OS Support For High-Performance Hardware

Lectures on Distributed Operating Systems (SS’23)

till.miemietz@barkhauseninstitut.org
Recap: Traditional I/O – Receiving a Network Packet

- Hereinafter: Memory allocation in red for kernel, gray for userspace processes
Recap: Traditional I/O – Receiving a Network Packet

- CPU executes process S (high priority), that is doing network I/O
Recap: Traditional I/O – Receiving a Network Packet

- Server process S (high priority) is blocked while waiting for network input
Recap: Traditional I/O – Receiving a Network Packet

- Instead of S, CPU executes an other process A (with low priority)
Recap: Traditional I/O – Receiving a Network Packet

- Packet arrives at the NIC
Recap: Traditional I/O – Receiving a Network Packet

- NIC performs demodulation etc., saves packet in RAM of NIC
Recap: Traditional I/O – Receiving a Network Packet

- NIC emits an Interrupt (IRQ) to the CPU
Recap: Traditional I/O – Receiving a Network Packet

- CPU interrupts user program, executes IRQ handler set by OS

![Diagram showing CPU, NIC, and IRQ connection](image-url)
Recap: Traditional I/O – Receiving a Network Packet

- OS sets up Direct Memory Access (DMA) transfer between NIC and host RAM

Diagram:
- CPU
- NIC
- DMA Setup
- Interrupt Handler (in Kernel)
Recap: Traditional I/O – Receiving a Network Packet

- DMA hardware transfers packet to in-kernel buffer
Recap: Traditional I/O – Receiving a Network Packet

- DMA hardware transfers packet to in-kernel buffer
Recap: Traditional I/O – Receiving a Network Packet

- DMA hardware transfers packet to in-kernel buffer
Recap: Traditional I/O – Receiving a Network Packet

- IRQ handler triggers execution of the in-kernel network stack
Recap: Traditional I/O – Receiving a Network Packet

- Eventually leads to unblocking the server process
Recap: Traditional I/O – Receiving a Network Packet

- Data still in kernel buffers: Copy data to a location accessible by the server
Recap: Traditional I/O – Receiving a Network Packet

- Data still in kernel buffers: Copy data to a location accessible by the server
Recap: Traditional I/O – Receiving a Network Packet

- Data still in kernel buffers: Copy data to a location accessible by the server
Recap: Traditional I/O – Receiving a Network Packet

- Server process can continue
Recap: Traditional I/O – Receiving a Network Packet

- How does it look like from the server process’ POV? (Schematic I/O procedure)

```c
int          fd = -1;
ssize_t      bytes_read;
unsigned char buffer[1024];

/* Obtain a handle to a device */
fd = open_func(“pathname”, <options>, <mode>);

/* Read data. I.e., wait for input. This blocks the *
* calling process if no data is available immediately */
memset(&buffer, 0, 1024);
bytes_read = recv_io_func(fd, &buffer, 1024);
```
Traditional I/O – Common Insights

- Communication with peripheral devices is very slow

- This creates a lot of leeway for “CPU-sided” I/O optimizations
  - Caching
  - I/O scheduling
  - Use asynchronous I/O and try to do something else in the meantime

- Avoid CPU idling due to I/O operations (switch to a different process, ...)

- “Performance of the I/O software itself is of little concern”
Modern Hardware – What has changed in the last ~15 years?

- **CPU [1]:**
  - Intel 7150 N (rel. 2007): 1 core @ 3500MHz
  - Intel Xeon Platinum 8358 (rel. 2021): 32 (64) cores @ 2600 MHz

- **Storage [2,3], including a technology shift from HDDs to SSDs:**
  - Seagate Barracuda 7200.11 (rel. 2007): 1.5 TB, up to 120 MB/s
  - Samsung 990 pro (rel. 2022): 2.0 TB, 7400 MB/s (read) / 6900 MB/s (write)

- **Network [4,5]:**
  - Mellanox Connect-X2 (rel. ~2010): up to 40 Gbit/s per port
  - Mellanox Connect-X7 (rel. 2022): up to 400 Gbit/s per port
Modern I/O Devices – Takeaways

- Performance improvement for peripheral devices much higher than for the CPU

- Strong trend towards more parallelism
  - Helps at increasing scalability
  - Sometimes leveraged by hardware layout (flash memory)

- A similar increase in performance can be observed on the system bus (PCIe)
Modern I/O Devices: Any Impact on the OS?

- Nowadays, I/O operations may take only a couple of microseconds!
  - Compared to several milliseconds ~15 years ago

- Systems software is becoming a bottleneck!
The OS Is Becoming a Bottleneck – Latency / Throughput

- Case study for modern SSDs [7]:

![Graph showing comparison between Linux Kernel and SPDK for submit and complete operations.](image-url)
The OS Is Becoming a Bottleneck – Scalability

- Case study for modern SSDs [7]:

![I/O Performance on Single Intel® Xeon® core](image)
Why is That?

• Performance costs on hardware (from within the OS)
  - Writing 4 KiB to a modern SSD: ~15 µs
  - RTT for a 4 KiB Packet in an InfiniBand fabric: < 10 µs

• Compared to OS operations (carried out multiple times on the I/O path)
  - Copying 1 MiB in memory: ~ 1 µs
  - Performing a context switch: ~ 2 – 3 µs
Why is That? – Looking At the Intro Again

Diagram:
- CPU
- NIC
- IRQ
- DMA Transfer
- Process S
- Process A
- Kernel Software
Why is That? – Looking At the Intro Again

- CPU
- NIC
- Process S
- IRQ
- DMA Transfer
- Kernel Software
- Process A
Software-Induced Performance Barriers for Fast I/O

- Interrupt-based notification

- Context switches

- Copying data to / from intermediate buffers

- Inadequate design of drivers and applications
  - Parallelism of hardware not exploited in software (e.g. single queue block layer in Linux [7])
  - Poor locking schemes (coarse-grained locking, ...)
  - Complex “optimizations” on the hot path (→ I/O scheduling on SSDs)
Measures for Reducing Software Overhead in I/O Operations

- Polling-based event notification: avoid IRQs

- Drivers in userspace: avoid context switches, microkernel-like benefits

- IPC using shared memory: avoid context switches

- Implement critical I/O path in hardware (*offloading*): mitigates all previous issues
  - However, this trades speed for versatility!

- Software optimizations (lock-free programming, asynchronous I/O, ...)

Case Study – Remote Direct Memory Access (RDMA)
Remote Direct Memory Access (RDMA)

- Interface standard for high-performance NICs
  - Multiple implementations exist: RDMA over Converged Ethernet (RoCE), InfiniBand (IB), iWARP
  - While using different hardware, all approaches share a common API (verbs)

- Common design decisions [9]:
  - Partial offloading of the network stack to the NIC
  - Separation of data plane and control plane
  - Data plane implemented as a part of the application processes
  - Polling-based event notification
  - Several improvements of the network protocols compared to TCP/IP (out of scope for this lecture)
Control Plane and Data Plane in a Standard Network Stack

- Same path for data plane (e.g. send) and control plane (e.g. ioctl) operations
  - Too expensive for data plane operations that are frequently carried out

User Space
  - Application
  - TCP
  - IP
  - Ethernet
  - Device Driver

Kernel Space

Hardware
  - NIC

Crossing mode boundary
### Control Plane and Data Plane in a Standard Network Stack

- **Same path for data plane (e.g. send) and control plane (e.g. ioctl) operations**
  - Too expensive for data plane operations that are frequently carried out

<table>
<thead>
<tr>
<th>User Space</th>
<th>Kernel Space</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>TCP</td>
<td>NIC</td>
</tr>
<tr>
<td>IP</td>
<td>Ethernet</td>
<td></td>
</tr>
<tr>
<td>Device Driver</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Crossing mode boundary**
- **Trickling through several layers of software**
Control Plane and Data Plane in a Standard Network Stack

- Same path for data plane (e.g. send) and control plane (e.g. ioctl) operations
  - Too expensive for data plane operations that are frequently carried out

User Space
- Application
- TCP
- IP
- Ethernet
- Device Driver

Kernel Space

Hardware
- NIC

Crossing mode boundary
Trickling through several layers of software
Control Plane and Data Plane in a Standard Network Stack

- Same path for data plane (e.g. send) and control plane (e.g. ioctl) operations
  - Too expensive for data plane operations that are frequently carried out

```
User Space
  Application
  TCP
  IP
  Ethernet
  Device Driver

Kernel Space

Hardware
  NIC
```

Crossing mode boundary

Trickling through several layers of software
Control Plane and Data Plane in a Standard Network Stack

- Same path for data plane (e.g. send) and control plane (e.g. ioctl) operations
  - Too expensive for data plane operations that are frequently carried out

![Diagram showing layers of software and crossing mode boundary](image)

- User Space
  - Application
  - TCP
  - IP
  - Ethernet
  - Device Driver

- Kernel Space

- Hardware
  - NIC

Crossing mode boundary

Trickling through several layers of software
Control Plane and Data Plane in a Standard Network Stack

- Same path for data plane (e.g. send) and control plane (e.g. ioctl) operations
  - Too expensive for data plane operations that are frequently carried out

Diagram:

- User Space:
  - Application
  - TCP
  - IP
  - Ethernet
  - Device Driver

- Kernel Space:
  - Crossing mode boundary
  - Trickling through several layers of software
  - Propagation to device

- Hardware:
  - NIC
Control Plane and Data Plane in an RDMA Stack

- Data plane operations directly between NIC and application (*kernel bypass*)
  - All control operations, e.g. creating DMA mappings, go through the kernel (security enforcement)

```
User Space
Application
libibverbs
User Space Driver

Kernel Space
In-Kernel Driver

Hardware
RDMA NIC (HCA)
```

Control Operation handed over to verbs API
Control Plane and Data Plane in an RDMA Stack

- Data plane operations directly between NIC and application (*kernel bypass*)
  - All control operations, e.g. creating DMA mappings, go through the kernel (security enforcement)

Diagram:

- Application
- In-Kernel Driver
- User Space Driver
- libibverbs
- User Space
- Kernel Space
- Hardware
- RDMA NIC (HCA)

Control Operation handed over to verbs API

Handed over to kernel, kernel does permission checks etc.
Control Plane and Data Plane in an RDMA Stack

- Data plane operations directly between NIC and application (*kernel bypass*)
  - All control operations, e.g. creating DMA mappings, go through the kernel (security enforcement)
Control Plane and Data Plane in an RDMA Stack

- Data plane operations directly between NIC and application (*kernel bypass*)
  - All control operations, e.g. creating DMA mappings, go through the kernel (security enforcement)

```
User Space
  Application
  libibverbs
  User Space Driver

Kernel Space
  In-Kernel Driver

Hardware
  RDMA NIC (HCA)
```

- Control Operation handed over to verbs API
- Handed over to kernel, kernel does permission checks etc.
- Propagation to device
Control Plane and Data Plane in an RDMA Stack

- Data plane operations directly between NIC and application (*kernel bypass*)
  - All control operations, e.g. creating DMA mappings, go through the kernel (security enforcement)

```
User Space
  Application
  libibverbs
  User Space Driver

Kernel Space
  In-Kernel Driver

Hardware
  RDMA NIC (HCA)
```
Control Plane and Data Plane in an RDMA Stack

- Data plane operations directly between NIC and application (*kernel bypass*)
  - All control operations, e.g. creating DMA mappings, go through the kernel (security enforcement)

```
User Space
libibverbs
User Space Driver

Kernel Space
In-Kernel Driver

Hardware
RDMA NIC (HCA)
```

- Data Operation handed over to verbs API
- Userspace driver forwards operation to HCA
Control Plane and Data Plane in an RDMA Stack

- Data plane operations directly between NIC and application (*kernel bypass*)
  - All control operations, e.g. creating DMA mappings, go through the kernel (security enforcement)

![Diagram showing the control plane and data plane in an RDMA stack.](image-url)

- **User Space**
  - Application
  - libibverbs
  - User Space Driver

- **Kernel Space**
  - In-Kernel Driver

- **Hardware**
  - RDMA NIC (HCA)

  - Data Operation handed over to verbs API
  - Userspace driver forwards operation to HCA
  - Data reaches NIC without going through the kernel, saving copies and mode transitions
Receiving a Network Packet With (Two-Sided) RDMA

- Hereinafter: Memory allocation in red for kernel, gray for userspace processes
Receiving a Network Packet With (Two-Sided) RDMA

- First, S asks kernel to set up DMA mapping from its address space to NIC
  - This is done only *once* when S starts using the NIC!
Receiving a Network Packet With (Two-Sided) RDMA

- Result: NIC is allowed to directly read from / write to application memory
  - One mapping for signaling (control buffer, *doorbell register*), another as a designated packet buffer
Receiving a Network Packet With (Two-Sided) RDMA

- Process S now signals to NIC that it is ready for receiving data (*receive request*)
  - Interaction between NIC and process S done by writing to memory windows established before
Receiving a Network Packet With (Two-Sided) RDMA

- Process S now signals to NIC that it is ready for receiving data (*receive request*)
  - Interaction between NIC and process S done by writing to memory windows established before
Receiving a Network Packet With (Two-Sided) RDMA

- With the receive request, a buffer for storing the next packet is specified
  - Must be accessible by the NIC!
Receiving a Network Packet With (Two-Sided) RDMA

- NIC is notified of receive request by monitoring the mappings shared with S
Receiving a Network Packet With (Two-Sided) RDMA

- S now starts polling for changes in the signaling memory window
  - This means busy waiting, comparable to a spinlock → CPU is effectively blocked
Receiving a Network Packet With (Two-Sided) RDMA

- Packet arrives at the NIC
Receiving a Network Packet With (Two-Sided) RDMA

- NIC performs demodulation, packet parsing, etc.
  - Protocol handling normally done in the kernel is performed directly by the NIC (in hardware)
Receiving a Network Packet With (Two-Sided) RDMA

- NIC uses P2P-DMA to move packet to designated RAM buffer
  - Note that this does not involve the CPU at all!
Receiving a Network Packet With (Two-Sided) RDMA

- NIC uses P2P-DMA to move packet to designated RAM buffer
  - Note that this does not involve the CPU at all!
Receiving a Network Packet With (Two-Sided) RDMA

• NIC uses P2P-DMA to move packet to designated RAM buffer
  - Note that this does not involve the CPU at all!
Receiving a Network Packet With (Two-Sided) RDMA

- Lastly, the NIC writes a new value to the doorbell register
  - Through constant polling, this change is immediately noticed by S
Receiving a Network Packet With (Two-Sided) RDMA

- S can access packet payload directly from predefined buffer
RDMA – Programming Model

- Shared memory windows are abstracted to buffers and *queue pairs*
  - Different queues for sending, receiving and completion notification (for more details see [8, 10])

![Diagram of RDMA - Programming Model](image)
RDMA – Programming Model

- Shared memory windows are abstracted to buffers and *queue pairs*
  - Different queues for sending, receiving and completion notification (for more details see [8, 10])

A single work queue entry specifies the type of operation, location of data buffers, etc.
RDMA – Programming Model

- Shared memory windows are abstracted to buffers and *queue pairs*
  - Different queues for sending, receiving and completion notification (for more details see [8, 10])

After working it off, the NIC posts the result of each work queue entry to a completion queue (technically not a part of a queue pair).
However, note the increased complexity compared to traditional POSIX APIs

- E.g. queue pairs have a state machine associated with them (for more details see [8, 10])

```c
/* This is only a snippet of pseudocode to showcase some of the complexity entangled with programming for RDMA devices. */
/* Many steps necessary to obtain an RDMA MWE are not depicted here. Also, all steps are over-simplified! */

/* Create a device context, similar to an fd obtained from open() */
dev_context = ibv_open_device();

/* This is eventually a syscall for setting up the memory mappings between this process and the NIC */
register_memory(dev_context, buffer);

/* Creates a new queue pair */
queue_pair = ibv_create_qp(dev_ctx, ...);

/* Move the queue pair into a fully operational state, this operation alone takes ~200 LOC if implemented manually */
transition_queue_pair(&queue_pair);

/* Tell the NIC that we are ready to receive a packet inside the previously registered buffer */
ibv_post_recv(buffer, ...);

/* This is the tight loop that polls the queue pair for incoming events from the NIC */
long no_events = 0;
while (no_event == 0) {
    no_events = ibv_poll_cq(dev_context->cq, ...);
}

/* After receiving a notification, data can be directly read from the buffer */
char *packet_data = buffer;
```
RDMA – Summary

- Data path avoids multiple performance bottlenecks
  - Kernel is not involved at all
  - No copying of data between in-kernel and application buffers
  - Communication between NIC and host done through polling instead of IRQs

- A lot of network-related code (protocol handling) implemented in NIC hardware

- Note that the APIs for communicating with the device are asynchronous
  - Instead of avoiding idle time, this is now a key feature to ensure low latency / high throughput!
High-Performance I/O – Some More Aspects
High-Performance I/O for Storage

• Similar problems to those of fast NICs exist with modern SSDs
  – Introduction of NVMe (parallel, low-overhead storage protocol on top of PCIe)
  – Advanced flash technology
  – Microsecond-scale of storage I/O operations

• Storage-Performance Development Kit (SPDK) [6]
  – Conceptually very similar to RDMA (userspace driver, avoiding interrupts, ...)

• Programming model for different classes of fast I/O devices is similar
  – Queue pairs and doorbell registers as central abstractions
High-Performance I/O – Eliminating System Calls

• System calls as a performance bottleneck [12, 13]
  - Broadly spoken, system calls are some form of interrupt as well
  - Multiple issues: Expensive mode transitions, loss of caches, address space switch possible ...

[Diagram showing userspace process, CPU, cache, and syscall]
High-Performance I/O – Eliminating System Calls

- System calls as a performance bottleneck [12, 13]
  - Broadly spoken, system calls are some form of interrupt as well
  - Multiple issues: Expensive mode transitions, loss of caches, address space switch possible ...

![Diagram](https://via.placeholder.com/150)

- Userspace Process
- CPU jumps to predefined entry point in kernel binary, clears microarchitectural state
High-Performance I/O – Eliminating System Calls

- System calls as a performance bottleneck [12, 13]
  - Broadly spoken, system calls are some form of interrupt as well
  - Multiple issues: Expensive mode transitions, loss of caches, address space switch possible ...

![Diagram showing kernel, CPU, and cache with a note on cache cleared]
High-Performance I/O – Eliminating System Calls

- Instead, use shared memory between user process and kernel (→ io_uring)
  - Both threads run on different CPU cores, polling on the shared memory window
  - Possible advantage: Use of kernel abstractions and drivers at lower cost
High-Performance I/O – Eliminating System Calls

- Instead, use shared memory between user process and kernel (\( \rightarrow \) io_uring)
  - Both threads run on different CPU cores, polling on the shared memory window
  - Possible advantage: Use of kernel abstractions and drivers at lower cost
High-Performance I/O – Eliminating System Calls

- Instead, use shared memory between user process and kernel (→ io_uring)
  - Both threads run on different CPU cores, polling on the shared memory window
  - Possible advantage: Use of kernel abstractions and drivers at lower cost
High-Performance I/O – Eliminating System Calls

- Instead, use shared memory between user process and kernel (→ io_uring)
  - Both threads run on different CPU cores, polling on the shared memory window
  - Possible advantage: Use of kernel abstractions and drivers at lower cost
  - Also, CPUs keep caches and other microarchitectural state
High-Performance I/O – A Grain of Salt

● Frameworks like RDMA / SPDK / ... move the device close to the application
  - Suddenly you may find yourself writing kernel-style code in userspace!
  - Hard to get right in the first place (the device is working)
  - Even harder to get the right performance (“RDMA does not scale”)
  - Use of more high-level libraries like openMPI (?)

● Lack of common OS abstractions
  - Multi-user management, live migration, ... (see also [11])

● High-Performance I/O might be an energy-efficiency nightmare
  - When polling, a CPU core runs at 100% load...
High-Performance I/O – Summary

• Modern I/O devices may challenge traditional OS designs
  - Using standard approaches data rates of modern NICs / SSDs are difficult to provide to applications
  - Systems software as a bottleneck (e.g. not accounting for parallelization of devices)

• Try to remove major OS parts (e.g. the kernel) from the critical data path
  - Device drivers in userspace
  - Function offloading
  - Use polling on doorbell registers instead of interrupts

• Often, a tradeoff between usability and performance has to be accepted
References for Further Reading


    (https://indico.cern.ch/event/218156/attachments/351725/490089/RDMA_Aware_Programming_user_manual.pdf)