6.5810: Host networking

Adam Belay <abelay@mit.edu>

Some motivation

- 2009 (Nehalem): 4 cores * 3GHZ / (10GbE / MTU)

= **13,440** cycles per packet

- 2022 (Ice Lake): 32 cores * 2.3GHZ / (200GbE / MTU)

= **4,121** cycles per packet

Challenge in paper is 3x harder today!

Cycle costs for perspective

Budget: ~4000 cycles

- **Interrupts:** 200–2000 cycles (depending on TLB etc.)
- **Malloc:** 30-200 cycles
- **Kernel transition:** 200+ cycles

Other operations: Protocol handling (e.g., TCP), pacing + QoS, memory copying, locking + synchronization, segmentation, descriptor ring management, ref counting, thread scheduling, socket API, etc.

Q: How can we saturate NICs with today's CPUs?

- Better question: How can we take advantage of NIC performance without wasting a ton of cores!
- Budget should be much smaller than 4000 cycles!

- Caladan today: ~2000 cycles per packet w/ TCP
  - Up to 10x faster than Linux depending on packet size + MTU
  - Includes thread scheduling and socket API cost

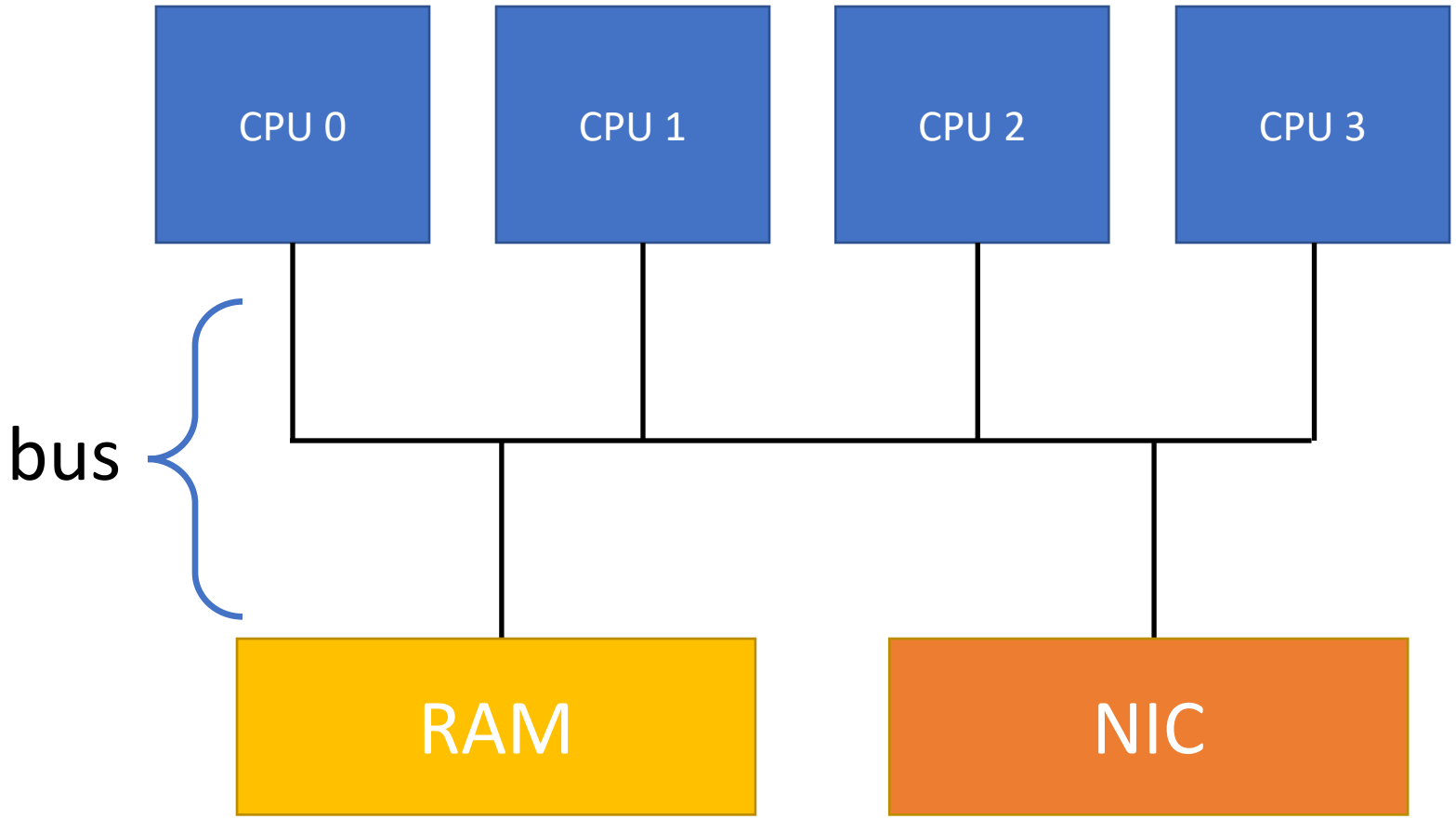
Sockets

- Abstraction for communicating between machines
  - Requires copying of data from packet buffers to application buffers
- **Datagram sockets**: Unreliable message delivery
  - e.g. UDP
  - Messages may be reordered or lost
  - Reads return the full message (if req len is large enough)
- **Stream sockets**: Bi-directional pipes
  - e.g. TCP
  - Bytes written on one end, are read on the other
  - Reads may not return the full amount requested

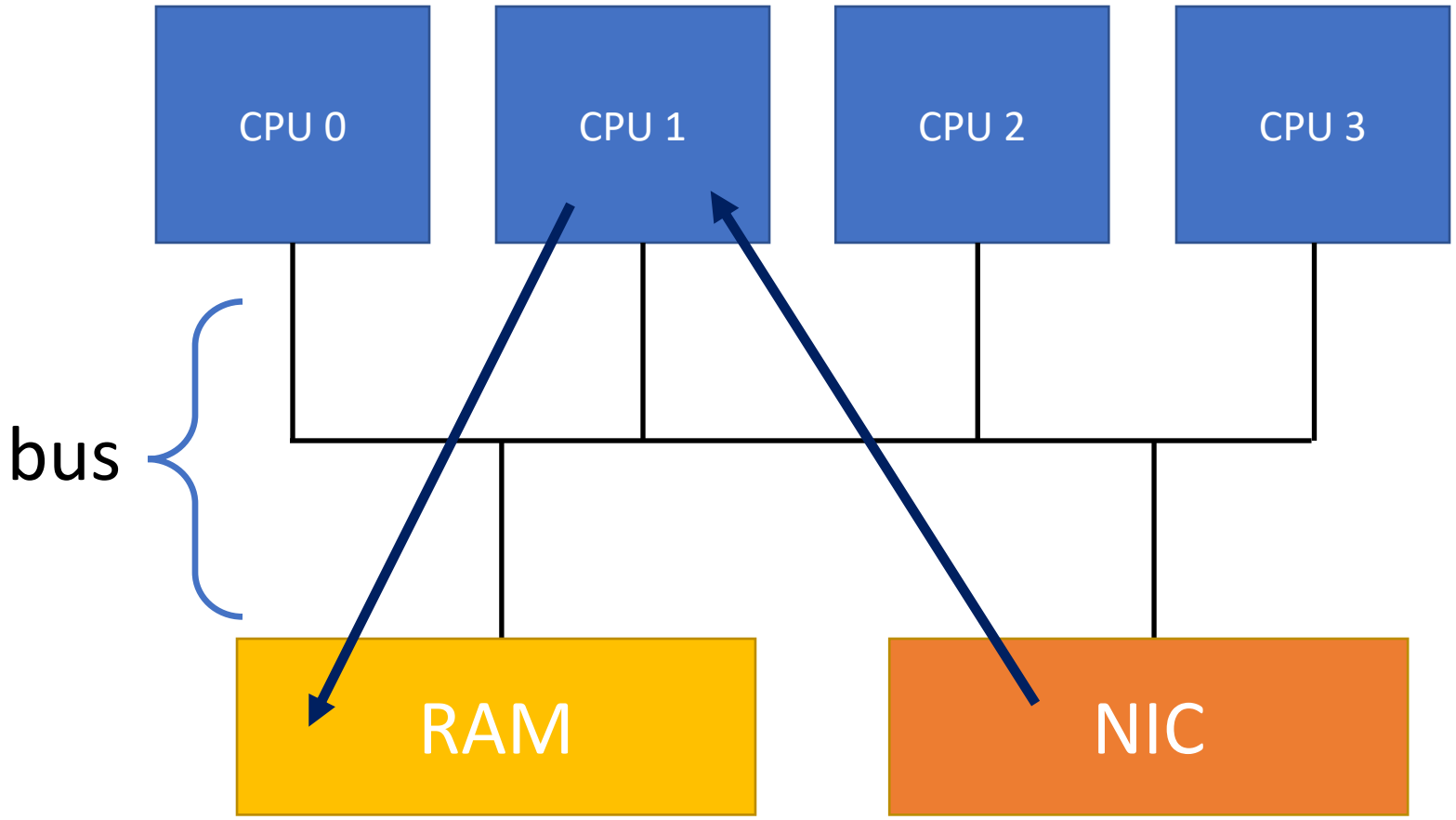
Socket implementation

- Both TCP and UDP name connection endpoints
  - 32-bit IP address specifies machine
  - 16-bit Port number demultiplexes within host
- Thus, a connection is named by 5 components
  - Protocol (UDP), local IP, local Port, remote IP, remote Port (called a 5-tuple)
- OS keeps connection state in PCB structures
  - Keep all PCBs in a hash table
  - When packet arrives, use 5-tuple to find PCB and use PCB to determine what to do with packet

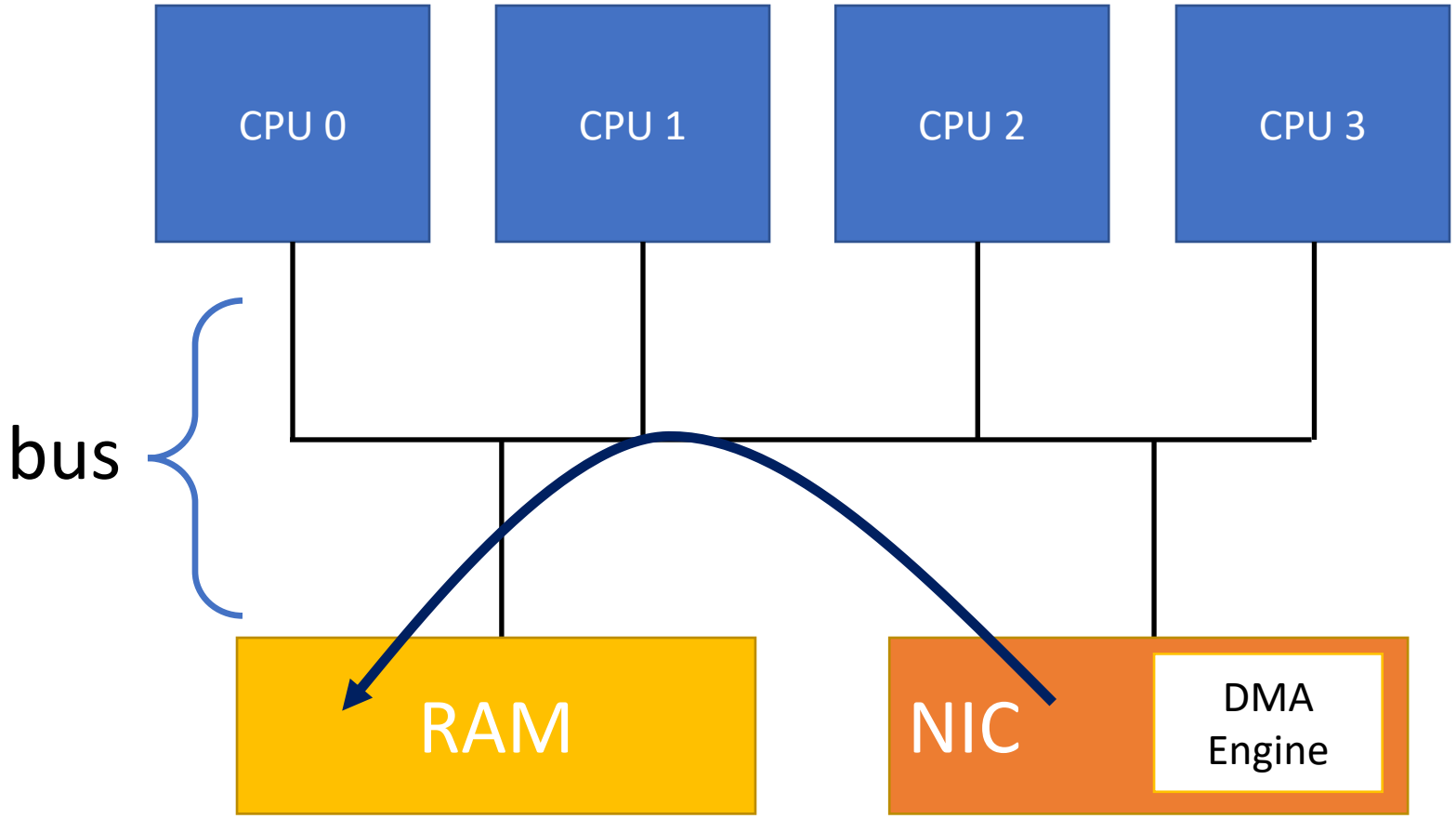
Recap: How to transfer pkts to/from NIC?



Idea: Have CPU copy to RAM

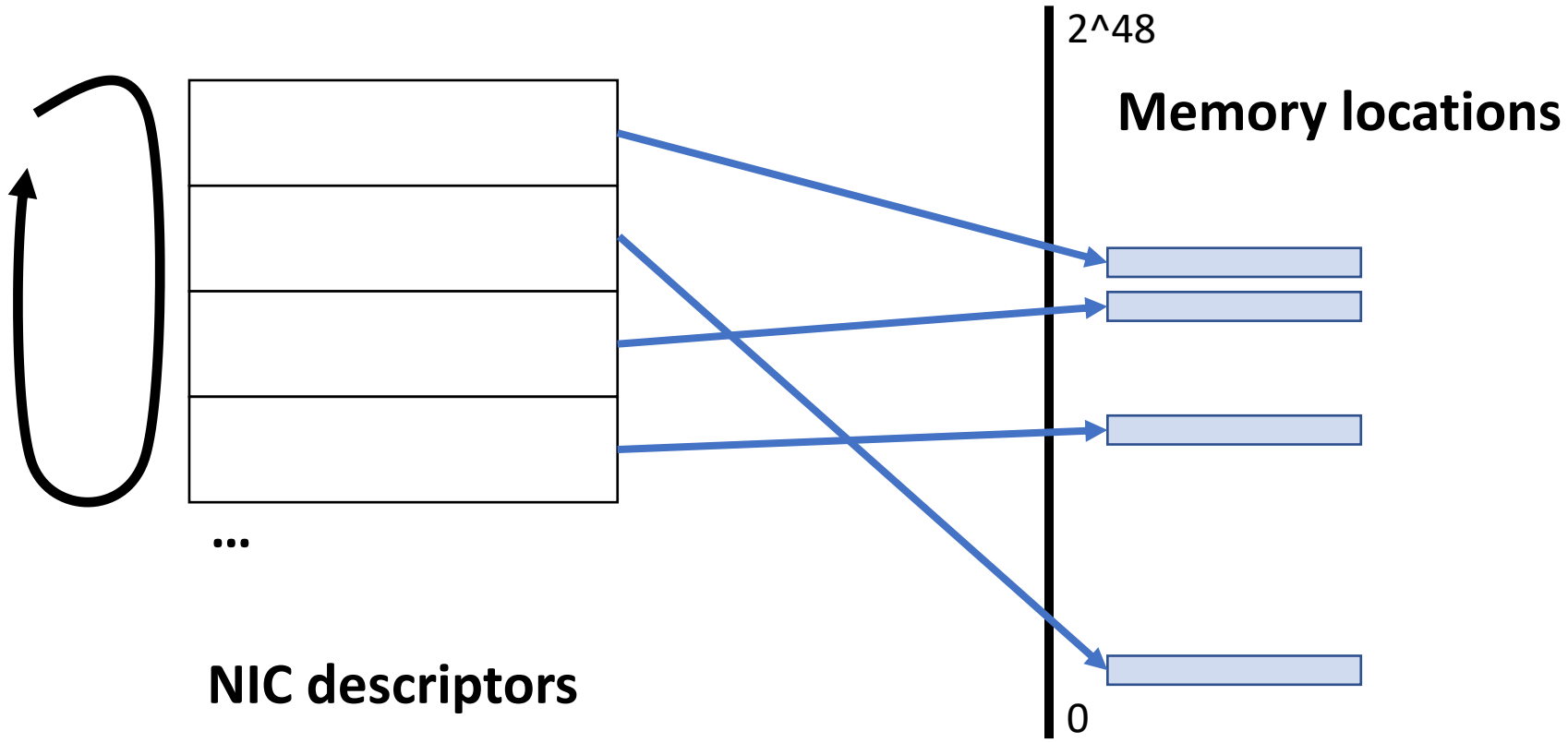


Better: Have NIC copy to RAM
Called direct memory access (DMA)



OS programs DMA engine

- Circular array of descriptors (fetched from memory by NIC)
- Each descriptor describes location to put packet in memory



DMA details

- OS provides NIC with locations to copy packet data
- NIC provides OS with notifications of finishing
  - Mechanism #1: Writes done flag to descriptor
  - Mechanism #2: Sends interrupts
- OS recycles descriptors
  - Gives previous buffer to networking stack (or frees it)
  - Then allocates and programs new buffer into descriptor
- Descriptors often contain flags and metadata about how to receive or transmit a packet
- Separate descriptor rings for receive and transmit

Don’t starve the DMA engine

- Need to keep descriptor rings full!
- What if receive ring goes empty?
  - NIC drops packets!
- What if transmit ring goes empty?
  - NIC wastes bandwidth! (doesn’t send)
- OS must constantly monitor descriptors!
  - Can poll, by checking them periodically
  - Or can program NIC to send interrupts

The Paper: TCP onloading

- Context: Written by Intel; vendor of both NICs and CPUs
  - Discusses several existing optimizations and proposes new optimizations
- Concern over whether CPUs could keep up with 10GbE networks
- Same problem is happening today with 200GbE+ networks

Existing optimizations (in 2004)

- **Interrupt moderation:** Accumulate packets before sending interrupts
  - Results in less interrupt overhead but **higher latency**
- **Checksum offload:** Calculate internet protocol checksums in HW
  - Some mistakes made in early hardware, but eventually a simple one's complement of the packet was sufficient to support nearly all checksums
  - **Only beneficial**, a great and simple offload
- **Large segment offload**: Segments TX transfers larger than MTU in HW
  - Requires just enough TCP state (in the descriptor) to set up the headers
  - Causes bursts in network fabric, resulting in **higher latency**

Challenges identified in the paper

1. **OS overhead:** software layers, interrupts, system calls, buffer mgmt.
2. **Poor cache locality:** DMA places packet buffer in RAM, not cache
3. **Data copying:** An extra copy from packet buffer to application buffer is required

TCP Offload (TOE)?

- Idea: Put the entire TCP network stack in the NIC
- Sounds great, but many challenges:
  - Most of the overhead isn't TCP: interrupts, buffers, copying, system calls, etc.
  - TCP has a ton of branches and memory state, normal CPUs are great at this
  - Generally, the NIC has worse caching, less local memory, less bandwidth to DRAM, less investment in silicon process
  - Flexibility: Hard or impossible to evolve TCP protocol (e.g., BBR)
  - Kernel developers wanted more control of networking
- Little adoption in practice

Future enhancements discussed in paper

- Async I/O: More efficient but hard to use programming model
- Header splitting: Put headers in a separate place from data
- Receive side scaling: Spread packets across cores with hashing

RSS was a huge win; but the others have mixed value today

What does TCP onloading mean?

- Plan: Make CPUs better at processing packets
- Onload is the opposite of offload, or placing the work on the NIC
- Intel developed a collection of CPU changes to reduce overhead, some of which made it into today's hardware

Direct cache access (DCA)

- Problem: Packet buffers are DMAed into DRAM, simply accessing the memory will eventually be too costly
- Solution: Place descriptors and payloads directly in the CPU cache
- Some interesting problems, such as cache interference with applications
- Today Intel calls this DDIO

Copy engine

- Problem: CPU cores spend a nontrivial time moving data
- Solution: Add dedicated hardware to perform memcpy (not cores)
- Completion processing is still hard and potentially costly
  - Interrupts or polling, locality etc.
- Rapidly evolving space even today

Lightweight threading (hardware)

- Problem: CPU cores stall waiting for memory loads
- Solution: Hardware threading (similar to hyperthreading)
- Challenge: Too much threading can harm latency
- Also, DCA reduces these stalls
- So far, this hardware has not materialized

Other ideas

- Increase the maximum transmission unit (MTU)
  - 1400 bytes -> 9000 bytes possible on ethernet (jumbo frames)
  - Reduces packet rate, enormous benefits for bulk transfers
- Receive-side coalescing (RSC)
  - NIC combines in-order TCP frames
  - Effect almost the same as larger MTU in practice
  - But fails when there is pacing or many active connections
- Change the socket API to support zero-copy
  - Worth it for bulk transfers; no benefit for small transfers

Recent developments

- RDMA: NIC can read remote memory without CPU involvement (on the remote side)
  - Like TOE, protocol-level handling happens in the NIC
- CXL: PCIe bus will soon be cache coherent with the CPU
  - CPU can directly load and store data exposed over bus
  - CXL 2.0 (not out yet) will expose memory over a switch to many machines
  - Latency will be higher than DRAM; potential performance challenges

Recap: Primary overheads

1. Interrupts
2. System calls
3. Copying
4. TCP/IP protocol
5. Buffer management
6. Driver
7. Core Scheduling / Threading

Hard to eliminate any of these steps in a general solution

DPDK is specialized; can skip interrupts, system calls, copying, and scheduling; but can only support a limited set of applications

Discussion: What should we do next to reduce host networking overhead?