized chunks

# Linux Network Stack Data Path Walk-Through



- Sender #8: Packets (`skbs`) are enqueued in the NIC TX queues

# Linux Network Stack Data Path Walk-Through



Sender

Receiver

App

App

write

User Space

socket (skbs)

TCP/IP State

Netfilter

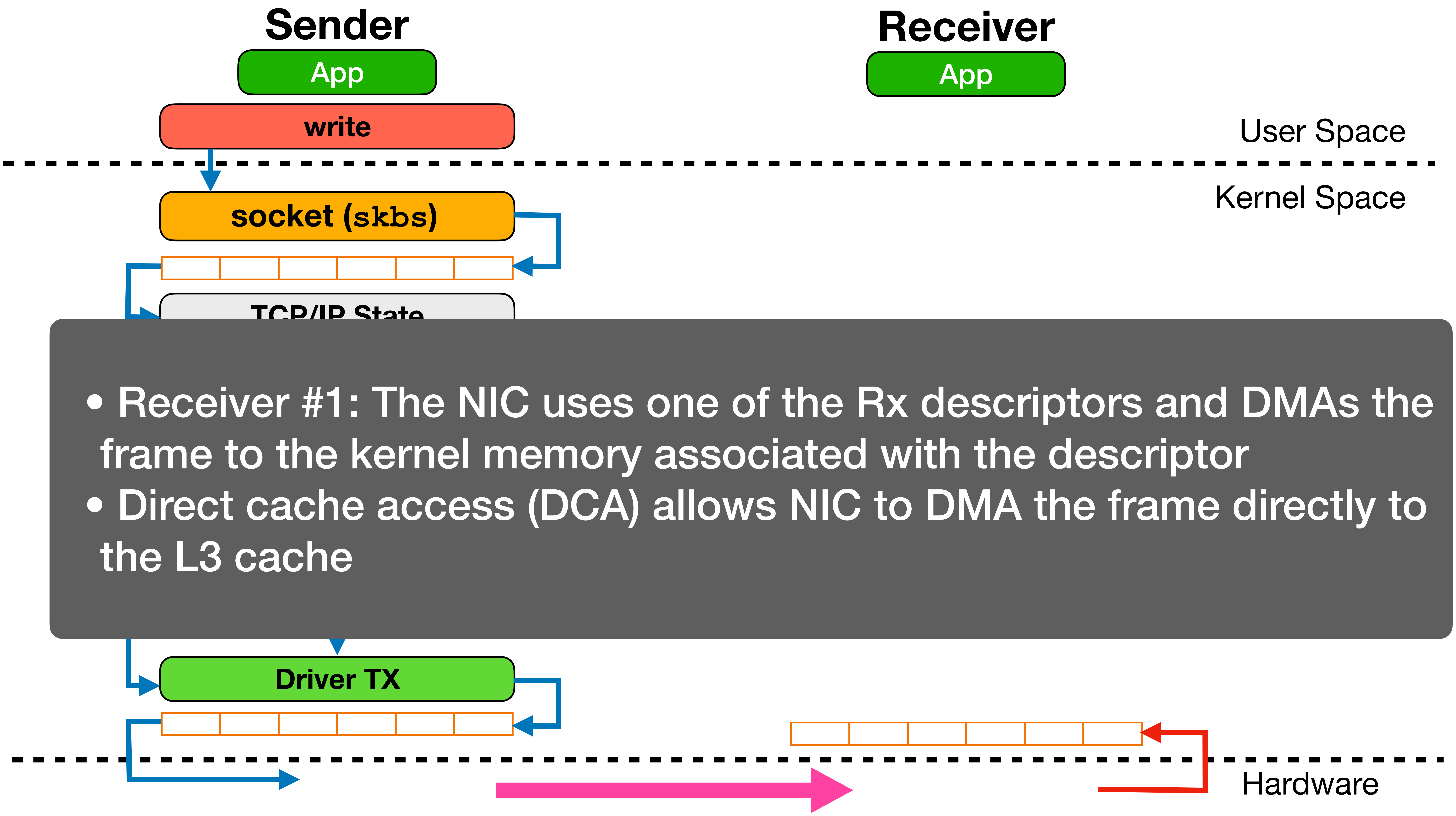
GSO

Driver TX

Hardware

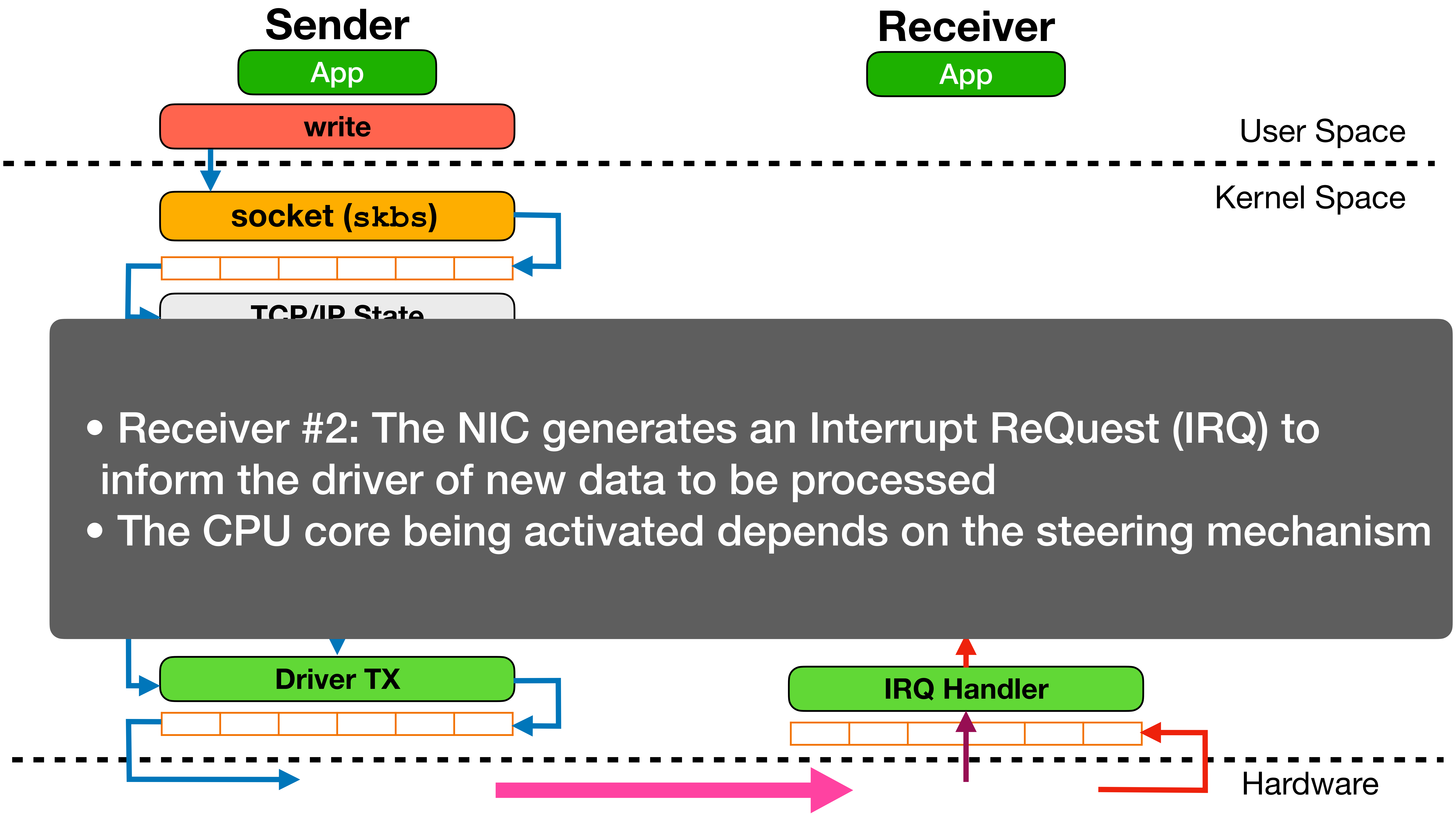
- Packets traverse the networking fabric

# Linux Network Stack Data Path Walk-Through



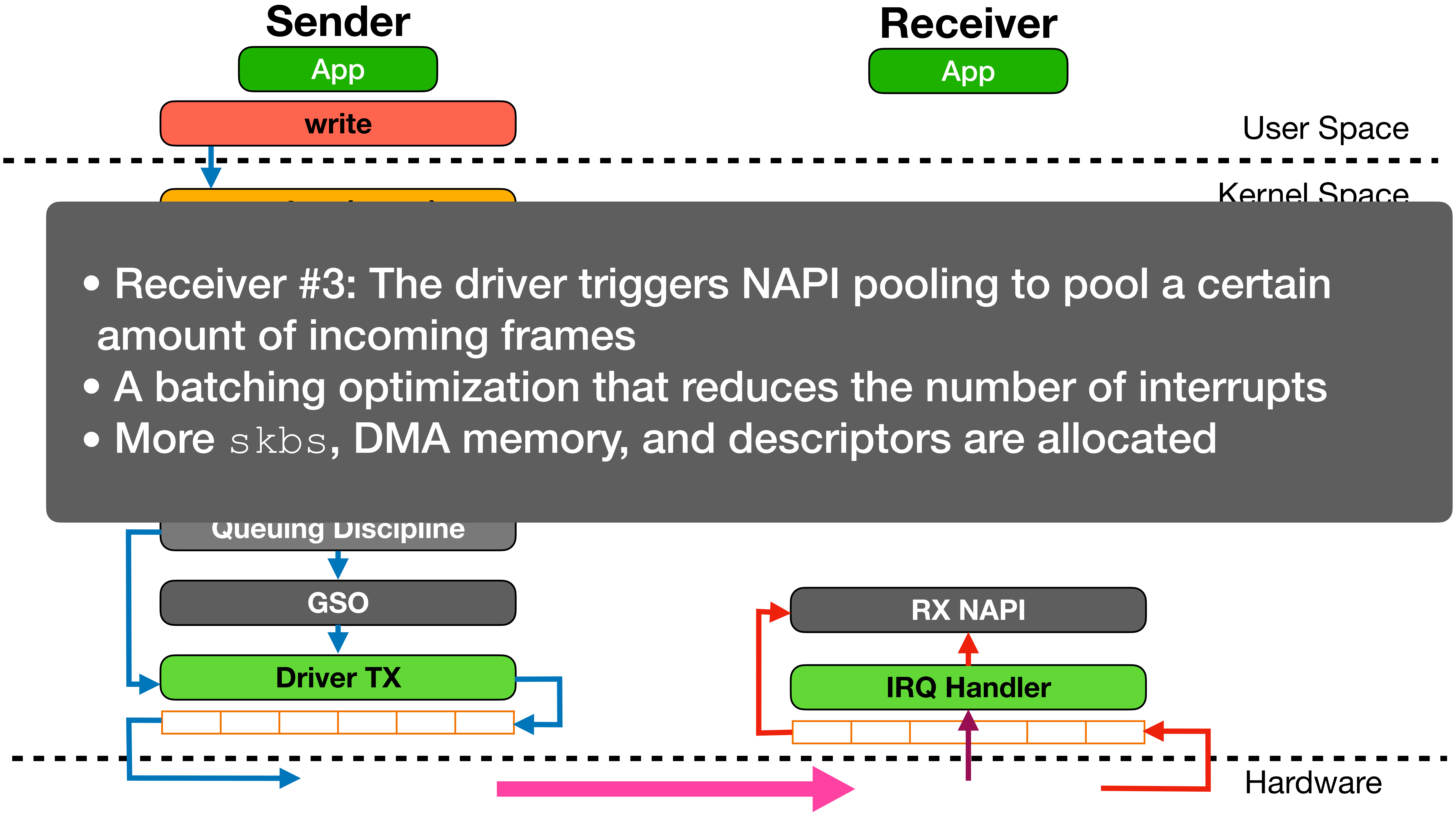
- Receiver #1: The NIC uses one of the Rx descriptors and DMAs the frame to the kernel memory associated with the descriptor
- Direct cache access (DCA) allows NIC to DMA the frame directly to the L3 cache

# Linux Network Stack Data Path Walk-Through



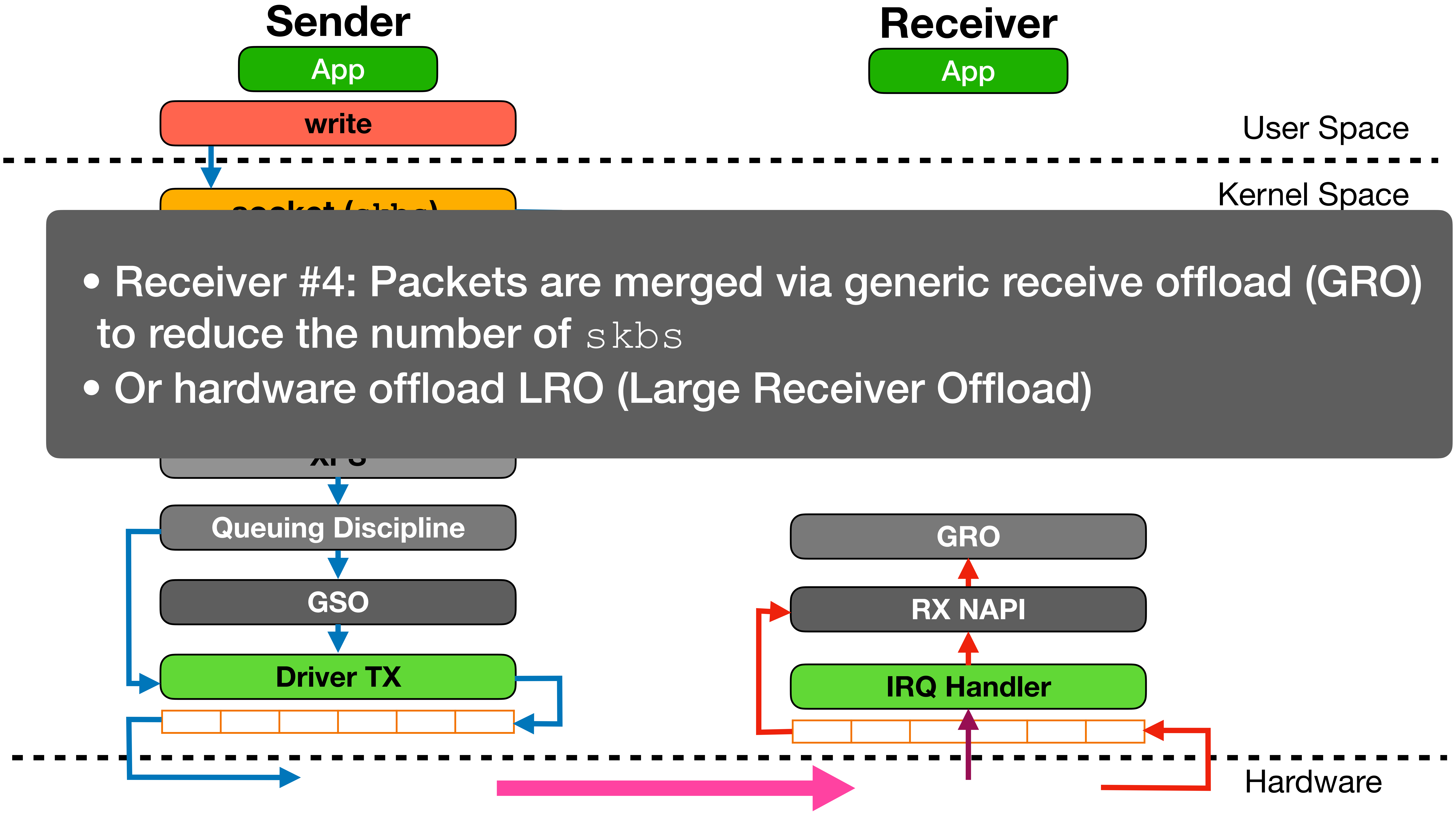
- Receiver #2: The NIC generates an Interrupt ReQuest (IRQ) to inform the driver of new data to be processed
- The CPU core being activated depends on the steering mechanism

# Linux Network Stack Data Path Walk-Through



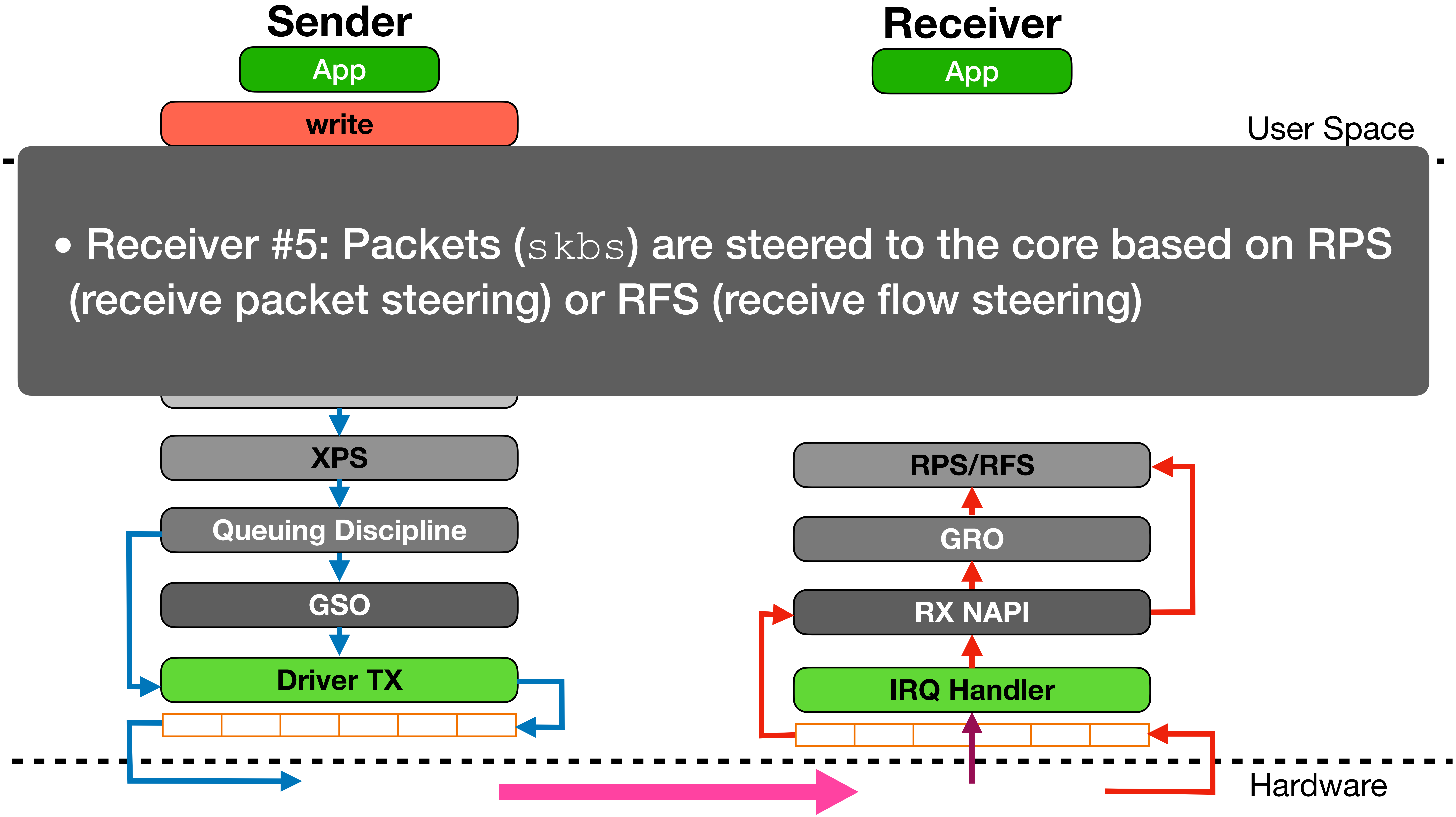
- Receiver #3: The driver triggers NAPI pooling to pool a certain amount of incoming frames
- A batching optimization that reduces the number of interrupts
- More `skbs`, DMA memory, and descriptors are allocated

# Linux Network Stack Data Path Walk-Through



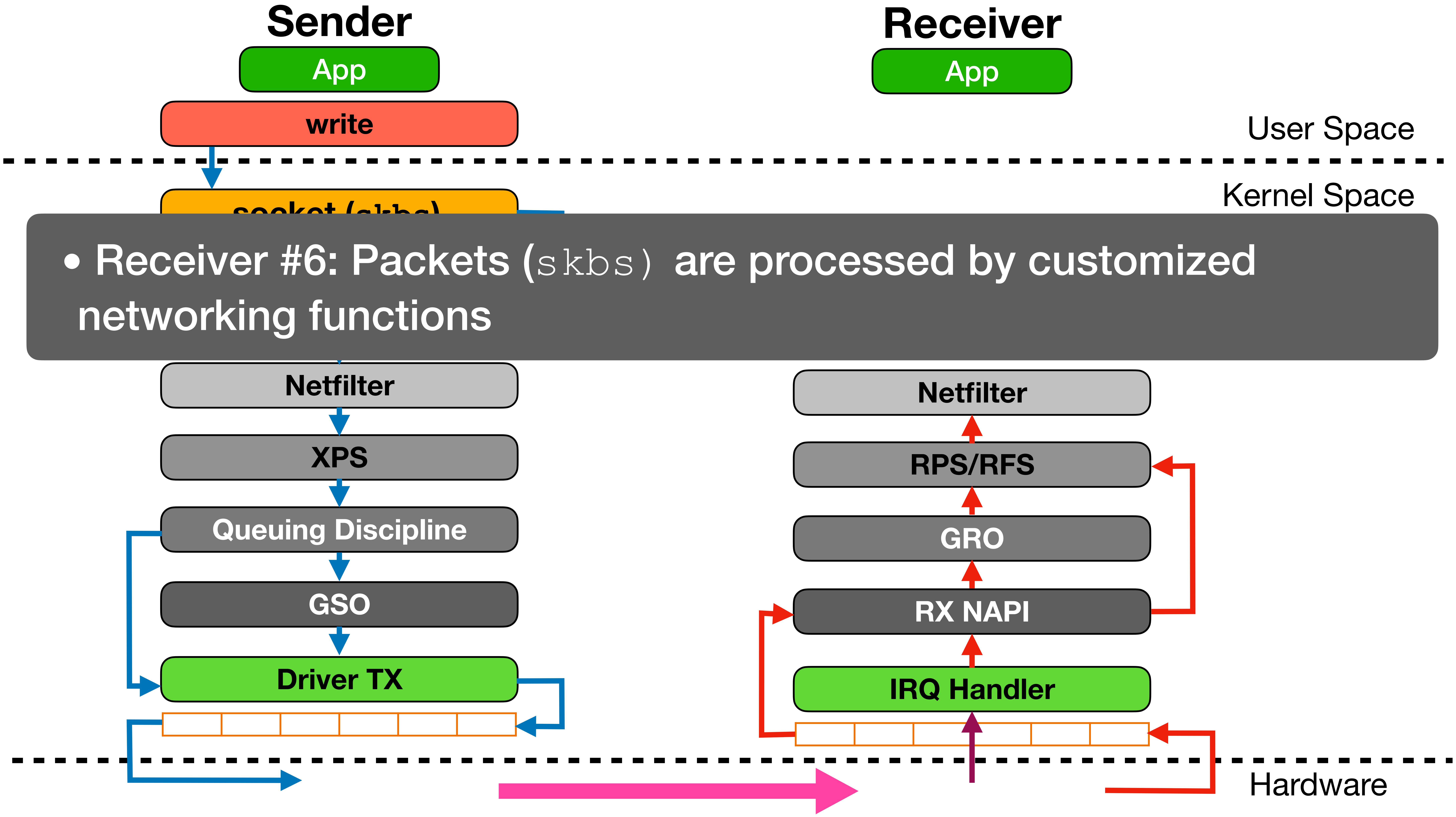
- Receiver #4: Packets are merged via generic receive offload (GRO) to reduce the number of `skbs`
- Or hardware offload LRO (Large Receiver Offload)

# Linux Network Stack Data Path Walk-Through



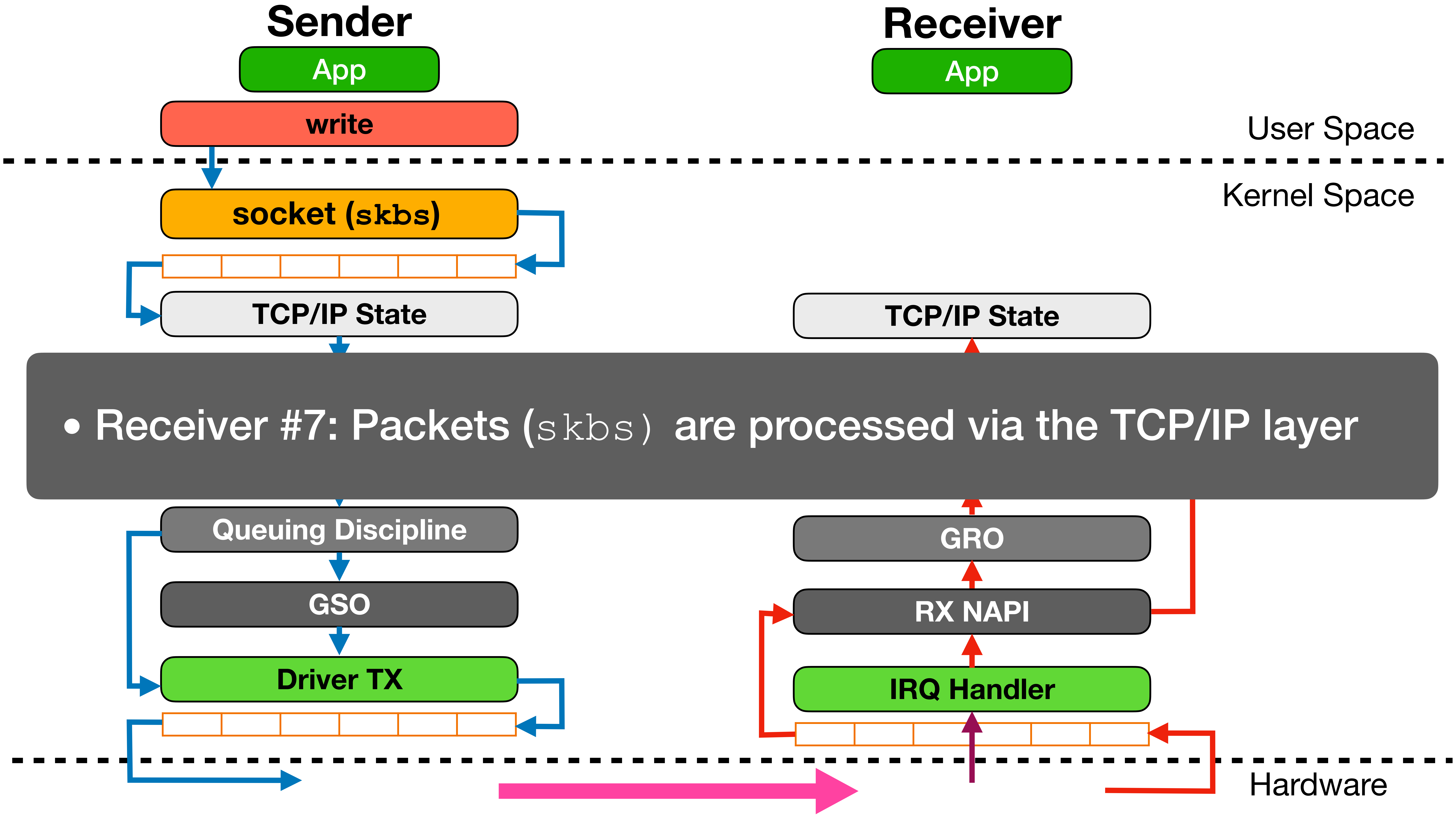
- Receiver #5: Packets (`skbs`) are steered to the core based on RPS (receive packet steering) or RFS (receive flow steering)

# Linux Network Stack Data Path Walk-Through



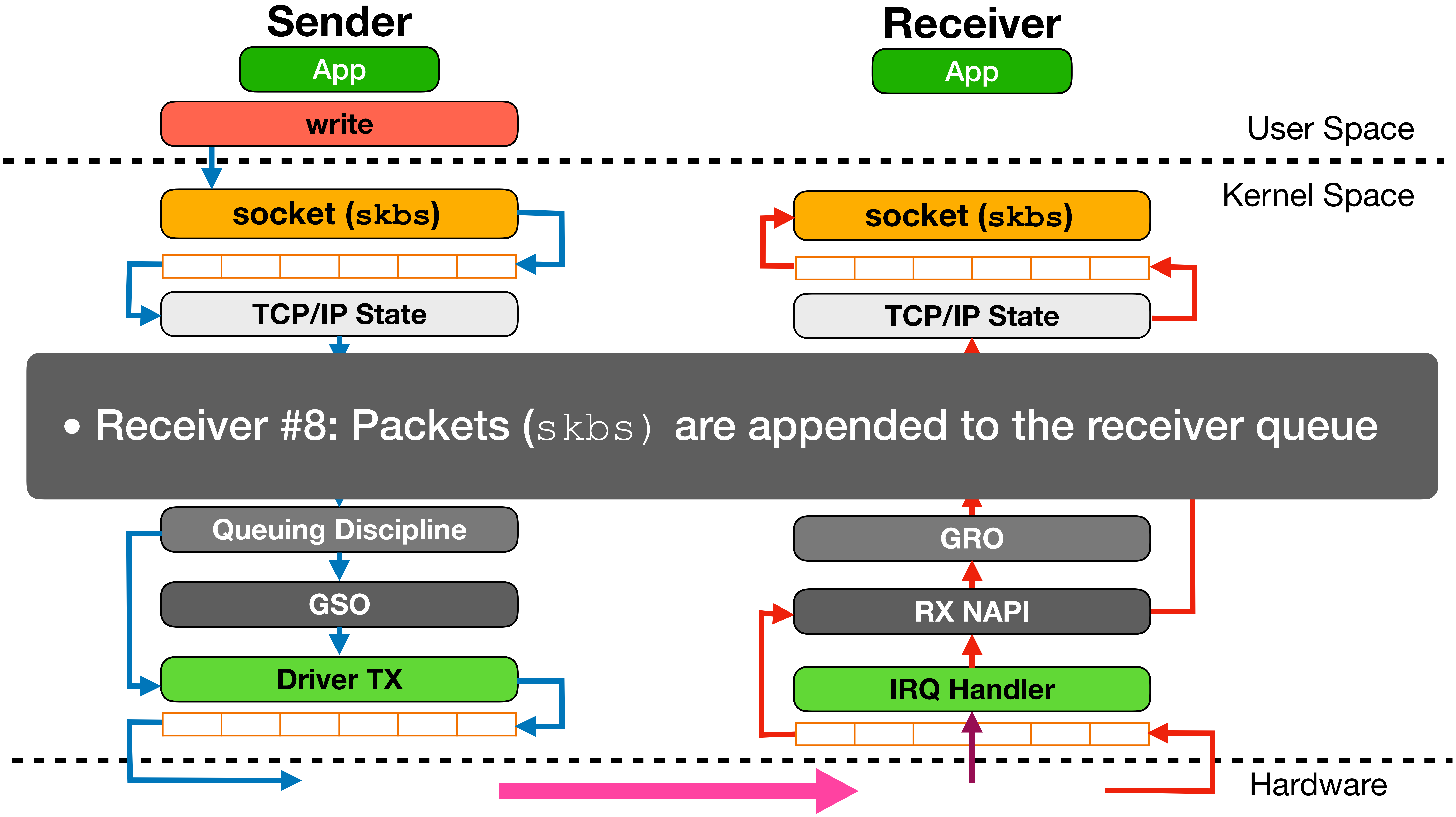
- Receiver #6: Packets (`skbs`) are processed by customized networking functions

# Linux Network Stack Data Path Walk-Through



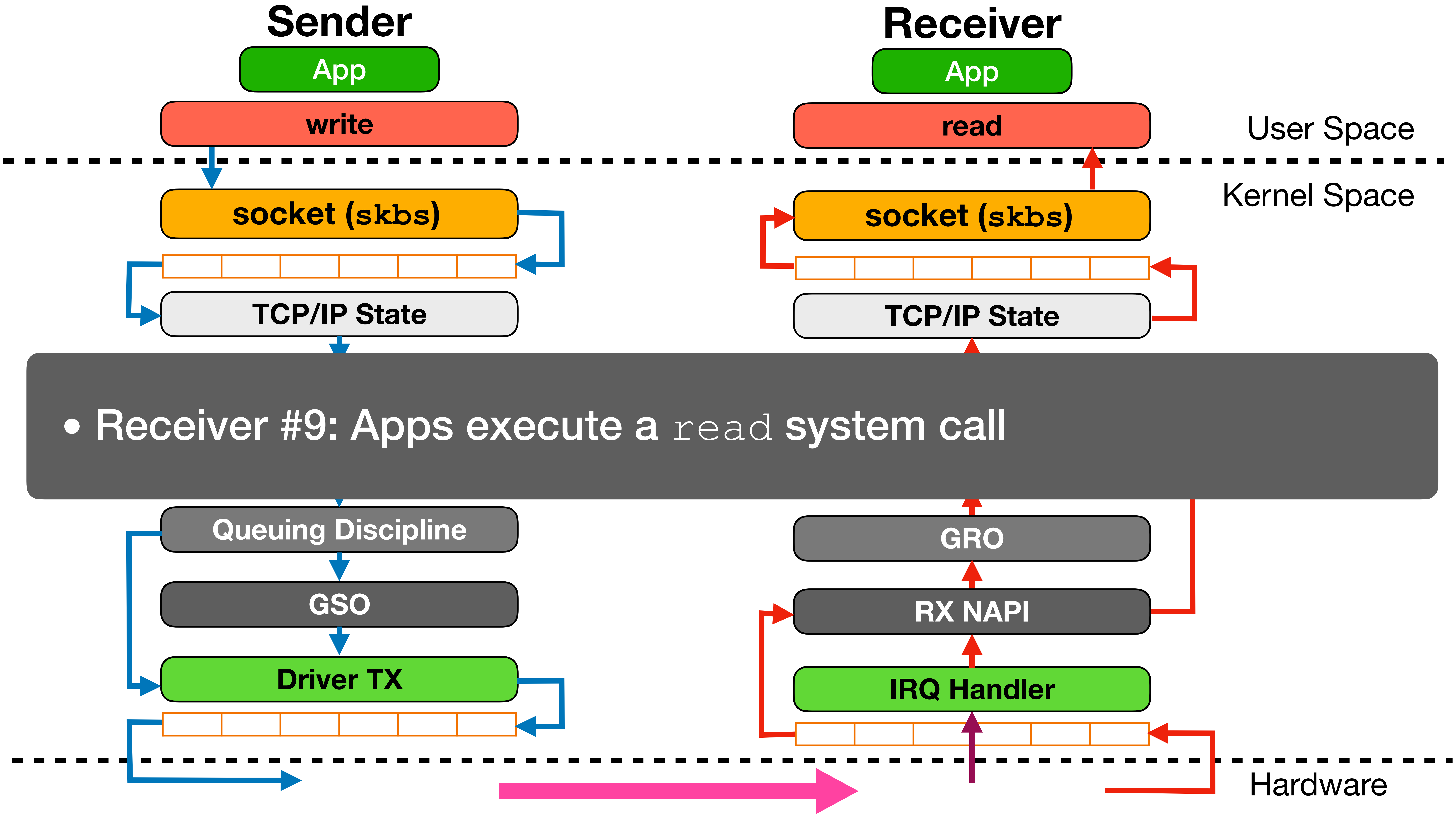
- Receiver #7: Packets (`skbs`) are processed via the TCP/IP layer

# Linux Network Stack Data Path Walk-Through



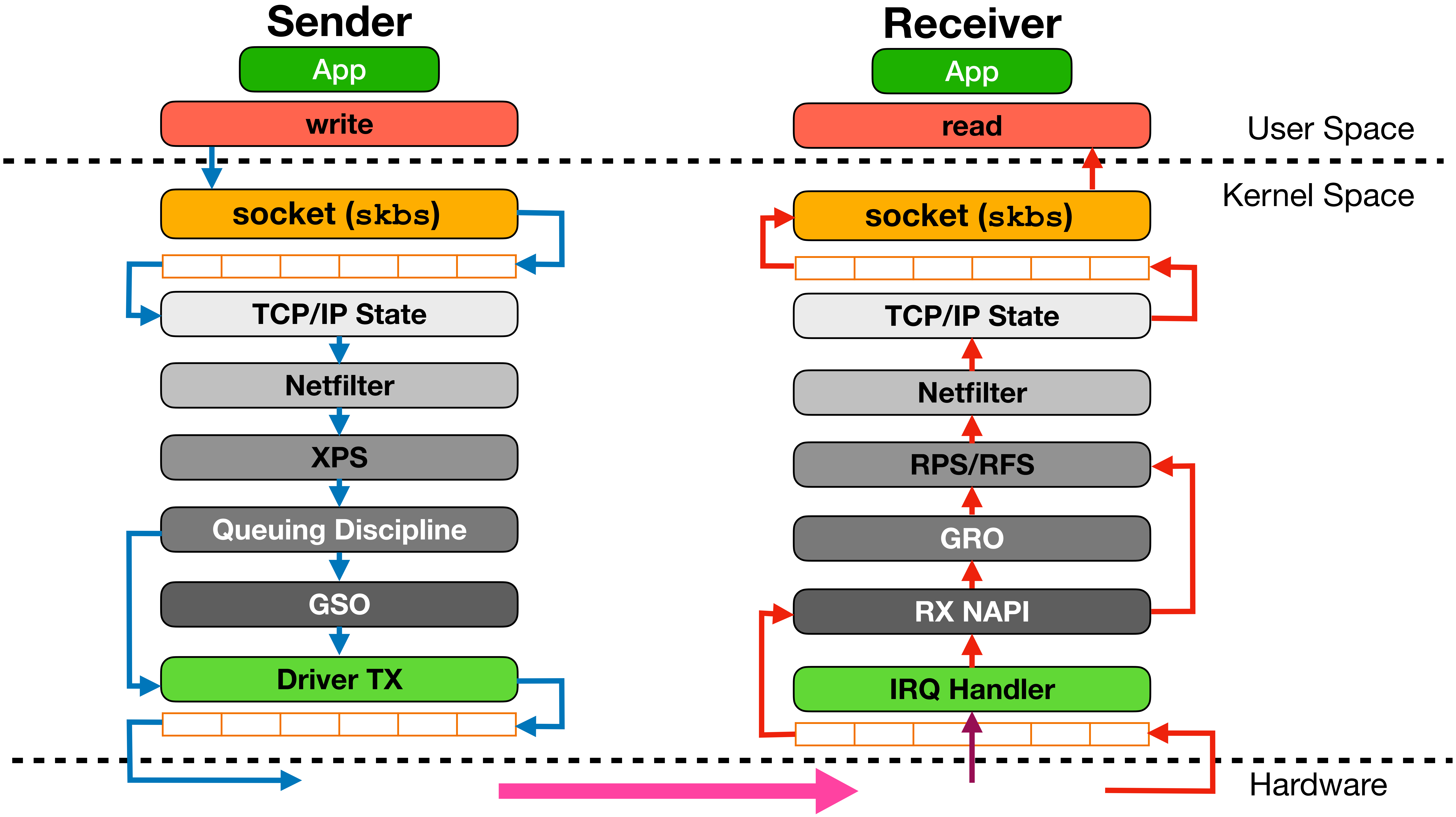
- Receiver #8: Packets (`skbs`) are appended to the receiver queue

# Linux Network Stack Data Path Walk-Through



- Receiver #9: Apps execute a `read` system call

# Linux Network Stack Data Path Walk-Through



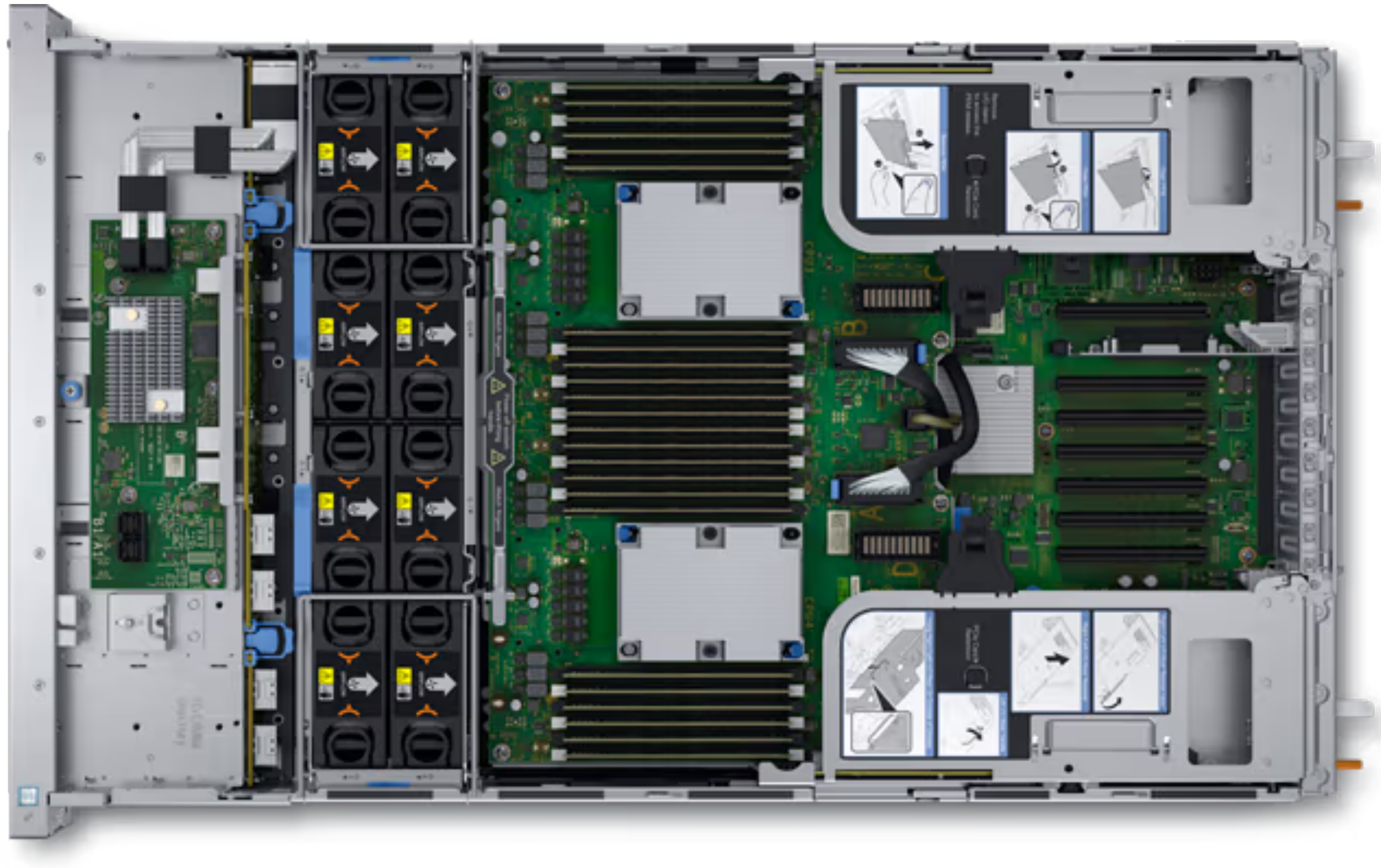
# How can we design the experimental methodology?

# Experimental Methodology

- #1: What is the hardware testbed?

# Experimental Methodology

- #1: What is the hardware testbed?



- 4-socket
- Each socket:
  - Intel Xeon Gold 6-core 6128 3.4GHz CPU
  - L1/L2/L3: 32KB/1MB/20MB
- 256GB
- 100G Mellanox CX5 NIC

# Experimental Methodology

- #1: What is the hardware testbed?

- #2: What is the software stack?

# Experimental Methodology

- #1: What is the hardware testbed?

- #2: What is the software stack?

- Ubuntu 16.04 + Kernel 5.4.43
- Mellanox OFED driver
- iPerf/netperf

# Experimental Methodology

- #1: What is the hardware testbed?
- #2: What is the software stack?
- #3: What are the performance metrics?

# Experimental Methodology

- #1: What is the hardware testbed?

- #2: What is the software stack?

- #3: What are the performance metrics?

- Throughput/Latency
- CPU Utilization
- LLC Cache Miss Rate

# Experimental Methodology

- #1: What is the hardware testbed?

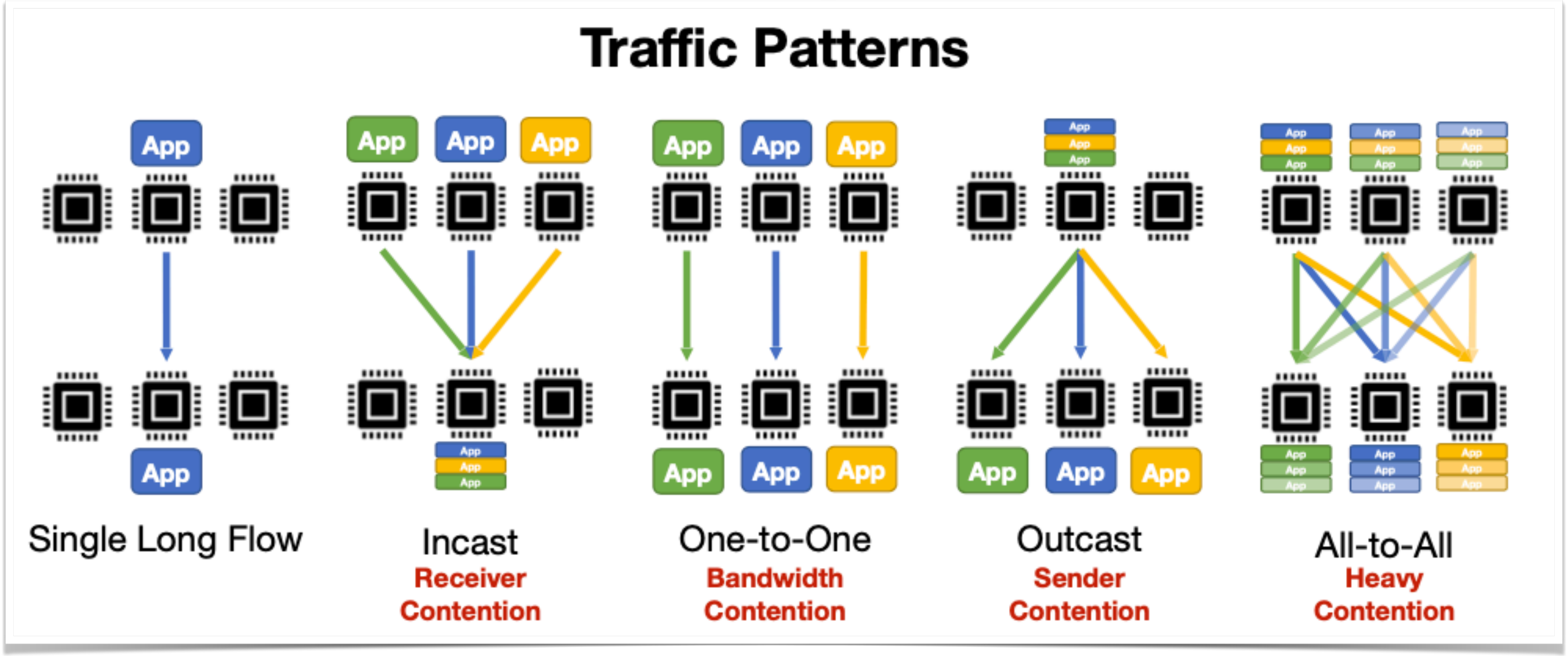
- #2: What is the software stack?

- #3: What are the performance metrics?

- #4: What is the experimental setup?

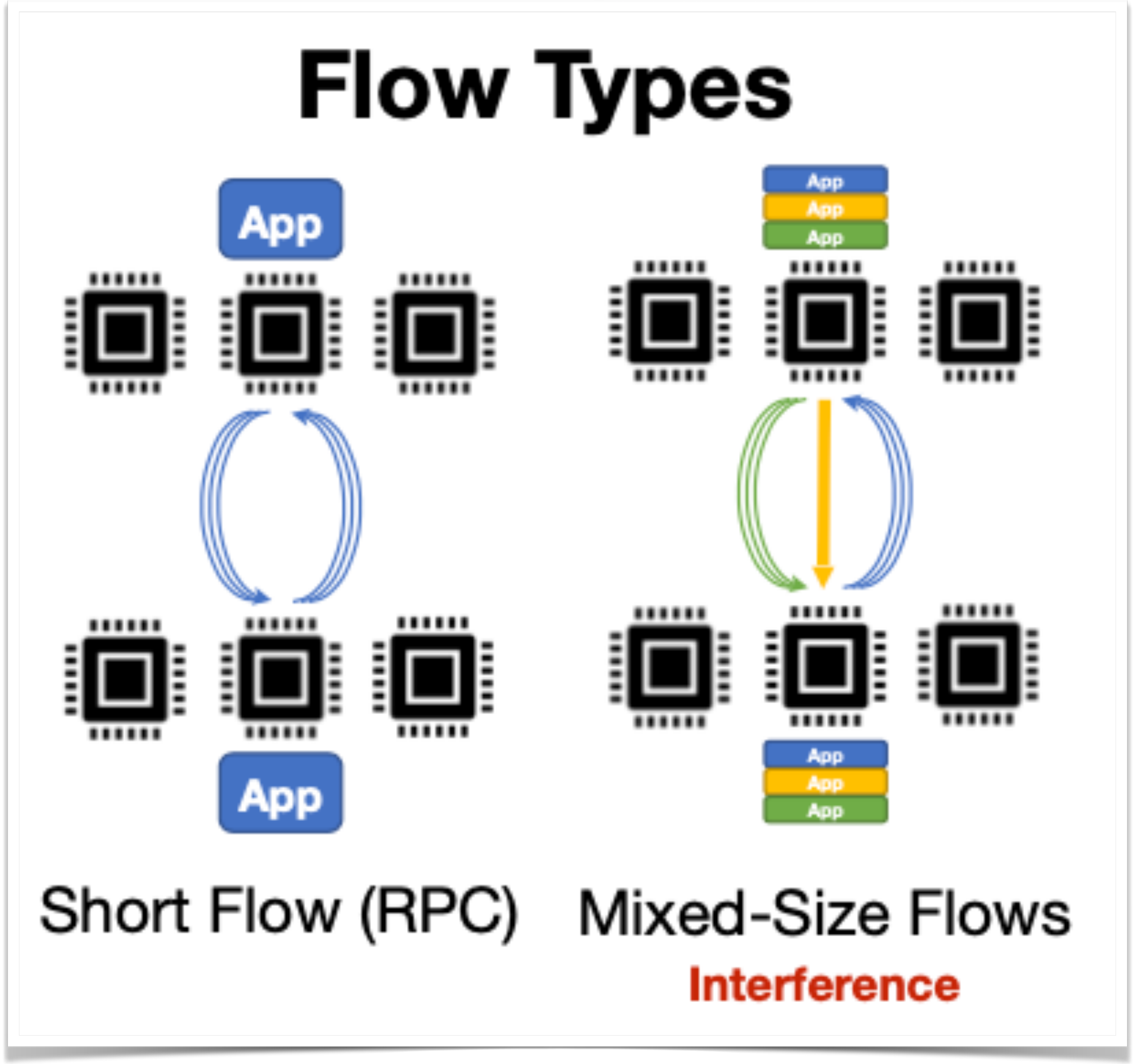
# Experimental Setup

- Many dimensions
  - Knob1: traffic patterns



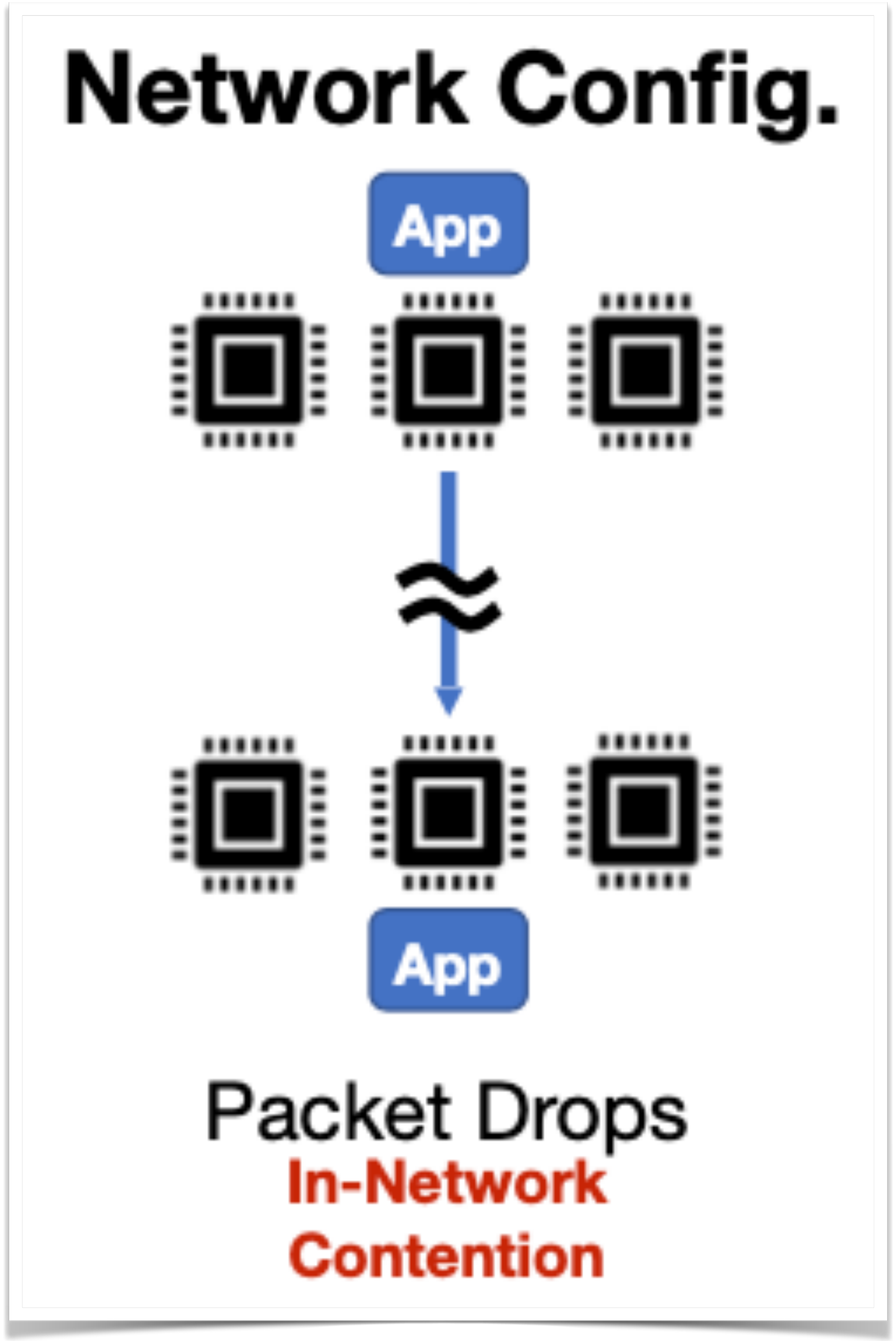
# Experimental Setup

- Many dimensions
  - Knob1: traffic patterns
  - Knob2: flow types



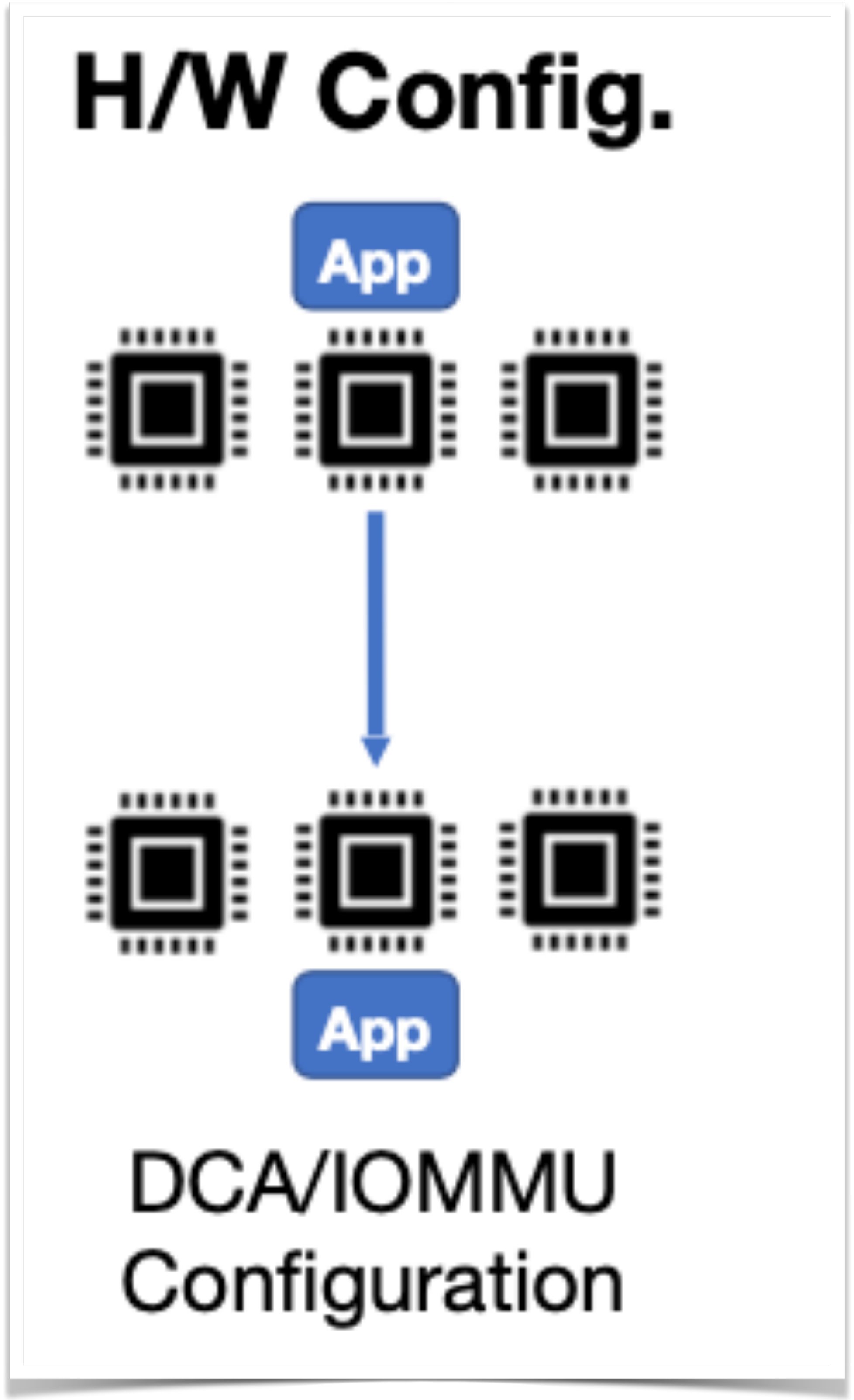
# Experimental Setup

- Many dimensions
  - Knob1: traffic patterns
  - Knob2: flow types
  - Knob3: network configurations



# Experimental Setup

- Many dimensions
  - Knob1: traffic patterns
  - Knob2: flow types
  - Knob3: network configurations
  - Knob4: H/W configurations



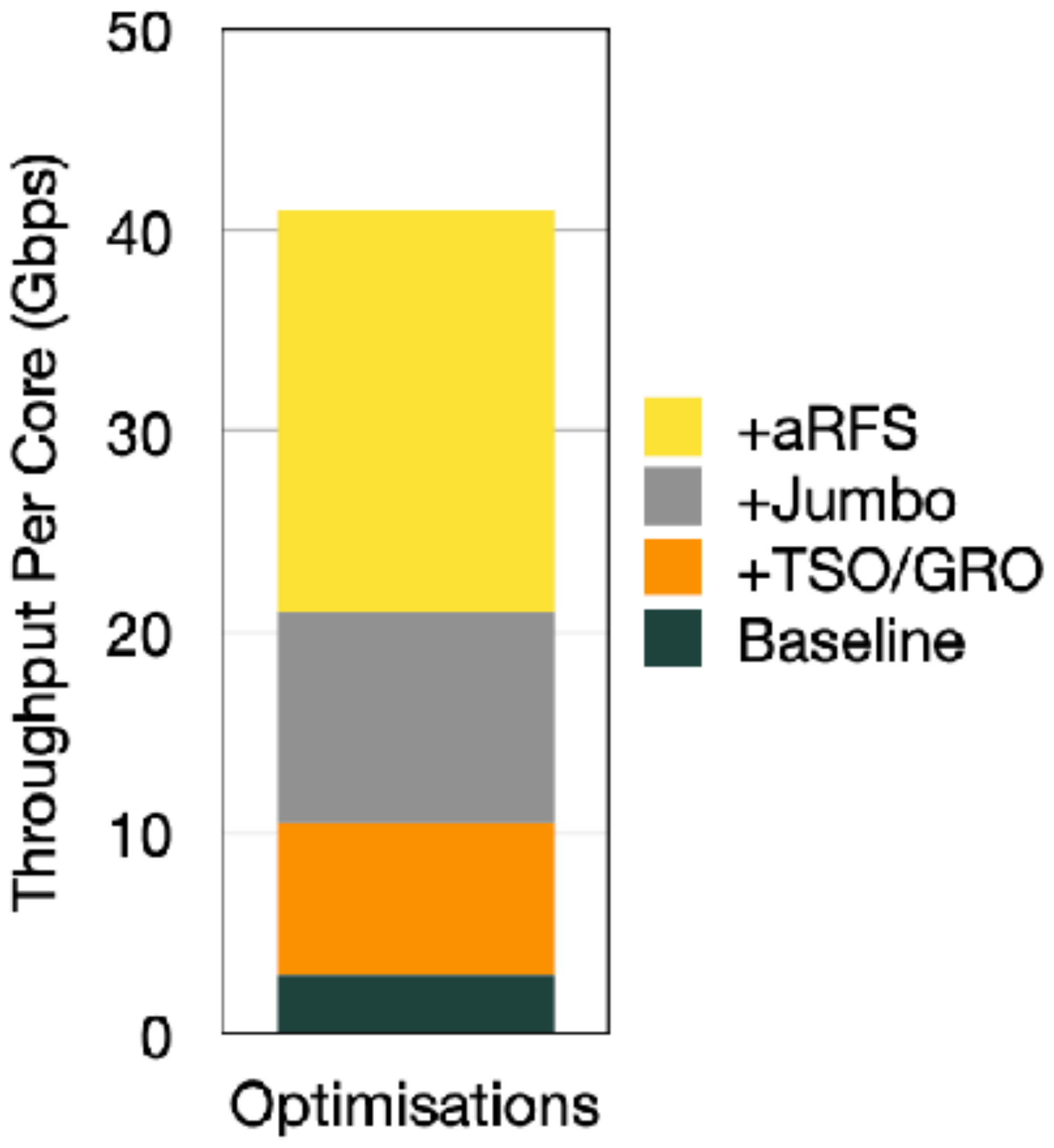
# What are the new insights?

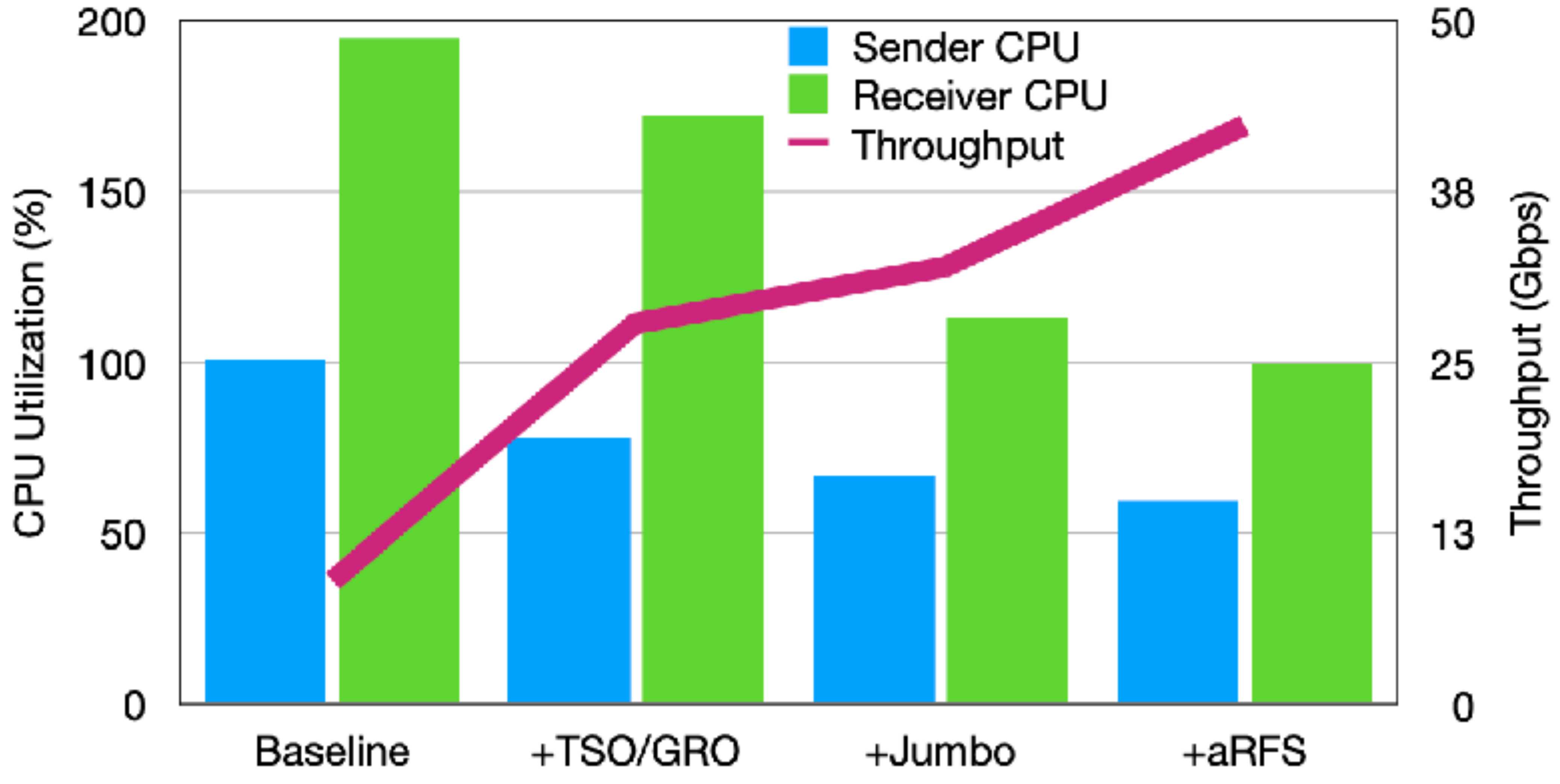
# Observation #1: Single Flow Performance

- Can we saturate the network bandwidth using a single core?

# Observation #1: Single Flow Performance

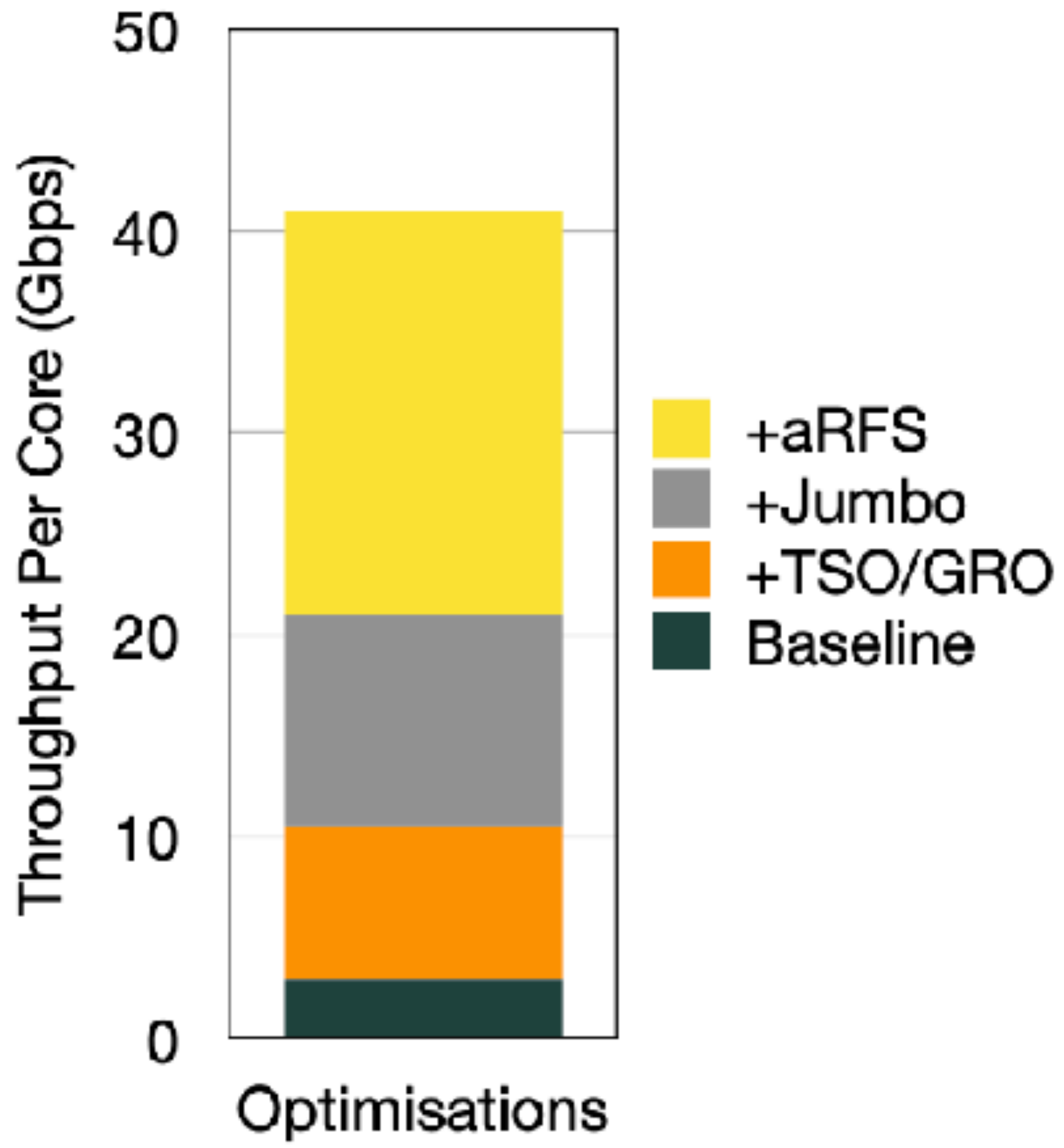
- Can we saturate the network bandwidth using a single core?

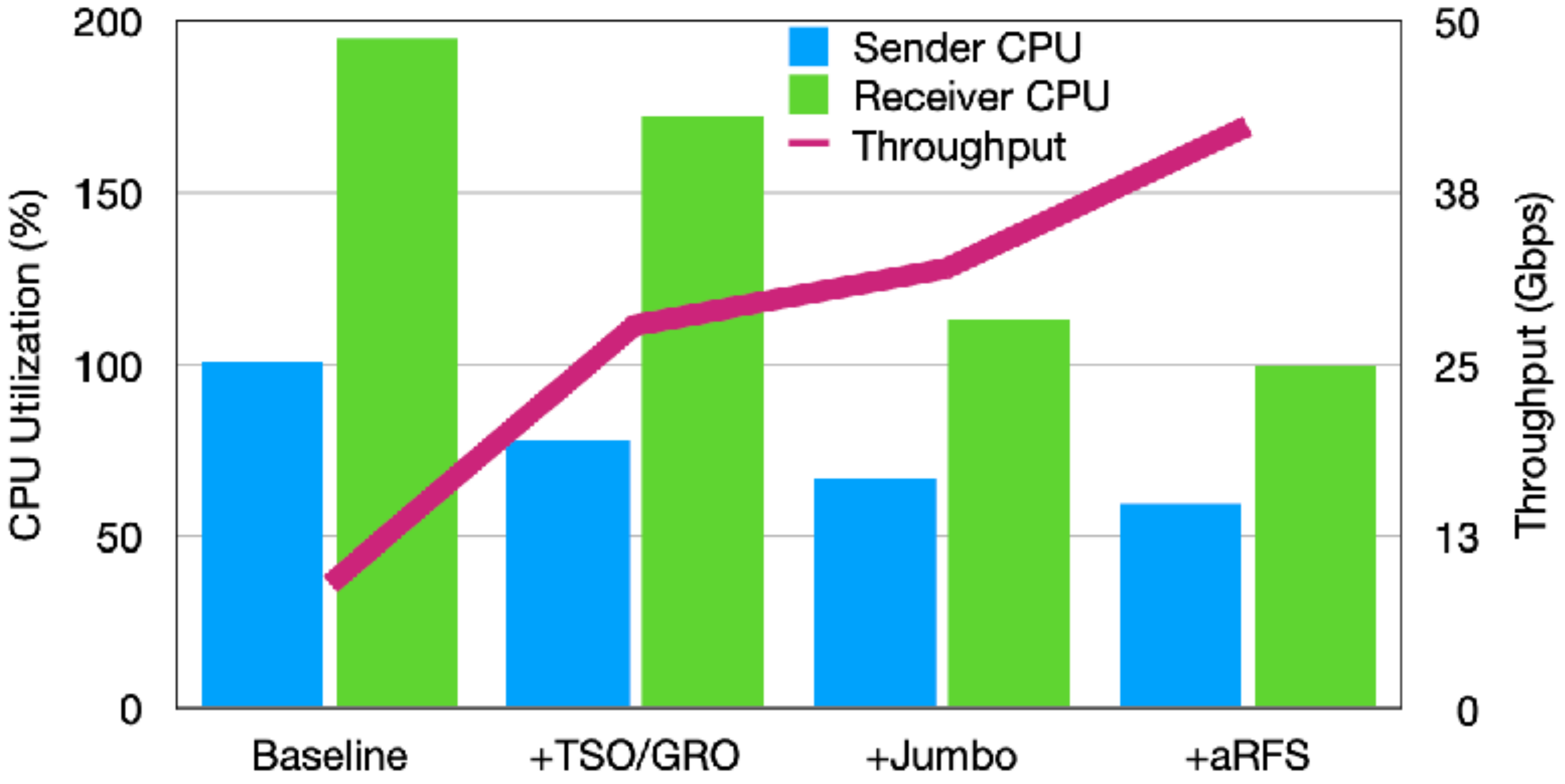




# Observation #1: Single Flow Performance

- Can we saturate the network bandwidth using a single core?

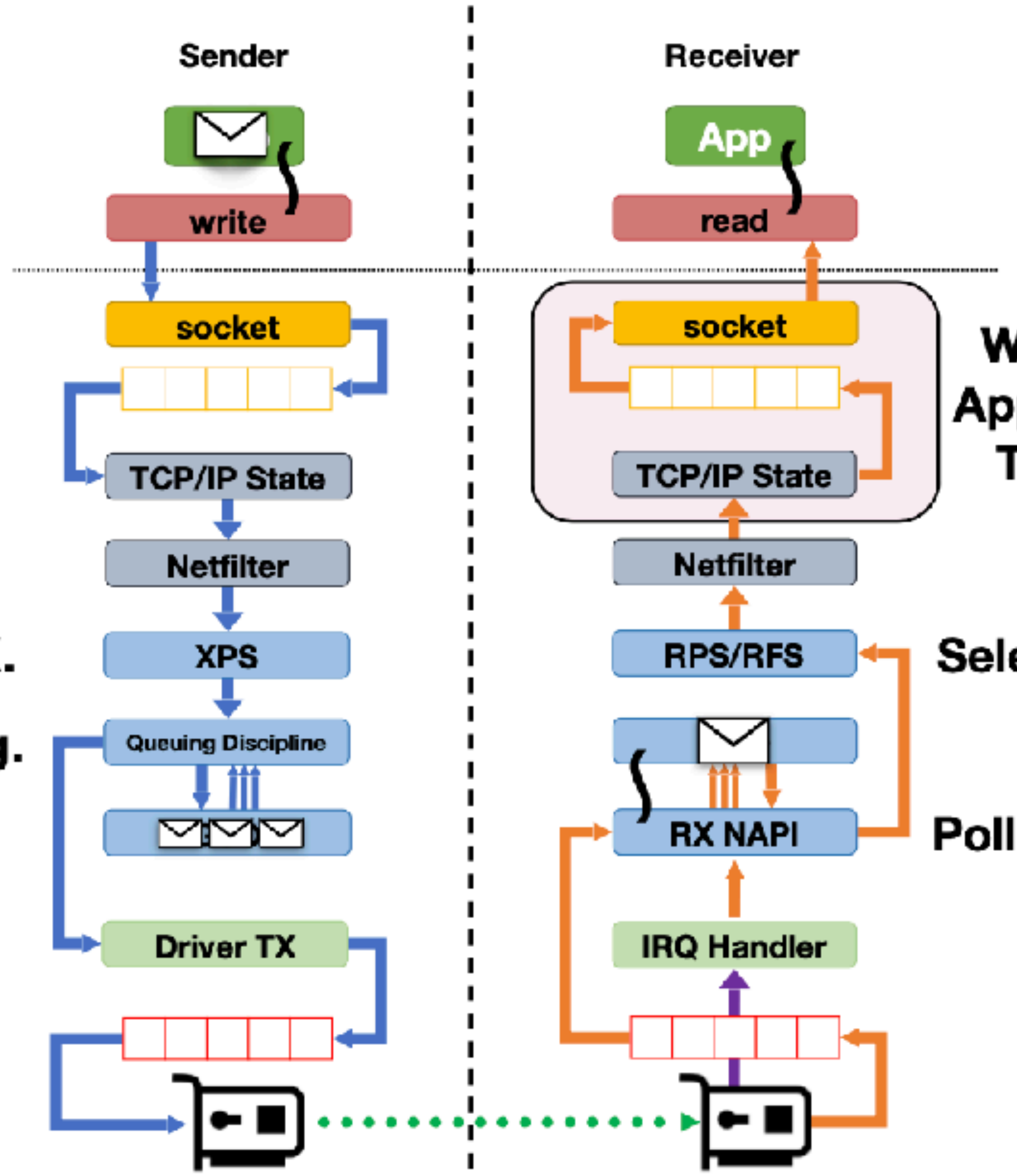




- One core is not enough
- The bottleneck is at the receiver side

# Observation #2: Bottleneck Localization

- Breakdown the data path



Select the Hardware Queue for TX.

Packet Scheduling.

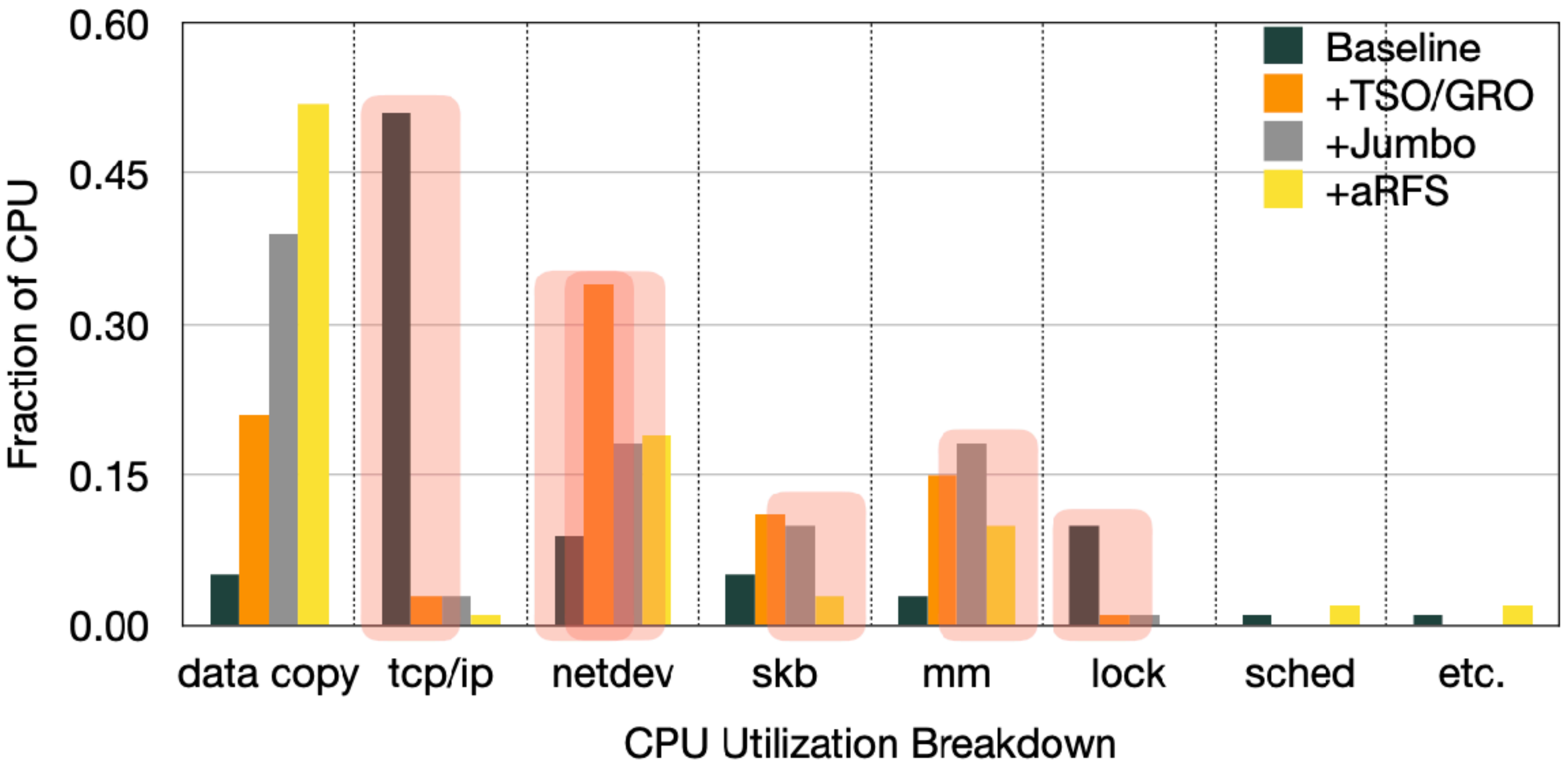
Wake-up Application Thread.

Select the CPU for TCP/IP Processing.

Poll for RX Packets.

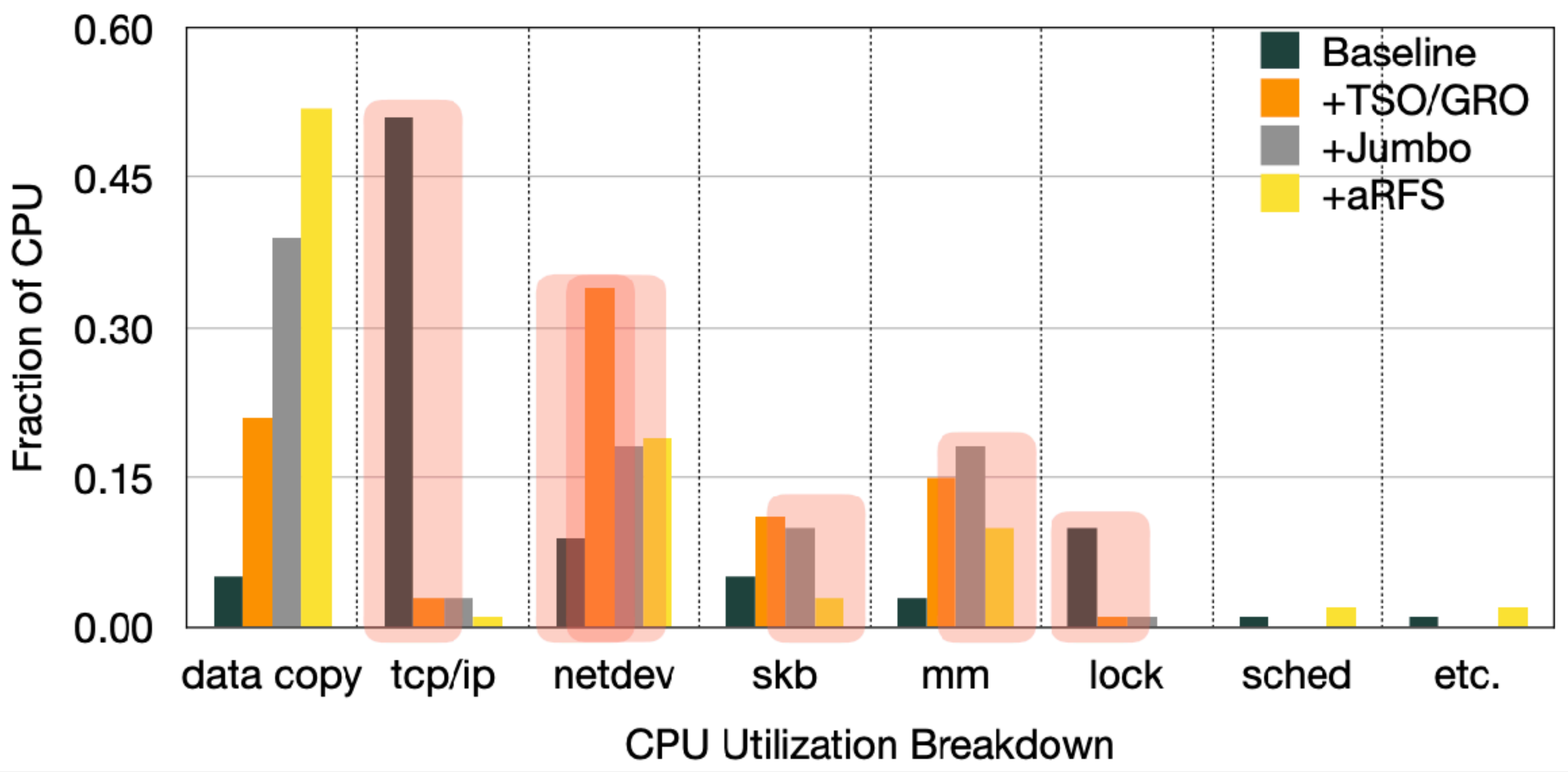
# Observation #2: Bottleneck Localization

- Breakdown the data path



# Observation #2: Bottleneck Localization

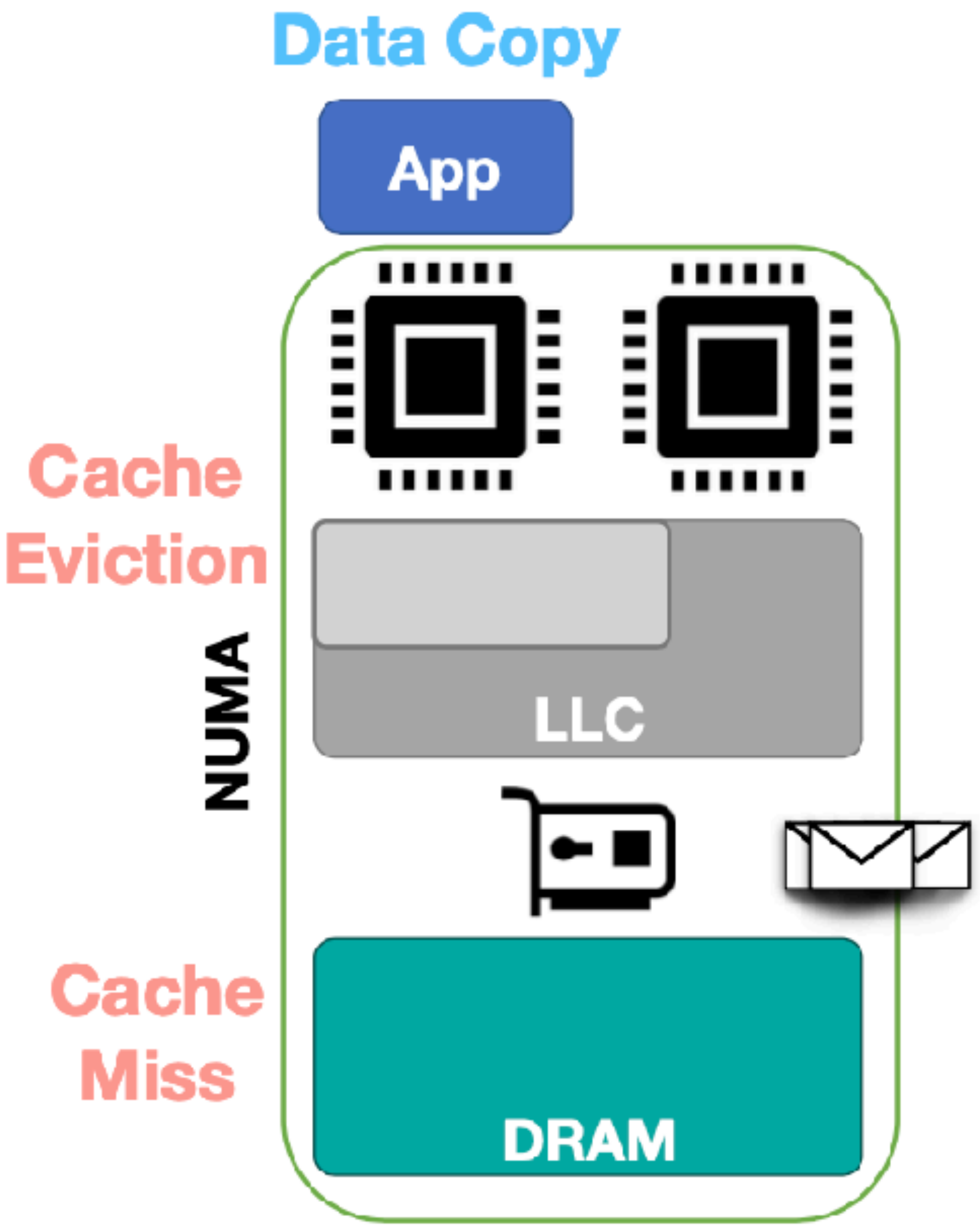
- Breakdown the data path



- Data copy dominates the receive-side processing

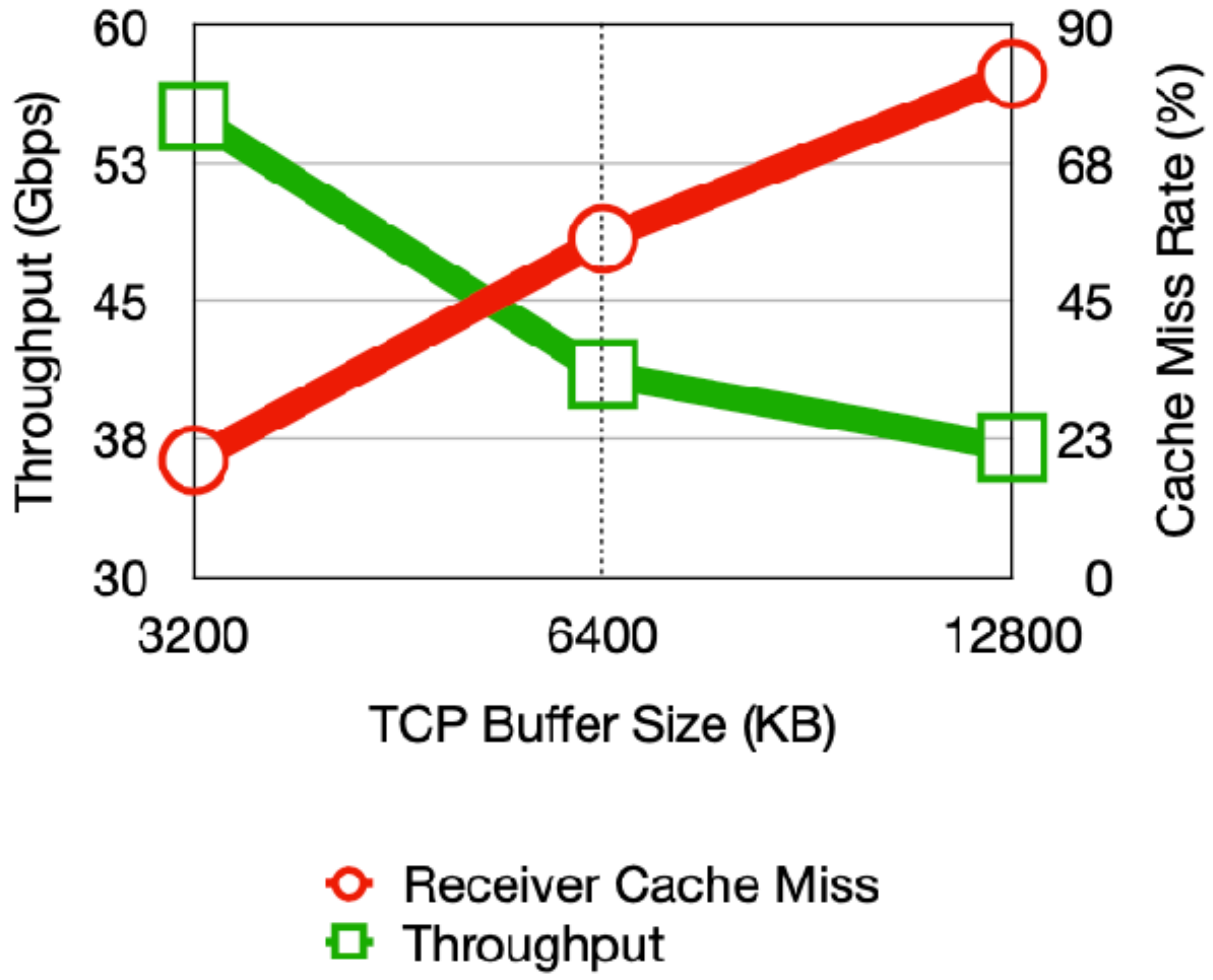
# Lesson #1: Bottlenecks have shifted from packet processing to data copy.

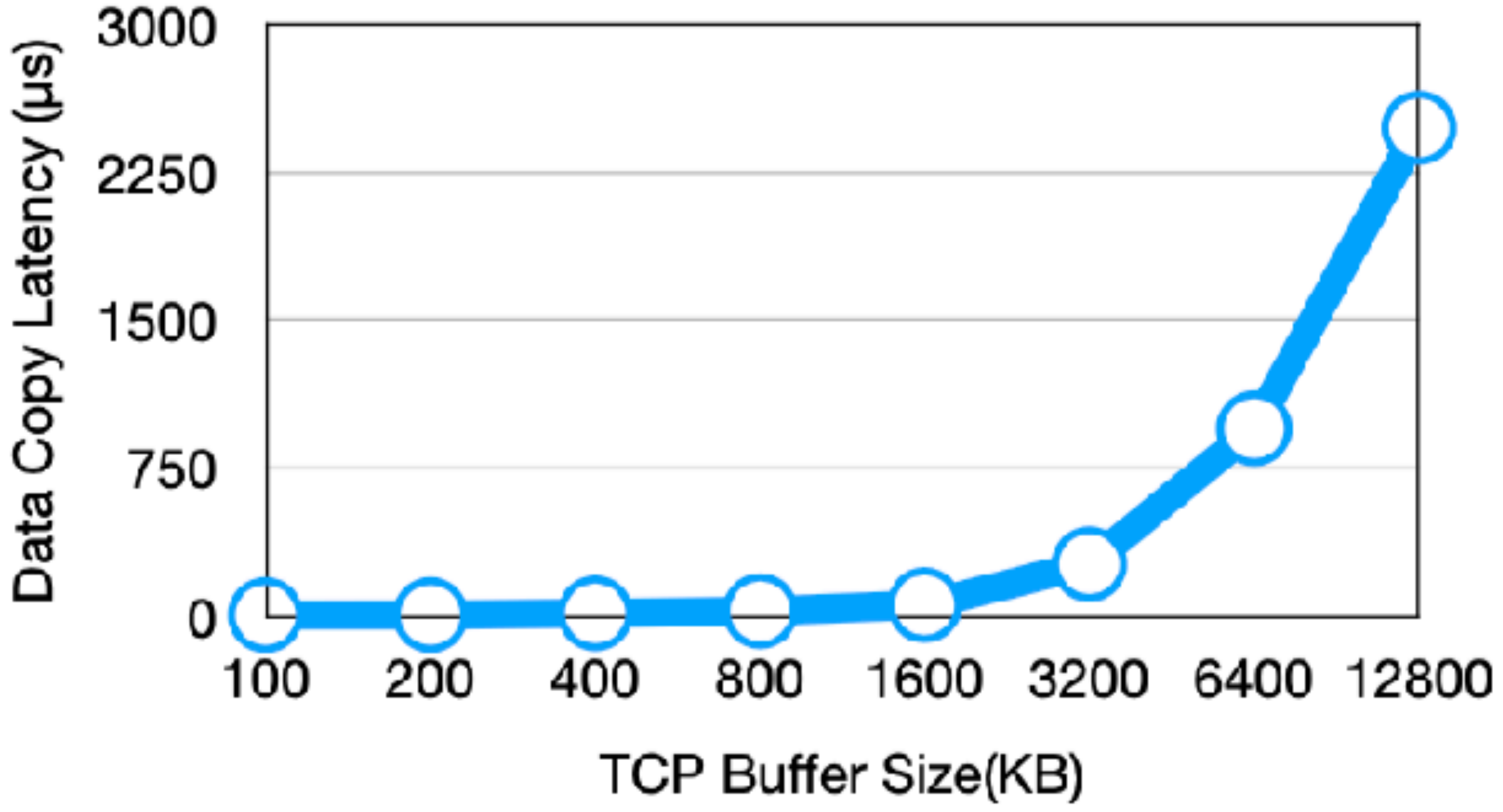
# Observation #3: Cache Misses



# Observation #3: Cache Misses

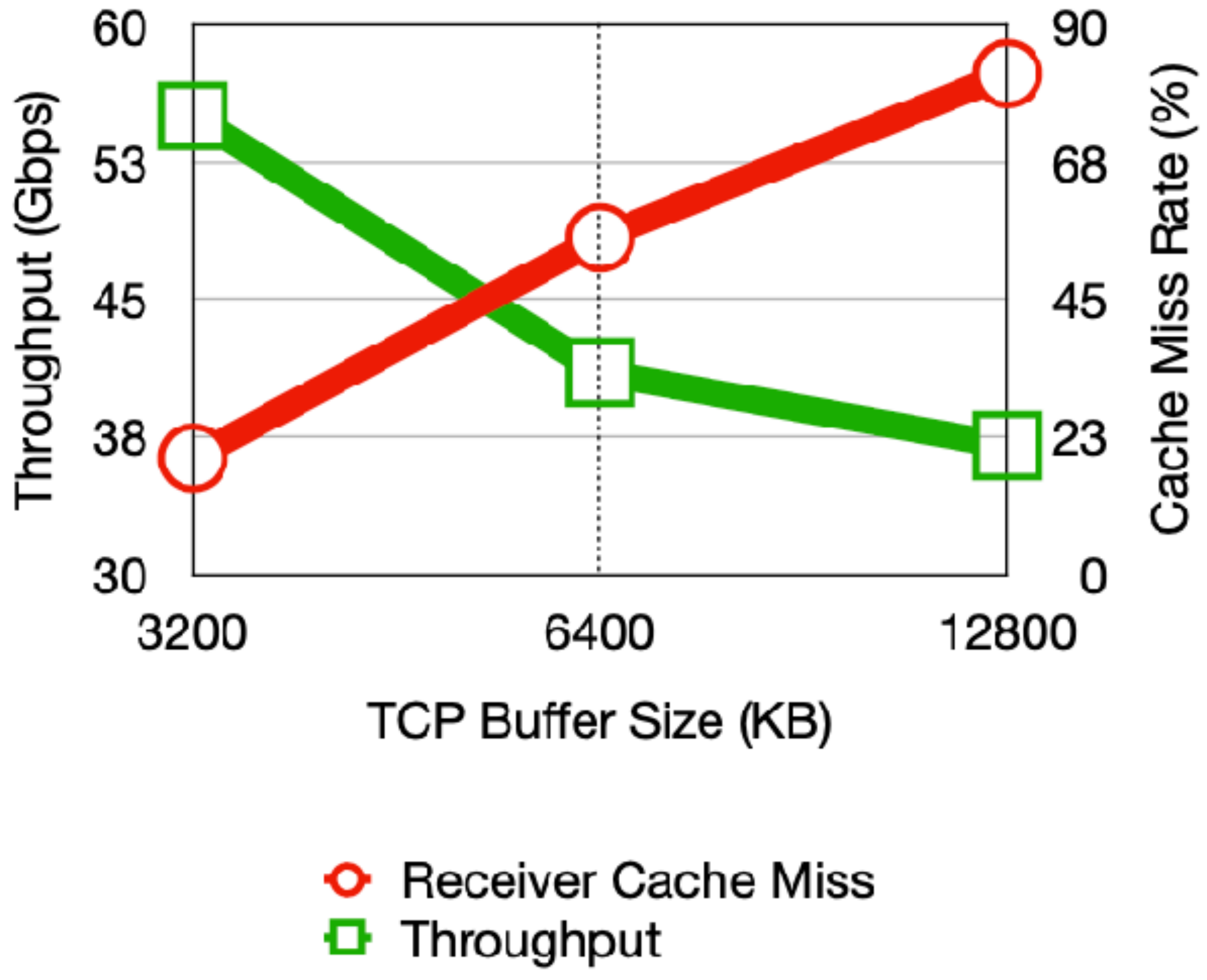
- Large TCP buffer

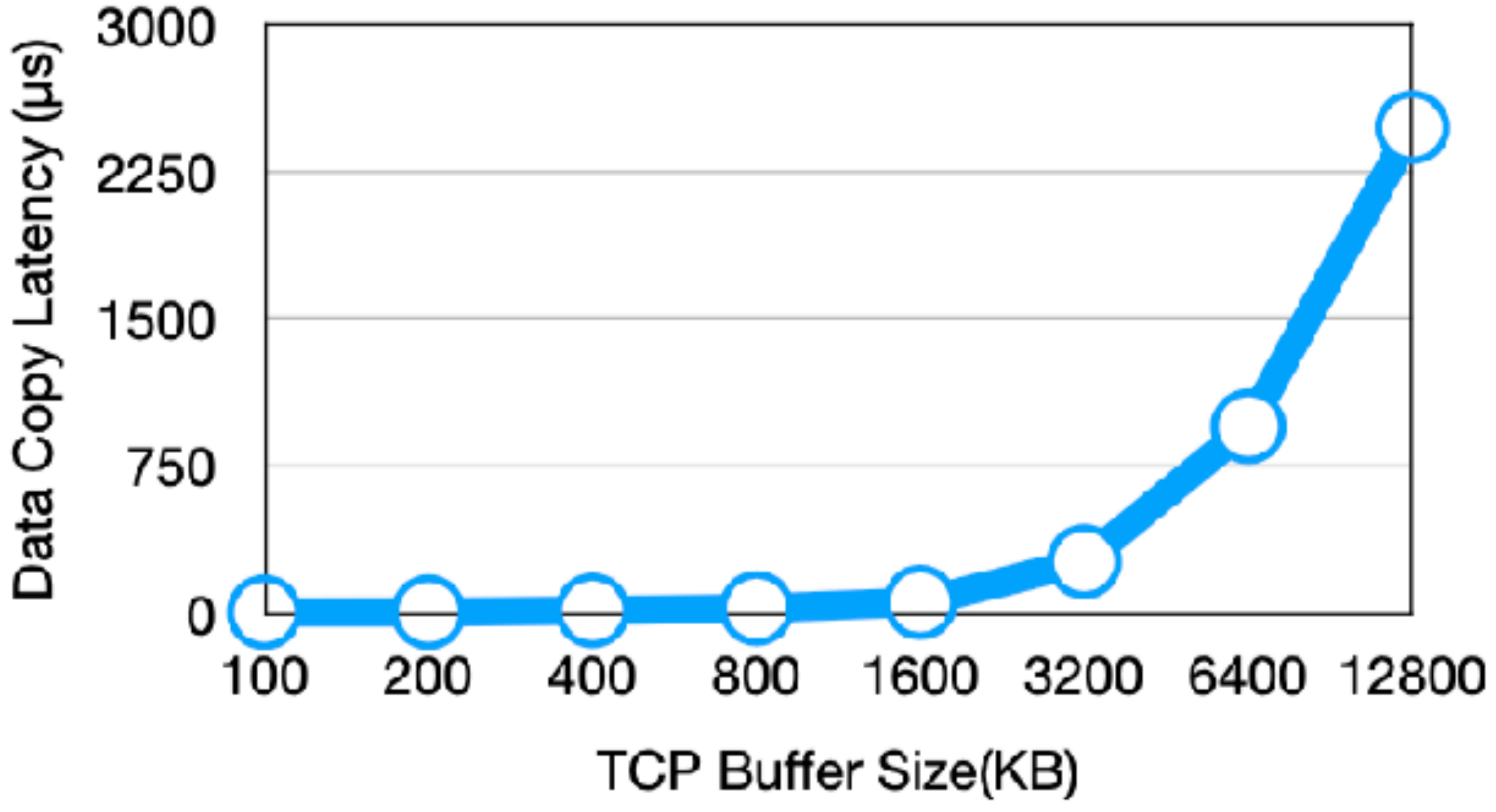




# Observation #3: Cache Misses

- Large TCP buffer

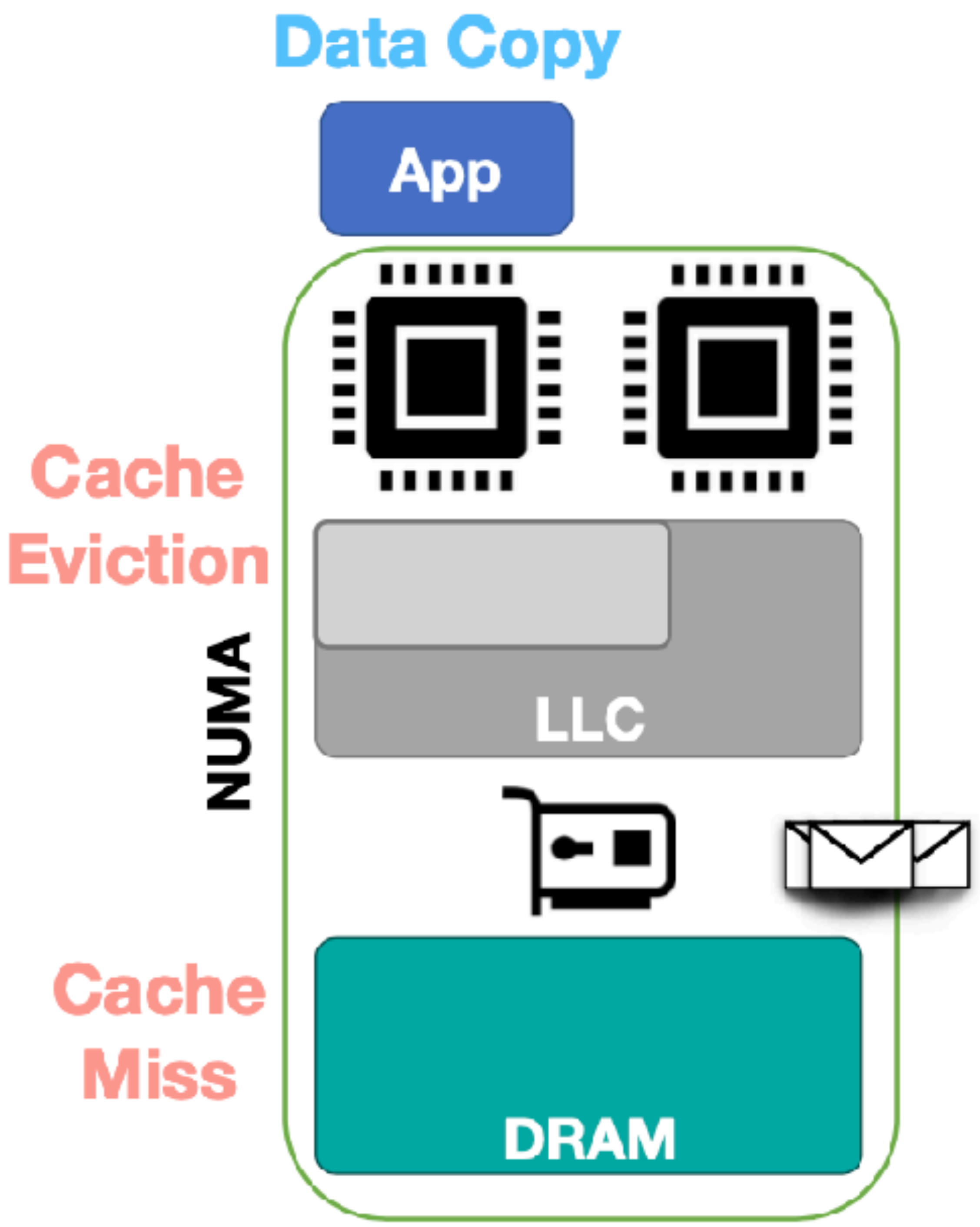


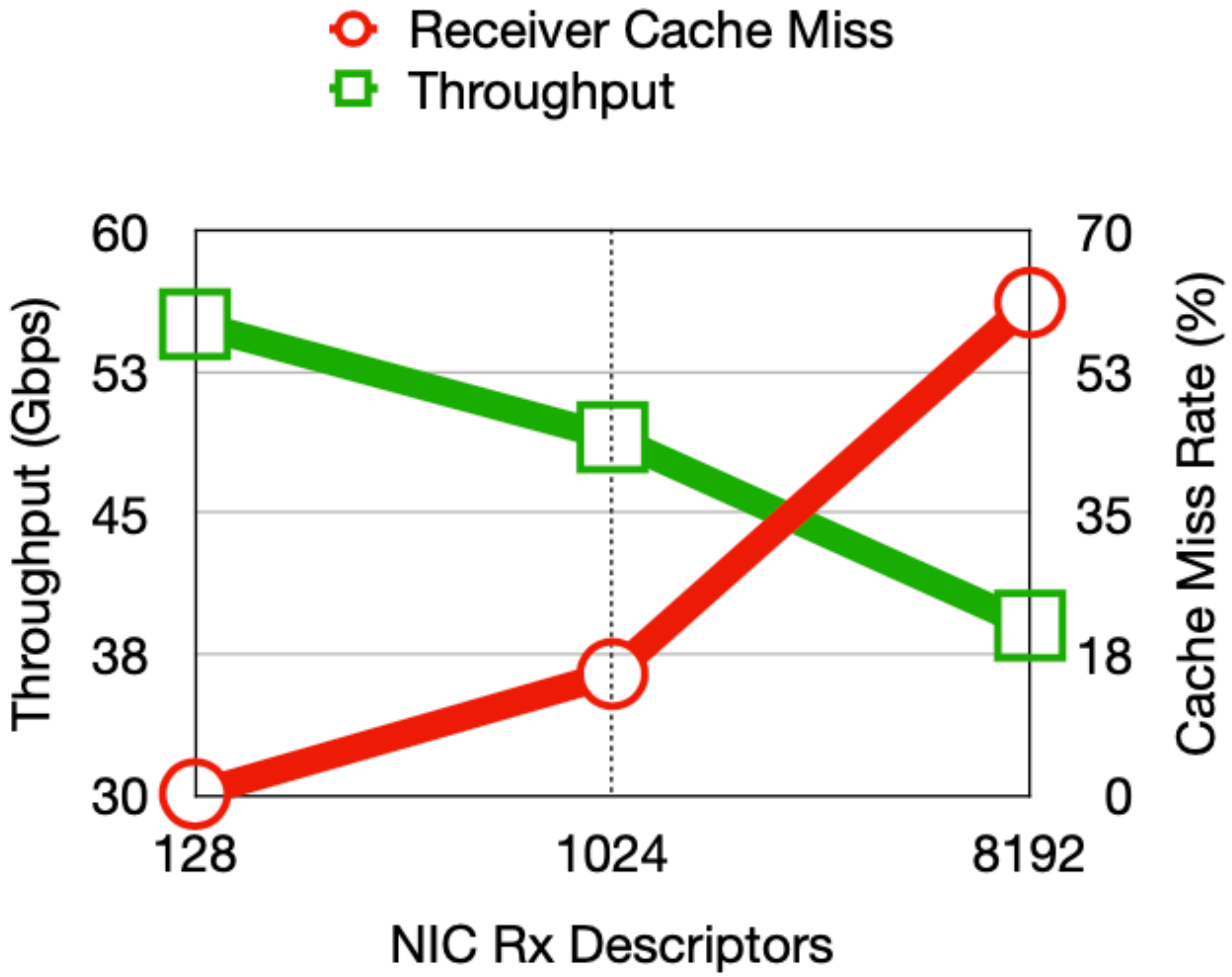


- High LLC miss rate

# Observation #3: Cache Misses

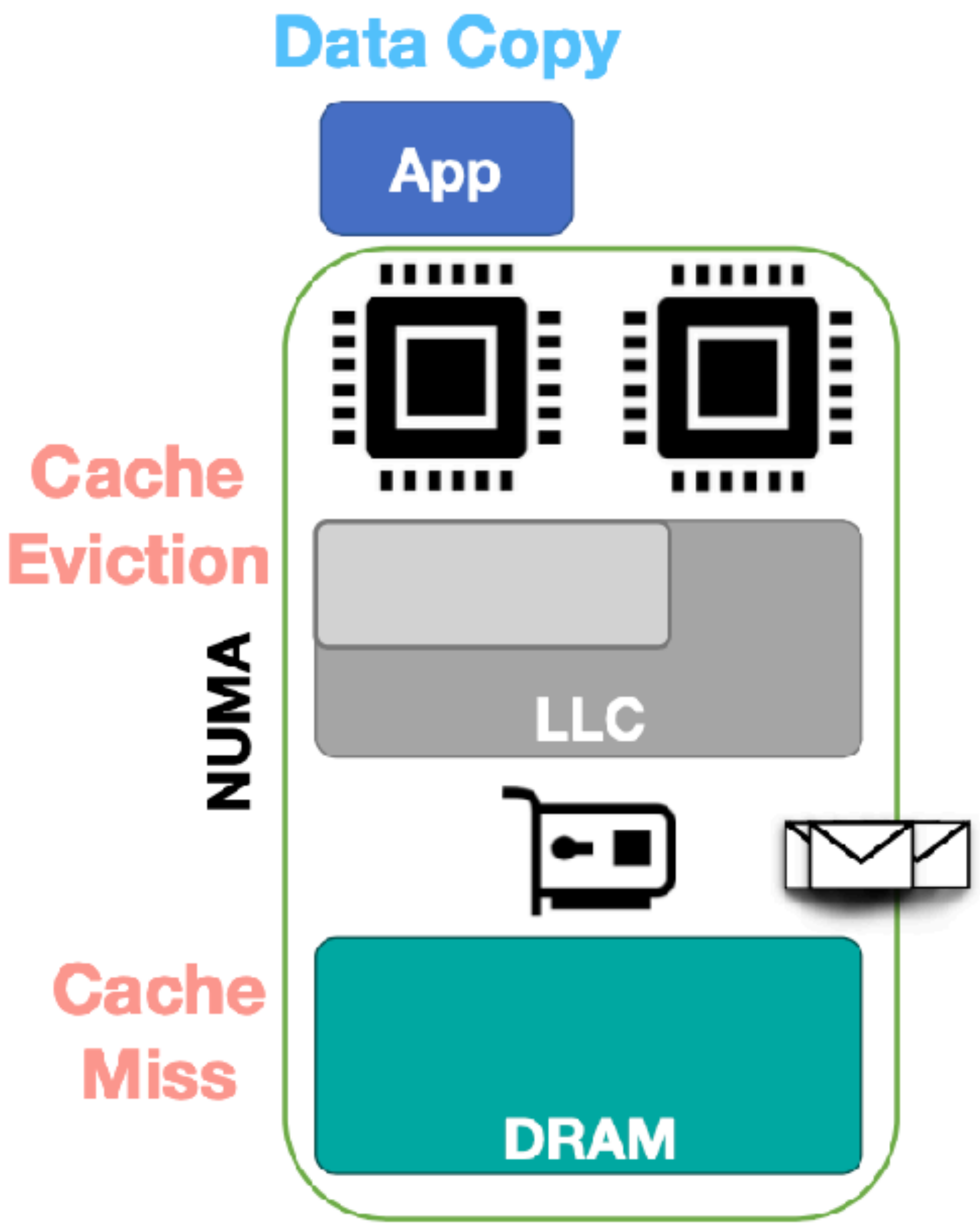
- More NIC RX descriptors

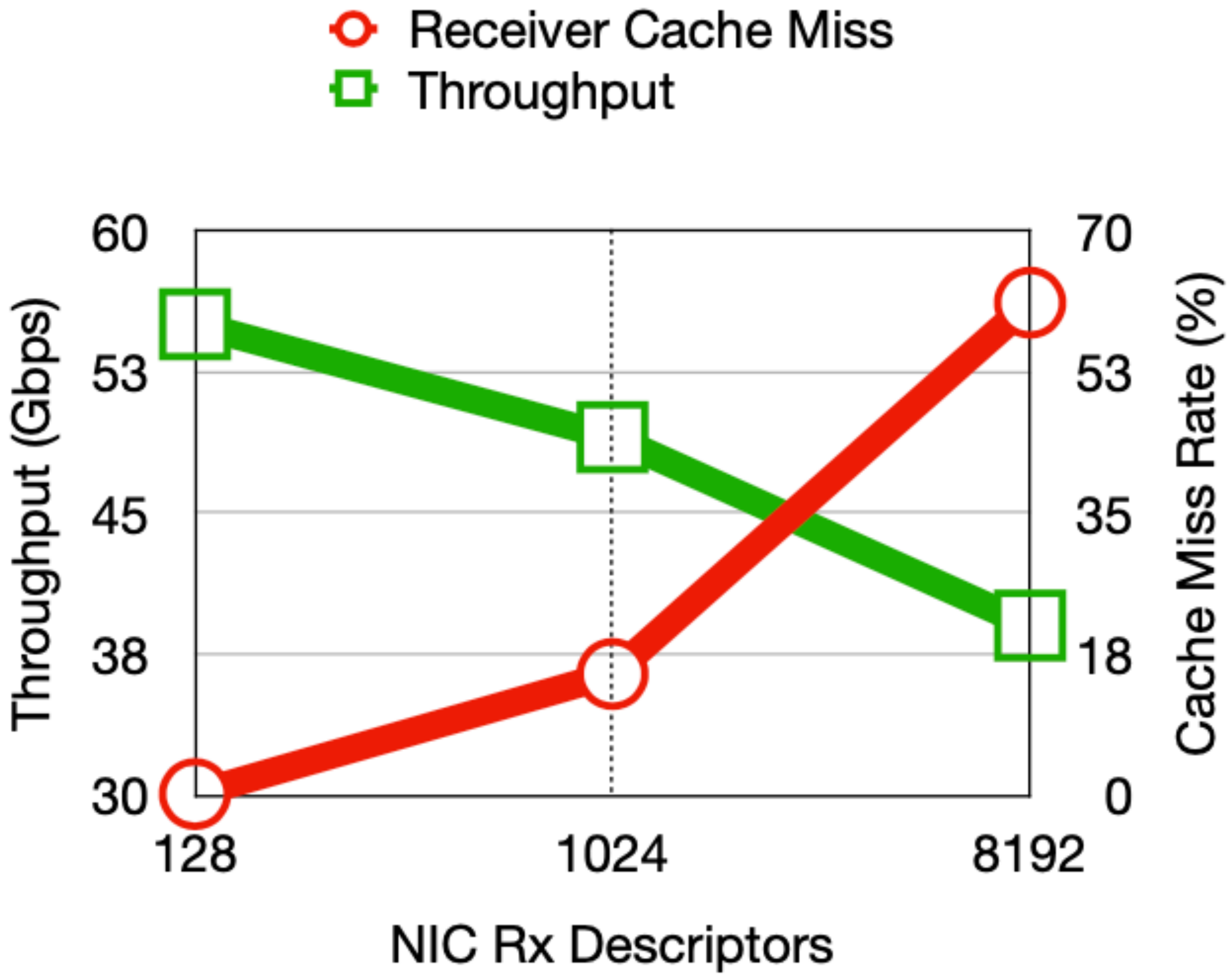




# Observation #3: Cache Misses

- More NIC RX descriptors



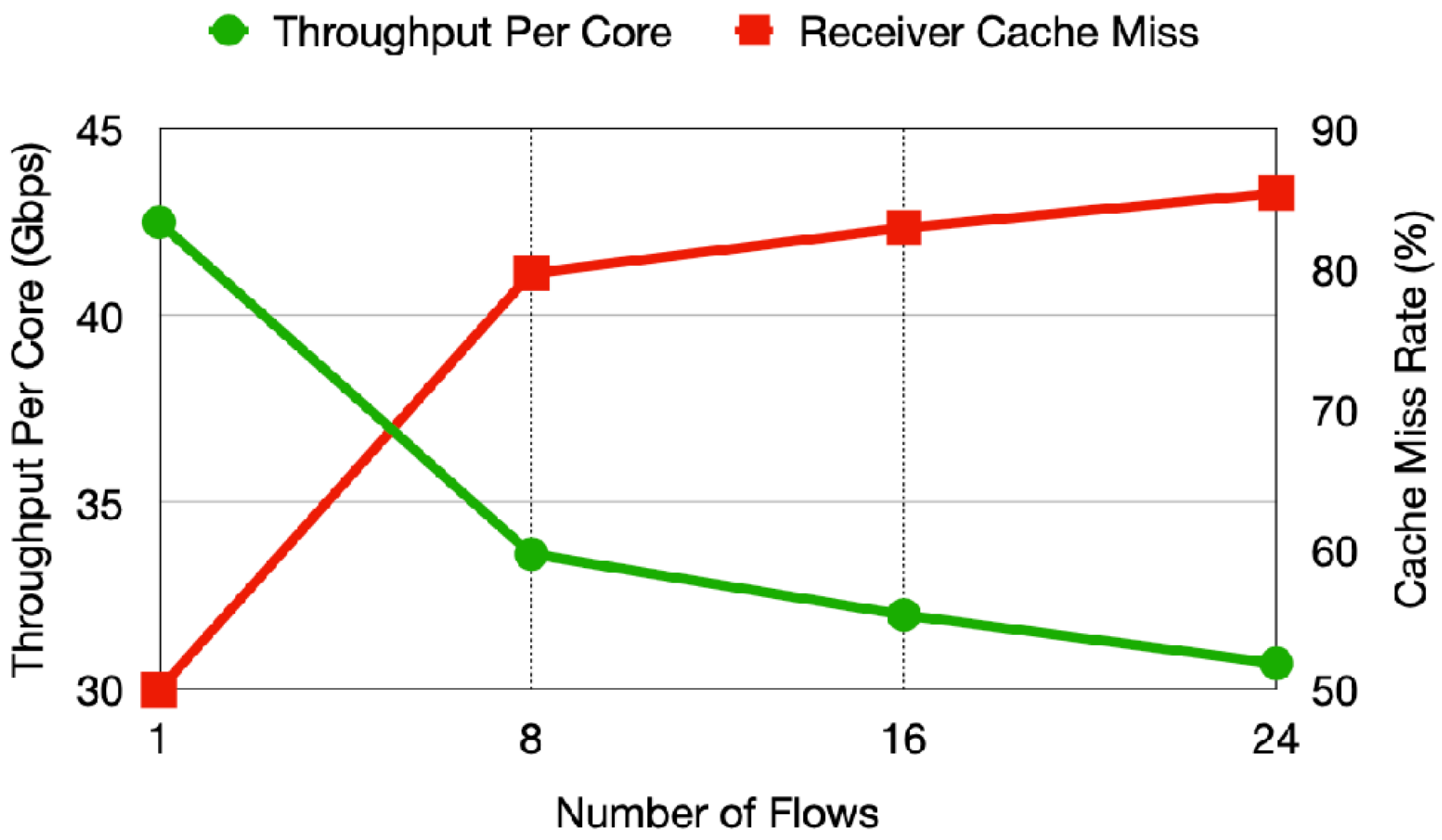


- High cache eviction

# Lesson #2: The NIC DMA pipeline is inefficient.

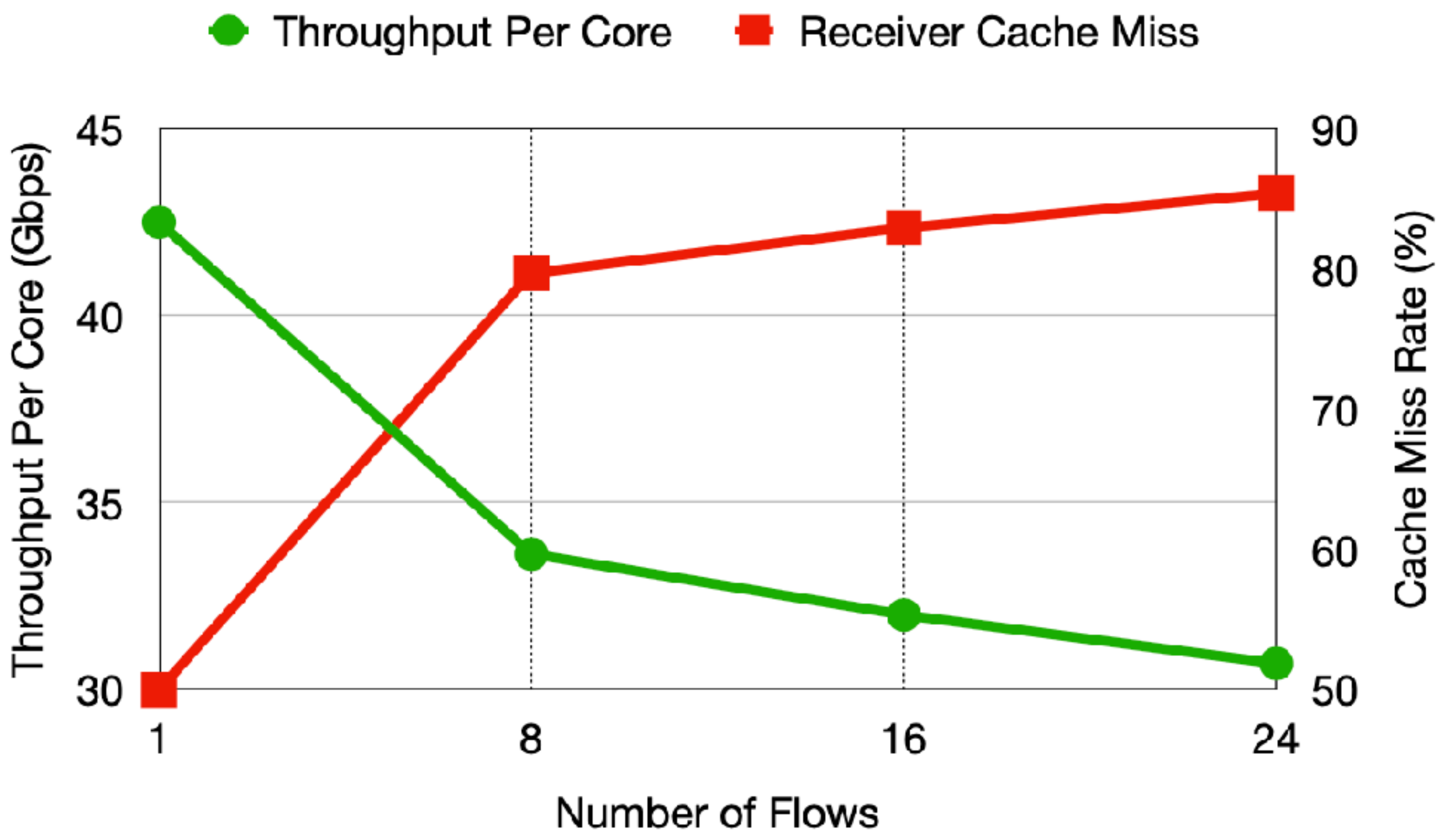
# Observation #4: Cache Contention under Incast

- Throughput per core drops under multiple flows



# Observation #4: Cache Contention under Incast

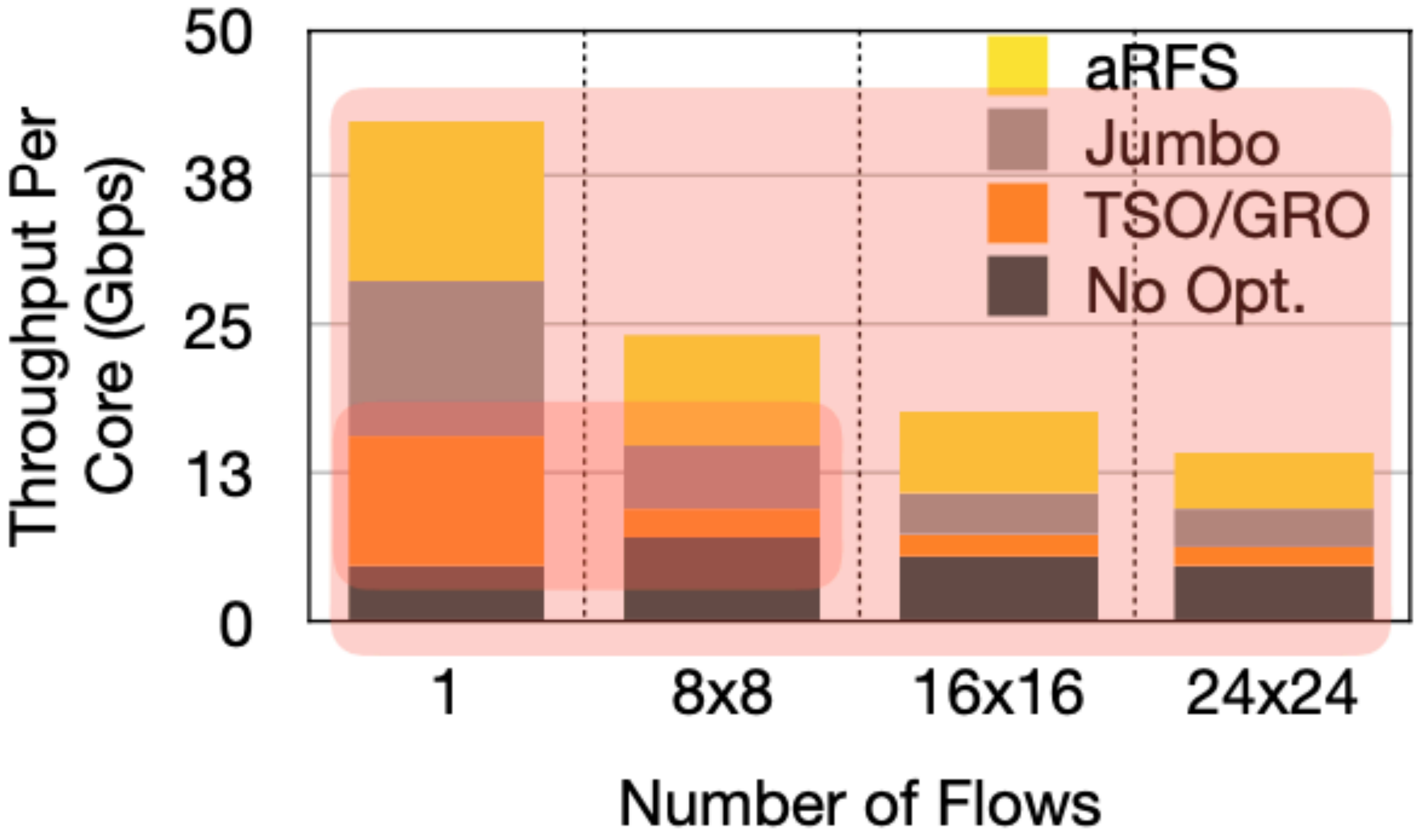
- Throughput per core drops under multiple flows



- Cache contention

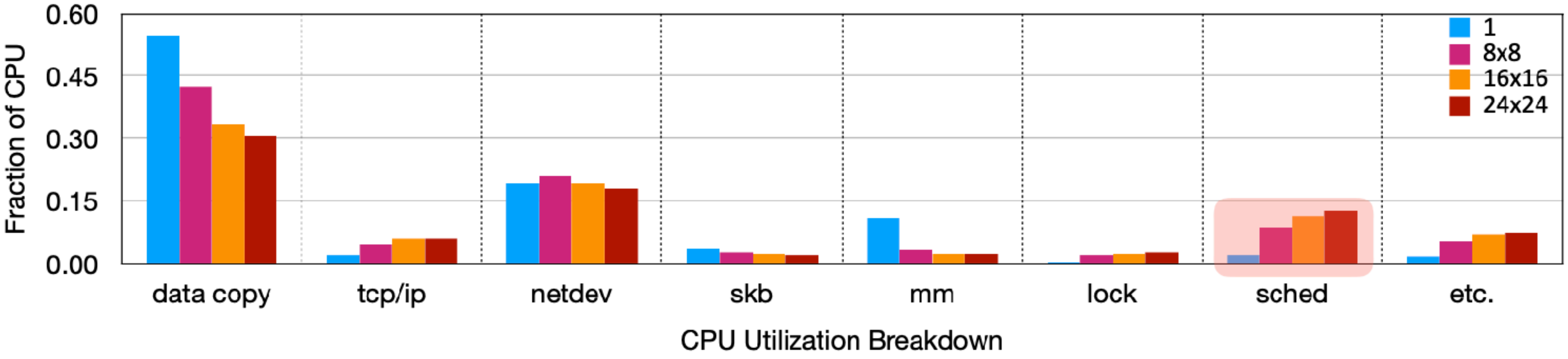
# Observation #5: Heavy Contention via All-to-All

- Throu./core drops by 67% due to contention at CPU/network



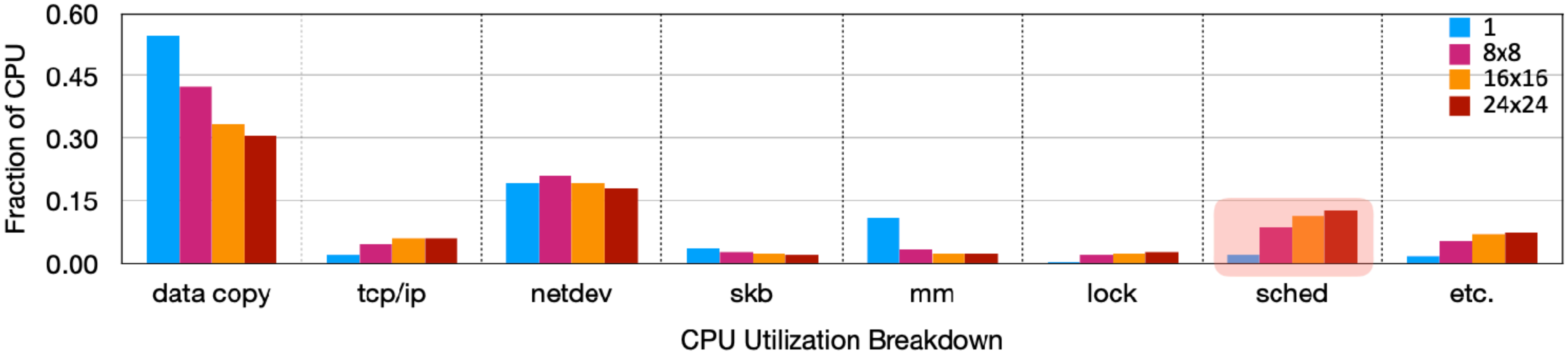
# Observation #5: Heavy Contention via All-to-All

- Scheduling overheads increases!



# Observation #5: Heavy Contention via All-to-All

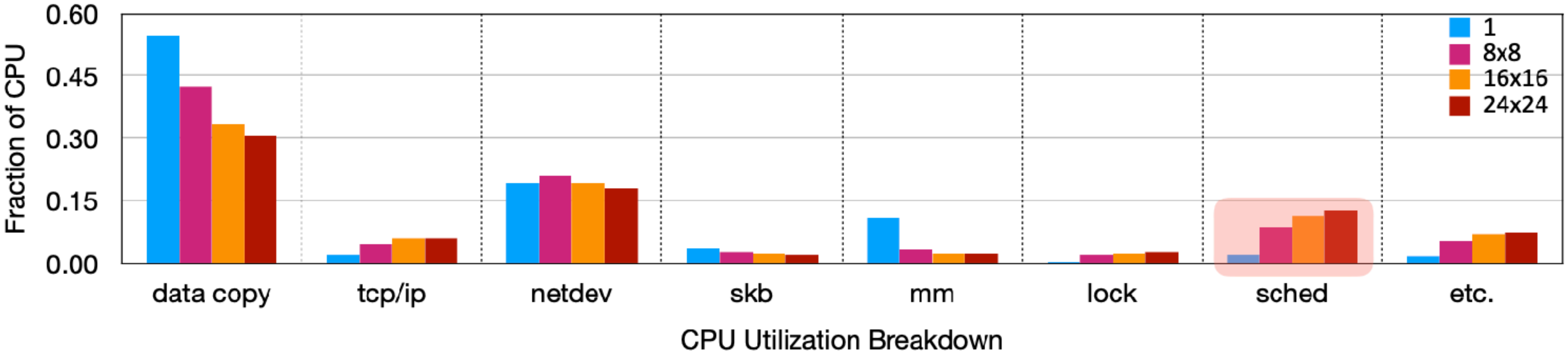
- Scheduling overheads increases!



- Applications sleep and wake-up frequently!

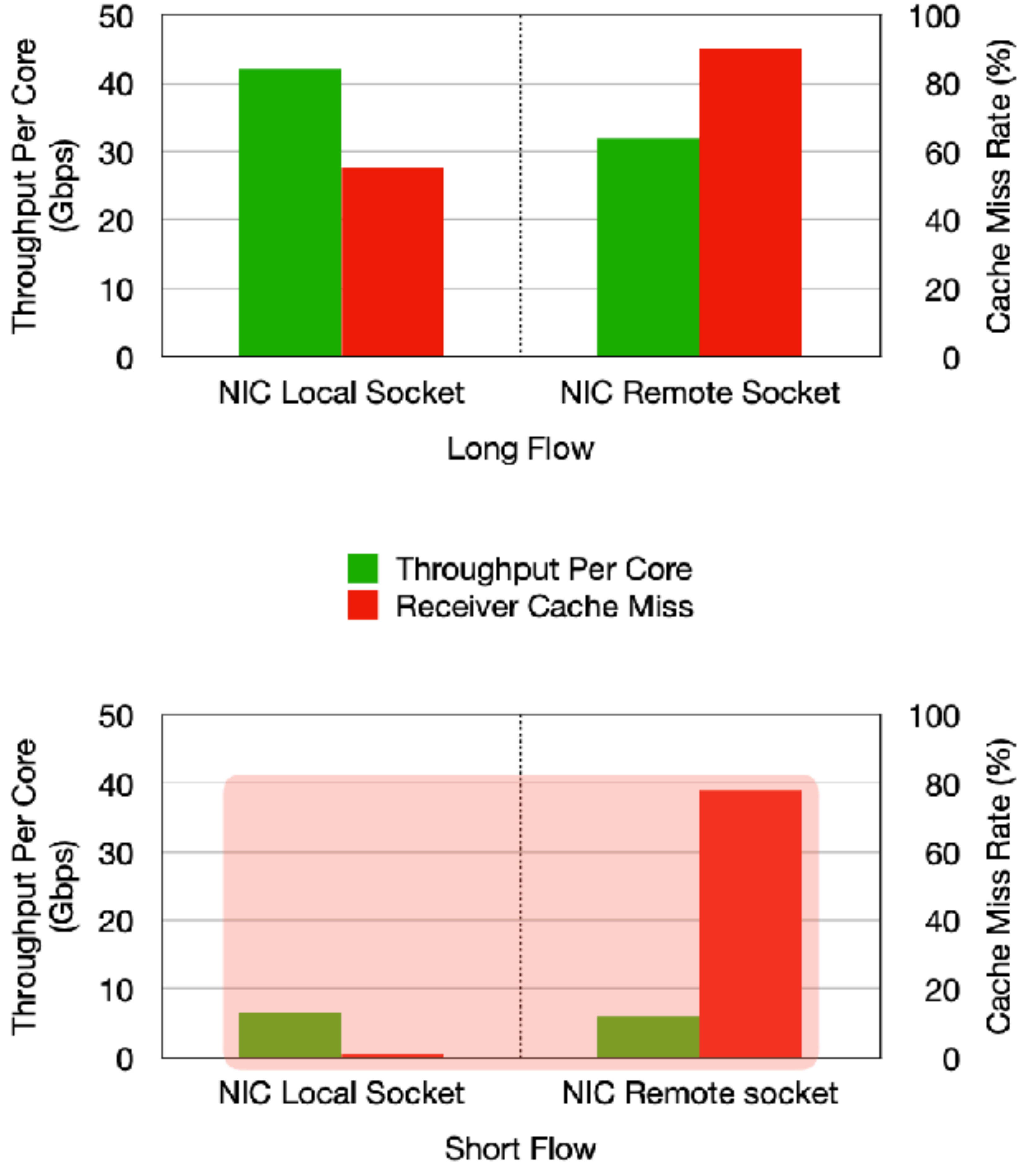
# Observation #5: Heavy Contention via All-to-All

- GRO efficiency drops and packet processing overheads rise

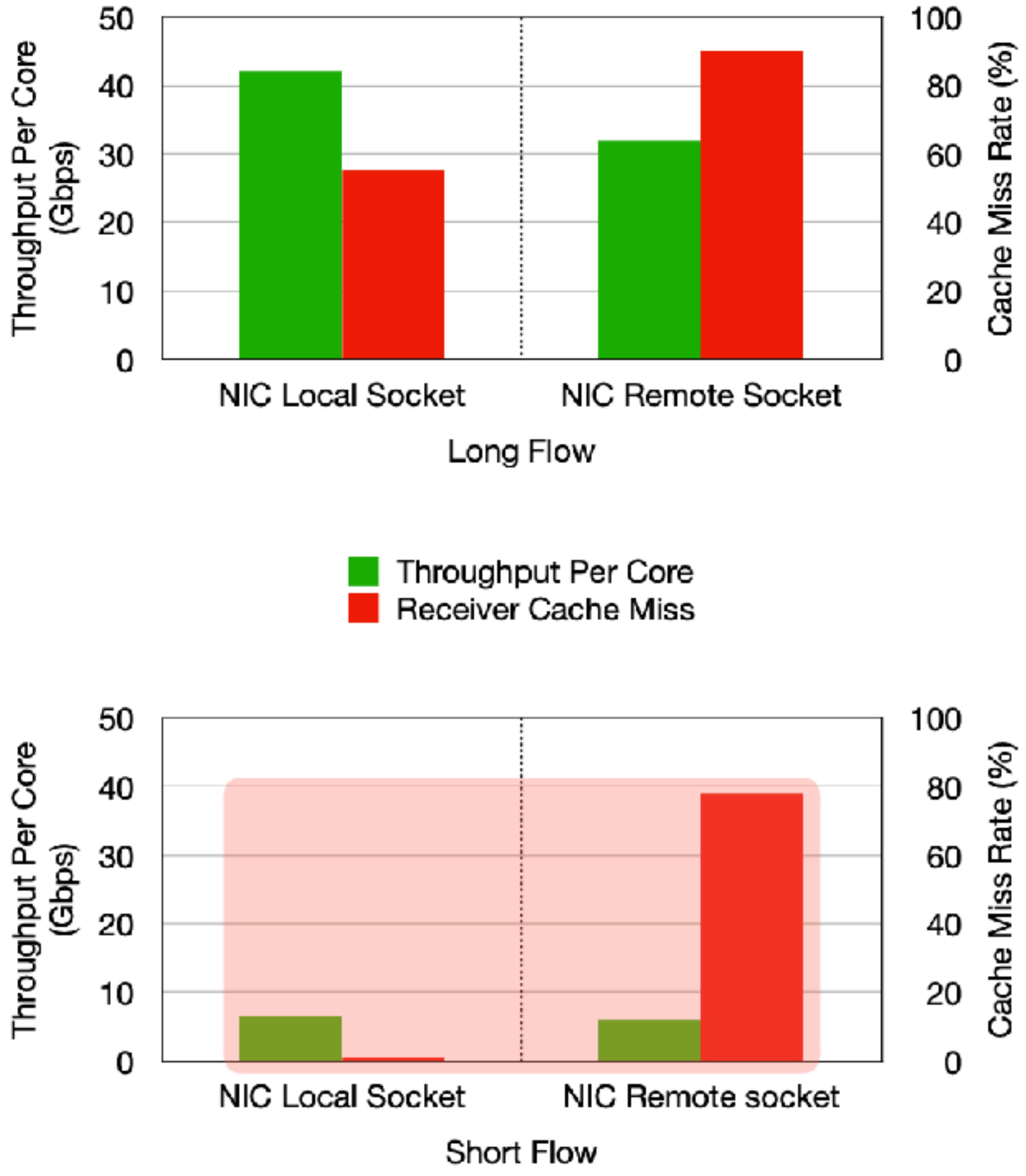


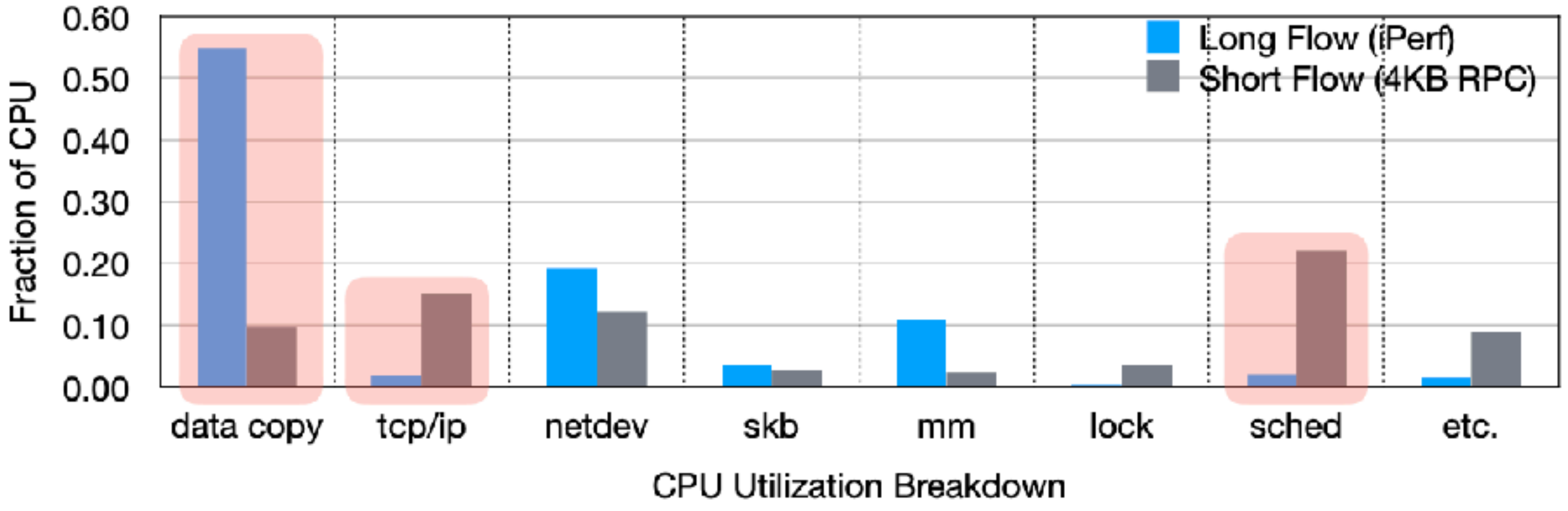
# Lesson #3: Endhost resource sharing is considered harmful.

# Observation #6: Mixed Flows

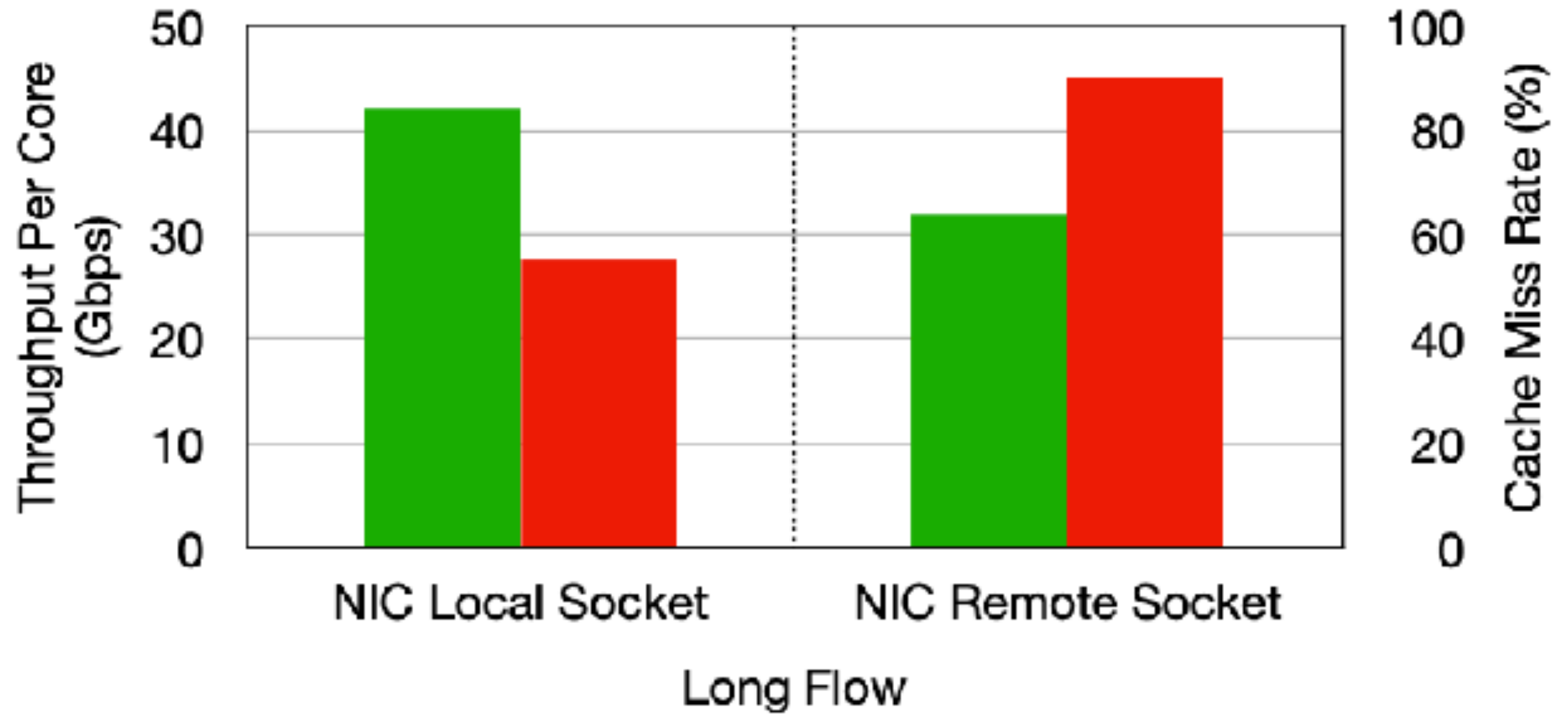


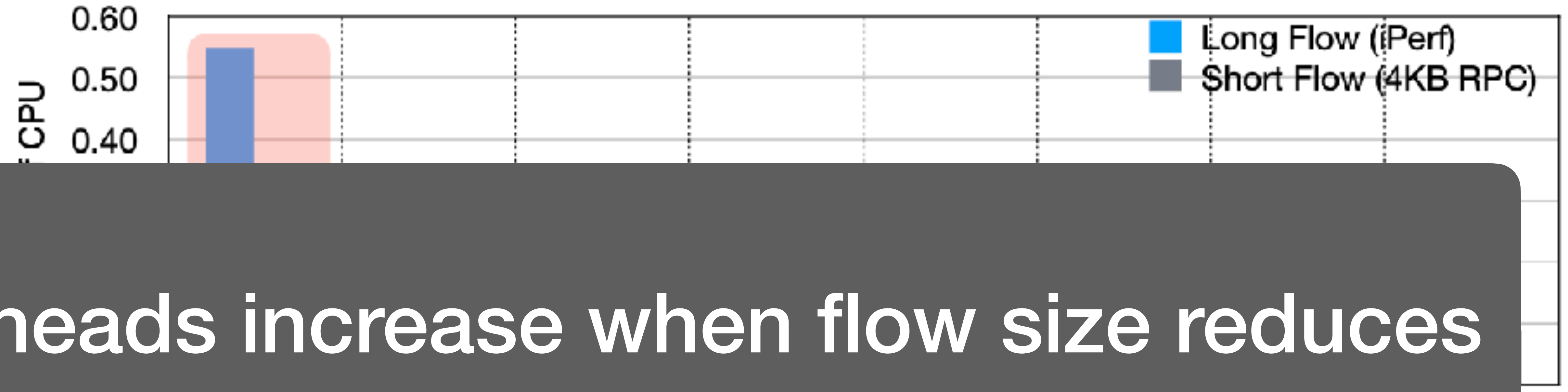
# Observation #6: Mixed Flows



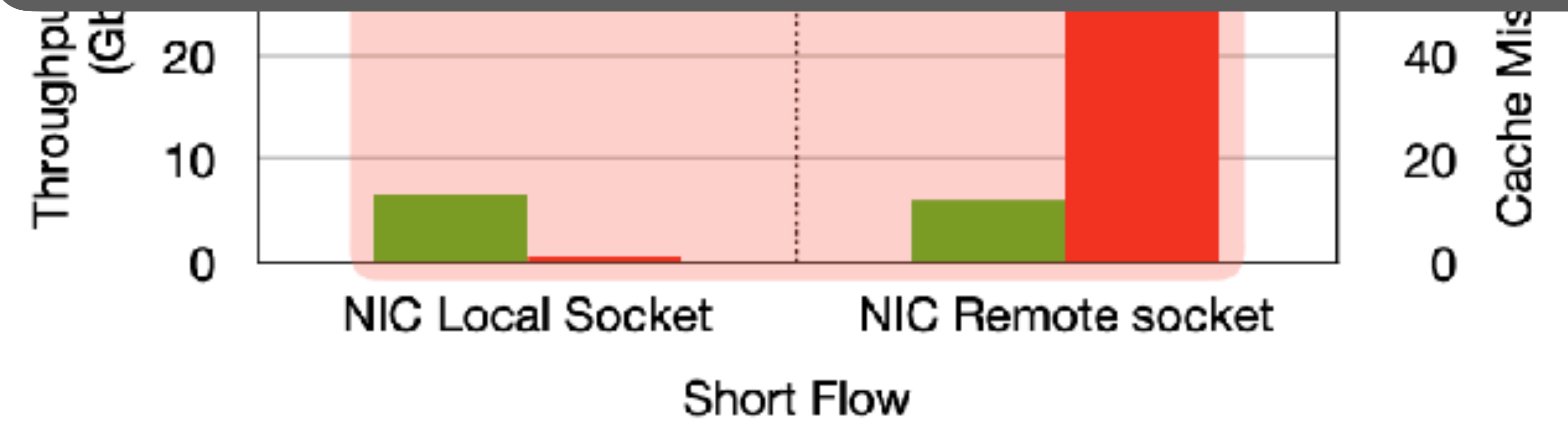


# Observation #6: Mixed Flows

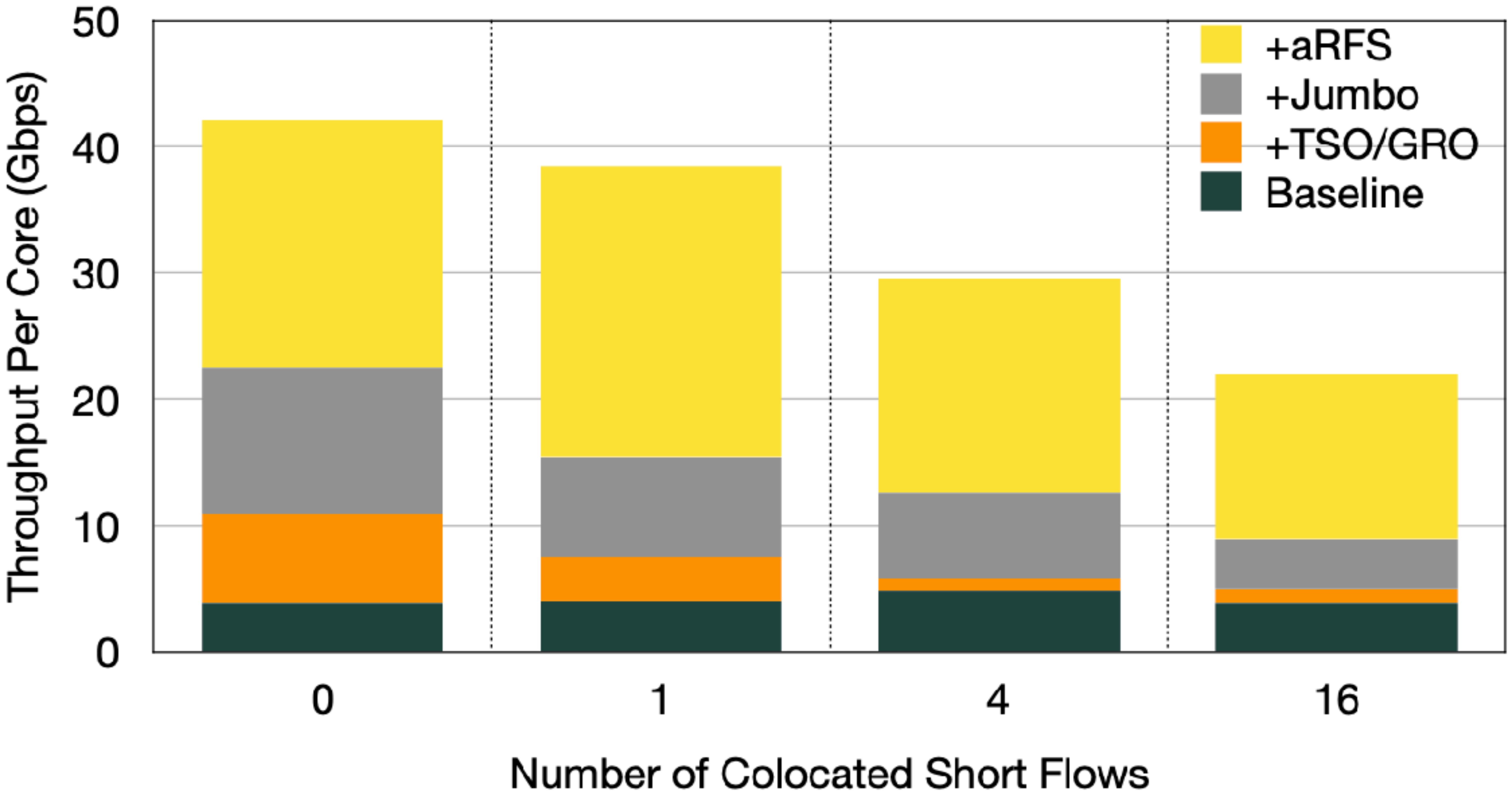




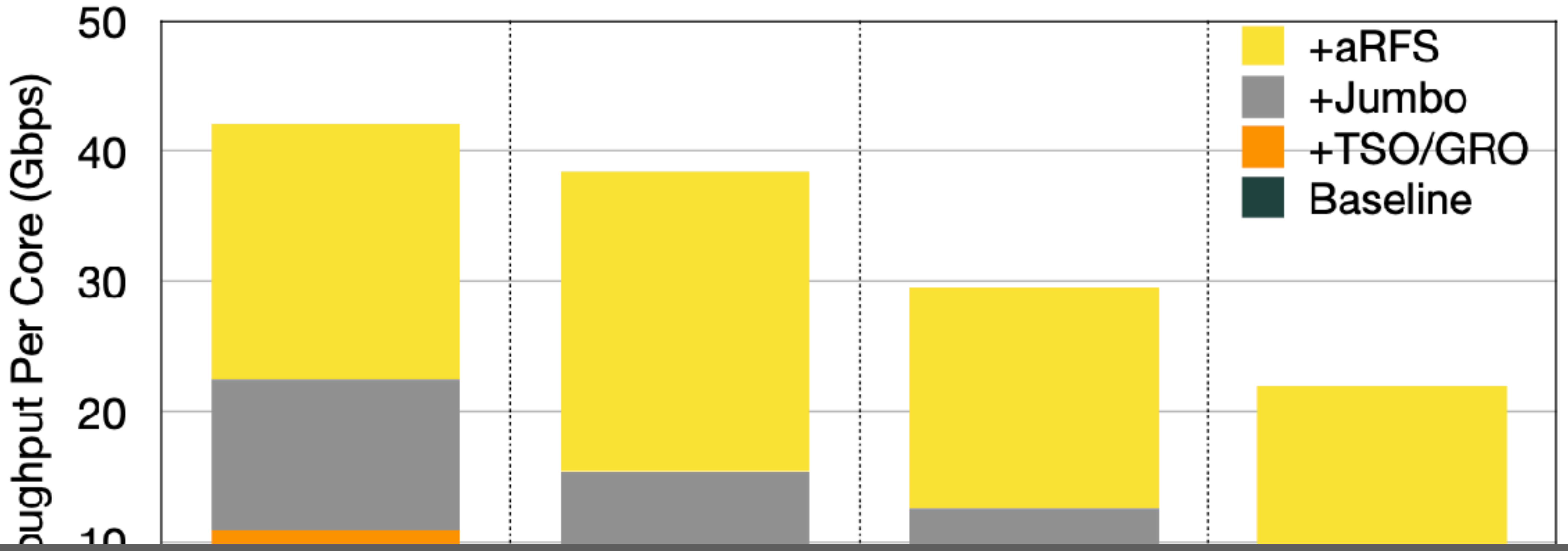
- TCP/IP and scheduling overheads increase when flow size reduces
- Intel DCA has little impact on the throughput of short flows



# Observation #6: Mixed Flows



# Observation #6: Mixed Flows



- Overall throughput per core drops by 47%
- Long/short flow performance degrades by 52/57%

# Lesson #4: Need redesigned networks and network-aware CPU schedulers.

# Summary

- Today
  - Linux NStack

- Next
  - SNAP (SOSP’19)