DMA
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
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
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

Example (Embedded Processor)
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.
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

Host system

PCI endpoint
Direct Memory Access

Input Arbiter
Output Port Lookup

Output Queues

AXI Interconnect

10GE
10GE
10GE
10GE
Processing Overheads

- Processing in the kernel takes a lot of time…

<table>
<thead>
<tr>
<th>Component</th>
<th>Time [us]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Driver RX</td>
<td>0.60</td>
</tr>
<tr>
<td>Ethernet &amp; IPv4 RX</td>
<td>0.19</td>
</tr>
<tr>
<td>TCP RX</td>
<td>0.53</td>
</tr>
<tr>
<td>Socket Enqueue</td>
<td>0.06</td>
</tr>
<tr>
<td>TCP TX</td>
<td>0.70</td>
</tr>
<tr>
<td>IPv4 &amp; Ethernet TX</td>
<td>0.06</td>
</tr>
<tr>
<td>Driver TX</td>
<td>0.43</td>
</tr>
</tbody>
</table>

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)
- \( \times 10 \) 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**
  - Application
  - Socket API
  - TCP/IP/ETH
  - OS packet I/O
  - Device driver
  - NIC
  - Hardware

- **Partly within Kernel**
  - Application
  - Framework
  - Buffers
  - TCP/IP/ETH
  - Device driver
  - NIC
  - Hardware

- **Completely in User Space**
  - Application
  - Library
  - Device driver
  - NIC
  - Hardware
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
Inferring Switch Architecture
refer to:
http://www.mellanox.com/tolly/
The following graph shows the latency CDF of a single packet going through the switch:

What does this graph tell us?
The following graph shows the maximum latency of packets sent from Port 0 to Port 1, (a) when there is no other traffic and (b) when there are (in parallel) packets sent from Port 1 to Port 0:

What does this graph tell us?
The following graph shows the latency CDF of packets sent from Port 0 to Port 1, where there are (in parallel) packets sent from Port 1 to Port 0:

What does this graph tell us?
P51 Summary

• Architecture of high-performance network devices
• High throughput devices
• Low latency devices
• Programmable network devices

• This is just a little glimpse into high performance networking…