UNIVERSITY OF CAMBRIDGE

P51: High Performance Networking

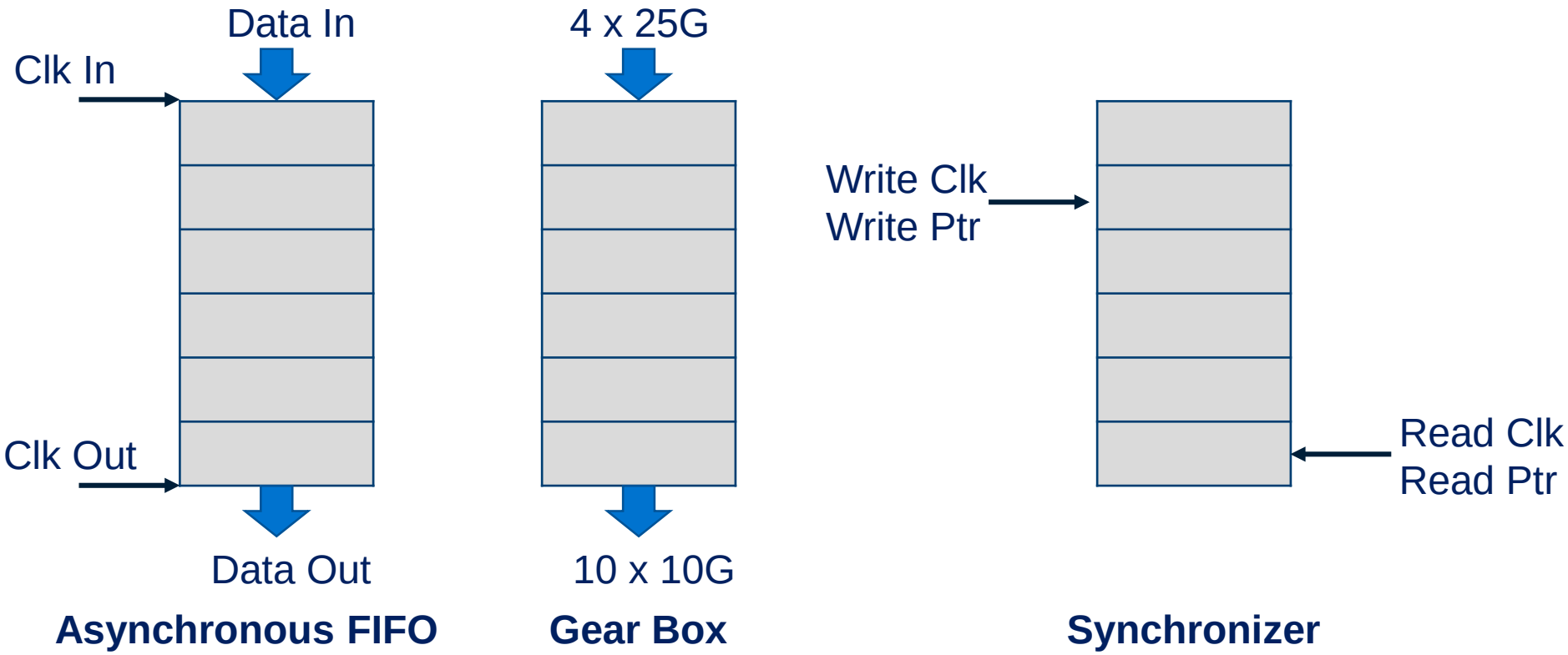
Lecture 4: Low Latency Devices

Dr Noa Zilberman
noa.zilberman@cl.cam.ac.uk

Lent 2017/18

Crossing Clock Domains

- Last week we discussed the clock frequency required in different places in the design.
- Crossing clock domains requires careful handling

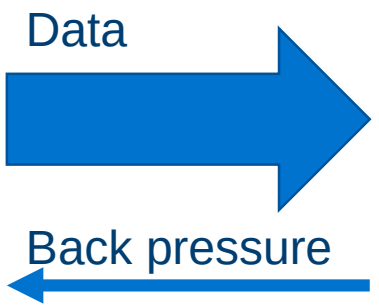


Crossing Clock Domains

- Why do we care about clock domain crossing?
- Adds latency
- The latency is not deterministic
  - But bounded
- Crossing clock domains multiple times increases the jitter
- Using a single clock is often not an option:
  - Insufficient packet processing rate
  - Multiple interface clocks
  - Need speed up (e.g., to handle control events)

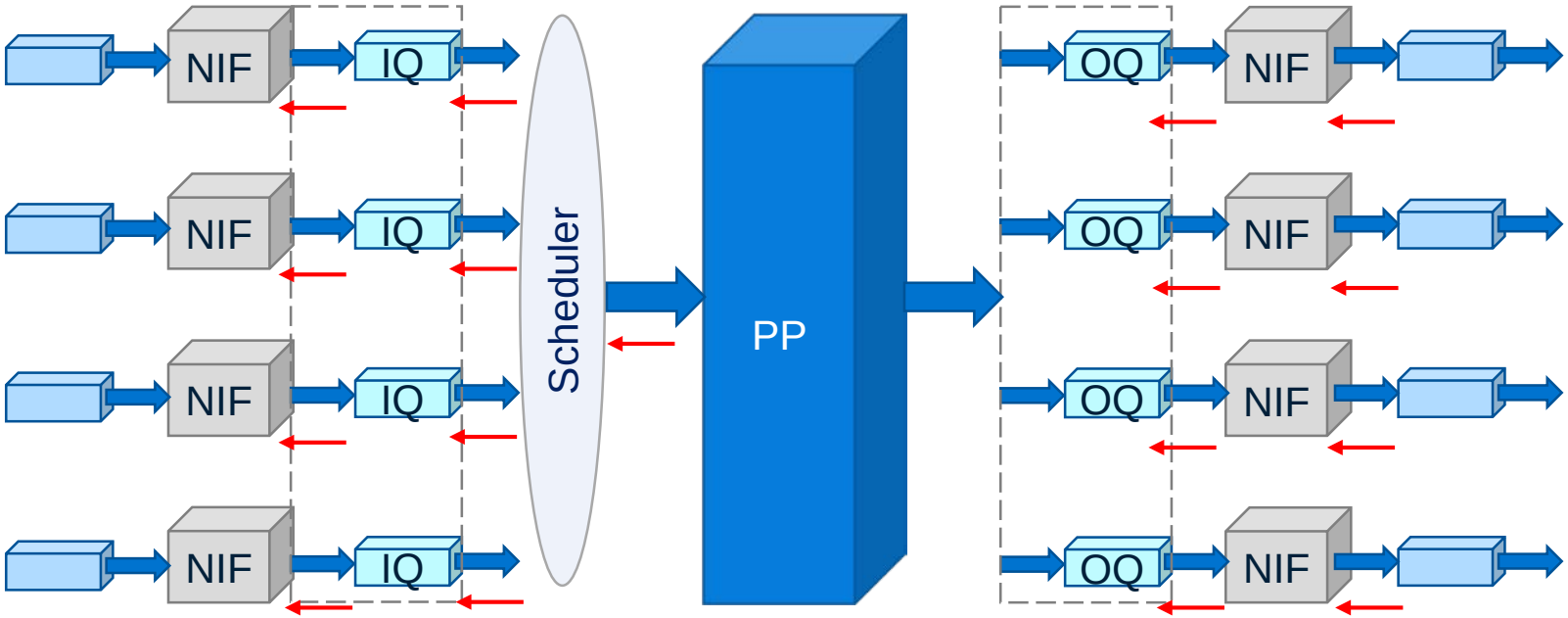
Flow Control

- The flow of the data through the device (the network) needs to be regulated
- Different events may lead to stopping the data:
  - An indication from the destination to stop
  - Congestion (e.g. 2 ports sending to 1 port)
  - Crossing clock domains
  - Rate control
  - …



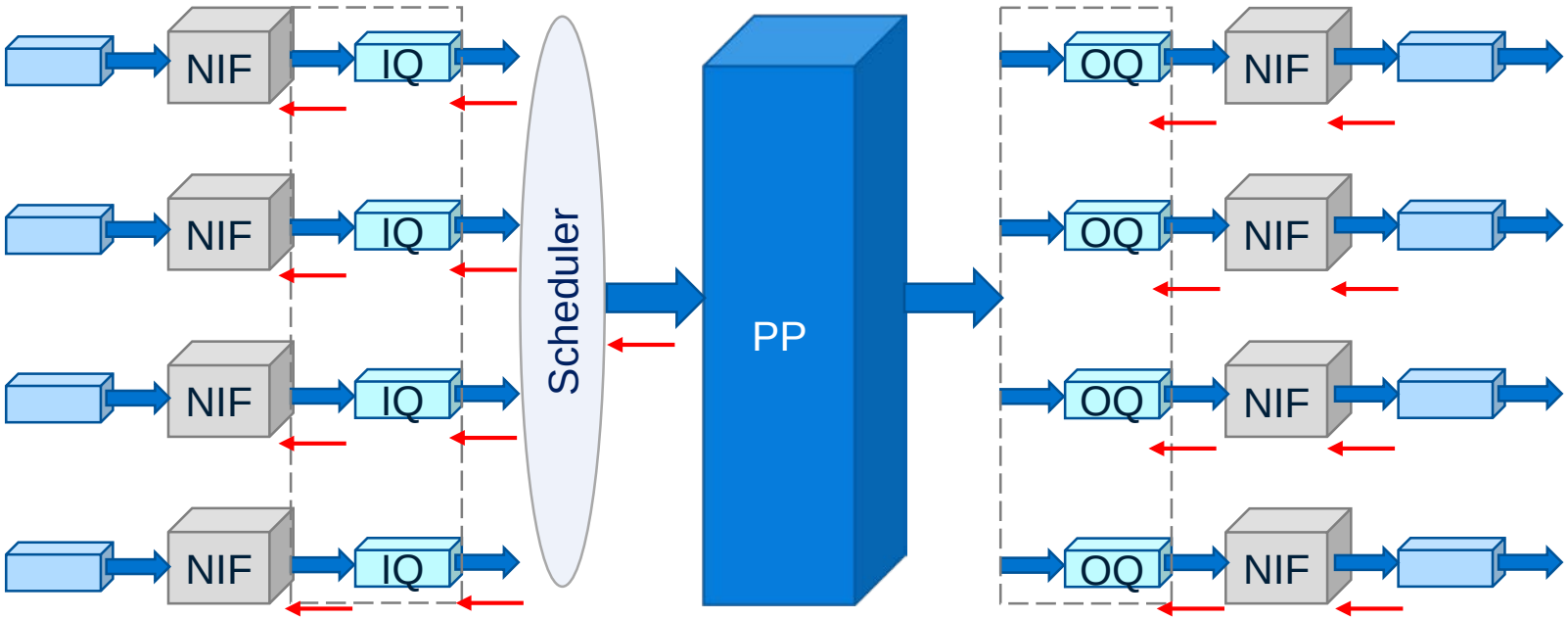
Flow Control

- Providing back pressure is not always allowed
- In such cases, need to make amendments in the design



Flow Control

- What to do if an output queue is congested?



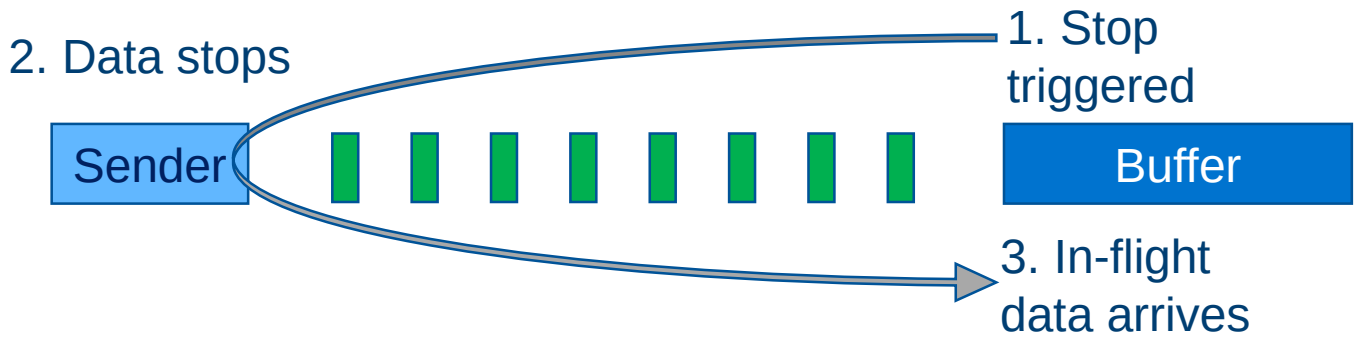
Flow Control and Buffering

- Back pressure may take *time*

- Need to either:
  - Assert back pressure sufficient time before traffic needs to stop
    OR
  - Provide sufficient buffering

Flow Control and Buffering

Calculating buffer size:



Intuitively:

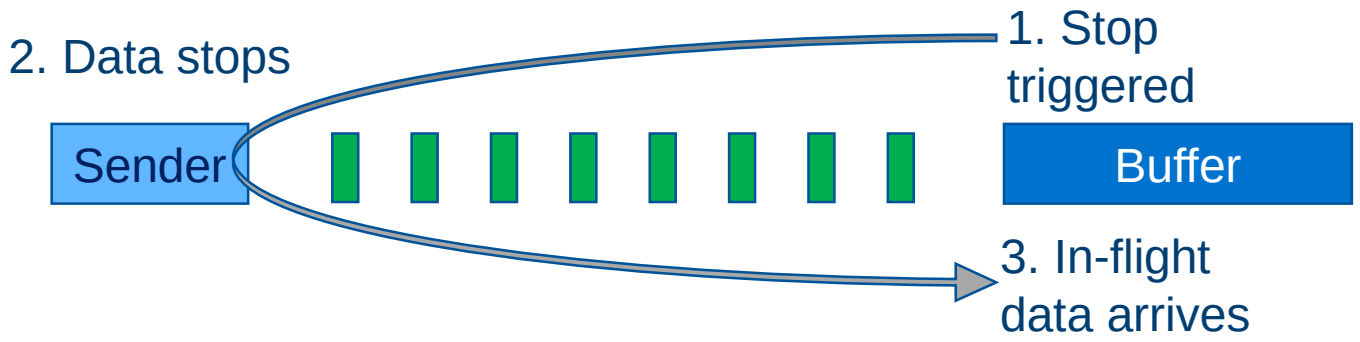
Nearby sender: Buffer size $\geq$ Reaction time $\times$ Data rate

Remote sender: Buffer size $\geq$ RTT $\times$ Data rate

Buffer size $\geq$ (RTT + Reaction time) $\times$ Data rate

Flow Control and Buffering

Calculating buffer size:



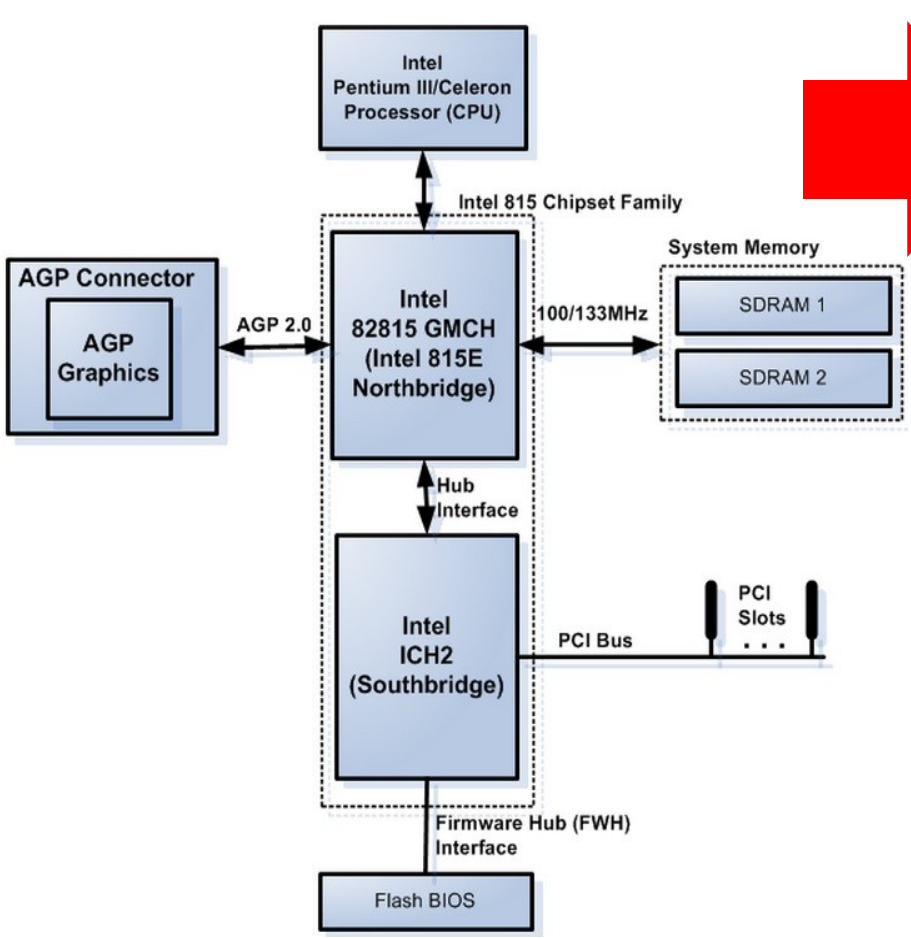
2 switches, connected using 100m fibre, 10G port, instantaneous response time:

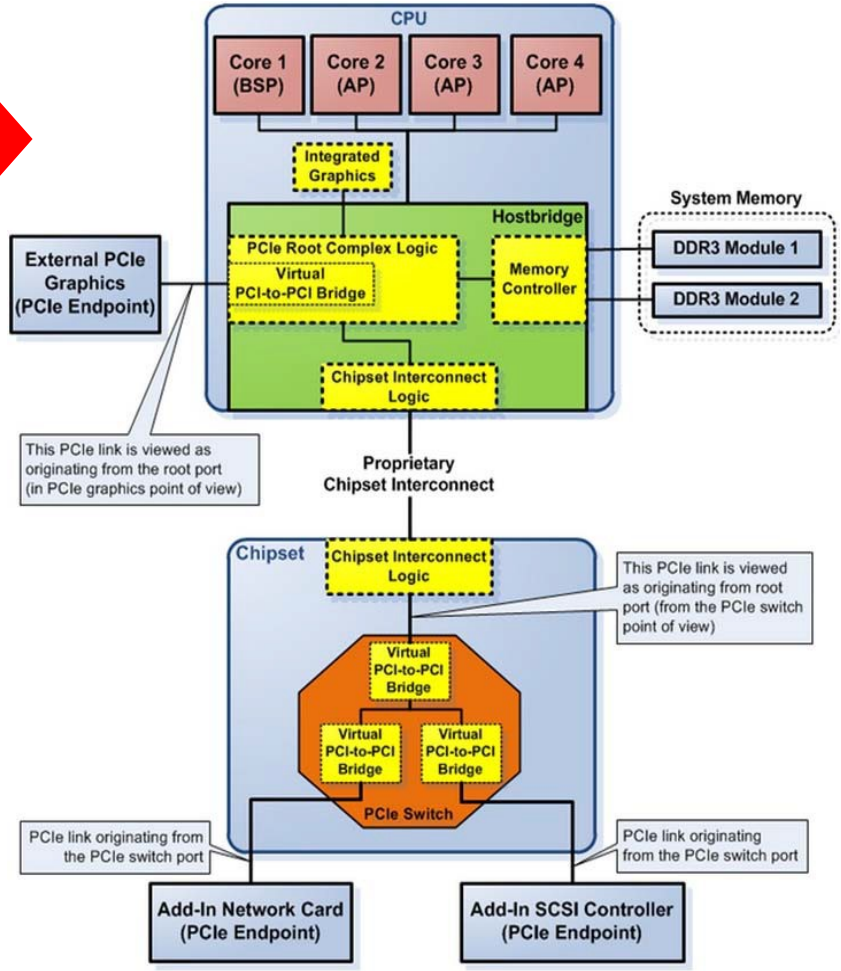
Propagation delay in a fibre is 5ns/m

Buffer size $\geq$ 1us $\times$ 10Gbps = ~1.25KB

DMA

Host architecture

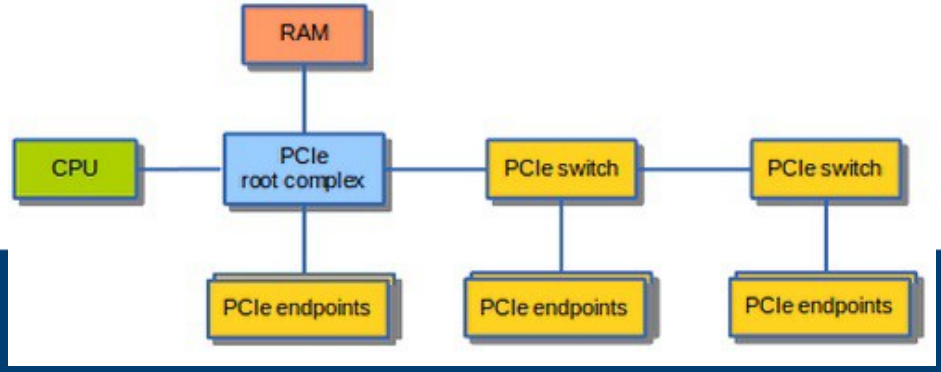




Legacy vs. Recent (courtesy of Intel)

Interconnecting components

- **Need interconnections between**
  - CPU, memory, storage, network, I/O controllers
- **Shared Bus: shared communication channel**
  - A set of parallel wires for data and synchronization of data transfer
  - Can become a bottleneck
- **Performance limited by physical factors**
  - Wire length, number of connections
- **More recent alternative: high-speed serial connections with switches**
  - Like networks



I/O System Characteristics

- **Performance measures**
  - Latency (response time)
  - Throughput (bandwidth)
  - Desktops & embedded systems
    - Mainly interested in response time & diversity of devices
  - Servers
    - Mainly interested in throughput & expandability of devices
- **Reliability**
  - Particularly for storage devices (fault avoidance, fault tolerance, fault forecasting)

I/O Management and strategies

- **I/O is mediated by the OS**
  - Multiple programs share I/O resources
    - Need protection and scheduling
  - I/O causes asynchronous interrupts
    - Same mechanism as exceptions
  - I/O programming is fiddly
    - OS provides abstractions to programs

Strategies characterize the *amount of work* done by the CPU in the I/O operation:

- **Polling**
- **Interrupt Driven**
- **Direct Memory Access**

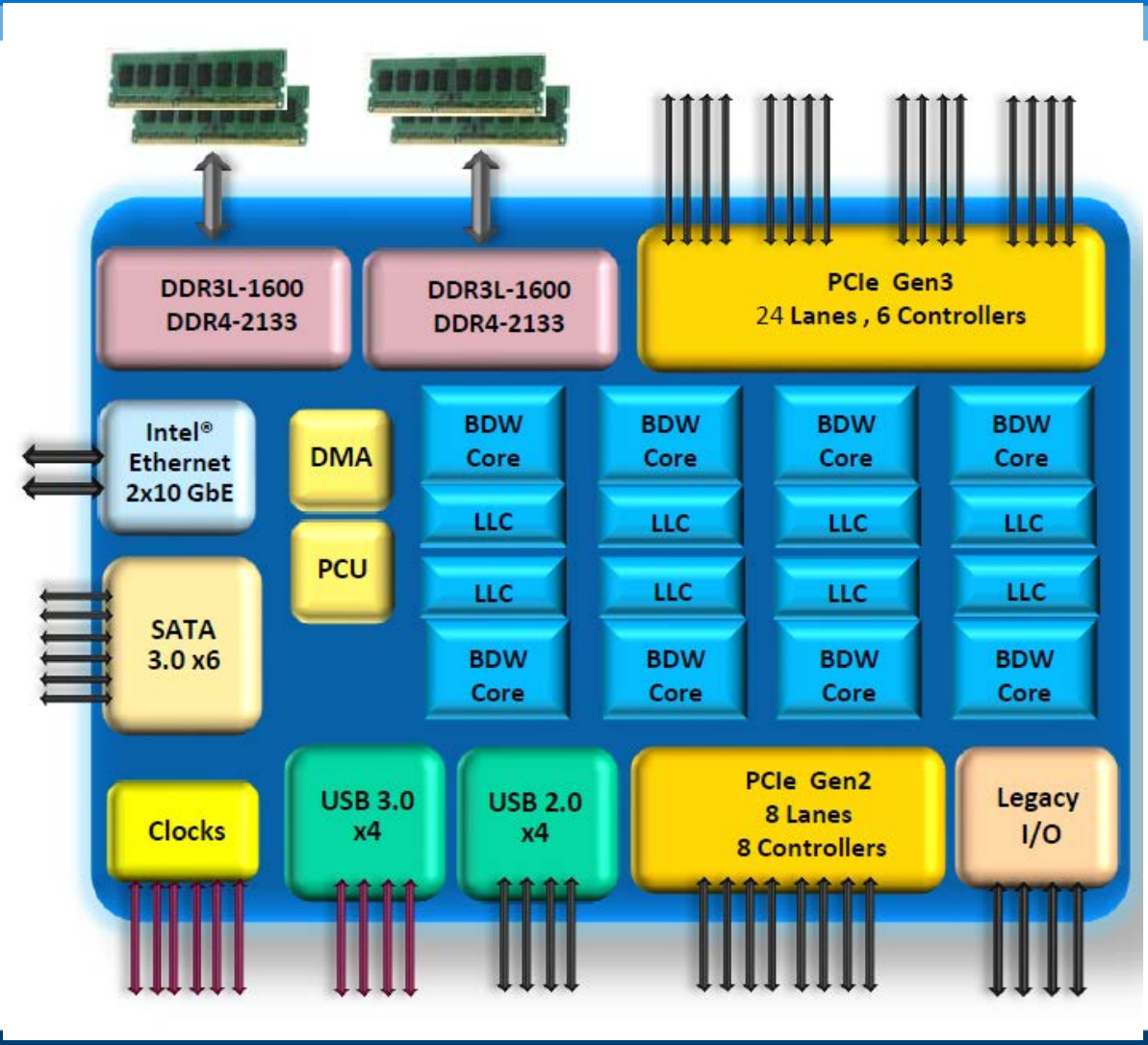
The I/O Access Problem

- Question: how to transfer data from I/O devices to memory (RAM)?
- Trivial solution:
  - Processor individually reads or writes every word
  - Transferred to/from I/O through an internal register to memory
- Problems:
  - Extremely inefficient – can occupy a processor for 1000's of cycles
  - Pollute cache

DMA

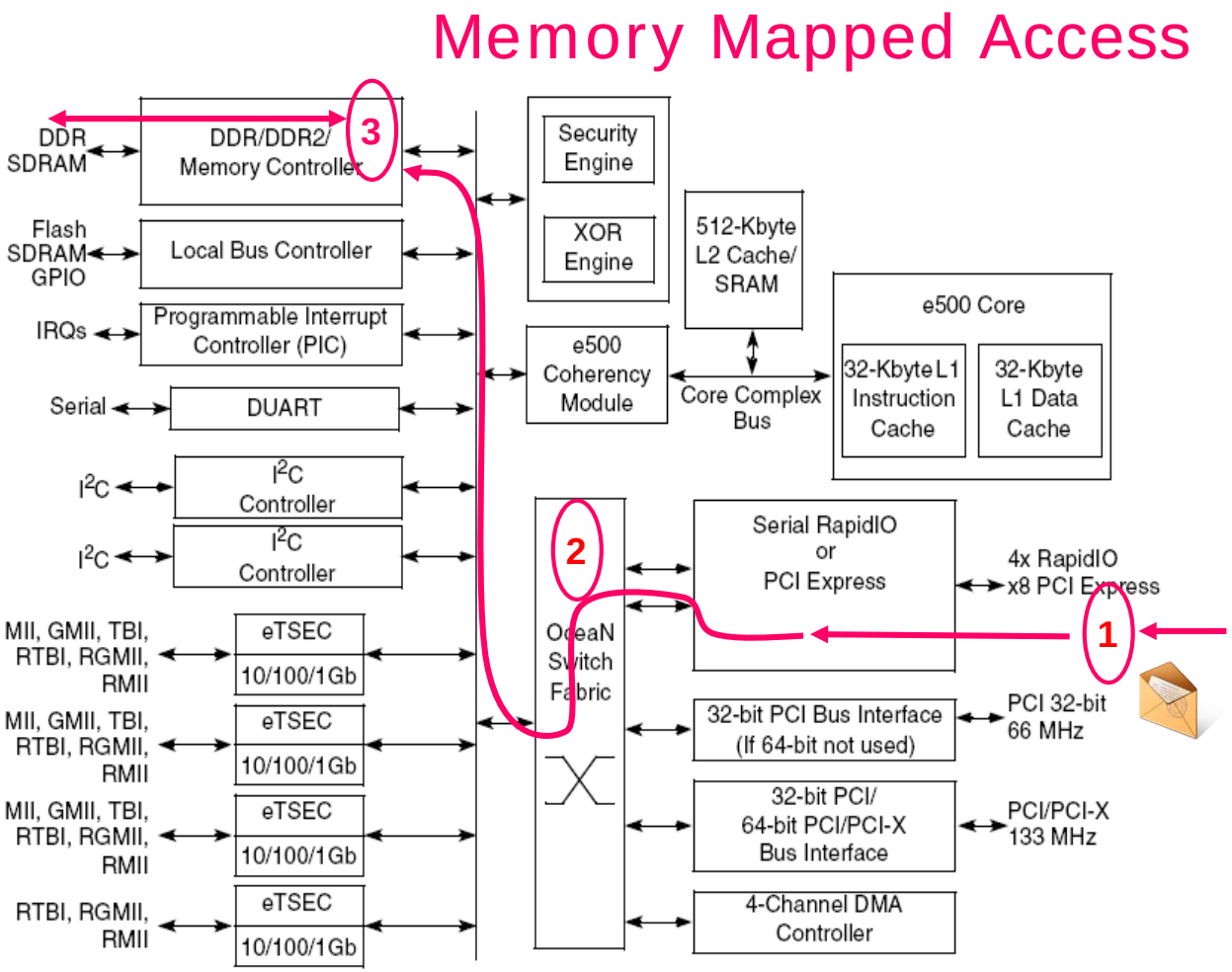
- DMA – Direct Memory Access
- A modern solution to the I/O access problem
- The peripheral I/O can issue read/write commands directly to the memory
  - Through the main memory controller
  - The processor does not need to execute any operation
- Write: The processor is notified when a transaction is completed (interrupt)
- Read: The processor issues a signal to the I/O when the data is ready in memory

Example – Intel Xeon D



Example (Embedded Processor)

1. Message arrives on I/O interface.
   Message is decoded to Mem read/write.
   Address is converted to internal address.
2. Mem Read/Write command goes through the switch to the internal bus and memory controller.
3. Memory controller executes the command to the DRAM.
   Returns data if required in the same manner.



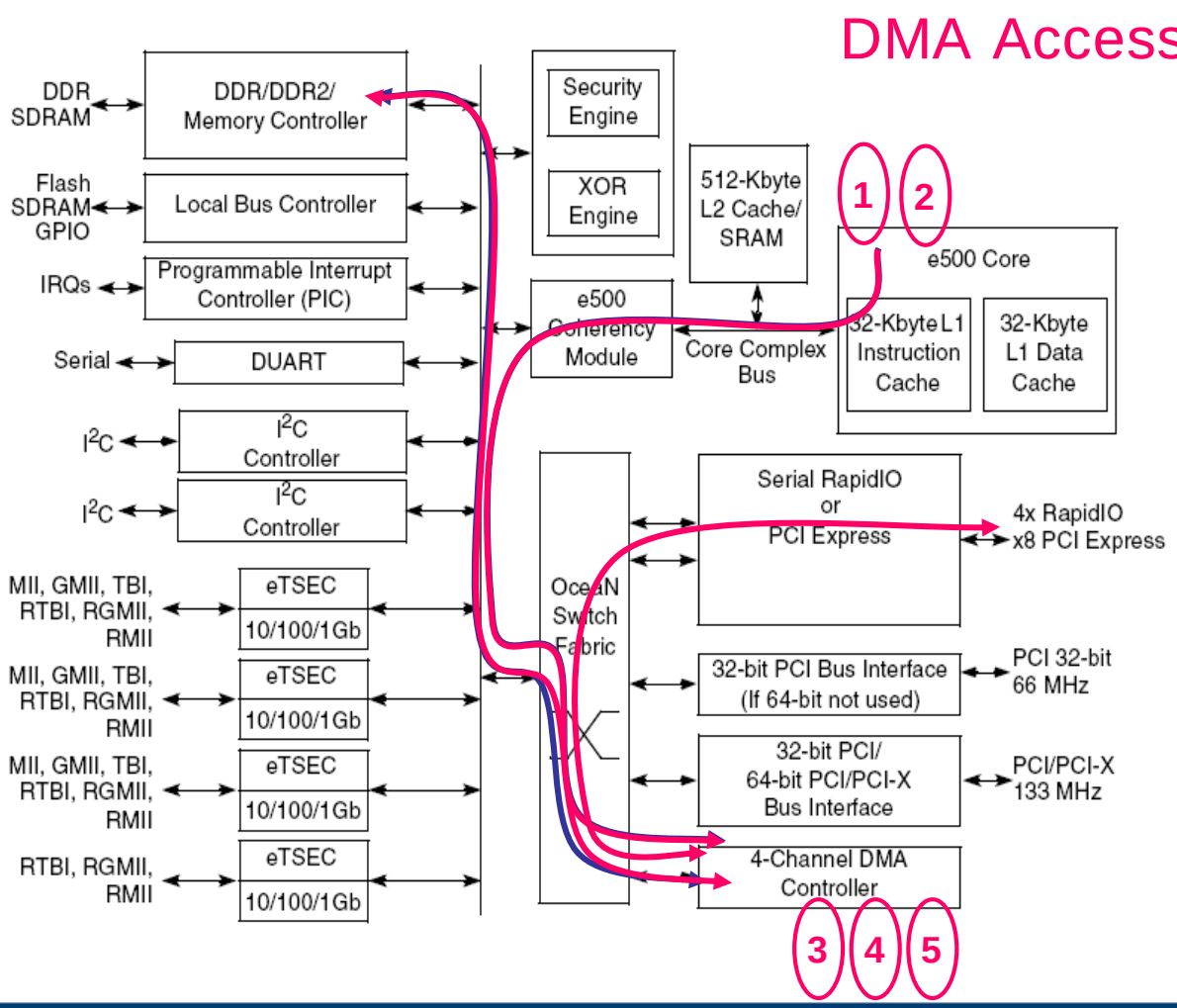
DMA

- DMA accesses are usually handled in *buffers*
  - Single word/block is typically inefficient
- The processors assigns the peripheral unit the buffers in advance
- The buffers are typically handled by *buffer descriptors*
  - Pointer to the buffer in the memory
  - May point to the next buffer as well
  - Indicates buffer status: owner, valid etc.
  - May include additional buffer properties as well

Example (Embedded Processor)

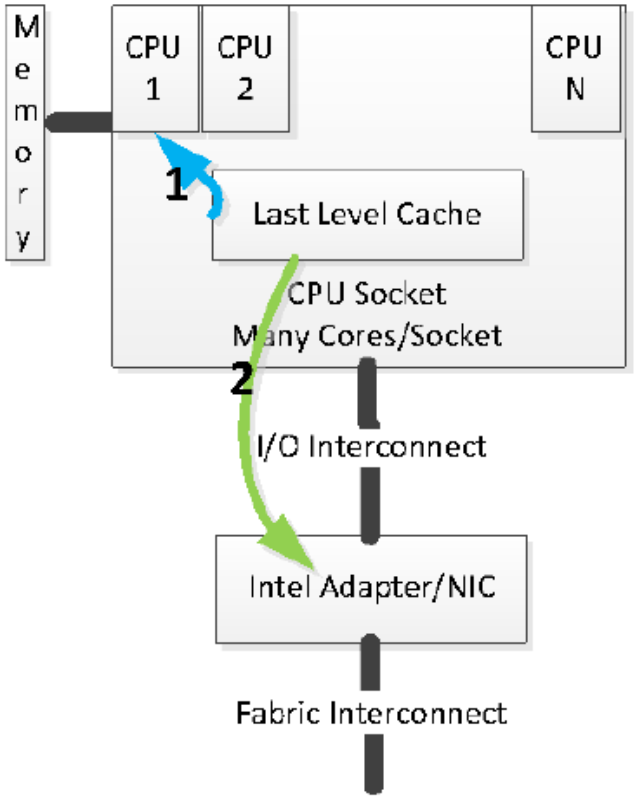
Transfers blocks of data between external interfaces and local address space

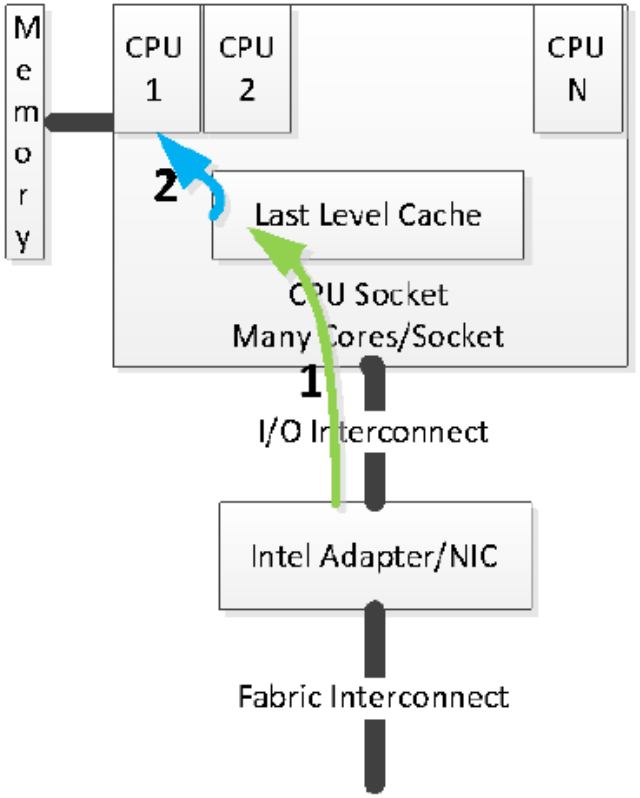
1. A transfer is started by SW writing to DMA engine configuration registers
2. SW Polls DMA channel state to idle and sets trigger
3. DMA engine fetches a descriptor from memory
4. DMA engine reads block of data from source
5. DMA engine writes data to destination



Intel Data Direct I/O (DDIO)

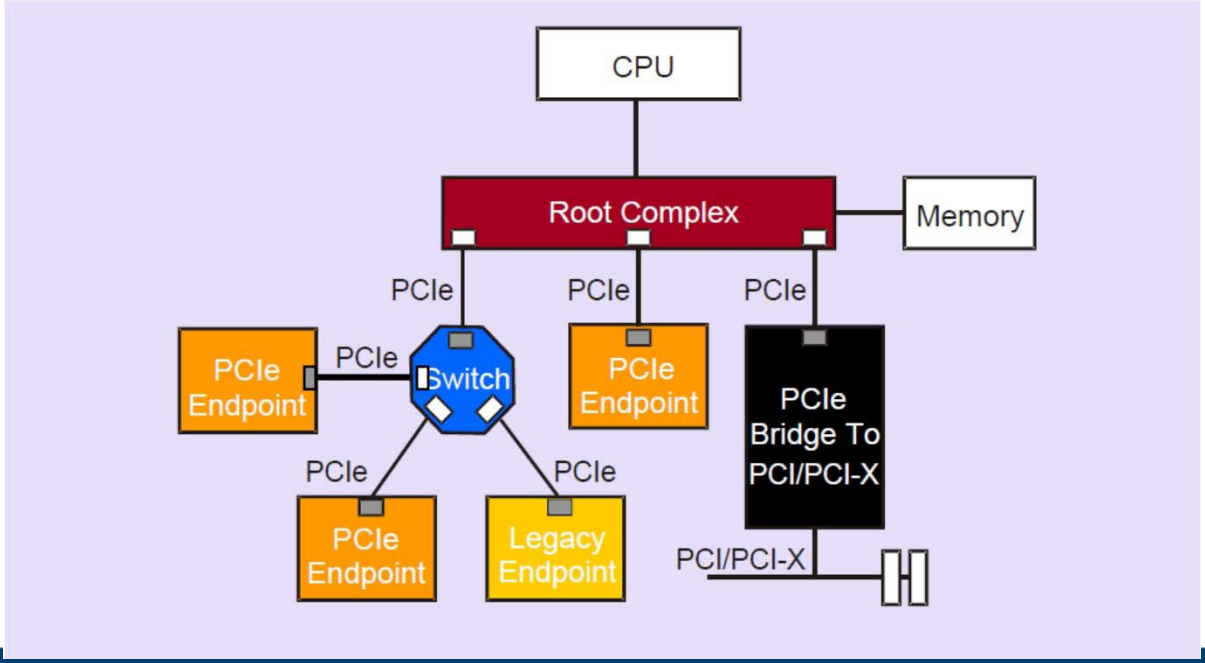
- Data is written and read directly to/from the last level cache





PCIe introduction

- PCIe is a *serial point-to-point interconnect* between two devices
- Implements *packet based protocol (TLPs)* for information transfer
- *Scalable performance* based on # of signal Lanes implemented on the PCIe interconnect
- Supports *credit-based* point-to-point flow control (not end-to-end)



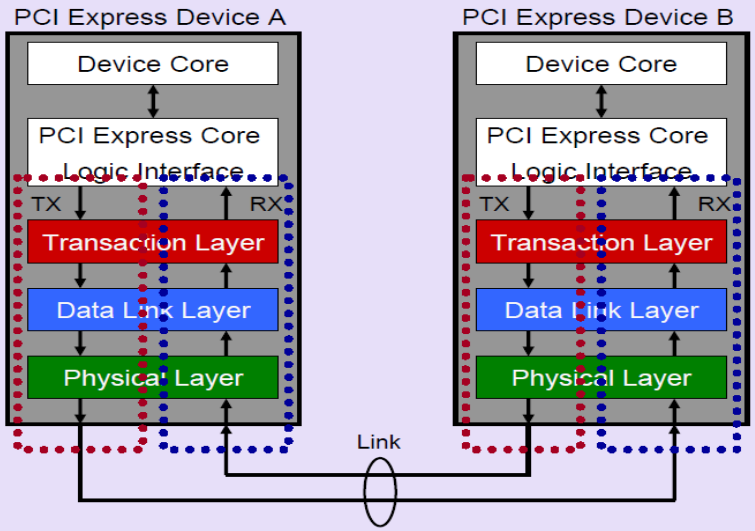
Provides:

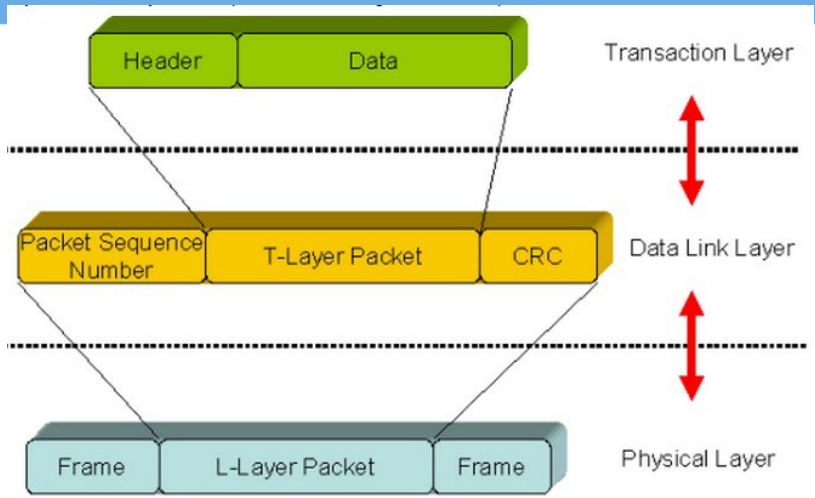
- Processor independence & buffered isolation
- Bus mastering
- Plug and Play operation

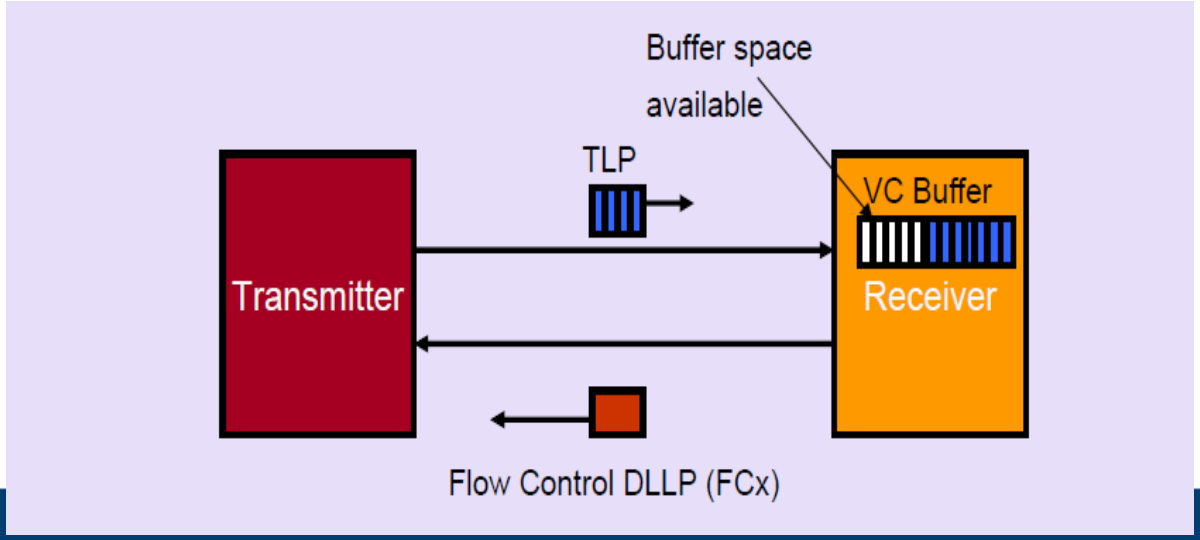
PCIe transaction types

- Memory Read or Memory Write. Used to transfer data from or to a memory mapped location
- I/O Read or I/O Write. Used to transfer data from or to an I/O location
- Configuration Read or Configuration Write. Used to discover device capabilities, program features, and check status in the 4KB PCI Express configuration space.
- Messages. Handled like posted writes. Used for event signaling and general purpose messaging.

PCIe architecture







Interrupt Model

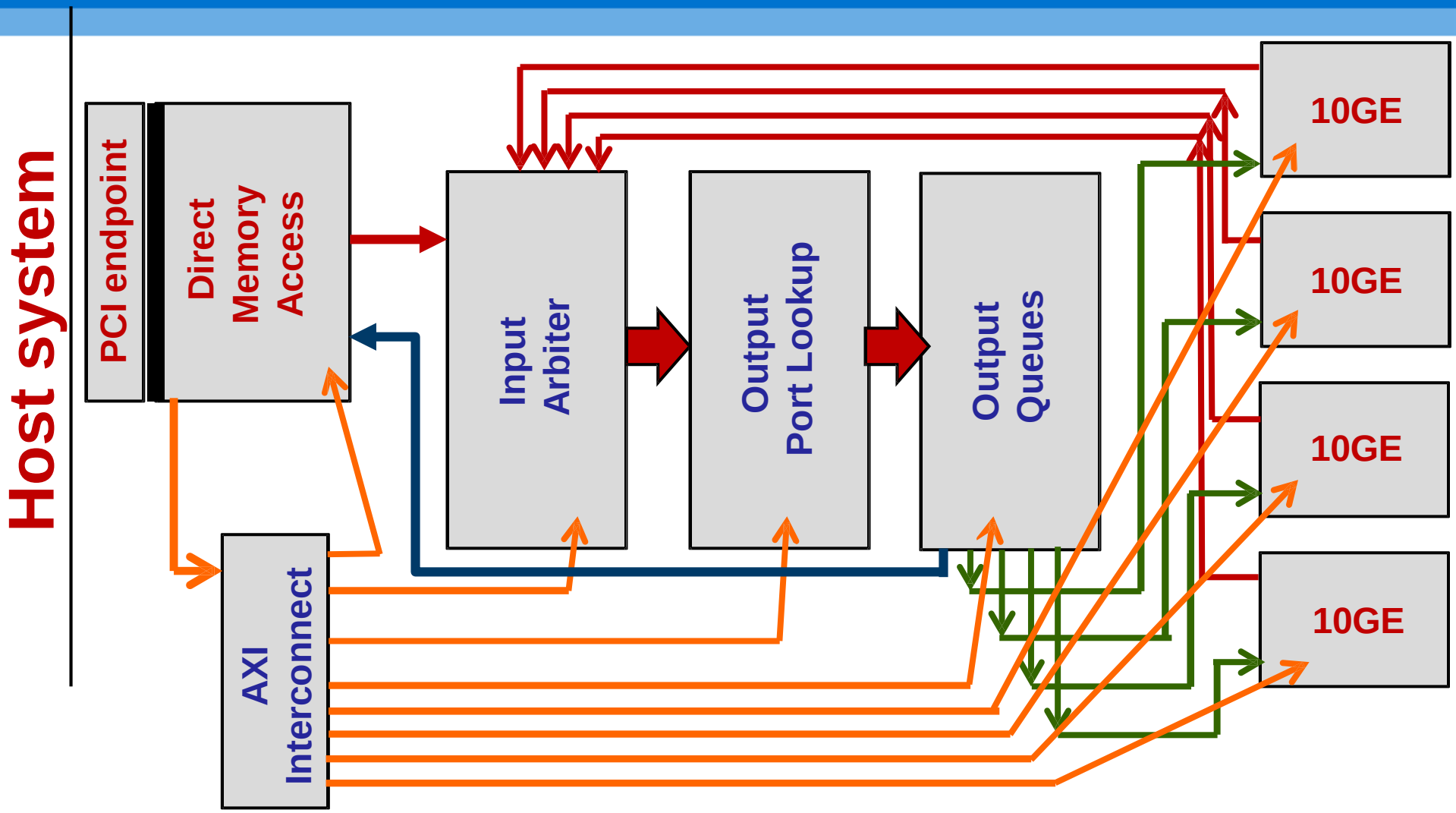
PCI Express supports three interrupt reporting mechanisms:

1. Message Signaled Interrupts (MSI)
   - interrupt the CPU by writing to a specific address in memory with a payload of 1 DW

2. Message Signaled Interrupts - X (MSI-X)
   - MSI-X is an extension to MSI, allows targeting individual interrupts to different processors

3. INTx Emulation
   four physical interrupt signals INTA-INTD are messages upstream
   - ultimately be routed to the system interrupt controller

NetFPGA Reference Projects



Processing Overheads

- Processing in the kernel takes a lot of time…

Component	Time [us]
Driver RX	0.60
Ethernet & IPv4 RX	0.19
TCP RX	0.53
Socket Enqueue	0.06
TCP TX	0.70
IPv4 & Ethernet TX	0.06
Driver TX	0.43

Source: Yasukata *et al*. “StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs”, Usenix ATC 2016

Processing Overheads

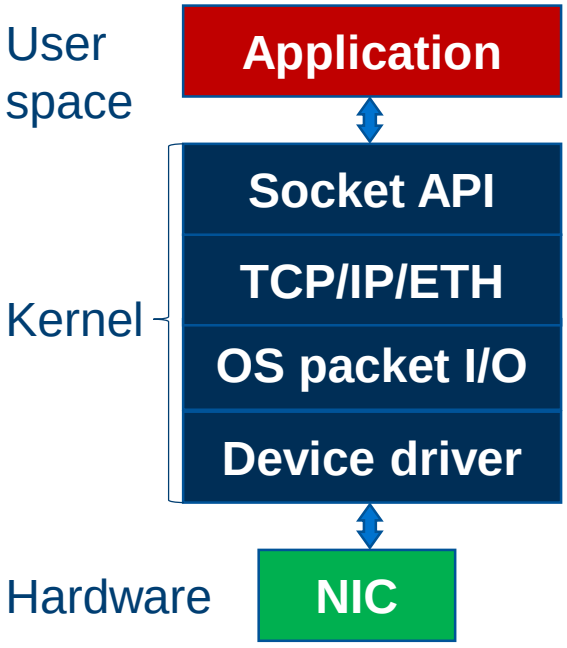
- Processing in the kernel takes a lot of time…
- Order of microseconds (~2-4us on Xeon E5-v4)
- ×10 the time through a switch

- Solution: don’t go through the kernel!

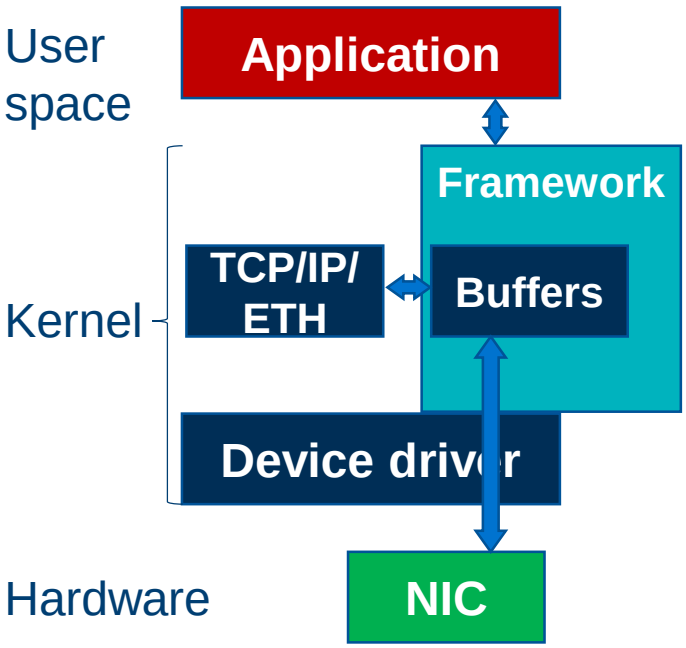
Kernel Bypass

- The Kernel is slow – lets bypass the Kernel!
- There are many ways to achieve kernel bypass
- Some examples:
  - Device drivers:
    - Customized kernel device driver. E.g. Netmap forks standard Intel drivers with extensions to map I/O memory into userspace.
    - Custom hardware and use bespoke device drivers for the specialized hardware.
  - Userspace library: anything from basic I/O to the entire TCP/IP stack

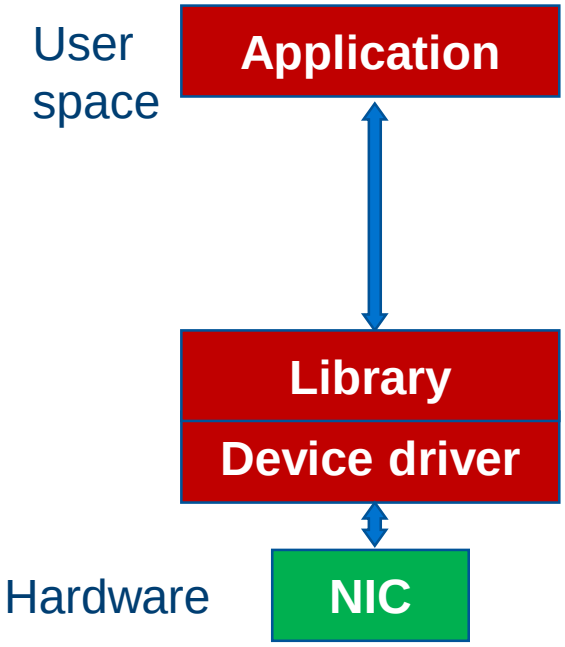
Kernel Bypass - Examples



No Bypass



Partly within Kernel



Completely in User Space

DPDK

- DPDK is a popular set of **libraries and drivers** for fast packet processing.
- Originally designed for Intel processors
  - Now running also on ARM and Power CPUs
- Runs mostly in Linux User space.
- Main libraries: multicore framework, huge page memory, ring buffers, poll-mode drivers (networking, crypto etc)
- It is ***not*** a networking stack

DPDK

- Usage examples:
  - Send and receive packets within minimum number of CPU cycles
    - E.g. less than 80 cycles
  - Fast packet capture algorithms
  - Running third-party stacks
- Some projects demonstrated 100’s of millions packets per seconds
  - But with limited functionality
  - E.g. as a software switch / router