Demystifying Network Cards

Paul Emmerich

December 27, 2017

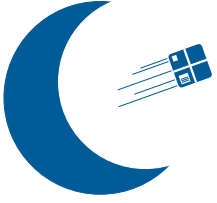
Chair of Network Architectures and Services
Department of Informatics
Technical University of Munich

About me

- PhD student at Technical University of Munich
- Researching performance of software packet processing systems
- Mostly working on my packet generator MoonGen

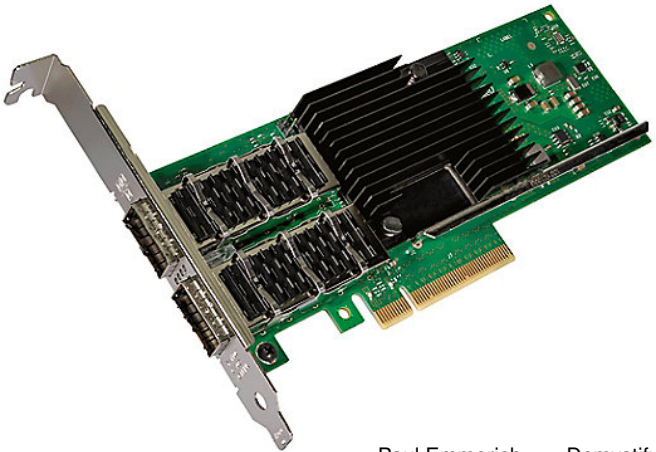
About me

- PhD student at Technical University of Munich
- Researching performance of software packet processing systems
- Mostly working on my packet generator MoonGen
  - Lightning talk about packet generators on Saturday

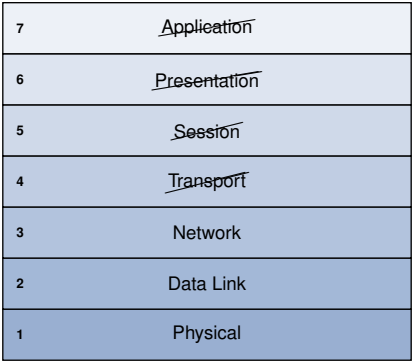


Network Interface Card (NIC)

Network Interface Card (NIC)



What I care about



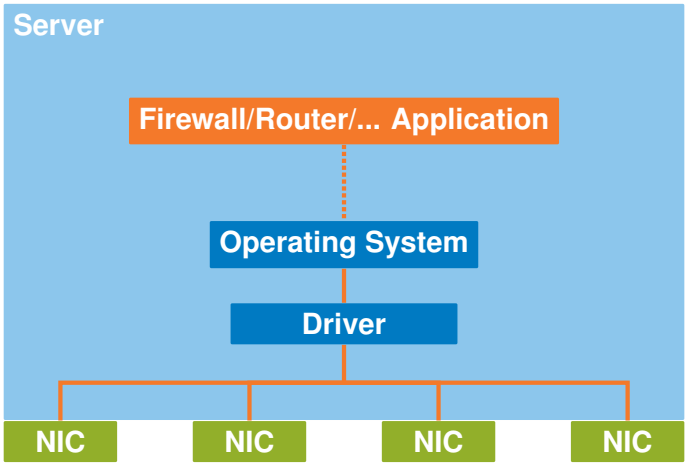
Not all apps run on top of HTTP

Lots of network functionality is moving from specialized hardware to software:

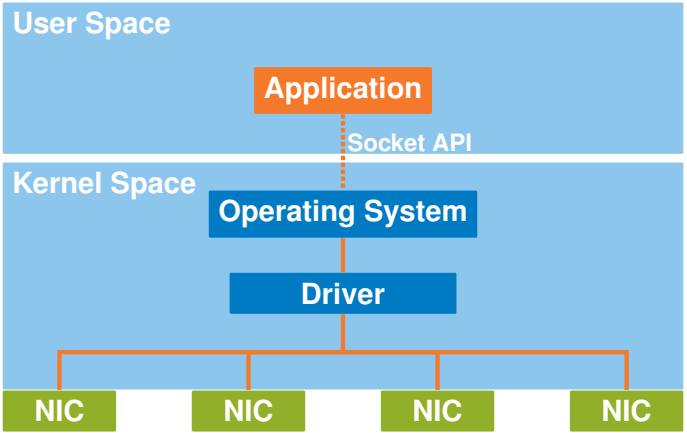
- Routers
- (Virtual) Switches
- Firewalls
- Middleboxes

Buzzwords: Network Function Virtualization, Service Function Chaining

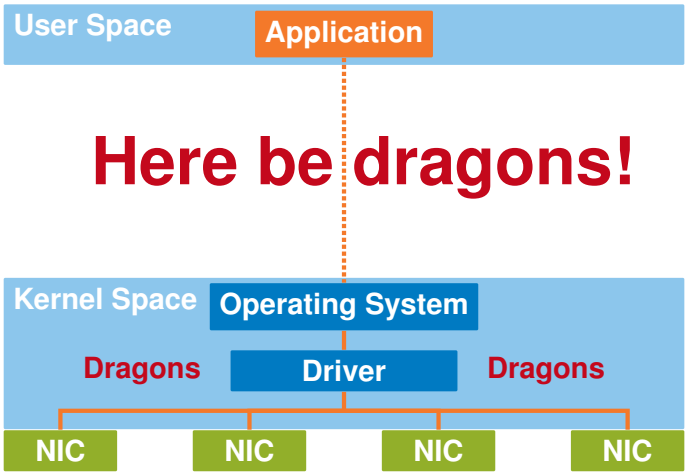
Example application



Normal applications



What it really looks like



Performance

- Packets per second on a 10 Gbit/s link: up to 14.88 Mpps

Performance

- Packets per second on a 10 Gbit/s link: up to 14.88 Mpps
- Packets per second on a 100 Gbit/s link: up to 148.8 Mpps

Performance

- Packets per second on a 10 Gbit/s link: up to 14.88 Mpps
- Packets per second on a 100 Gbit/s link: up to 148.8 Mpps
- Clock cycles per packet on a 3 GHz CPU with 14.88 Mpps: $\approx$ 200 cycles
- Typical performance target: $\approx$ 5 to 10 Mpps per CPU core for simple forwarding

Performance: User space app

- Typical performance target: $\approx$ 5 to 10 Mpps per CPU core for simple forwarding
- 5 to 10 Mpps = 300 to 600 cycles per packet at 3 GHz

Performance: User space app

- Typical performance target: $\approx$ 5 to 10 Mpps per CPU core for simple forwarding
- 5 to 10 Mpps = 300 to 600 cycles per packet at 3 GHz

- Time to cross the user space boundary: very very long

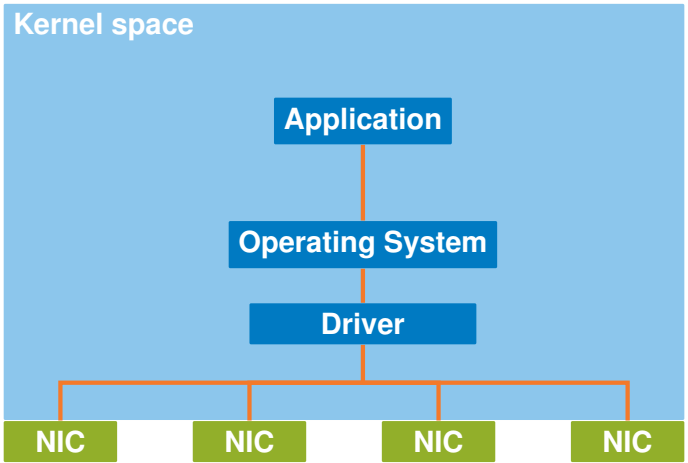
Performance: User space app

- Typical performance target: $\approx$ 5 to 10 Mpps per CPU core for simple forwarding
- 5 to 10 Mpps = 300 to 600 cycles per packet at 3 GHz

- Time to cross the user space boundary: very very long

- Single-core forwarding performance with sockets: $\approx$ 0.3 Mpps
- Single-core forwarding performance with libpcap: $\approx$ 1 Mpps

Move the application into the kernel



Move the application into the kernel

New problems:

- Cumbersome to develop
- Usual kernel restrictions (e.g., C as programming language)
- Application can (and will) crash the kernel

Performance: Kernel app

- Typical performance target: $\approx$ 5 to 10 Mpps per CPU core for simple forwarding
- 5 to 10 Mpps = 300 to 600 cycles per packet at 3 GHz

Performance: Kernel app

- Typical performance target: $\approx$ 5 to 10 Mpps per CPU core for simple forwarding
- 5 to 10 Mpps = 300 to 600 cycles per packet at 3 GHz

- Time to receive a packet in the Linux kernel: $\approx$ 500 cycles

Performance: Kernel app

- Typical performance target: ≈ 5 to 10 Mpps per CPU core for simple forwarding
- 5 to 10 Mpps = 300 to 600 cycles per packet at 3 GHz

- Time to receive a packet in the Linux kernel: ≈ 500 cycles
- Time to send a packet in the Linux kernel: ≈ 440 cycles

Performance: Kernel app

- Typical performance target: ≈ 5 to 10 Mpps per CPU core for simple forwarding
- 5 to 10 Mpps = 300 to 600 cycles per packet at 3 GHz

- Time to receive a packet in the Linux kernel: ≈ 500 cycles
- Time to send a packet in the Linux kernel: ≈ 440 cycles
- Time to allocate, initialize, and free a `sk_buff` in the Linux kernel: ≈ 400 cycles

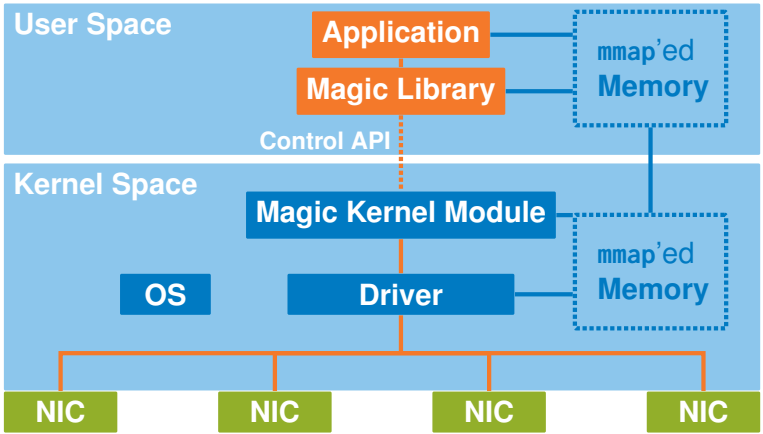
Performance: Kernel app

- Typical performance target: ≈ 5 to 10 Mpps per CPU core for simple forwarding
- 5 to 10 Mpps = 300 to 600 cycles per packet at 3 GHz

- Time to receive a packet in the Linux kernel: ≈ 500 cycles
- Time to send a packet in the Linux kernel: ≈ 440 cycles
- Time to allocate, initialize, and free a `sk_buff` in the Linux kernel: ≈ 400 cycles

- Single-core forwarding performance with Open vSwitch: ≈ 2 Mpps
- Hottest topic in the Linux kernel: XDP, which fixes some of these problems

Do more in user space?



User space packet processing frameworks

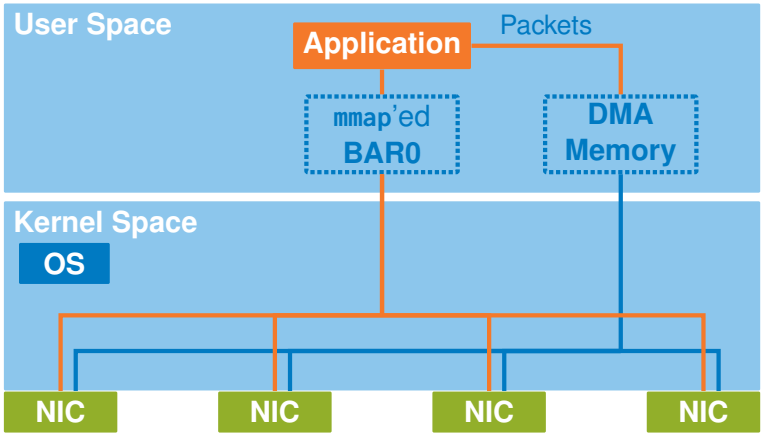
Examples for such frameworks

- netmap
- PF_RING ZC
- pfq

Problems

- Non-standard API, custom kernel module required
- Most frameworks require patched drivers
- Exclusive access to the NIC for one application
- No access to the usual kernel features
  - Limited support for kernel integration in netmap
- Poor support for hardware offloading features of NICs
- Framework needs explicit support for each NIC, limited to a few NICs

Do even more in user space?



User space driver frameworks

Examples for such frameworks

- DPDK
- Snabb

Problems

- Non-standard API
- Exclusive access to the NIC for one application
- Framework needs explicit support for each NIC model
  - DPDK supports virtually all ≥ 10 Gbit/s NICs
- Limited support for interrupts
  - Interrupts not considered useful at ≥ 0.1 Mpps
- No access to the usual kernel features

What has the kernel ever done for us?

- Lots of mature drivers

What has the kernel ever done for us?

- Lots of mature drivers
- Protocol implementations that actually work (TCP, ...)

What has the kernel ever done for us?

- Lots of mature drivers
- Protocol implementations that actually work (TCP, ...)
- Interrupts (NAPI is quite nice)
- Stable user space APIs
- Access for multiple applications at the same time
- Firewalling, routing, eBPF, XDP, ...
- ...and more

Are these frameworks fast?

- Typical performance target: $\approx$ 5 to 10 Mpps per CPU core for simple forwarding
- 5 to 10 Mpps = 300 to 600 cycles per packet at 3 GHz

Are these frameworks fast?

- Typical performance target: ≈ 5 to 10 Mpps per CPU core for simple forwarding
- 5 to 10 Mpps = 300 to 600 cycles per packet at 3 GHz

- Time to receive a packet in DPDK: ≈ 50 cycles
- Time to send a packet in DPDK: ≈ 50 cycles
- Other user space frameworks play in the same league

- Single-core forwarding with Open vSwitch on DPDK: ≈ 13 Mpps (2 Mpps without)
- Performance gains from: batching (typically 16 to 64 packets/batch), reduced memory overhead (no `sk_buff`)

Usage

Lots of packet processing apps have support for DPDK today:

- Open vSwitch
- Vyatta
- Juniper's vMX
- Cisco's VPP
- pfSense (soon)

Main reasons: performance and control over hardware features

Can we build our own user space driver?

Sure, but why?

- For fun ~~and profit~~
- To understand how NIC drivers work
- To understand how user space packet processing frameworks work
  - Many people see these frameworks as magic black boxes
  - DPDK drivers: $\geq$ 20k lines of code per driver
- How hard can it be?
- Turns out it's quite easy, I've written my driver Ixy in less than 1000 lines of C

Hardware: Intel `ixgbe` family (10 Gbit/s)

- `ixgbe` family: 82599ES (aka X520), X540, X550, Xeon D embedded NIC
- Commonly found in servers or as on-board chips
- Very good datasheet publicly available
- Almost no logic hidden behind black-box firmware

How to build a full user space driver in 4 simple steps

1. Unload kernel driver
2. `mmap` the PCIe MMIO address space
3. Figure out physical addresses for DMA
4. Write the driver

Find the device we want to use

```
# lspci
03:00.0 Ethernet controller: Intel Corporation 82599ES 10-Gigabit SFI/SFP+ ...
03:00.1 Ethernet controller: Intel Corporation 82599ES 10-Gigabit SFI/SFP+ ...
```

Find the device we want to use

```
# lspci
03:00.0 Ethernet controller: Intel Corporation 82599ES 10-Gigabit SFI/SFP+ ...
03:00.1 Ethernet controller: Intel Corporation 82599ES 10-Gigabit SFI/SFP+ ...
```

Unload the kernel driver

```
echo 0000:03:00.1 > /sys/bus/pci/devices/0000:03:00.1/driver/unbind
```

mmap the PCIe configuration address space from user space

```
int fd = open("/sys/bus/pci/devices/0000:03:00.0/resource0", O_RDWR);
struct stat stat;
fstat(fd, &stat);
uint8_t* registers = (uint8_t*) mmap(NULL, stat.st_size, PROT_READ | PROT_WRITE,
                                     MAP_SHARED, fd, 0);
```

Device registers

Table 8-2 Register Summary

Offset / Alias Offset	Abbreviation	Name	Block	RW	Reset Source	Page
General Control Registers						
0x00000 / 0x00004	CTRL	Device Control Register	Target	RW		543
0x00008	STATUS	Device Status Register	Target	RO		544
0x00018	CTRL_EXT	Extended Device Control Register	Target	RW		544
0x00020	ESDP	Extended SDP Control	Target	RW		545
0x00028	I2CCTL	I2C Control	Target	RW	PERST	549
0x00200	LEDCTL	LED Control	Target	RW		549
0x05078	EXVET	Extended VLAN Ether Type	Target	RW		551

Access registers: LEDs

```
#define LEDCTL 0x00200
#define LED0_BLINK_OFFS 7

uint32_t leds = *((volatile uint32_t*)(registers + LEDCTL));
*((volatile uint32_t*)(registers + LEDCTL)) = leds | (1 << LED0_BLINK_OFFS);
```

- Memory-mapped IO: all memory accesses go directly to the NIC
- One of the very few valid uses of `volatile` in C

How are packets handled?

- Packets are transferred via DMA (Direct Memory Access)
- DMA transfer is initiated by the NIC

- Packets are transferred via queue interfaces (often called rings)
- NICs have multiple receive and transmit queues
  - Modern NICs have 128 to 768 queues
  - This is how NICs scale to multiple CPU cores
  - Similar queues are also used in GPUs and NVMe disks

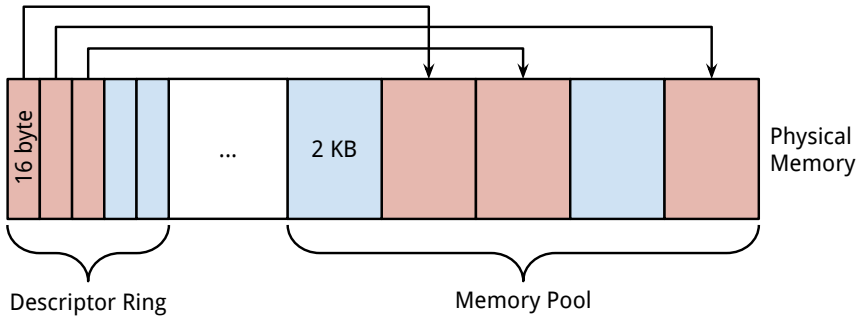
Rings/Queues

- Specific to `ixgbe`, but most NICs are similar
- Rings are circular buffers filled with DMA descriptors
  - DMA descriptors are 16 bytes: 8 byte physical pointer, 8 byte metadata
  - Translate virtual addresses to physical addresses using `/proc/self/pagemap`

Rings/Queues

- Specific to `ixgbe`, but most NICs are similar
- Rings are circular buffers filled with DMA descriptors
  - DMA descriptors are 16 bytes: 8 byte physical pointer, 8 byte metadata
  - Translate virtual addresses to physical addresses using `/proc/self/pagemap`
- Queue of DMA descriptors is accessed via DMA
- Queue index pointers (head & tail) available via registers for synchronization

Ring memory layout



Setting up a receive ring

```
#define RING_ENTRIES 512
size_t ring_size = RING_ENTRIES * sizeof(struct dma_desc);
struct dma_desc* ring_mem = (struct dma_desc*) malloc(ring_size);
set_reg(RDBAL_0, virt2phy(ring_mem));       // dma descriptor ring location
set_reg(RDBAH_0, virt2phy(ring_mem >> 32)); // dma descriptor ring location
set_reg(RDLEN_0, ring_size);                // dma descriptor size
for (int i = 0; i < RING_ENTRIES; i++) {
        ring_mem[i].dma_ptr = malloc(2048); // NIC will store packet here
}
set_reg(RDH_0, 0);             // head pointer
set_reg(RDT_0, RING_ENTRIES); // tail pointer: rx ring starts out full
```

(Simplified, can't use `malloc()` because the memory needs to be physically contiguous, real code uses hugetlbfs)

Receiving packets

- NIC writes a packet via DMA and increments the head pointer
- NIC also sets a status flag in the DMA descriptor once it's done
  - Checking the status flag is way faster than reading the MMIO-mapped RDH register

Receiving packets

- NIC writes a packet via DMA and increments the head pointer
- NIC also sets a status flag in the DMA descriptor once it's done
  - Checking the status flag is way faster than reading the MMIO-mapped RDH register
- Periodically poll the status flag
- Process the packet
- Reset the DMA descriptor, allocate a new packet or recycle
- Adjust tail pointer (RDT) register

What now?

- Transmit rings work the same way
- There's also a lot of boring initialization code in Ixy

What now?

- Transmit rings work the same way
- There's also a lot of boring initialization code in Ixy

Ideas for things to test with Ixy

- Performance: why are these frameworks faster than the Kernel?
- Obscure hardware/offloading features
- Security features: what about the IOMMU?
- Other NICs, other programming languages

Conclusion: Check out ixy

- Check out ixy on GitHub: `https://github.com/emmericp/ixy` (BSD license)
- $\leq$ 1000 lines of C code for the full framework
  - C is the lowest common denominator of programming languages
- Drivers are simple: don't be afraid of them. You can write them in any language!
- No kernel code needed :)

Backup slides

Backup Slides

Linux kernel: past, present, and future

Past

- Linux 2.4 suffered from problems when moving to 1 Gbit/s networks
- One interrupt per packet, servers live-locking under interrupt load

Present

- Linux 2.6 added NAPI
- Dynamic disabling of interrupts, batched receiving
- Lots of optimizations in the socket API

Linux kernel: past, present, and future

Present/Future

- Hottest Linux networking feature: eXpress Data Path (XDP)
- Run packet processing code in a fast path in the kernel using eBPF programs
  - Restrictions apply, typically programmed in a restricted subset of C
- Good integration with the kernel
  - Ideal for firewall applications for user space programs on the same host

Linux kernel: past, present, and future

Present/Future

- Hottest Linux networking feature: eXpress Data Path (XDP)
- Run packet processing code in a fast path in the kernel using eBPF programs
  - Restrictions apply, typically programmed in a restricted subset of C
- Good integration with the kernel
  - Ideal for firewall applications for user space programs on the same host
- Requires driver support (new memory model) and exclusive NIC access
- DPDK supports more NICs than XDP (as of Kernel 4.15)
- Work-in-progress, still lacks many features