CS 134
Operating Systems

April 24, 2019

OS Network Performance
IX

Based on OS Network Performance

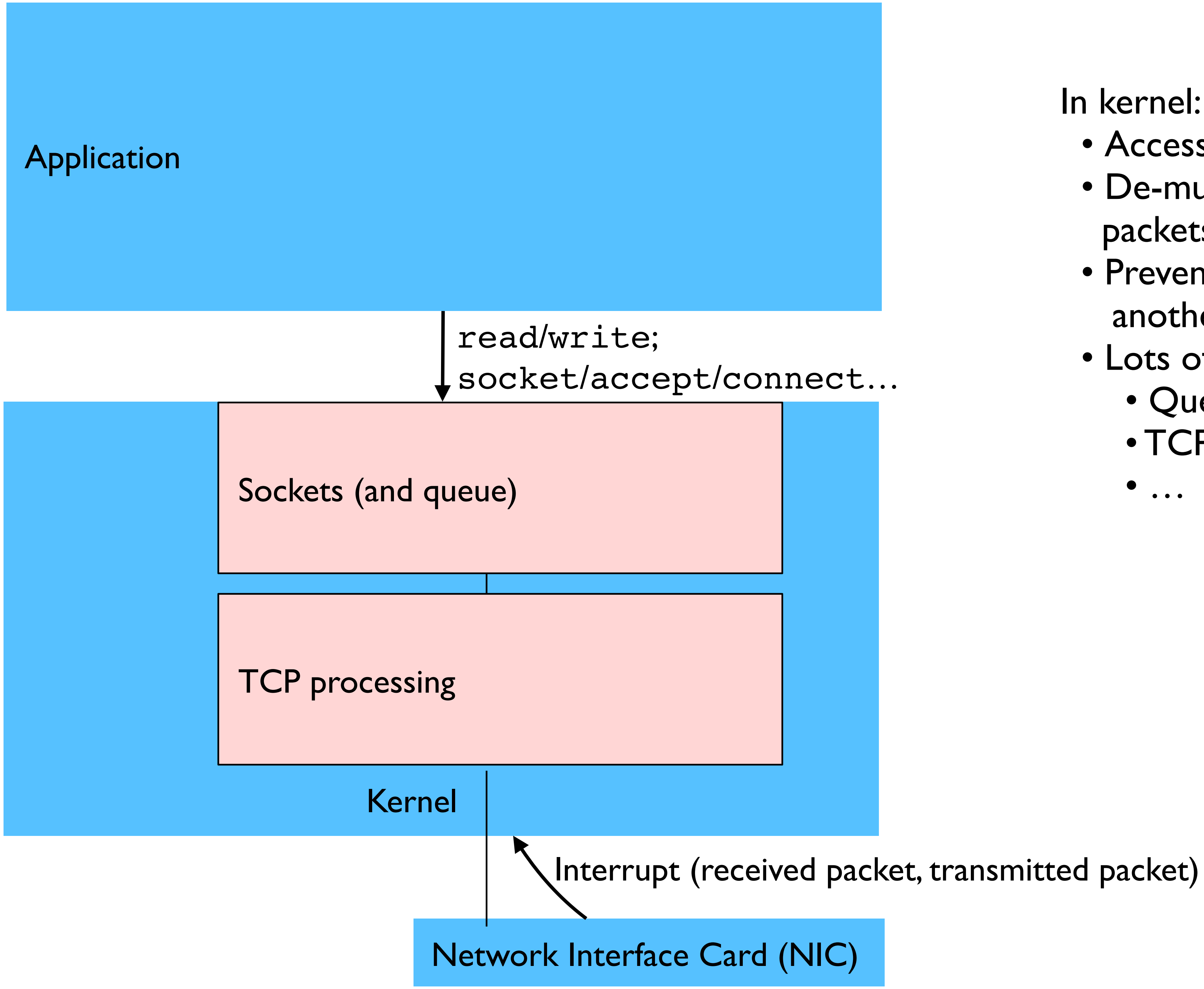
Outline

- OS Network Performance
- IX as a case study

Intel VT-x

- Makes x86 hardware “classically virtualizable” (as defined by Popek and Goldberg)
- Goal: **Direct execution** of most privileged instructions
- Introduces two CPU modes:
  - VMX root mode: for running VMM
  - VMX non-root mode: for running VMs (guest)
  - Each mode has its own rings (CPL0-CPL3)
- In-memory structure called VM Control Structure (VMCS) stores privileged register state and control flags

Linux network software structure



In kernel:
- Access to NIC hardware
- De-multiplex incoming packets (e.g., ARP/TCP)
- Prevent one app from messing with another app's connections
- Lots of locks and inter-core sharing:
  - Queues
  - TCP Connection state
  - …

High-performance network servers

- For example, memcached (in-memory key/value storage server)
  - High request rate
  - Short requests/responses
  - Lots of clients, lots of potential parallelism
  - Want high throughput under high load (request per second)
  - Want low latency under low/modest load (seconds per request)
  - Want low tail of latency distribution

What are the relevant HW limits?

- 10 Gb Ethernet: 15 million tiny packets/sec.
- 40 Gb Ethernet: 60 million tiny packets/sec.
- RAM: a few gigabytes/sec.
- Interrupts: 1 million/sec.
- System calls: a few million/sec.
- Contended locks: 1 million/sec.
- Inter-core data movement: a few million/sec.
- So:
  - If limited by Ethernet and RAM: XX million/sec.
  - If limited by interrupts, locks, etc.: Y million/sec.

Latency ingredients

- Latency important for e.g., web page with hundreds of items
- Low load: sum of a sequence of steps:
  - Network speed-of-light and switch round-trip time
  - Interrupt
  - queue operations
  - sleep/wakeup
  - system calls
  - inter-core data movement
  - RAM fetches

Latency ingredients

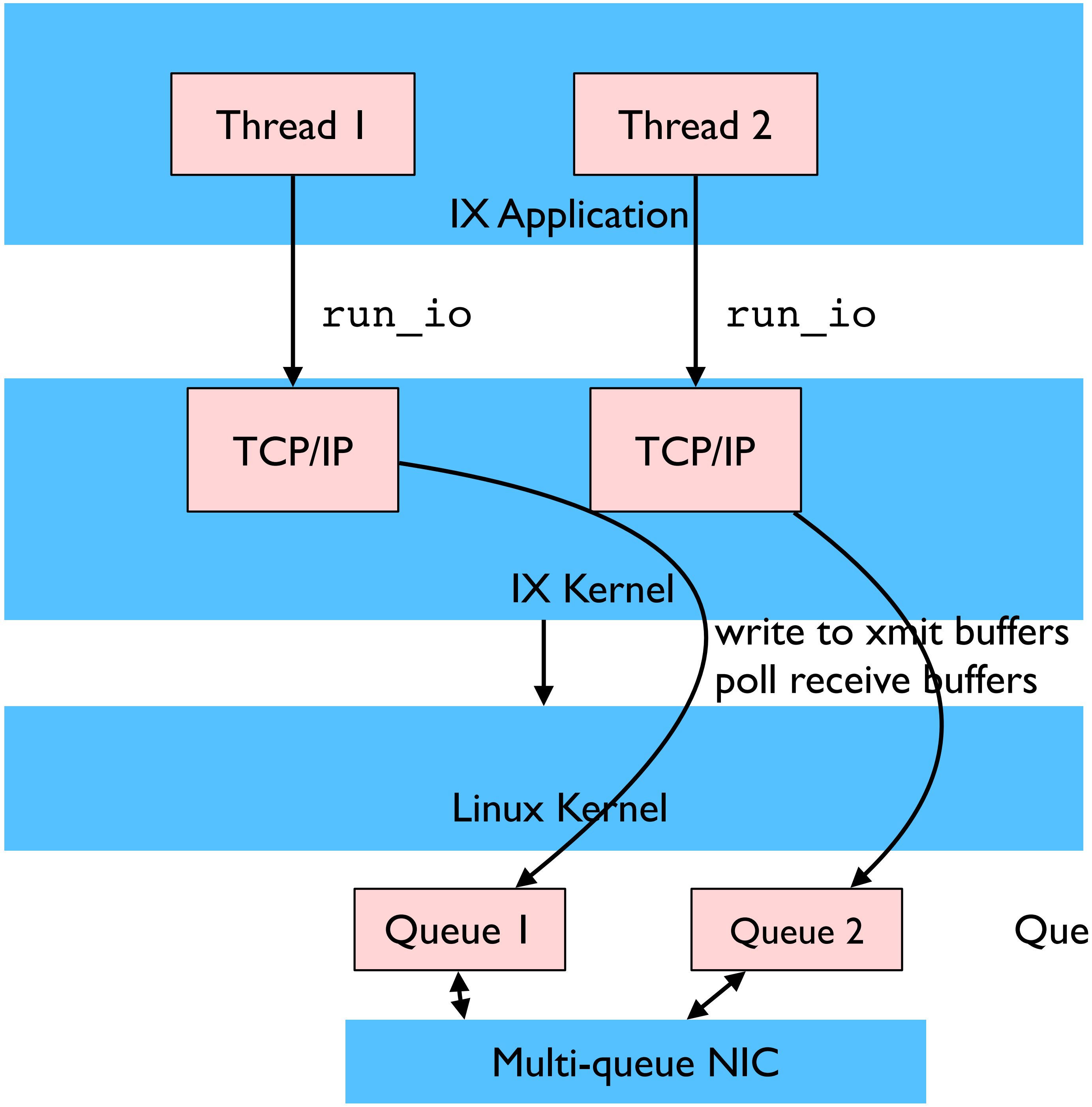
- Latency important for e.g., web page with hundreds of items
- High load: sum of a sequence of steps:
  - Latency is largely determined by wait time: queueing
  - Efficiency (high throughput) reduces queueing time
  - Bursty arrivals increase queue time
  - Bursty service times increase queue time
  - Structural problems can increase queue time
    - Load imbalance, or nobody servicing a queue

- Latency is hard to reason about: hard to improve

IX: a design for a high-performance network stack

- Built on top of Linux (with Dune kernel module)
- Different syscall API for networking (doesn’t preserve Linux API)
- Different TCP/IP stack architecture (doesn’t use Linux TCP/IP stack code or design)

Linux network software structure



Queues are actually in IX Kernel memory

IX Notes

- IX runs in VMX non-root (guest) mode using Dune
- IX Kernel at CPL 0
- IX App at CPL 3
- Linux kernel gives dedicated NIC queues and dedicated cores
  - After that, Linux isn't involved with networking
- IX application makes system call to IX kernel
  - To send and receive packets
- Packet buffers are in memory shared between IX kernel and IX application (and NIC)
  - So, packet data isn't copied (unlike Linux)

zero-copy!

Idea: batching system call interface

- The problem: System call overhead is big if messages are small
  - Want to send/recv more packets/sec than available syscalls/sec
- The solution: run_io()
  - `run_io()` argument contains one or more syscalls:
    - send to a TCP connection
    - done with a recv buffer
    - close/connect/accept
  - `run_io()` return contains:
    - Result of each of syscall, plus
      - recv on a connection
      - send completed
      - connection opened, connection terminated, …

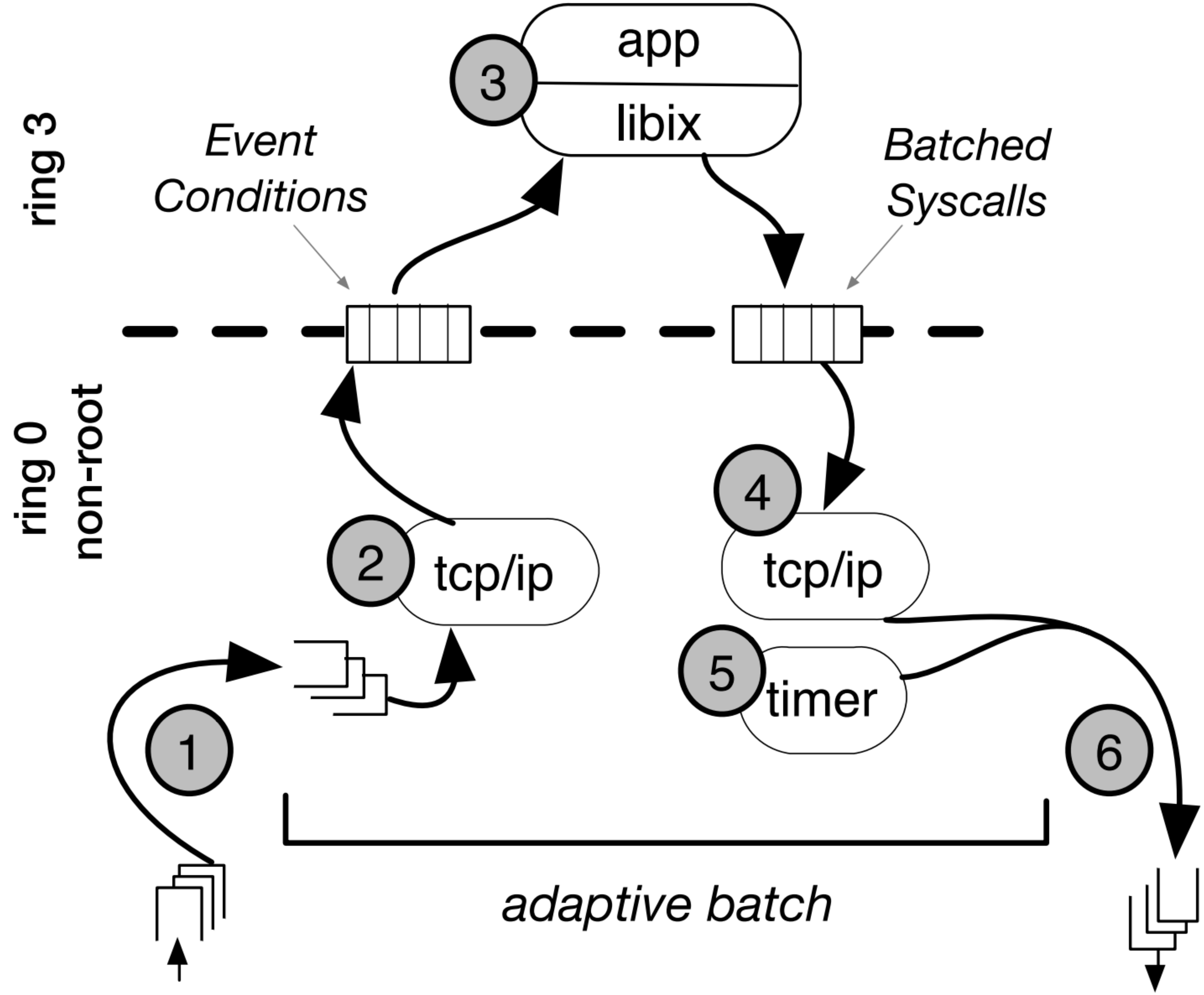
Idea: batching system call interface

- Each user/kernel crossing does lots of work
  - Amortizes syscall cost across lots of packets

```
while True:
  run_io(in, out)
  for msg in in:
    process msg
    out.append(reply)
```

pseudo-code for IX app thread

Idea: run to completion



(b) Interleaving of protocol processing and application execution.

Idea: run to completion

- The problem:
  - Linux uses CPU time moving packets through stages and queues
  - Queues:
    - Good if application is doing something else
    - Bad for network performance (locks, core-to-core, cache eviction)
- What is run-to-completion?

Idea: run to completion

- What is run to completion?
  - Complete the processing of one batch of inputs before starting on the next batch
  - Really complete: driver, TCP, application, enqueue reply
- How?
  - `run_io()` calls down to driver, returns packet all the way to app
  - app's next call to `run_io()` has reply message
- Why?
  - Single thread carries batch of packets thru all steps
  - Avoids queues, sleep/wakeup, context switch, core-to-core transfers
  - Keeps packet batch in CPU data cache
  - No problem balancing processing rate in each stage

Idea: polling rather than interrupts

- The problem:
  - Interrupts are expensive
  - Interrupts are redundant if input is always likely waiting
- What is polling?
  - Periodically check NIC DMA queues for new input
- Why hard?
  - Where to put the checks? In what loop?
  - Might check too often—waste CPU
  - Might check too rarely—high latency, queue overflow

Idea: polling rather than interrupts

- IX's solution:
  - Each application thread has a dedicated core:

```
while True:
  run_io(in, out)
  for msg in in:
    process msg
    out.append(reply)
```

  - `run_io` polls NIC DMA queues
  - No waste: if no input, nothing for the core to do anyway
  - If input, grabs a batch and returns it to the application
    - Never waits for a batch; just grabs what's there
  - Automatically polls more often with low load, less at high load
    - Paper calls this *adaptive polling*

What about multi-core parallelism?

- The problem:
  - One core often can’t deliver enough throughput
  - Will leave most of a 10Gb Ethernet idle
- Opportunity
  - Lots of clients
  - Work for each client is often independent
  - All modern machines have multiple cores
- The dangers
  - Lock contention is expensive
  - Data movement (between cores) is expensive

What about multi-core parallelism?

- To avoid data movement and lock contention:
  - All actions for a client, TCP, and packet should be on the same core
  - No data should be used on more than one core
- Examples of potentially shared data:
  - packet content
  - NIC queues
  - packet free lists
  - TCP data structures
  - Application data (e.g., memcached’s in-memory DB)

Idea: multiple NIC queues for parallelism

- Modern NICs support many independent DMA queues
  - NIC uses filters and hashing to pick the queue
- Linux sets up a separate set of NIC queues for each IX application
  - One queue per core for each IX application
  - Linux tells NIC a filter for each IX application

Idea: multiple NIC queues for parallelism

- NIC hashes client IP addr/port to pick the queue for each incoming packet
  - “flow-consistent hashing” or “receive-side scaling” (RSS)
  - NIC gives all packets for a given TCP connection to the same core
  - No need to share TCP connection state among all cores
  - No need to move packet data between cores
- `run_io` looks at NIC DMA queue for just its own core
- A new connection is given to the core determined by the NIC’s hash
  - Hopefully uniform and results in a balanced load

Idea: zero copy

- How to avoid IX/user and user/IX copies of packet data?
  - Across the CPL 0/CPL 3 boundary (like user/kernel)
  - 40 Gb/sec may stress RAM throughput
- IX uses page table to map packet buffers into both IX and application
  - NIC DMAs to/from this memory
  - `run_io` carries pointers into this memory
- App/IX cooperate to note when received/sent buffer is free
  - freed buffers reported via `run_io`

IX design limitations

- Assumes many parallel clients making small requests
  - You'd want something else for a single 40-Mb/sec transfer
- Assumes good load balancing across cores
  - Clients and requests evenly distributed across cores
  - Requests all take about the same amount of time
  - Could reassign flows to NIC queues?
  - Could steal work from other cores?
- Assumes non-blocking request handling
  - Service code computes and then replies
  - Does not: read the disk, send an RPC and wait, etc.
  - Blocking would cause an idle core and expanding queue
  - Could shift blocked requests to a dedicated thread/core?

Evaluation

- What should we look for?
  - High throughput under high load—especially for small messages
  - Low latency under light load
  - Throughput proportional to number of cores

Evaluation

- Low latency test
  - Single message ping-ponged between two servers on a 10Gb connection
  - Latency for a 64 byte message:
    - Between two IX servers: 5.7µs
    - Between two Linux servers: 24µs

Goodput: app-level throughput

Why increases?
Amortizes fixed costs over
larger amounts of data
Limited by 10Gb Ethernet - minus headers

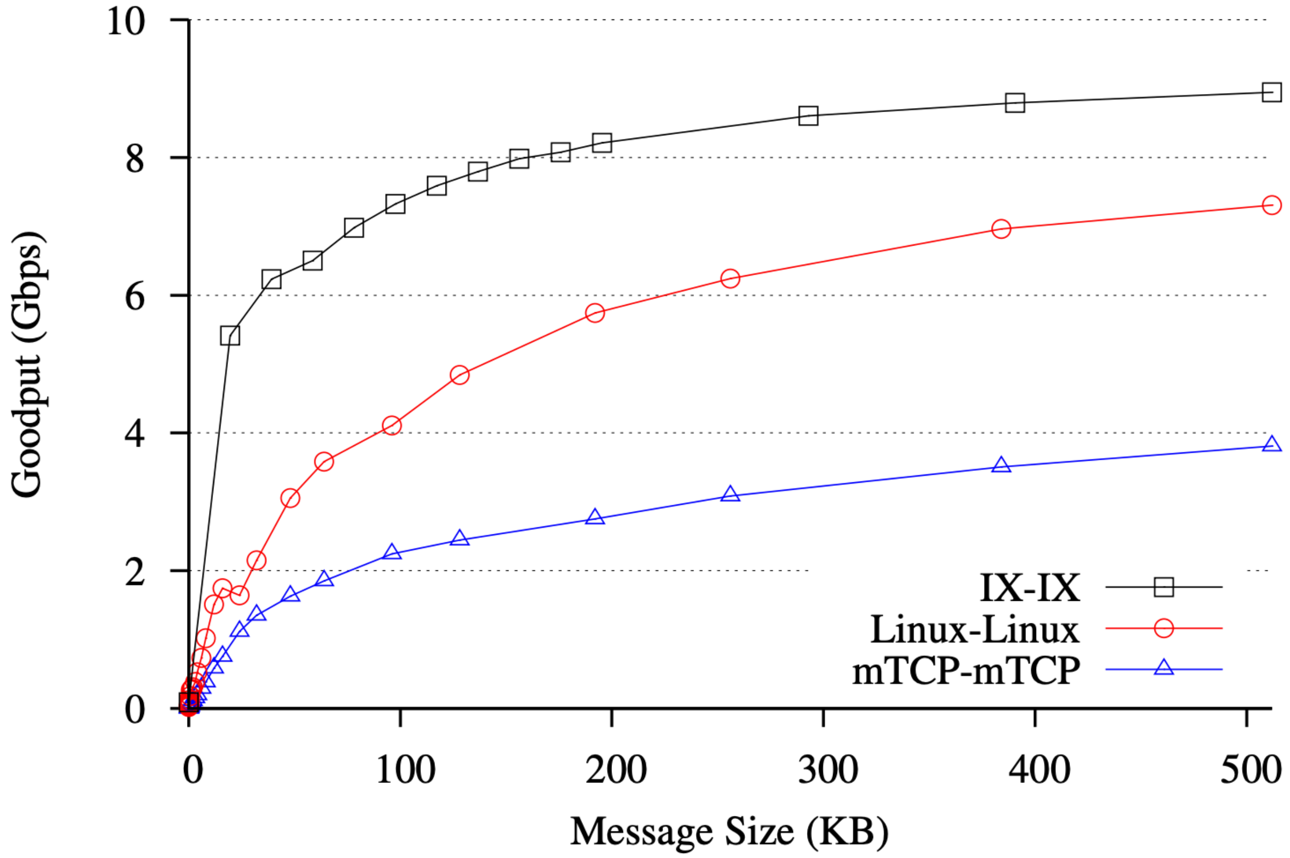


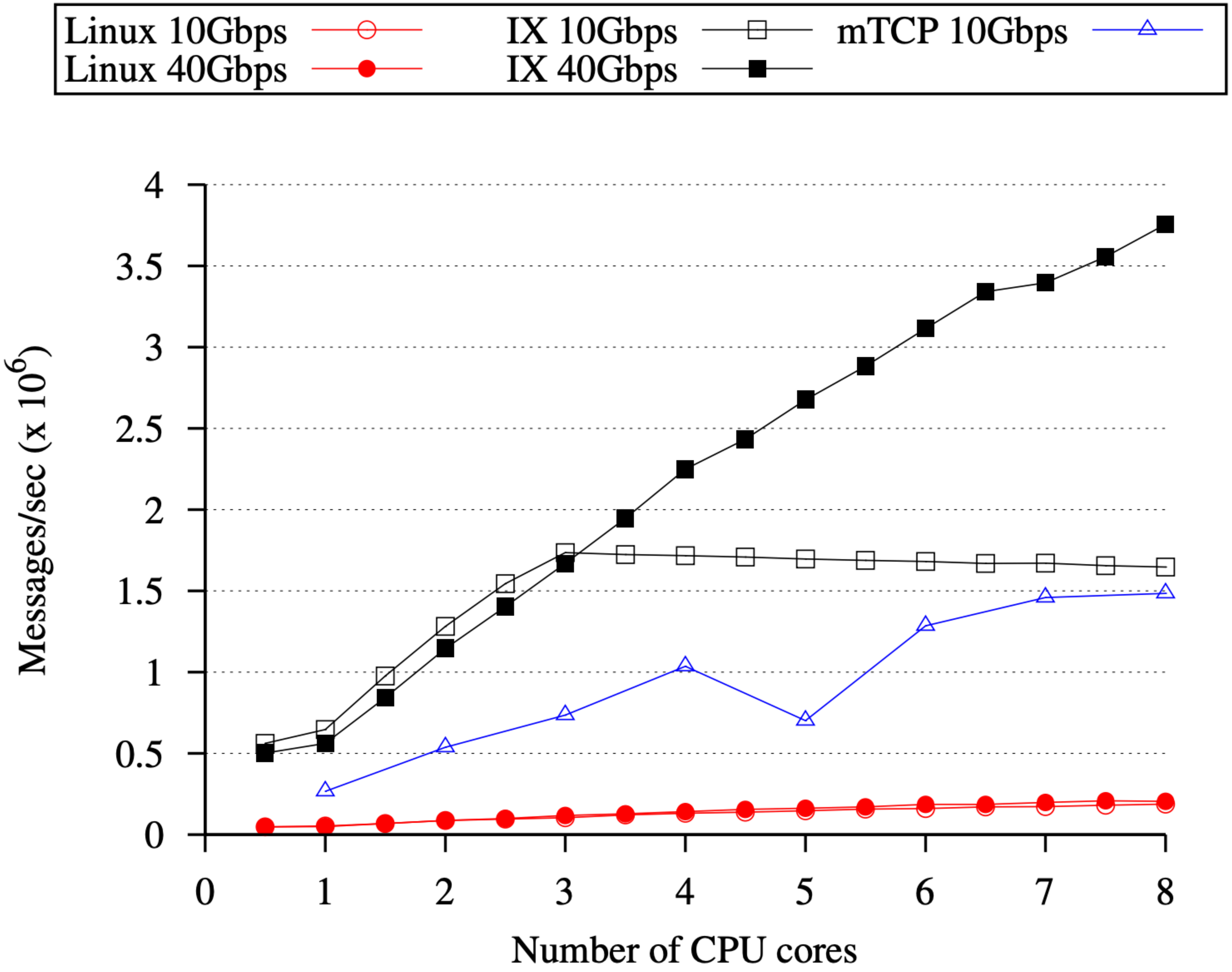
Figure 2: NetPIPE performance for varying message sizes and system software configurations.

Evaluation

- Low latency test with small packets
  - Why does IX beat Linux on goodput?
    - Latency-limited
    - IX polling sees the message sooner
    - IX has no interrupt/queuing/sleep/wakeup
    - Fewer user/kernel crossings

Evaluation

- Multi-core scalability



(a) Multi-core scalability (n=1, s=64B)

Summary: IX makes many big architectural decisions differently

- Per-application network stack
    - Rather than single shared stack
    - Allows packet buffers to be shared: zero copy
- Dedicated cores to application threads
    - Rather than shared cores multiplexed by kernel
    - Allows polling and run to completion
    - Helps make the software more efficient, and simpler
    - Requires plentiful cores
- Dedicated NIC queues to application threads
    - Rather than shared queues, multiplexed by kernel
    - More direct access for better efficiency
    - Requires plentiful NIC queues

Questions

- What is adaptive batching?
  - Never wait for packets
  - Upper bound on size of batch
- Could a single app disable reception for all other apps by acquiring all the buffers?
- What is *zero-copy?*
- When would one not necessarily want high throughput and low latency?
- What is a *data plane?*
  - The code responsible for manipulating the packets
- What is the hardware/OS mismatch?
  - Hardware should support high throughput/low latency
  - Most OSes are not designed to use the hardware well

Questions

- What are tradeoffs of using IX (other than only being able to run one app)?
  - Different API
  - Possibly-wasted/underutilized cores
  - Actually, can run ≥1 app
- What is RDMA?
  - User-level reads/writes of remote memory
  - Fast because goes directly from local NIC to remote NIC to registered memory, bypassing the remote OS
- Are elastic threads some type of sthread?
  - No, just thread with its own CPU and NIC queue