

Advanced Network Programming (ANP)

XB_0048

Multicore scalability

Animesh Trivedi
Autumn 2020, Period 1



Layout of upcoming lectures - Part 1

Sep 1st, 2020 (today): ~~Introduction and networking concepts~~

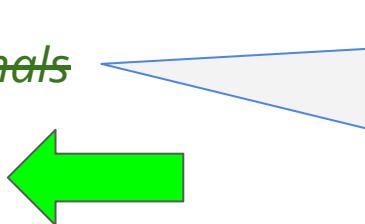
Sep 3rd, 2020 (this Tuesday): ~~Networking concepts (continued)~~

Sep 8th, 2020 : ~~Linux networking internals~~

Sep 10th 2020: *Multicore scalability*

Sep 15th 2020: *Userspace networking stacks*

Sep 17th 2020: *Introduction to RDMA networking*

- 
- NAPI
 - SoftIRQ (TH, BH)
 - Locking order
 - SKB
 - Zero-copy stacks
 - Per-packet/byte overheads

The System Model

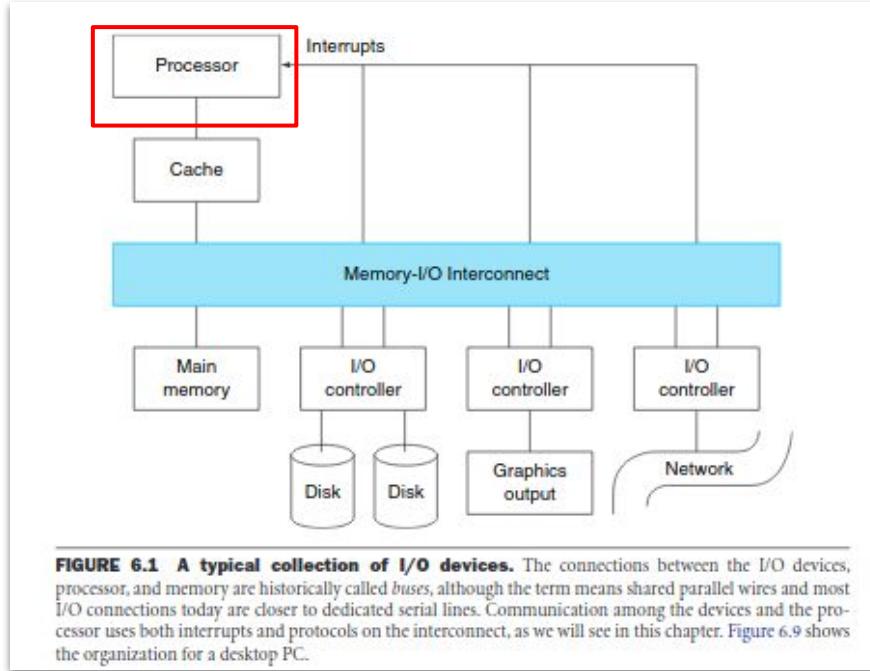
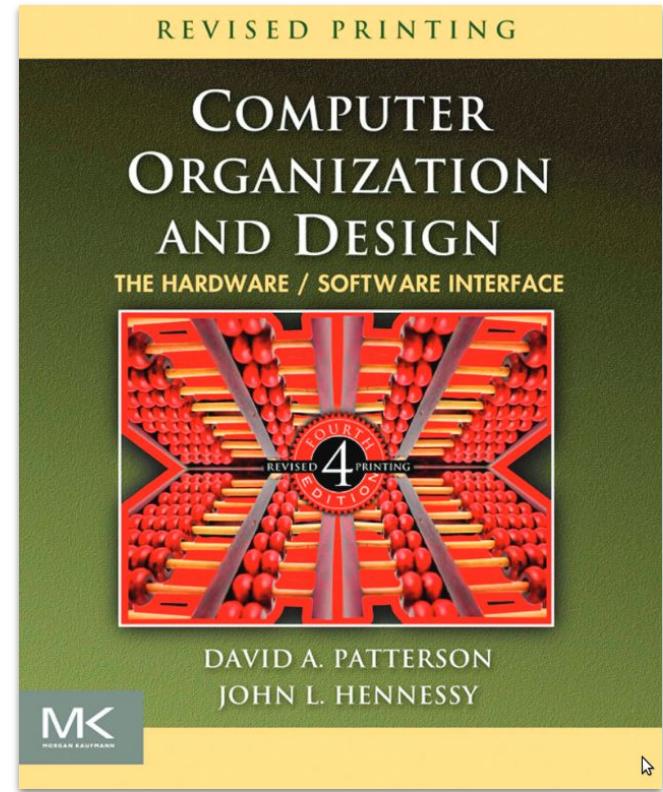


FIGURE 6.1 A typical collection of I/O devices. The connections between the I/O devices, processor, and memory are historically called *buses*, although the term means shared parallel wires and most I/O connections today are closer to dedicated serial lines. Communication among the devices and the processor uses both interrupts and protocols on the interconnect, as we will see in this chapter. Figure 6.9 shows the organization for a desktop PC.

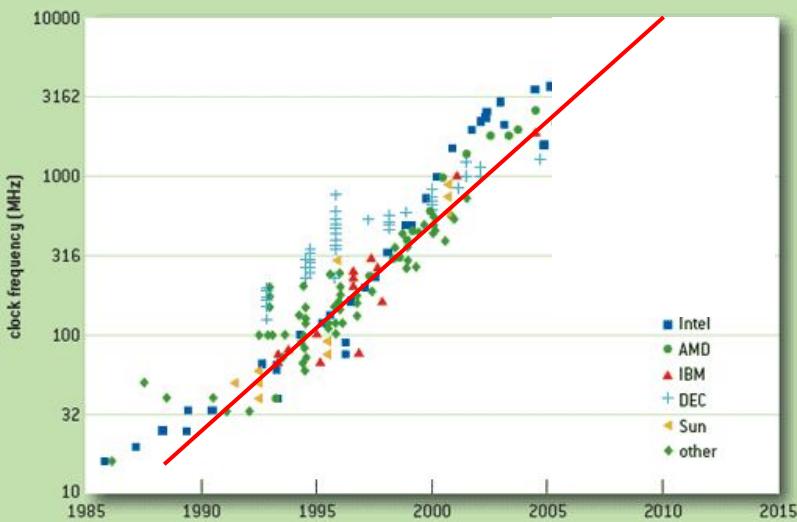


The processor is **the primary workhorse** of the system - executes a series of **instructions per cycle**
Have a number of **cycles/second** → roughly determined by the **CPU frequency**

The year is 2003

FIGURE
7

Processor Frequency Scaling Over Time



Well on the way for a 10 GHz CPU
(we know how that went)



<https://mobile-review.com/print.php?filename=/articles/2003/smartphones-en.shtml>

CPU DB: Recording Microprocessor History, <https://queue.acm.org/detail.cfm?id=2181798> (really cool read!)
<https://www.deviantart.com/darhymes/art/Back-to-the-Future-Icon-276270476>

Dennard's Scaling (1972)

$$\text{Power} = (N \times C \times F \times V^2) + (V \times I \text{ (leakage)})$$

N = transistors (following Moore's Law)

C = Capacitance (decreases for small transistors)

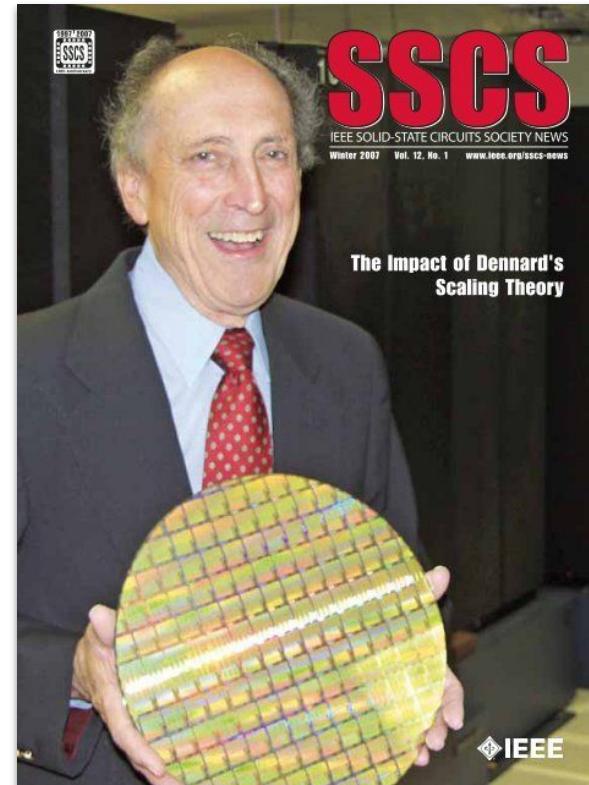
F = Frequency (increased, small transistors → low delay)

V = Operational voltage

I = Leakage current (mostly constant)

For a given power budget, we continue to push for more transistors within a given power budget

- frequency scaling
- more caches
- more instructions



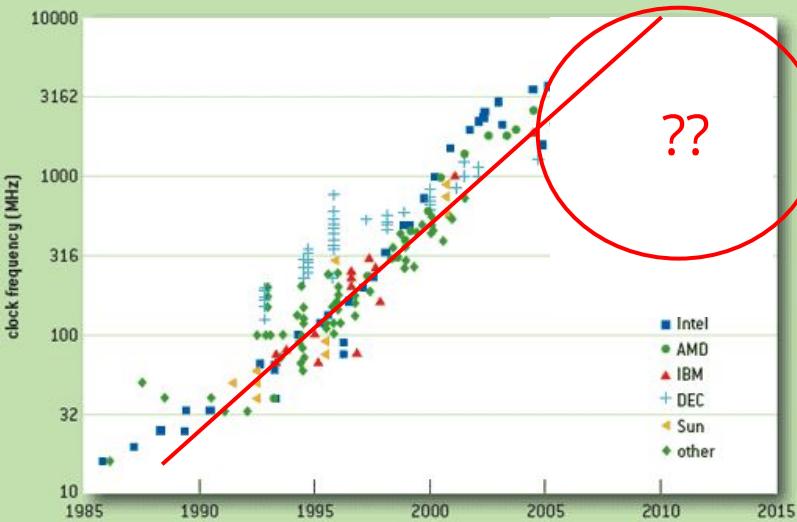
The year is 2003

Well on the way for a 10 GHz CPU
(we know how that went)



FIGURE
7

Processor Frequency Scaling Over Time



But then something happened here, and all our dreams of 10 GHz CPU were shattered ;)



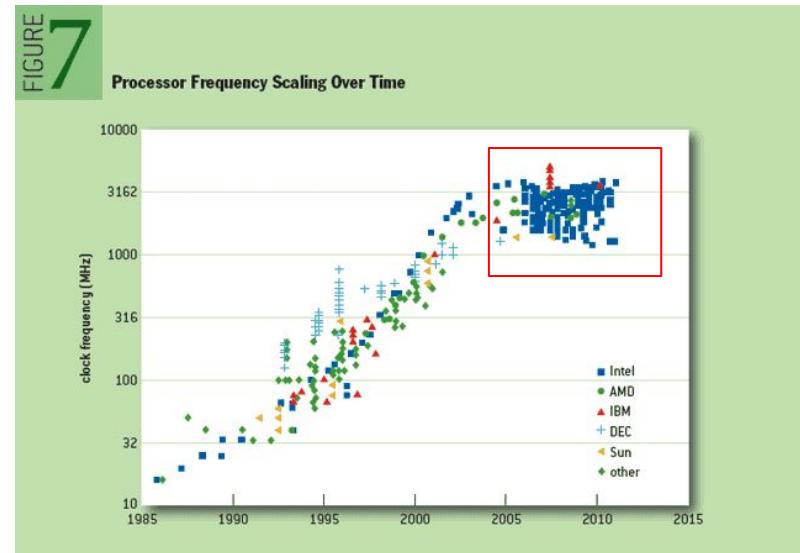
Dennard's Scaling (1972)

$$\text{Power} = (N \times C \times F \times V^2) + (V \times I \text{ (leakage)})$$

Slowing down the of Dennard's scaling

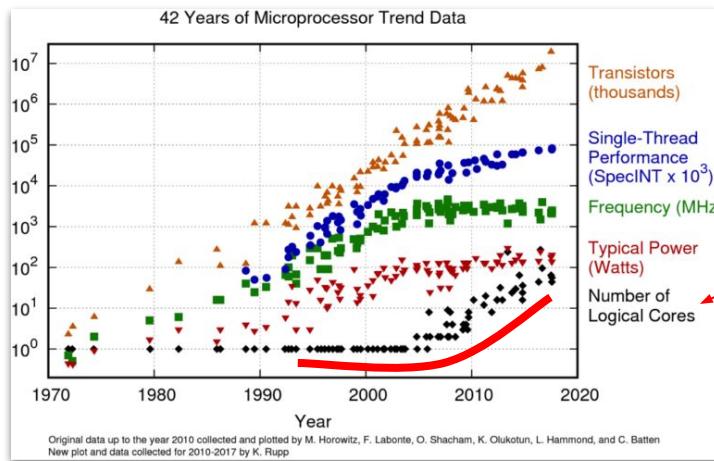
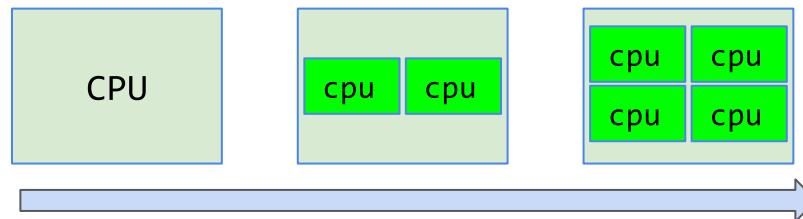
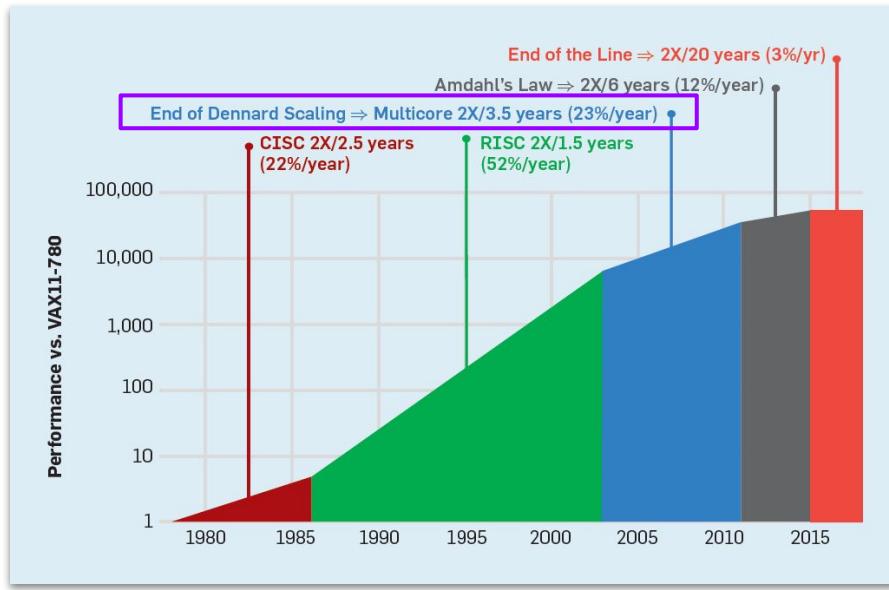
Around 2003-2004 years : **65 nanometer**

- Leakage current become dominant
- Cannot increase voltage (TDP)
- Power dissipation become challenging in a small chip



Hence, frequency scaling stalled, but the number of transistors were still increasing ...

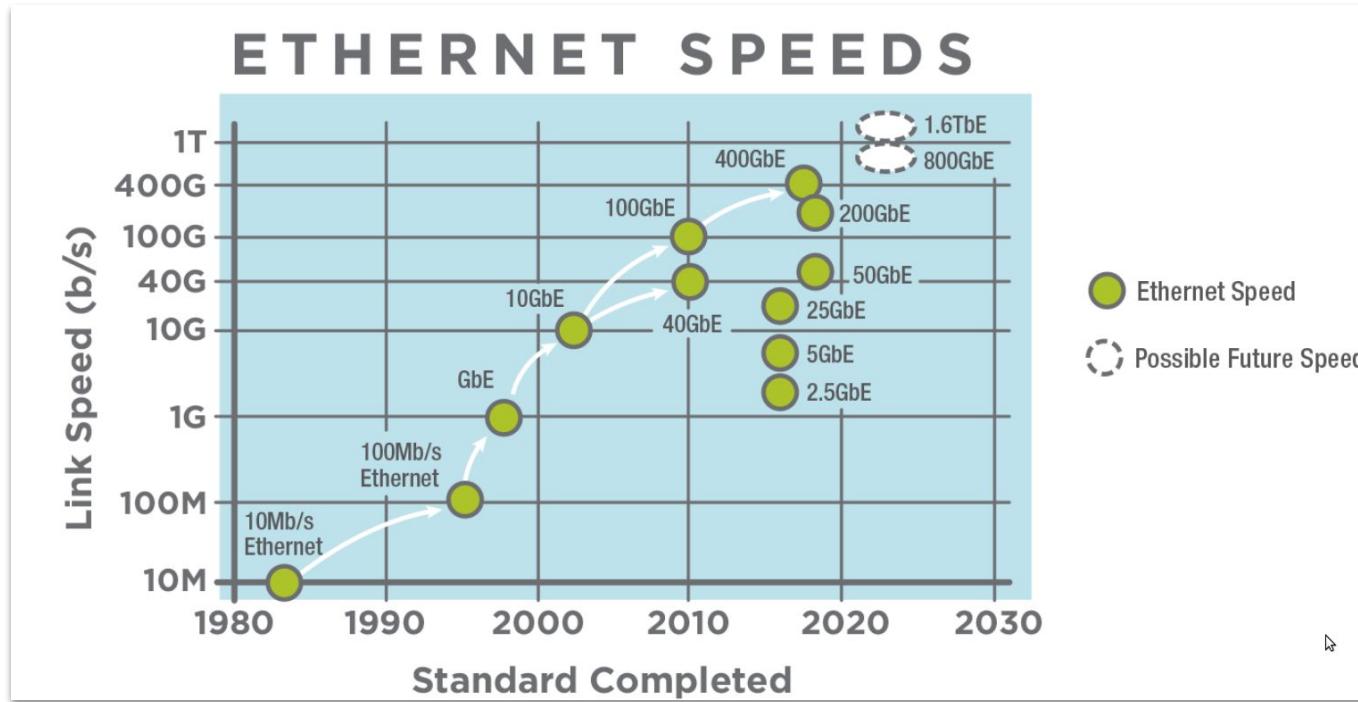
Welcome to the world of Multicore Processing



<https://drivescale.com/2020/01/multicore-crisis/>

<https://cacm.acm.org/magazines/2019/2/234352-a-new-golden-age-for-computer-architecture/fulltext>

Networking speed are growing exponentially ...



<https://ethernetalliance.org/wp-content/uploads/2018/02/Ethernet-Alliance-Roadmap-Panel-OFC-2018.pdf>

Three Phases of Evolution

1. When Moore's Law was valid (**pre 2005**)
 - a. CPU was faster than the network
 - b. Can (mostly) cope up with the demands of commodity network processing (not HPC!)
 - c. CPU performance doubled every 18 months ← **important**
2. When frequency scaling slowed down, manycore era (**2005 - 2015**)
 - a. Ethernet continued to improve ($1 \rightarrow 10 \rightarrow 40$ Gbps)
 - b. Innovation in the networking stack: *Jumbo frames, interrupt management, stateless offloading* (→ all beneficial for a single core, single connection processing!)
 - c. Many core scalability efforts
3. Now CPU performance is not increasing dramatically (**2015 - now**)
 - a. Performance delivery by **specialization** : hardware and software

Manycore Scalability



What are the high-level problems here with multicores

- **Synchronization:** all of them can read and write the same data structure concurrently
 - Yes, you can take locks, mutexes, but then you stall the other core
- **Cache pollution:** A poorly designed data structure can lead to what is known as
 - (a) cache line ping-pong; (b) shared cache pollution
 - Need a careful data layout and alignment

Manycore Scalability : Counter Example

CPU0

CPU1

CPU2

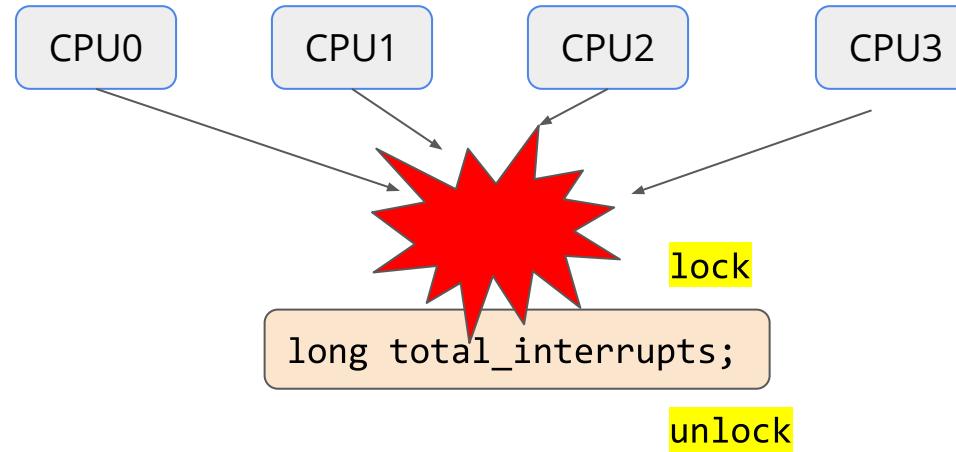
CPU3

lock

```
long total_interrupts;
```

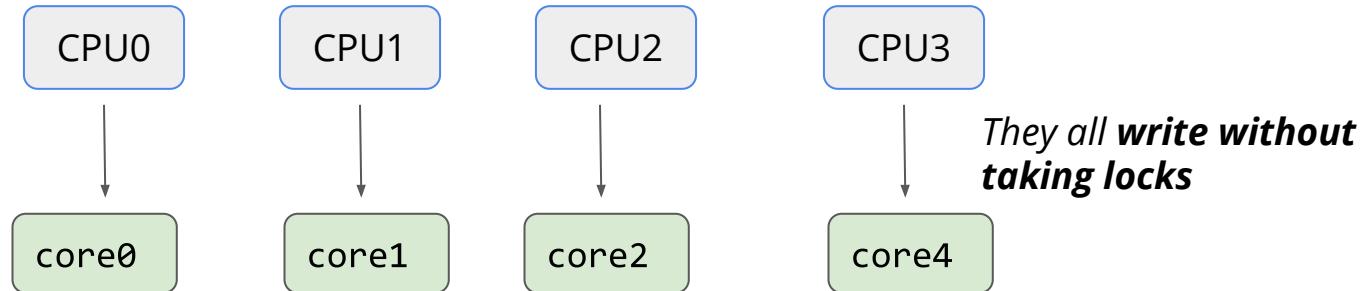
unlock

Manycore Scalability : Counter Example



Manycore Scalability : Counter Example

```
struct intx {  
    long core0;  
    long core1;  
    long core2;  
    long core3;  
};
```

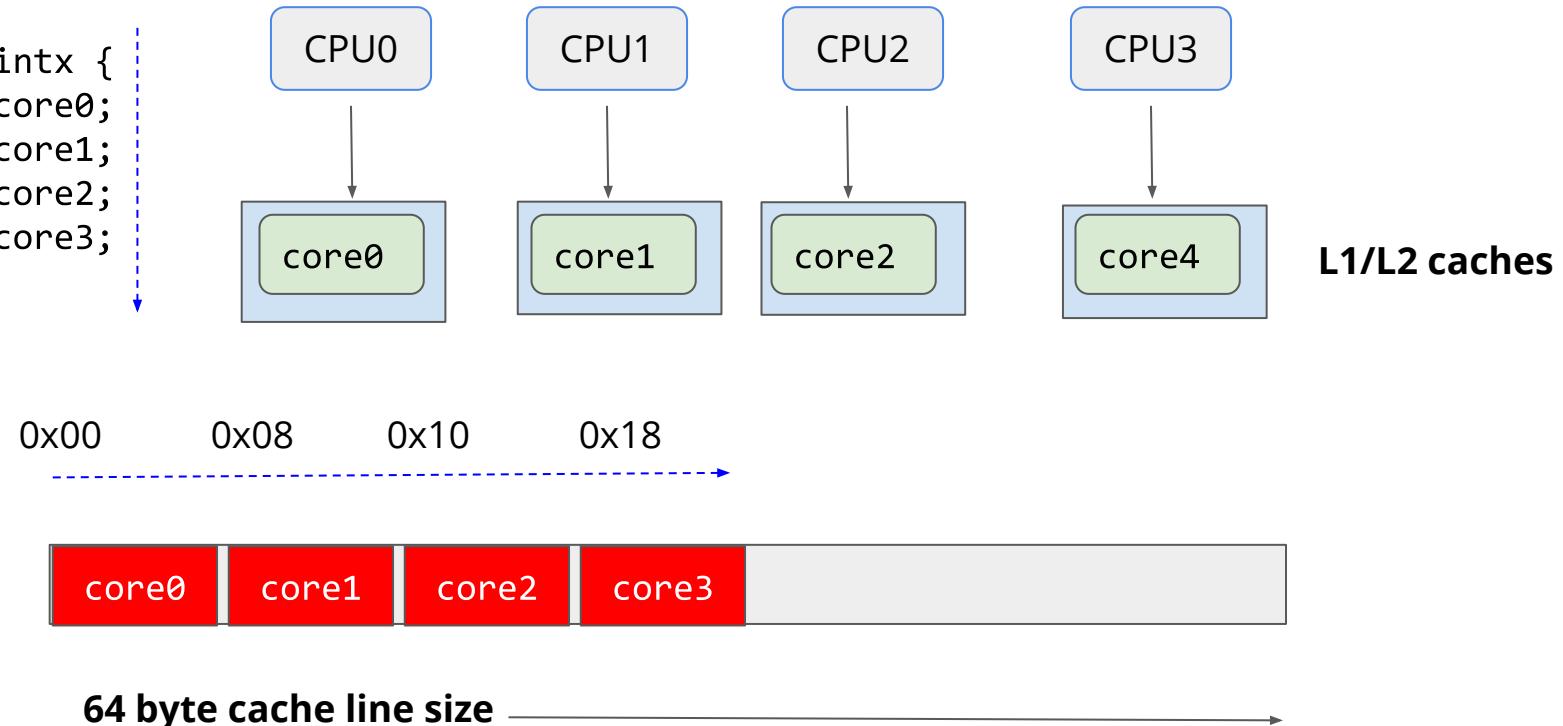


```
sum = rlock(core0 + core1 + core2 + core3);
```

What else can possibly go wrong here? ;)

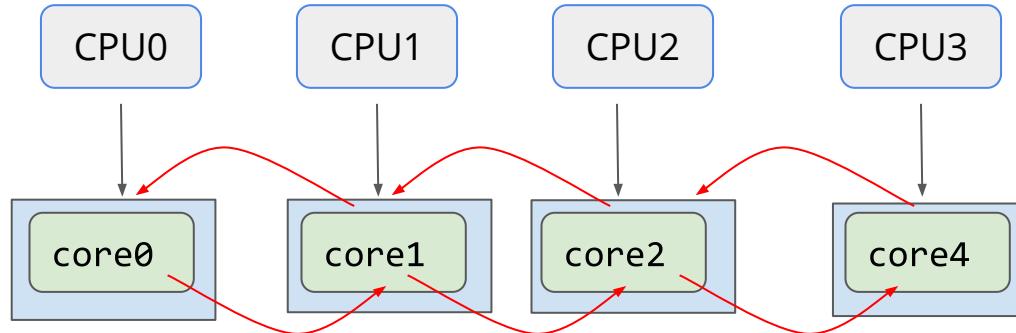
Manycore Scalability : Counter Example

```
struct intx {  
    long core0;  
    long core1;  
    long core2;  
    long core3;  
};
```



Manycore Scalability : Counter Example

```
struct intx {  
    long core0;  
    long core1;  
    long core2;  
    long core3;  
};
```



Whichever core needs to access the counter (cache line ping-pong)

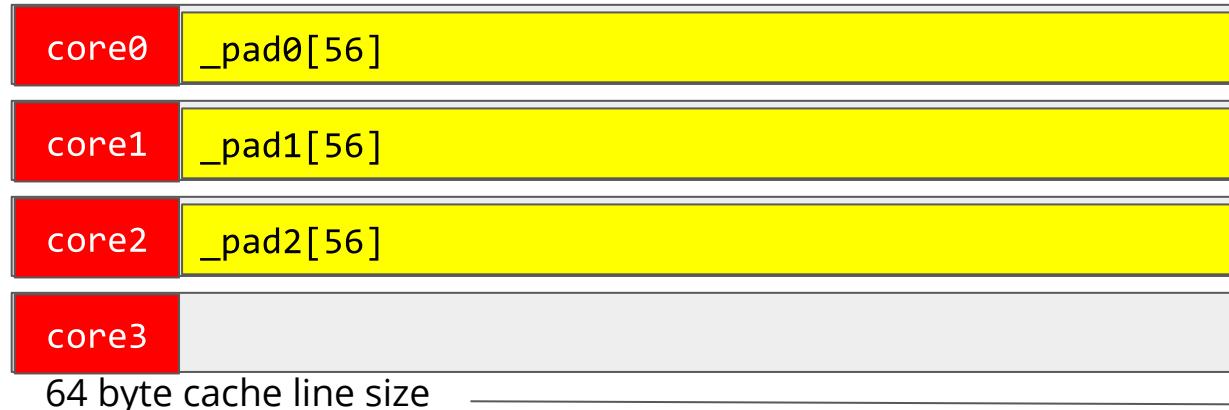
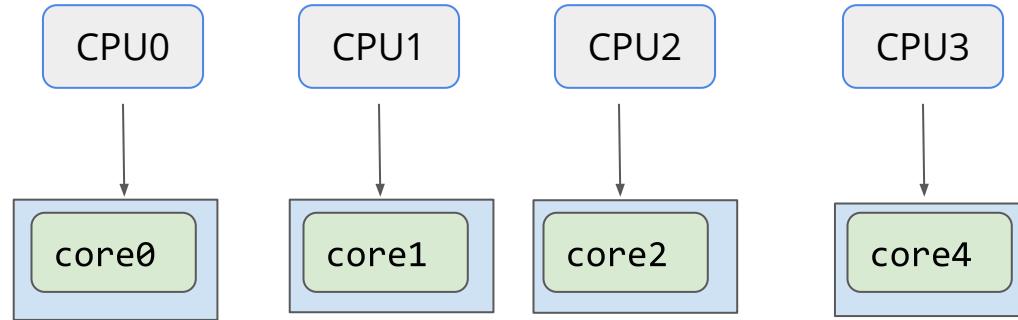
- L1 cache miss
- Fetch the **ENTIRE** cache line



64 byte cache line size

Manycore Scalability : Counter Example

```
struct intx {  
    long core0;  
    uint8_t _pad0[56];  
    long core1;  
    uint8_t _pad1[56];  
    long core2;  
    uint8_t _pad2[56];  
    long core3;  
};
```



Manycore Scalability

CPU0

CPU1

CPU2

CPU3

Goal:

- + *Minimize synchronization*
- + *Minimize cache pollution*
- + *Minimize shared data structures*



How should NIC and CPUs coordinate network packet processing to deliver

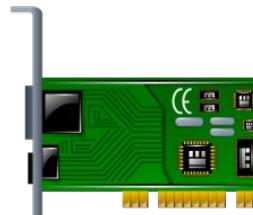
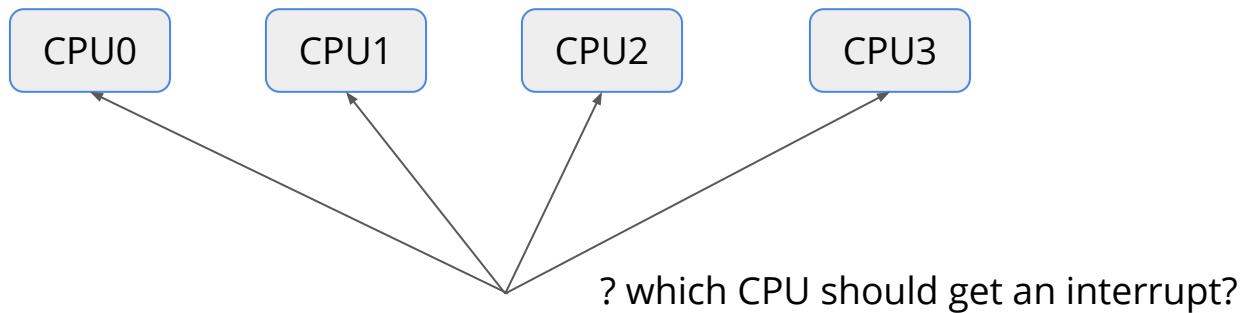
- Highest bandwidth
- Lowest latency
- Millions of small packet processing per second

while not stepping on each others' toe.

ideas?

Let's start from the beginning

What is the first thing NIC does when receiving or sending packets? **Interrupts**



Incoming network packets

Typically, the cpu0 is what gets all the interrupts

Why?

- Because the basic assumption that it is the CPU core that is always on
- Easy to configure
- In booting cpu0 comes up first and bring other cores up
- Generally, core0 is a bit special

CPU0 handling majority of interrupts

	CPU0	CPU1	CPU2	CPU3		
0:	3710374484	0	0	0	IO-APIC-edge	timer
1:	20	0	0	0	IO-APIC-edge	i8042
6:	5	0	0	0	IO-APIC-edge	floppy
7:	0	0	0	0	IO-APIC-edge	parport0
8:	0	0	0	0	IO-APIC-edge	rtc
9:	0	0	0	0	IO-APIC-level	acpi
12:	240	0	0	0	IO-APIC-edge	i8042
14:	11200026	0	0	0	IO-APIC-edge	ide0
51:	61281329	0	0	0	IO-APIC-level	ioc0
59:	1	0	0	0	IO-APIC-level	vmci
67:	19386473	0	0	0	IO-APIC-level	eth0
75:	94595340	0	0	0	IO-APIC-level	eth1
NMI:	0	0	0	0		

Interrupt Balancing

```
atr@atr:~$ service irqbalance status
● irqbalance.service - irqbalance daemon
  Loaded: loaded (/lib/systemd/system/irqbalance.service; enabled; vendor preset: enabled)
  Active: active (running) since Tue 2020-08-25 11:37:27 UTC; 13min ago
    Main PID: 831 (irqbalance)
      Tasks: 2 (limit: 4915)
     CGroup: /system.slice/irqbalance.service
             └─831 /usr/sbin/irqbalance --foreground

Aug 25 11:37:27 atr systemd[1]: Started irqbalance daemon.
```

Linux has a framework (service) called : irqbalance

Tries to distributed interrupt load across CPU cores, you can configure it to

1. Balance interrupt once or periodically (static vs dynamic)
2. Tell which interrupts should go to which CPU (affinity)
3. Tell which CPUs should not handle which interrupts (!affinity)
4. Which interrupts should not be balanced (manual pinning)

How do you assign an interrupts to CPU cores?

How do I find out which NIC has which interrupt numbers?

Interrupt Investigation

```
atr@atr:~$ ifconfig
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.1.161 netmask 255.255.255.0 broadcast 192.168.1.255
        inet6 fe80::a00:27ff:fe25:9e74 prefixlen 64 scopeid 0x20<link>
            ether 08:00:27:25:9e:74 txqueuelen 1000 (Ethernet)
            RX packets 395305 bytes 59640226 (596.4 MB)
            RX errors 0 dropped 0 overruns 0 frame 0
            TX packets 22559 bytes 1736302 (1.7 MB)
            TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
        inet6 ::1 prefixlen 128 scopeid 0x10<host>
            loop txqueuelen 1000 (Local loopback)
            RX packets 26 bytes 2202 (2.2 KB)
            RX errors 0 dropped 0 overruns 0 frame 0
            TX packets 26 bytes 2202 (2.2 KB)
            TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

```
atr@atr:~$ cat /sys/class/net/enp0s3/device/irq
19
atr@atr:~$
```

	CPU0	CPU1			
0:	30	0	IO-APIC	2-edge	timer
1:	59	0	IO-APIC	1-edge	i8042
8:	0	0	IO-APIC	8-edge	rtc0
9:	0	0	IO-APIC	9-fasteoi	acpi
12:	0	156	IO-APIC	12-edge	i8042
14:	0	0	IO-APIC	14-edge	ata_piix
15:	0	902	IO-APIC	15-edge	ata_piix
18:	0	1	IO-APIC	18-fasteoi	vboxvideo
19:	0	24282	IO-APIC	19-fasteoi	enp0s3
20:	315	0	IO-APIC	20-fasteoi	vboxguest
21:	12930	945	IO-APIC	21-fasteoi	ahci[0000:00:0d.0]
22:	24	0	IO-APIC	22-fasteoi	ohci_hcd:usb1

A single device can have multiple interrupts assigned to it for various purposes

Figure out details about interrupts

```
atr@atr:~$ cat /proc/irq/
0/          12/          18/          21/          5/          9/
1/          13/          19/          22/          6/          10/
10/         14/          2/           3/          7/          11/
11/         15/          20/          4/          8/          default_smp_affinity
```

```
atr@atr:~$ cat /proc/irq/19/smp_affinity_list
1
atr@atr:~$
```

Keyboard interrupts at cpu0 and cpu1

NIC interrupts at cpu1

```
atr@atr:~$ cat /proc/irq/1/smp_affinity_list
0-1
atr@atr:~$
```

For every interrupt there is a file: /proc/irq/**irq_number**/smp_affinity that shows the CPU bitmask mask where interrupts can go

- In a 8 core system if you have : 0xFF (all CPUs can get interrupt)
 - 0xF0 (only cpus 4-7 are allowed, not 0-3)
 - 0x11 (only cpu 0 and 4 are allowed)

Lets try to change IRQ affinity

```
root@atr:/home/atr# cat /proc/interrupts
```

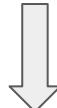
	CPU0	CPU1			
0:	30	0	IO-APIC	2-edge	timer
1:	59	0	IO-APIC	1-edge	i8042
8:	0	0	IO-APIC	8-edge	rtc0
9:	0	0	IO-APIC	9-fasteoi	acpi
12:	0	156	IO-APIC	12-edge	i8042
14:	0	0	IO-APIC	14-edge	ata_piix
15:	0	1506	IO-APIC	15-edge	ata_piix
18:	0	1	IO-APIC	18-fasteoi	vboxvideo
19:	0	25726	IO-APIC	19-fasteoi	enp0s3

```
root@atr:/home/atr# cat /proc/irq/19/smp_affinity_list
```

1

```
root@atr:/home/atr# cat /proc/irq/19/smp_affinity
```

2



```
root@atr:/home/atr# echo 1 > /proc/irq/19/smp_affinity
```

```
root@atr:/home/atr# cat /proc/irq/19/smp_affinity_list
```

0

```
root@atr:/home/atr# cat /proc/interrupts
```

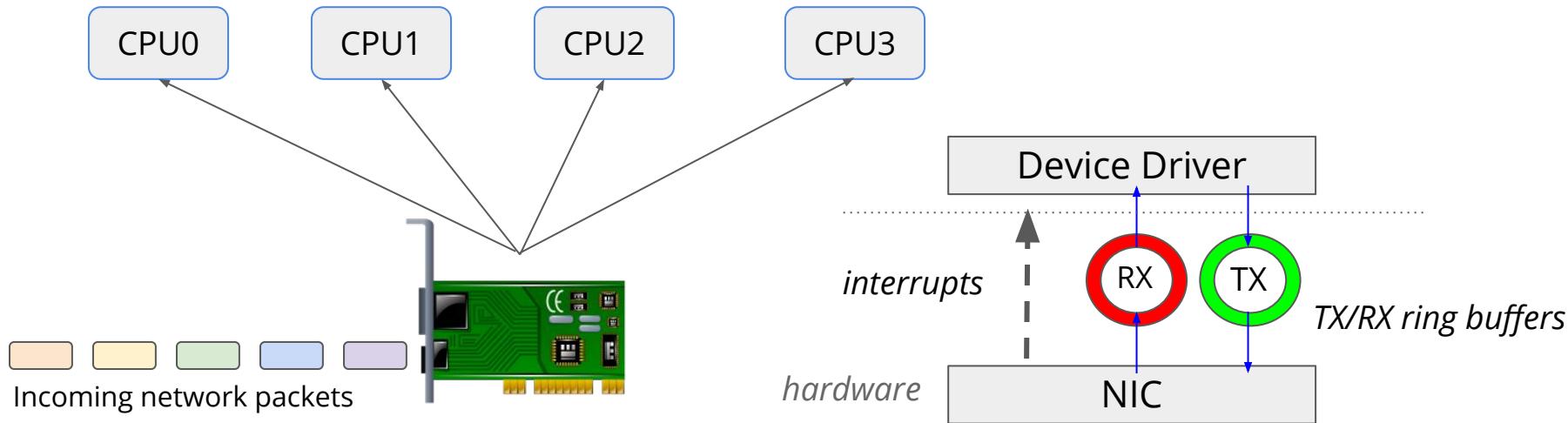
	CPU0	CPU1			
0:	30	0	IO-APIC	2-edge	timer
1:	59	0	IO-APIC	1-edge	i8042
8:	0	0	IO-APIC	8-edge	rtc0
9:	0	0	IO-APIC	9-fasteoi	acpi
12:	0	156	IO-APIC	12-edge	i8042
14:	0	0	IO-APIC	14-edge	ata_piix
15:	0	1576	IO-APIC	15-edge	ata_piix
18:	0	1	IO-APIC	18-fasteoi	vboxvideo
19:	67	26011	IO-APIC	19-fasteoi	enp0s3
20:	585	0	IO-APIC	20-fasteoi	vboxguest
21:	13291	945	IO-APIC	21-fasteoi	ahci[0000:00:0d.0]
22:	24	0	IO-APIC	22-fasteoi	ohci_hcd:usb1

Step 2: What happens then?

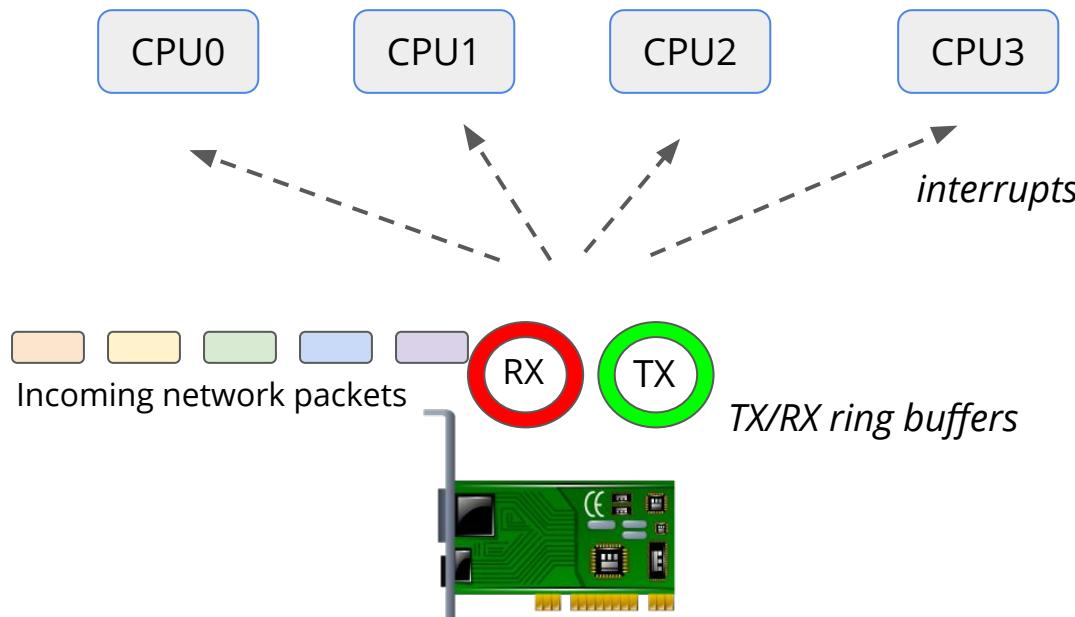
Recall we talk about the rings (or queue) where the outgoing and incoming packets are queued while they wait for processing from the NIC

- Let's say we have mapped that all interrupts should go to all CPUs
- Now we have incoming packets

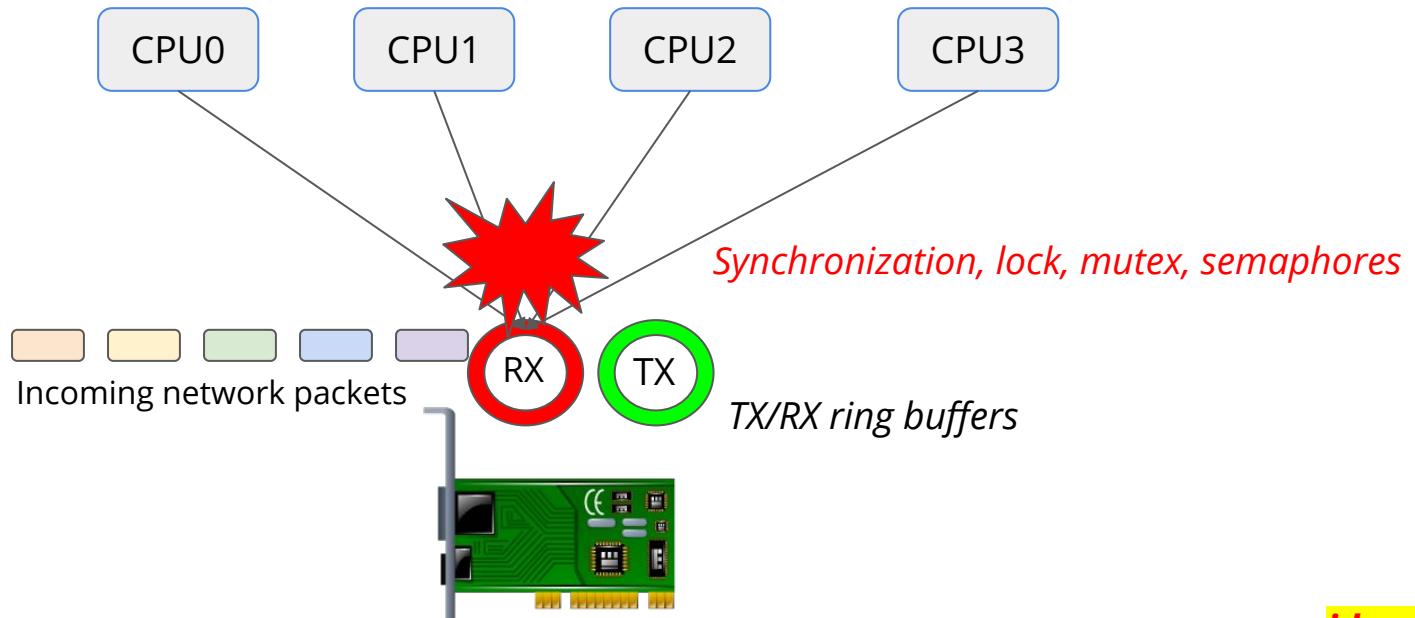
After getting interrupts all of them try to run the interrupt handler, and then?



Step 2: What happens then?

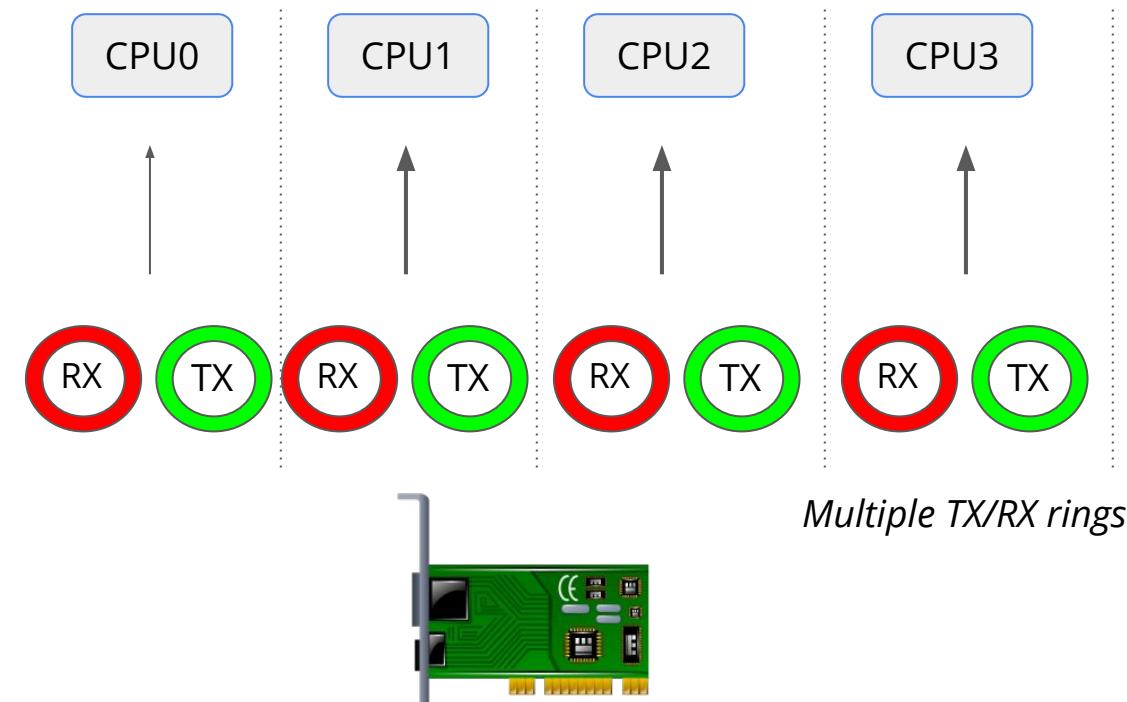


Step 2: What happens then?

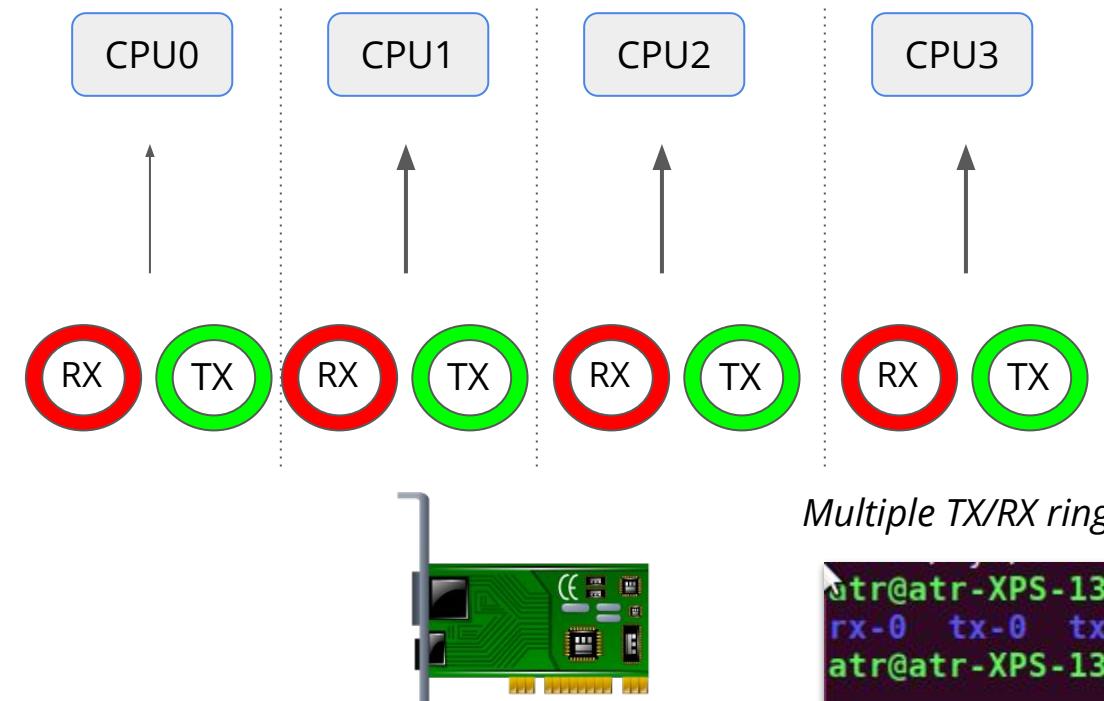


The ring data structure needs to be protected by locks
All CPUs trying to access to the same data structure
Lots of lock contention → Loss in performance !

Solution Multi-Queue NICs (and other devices)



Solution Multi-Queue NICs (and other devices)



```
atr@qemuss20:~$ ethtool -l ens3
Channel parameters for ens3:
Pre-set maximums:
RX:          0
TX:          0
Other:        0
Combined:    1
Current hardware settings:
RX:          0
TX:          0
Other:        0
Combined:    1
atr@qemuss20:~$
```

```
atr@atr-XPS-13:~$ ls /sys/class/net/wlp2s0/queues/
rx-0 tx-0 tx-1 tx-2 tx-3
atr@atr-XPS-13:~$
```

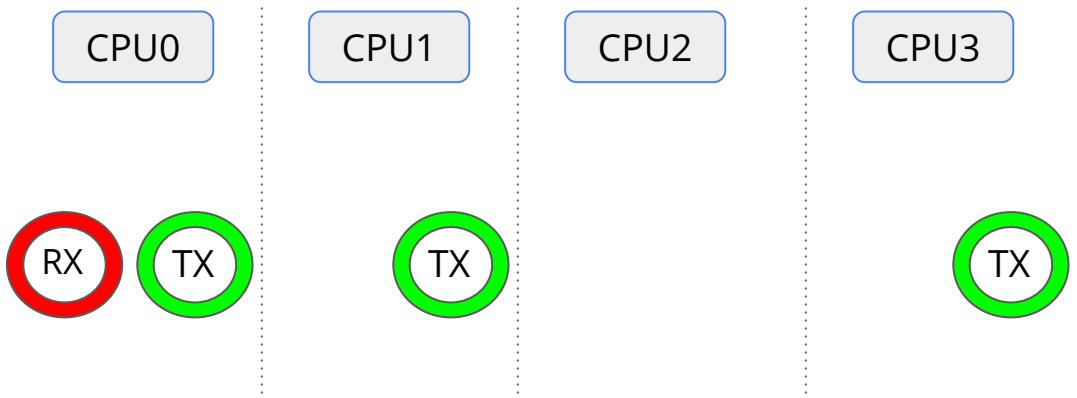
This way no contention between CPU cores, no locking and stepping on each other toe's

Solution Multi-Queue NICs (and other devices)

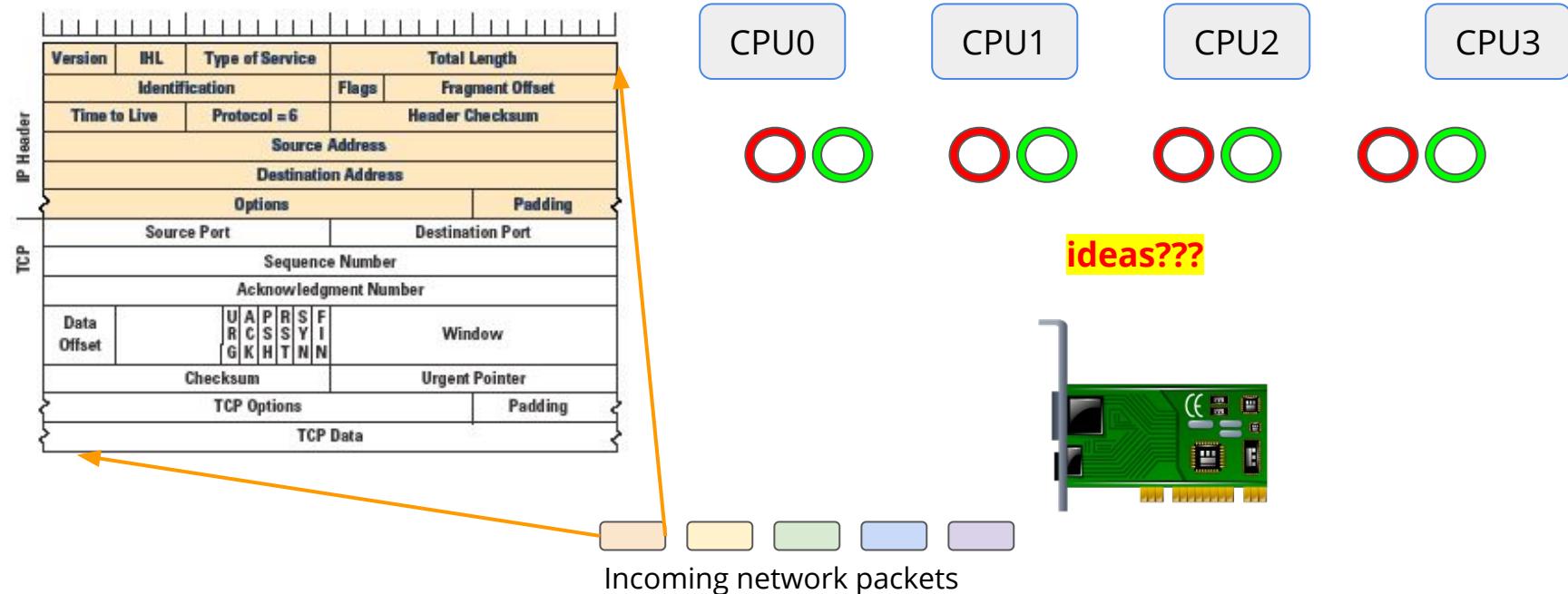
Caution: even though I am showing 4 queue pairs for 4 CPUs, in reality a NIC can have any number of TX and RX queues - 2, 4, 8, 16, so

- Multiple CPUs can share TX and RX queue
- Each queue (TX and RX) could work independently with their interrupts
- Multiple TX queues might share RX queues ($n:m$) mapping

Here it is 1:3 mapping (RX:TX) where CPU2 does not have a TX or RX queue



Next Challenge: which packet go to which queue?



Strategy 1: Random assignment

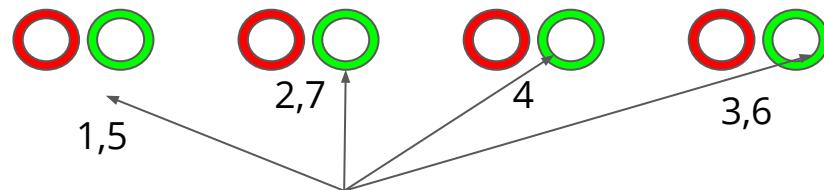
CPU0

CPU1

CPU2

CPU3

Connection 1
Connection 2



7 6 5 4 3 2 1

Incoming network packets

Will it work? YES

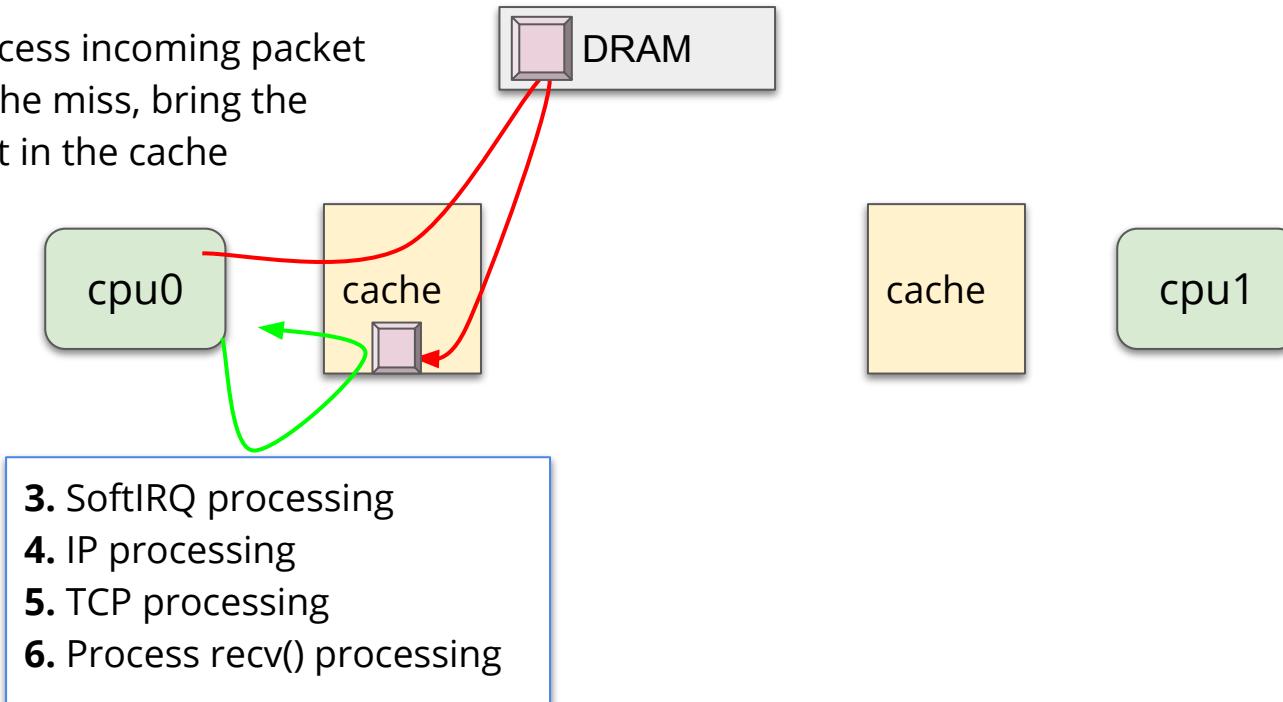
- + Simple and easy to implement
- + Load balanced at the *packet-level*
- What happens if packet 1,3,5 belong to *connection_1*, and 2,4,6,7 to *connection_2*?

Processing is distributed all over the place, locking, poor cache management

No connection based management

What is Poor Cache Locality Mean

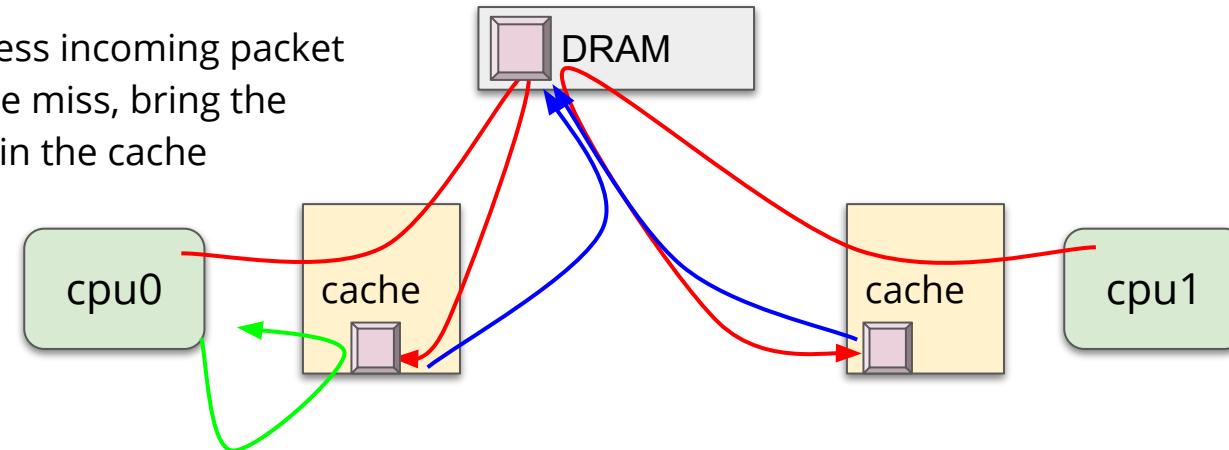
1. Process incoming packet
2. Cache miss, bring the packet in the cache



Done all in cache

What is Poor Cache Locality Mean

1. Process incoming packet
2. Cache miss, bring the packet in the cache



3. SoftIRQ processing
5. TCP processing

4. IP processing
6. Process recv() processing

Lots of unnecessary cache misses, write backs (blue lines), and hence, poor cache performance

We want to avoid this and pick and stick with which core is going to do packet processing for which packets, TCP connections, IP connections, etc. (flow-locality)

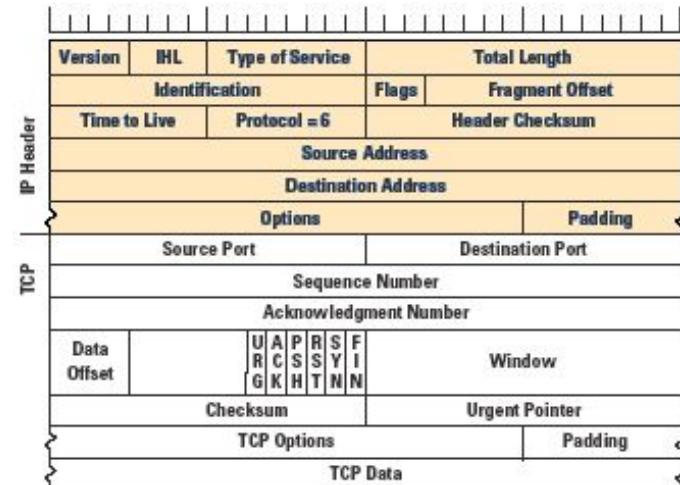
Strategy 2: Receive Side Scaling (RSS)

How do we identify a TCP flow?

4-Tuple {source_ip, destination_ip, source_port, destination_port}

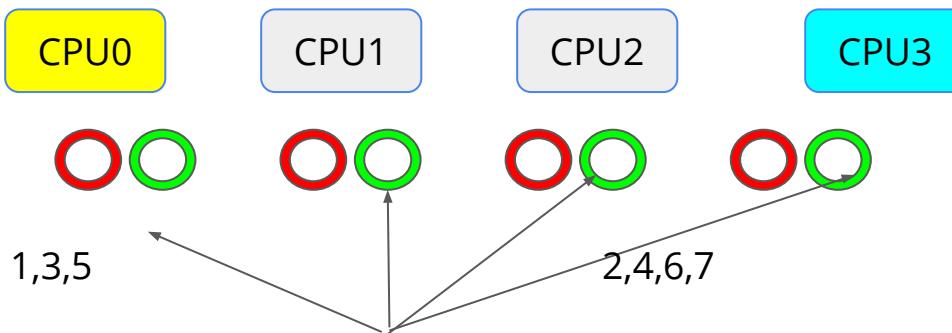
Execute a “hash” function over 4 values

- **Hash function:** deterministically map a set of values to another set, e.g., modulo (%) is an example hash function
- Same input -> same output
- Very low probability that two different values map to a same output hash



Hash function output is a the number of CPU core or queue number

Strategy 2: Receive Side Scaling (RSS)



```
RX_queue_number = hash('192.168.254.30', '192.168.254.1', 65400, 4321) %  
number_of_queues
```



7 6 5 4 3 2 1

A simple hash calculation :

IP addresses are 4 bytes
+ Port addresses are 2 bytes

Add all them up as a number then

Destination queue = sum % #queue

Now, packets 1,3,5 will be put to the same core, and 2,4,6,7 go to another

• *Intra connection parallelism is _HARD_*

RSS Advantages

Making sure that packets belonging to a same connection go to the same CPU

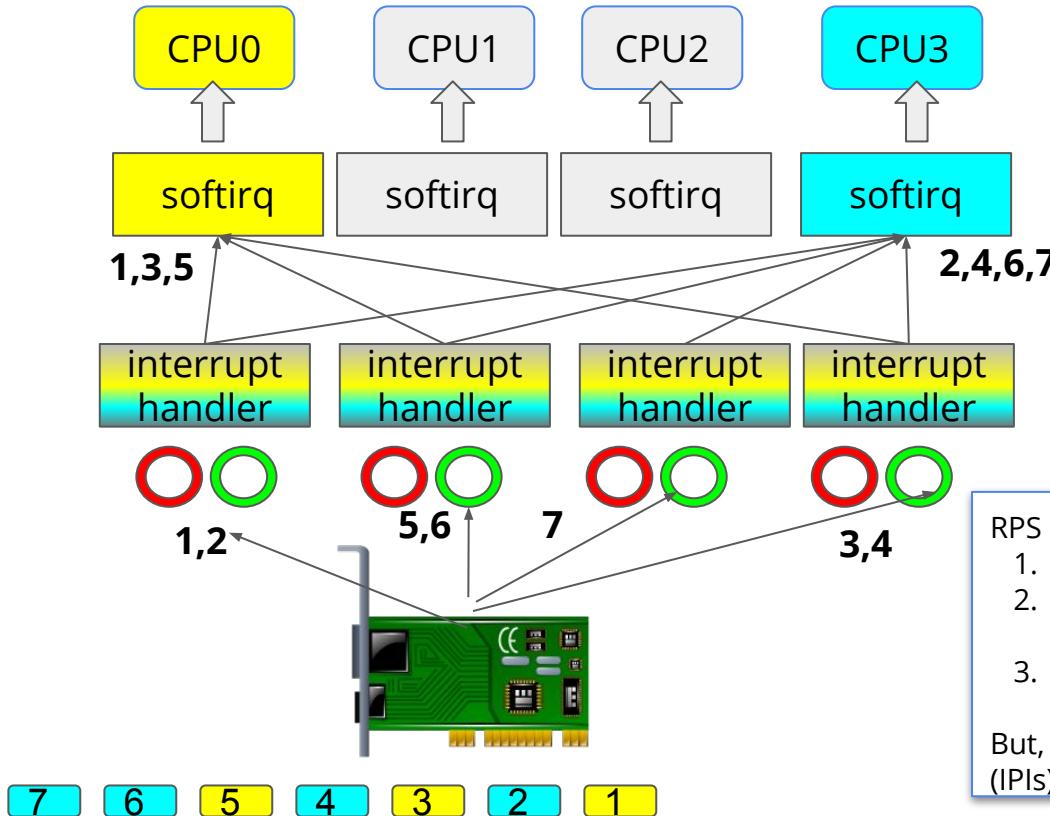
- Early decision on which CPU processing should be (in hardware) - early multiplexing
- Typically that CPU will have all the other data associated with that connection in the cache as well - cache locality ← *very important!*
- That CPU also knows that no other CPU can process packets for this connection - hence, no need to take locks
- Generally, good performance

However, it does not get support from the network card to be able to run a hash function for all incoming packets, *what if you don't have such hardware?*

<https://www.kernel.org/doc/Documentation/networking/scaling.txt>

<https://garycplin.blogspot.com/2017/06/linux-network-scaling-receives-packets.html>

Software Mechanism: RPS: Receive Packet Steering

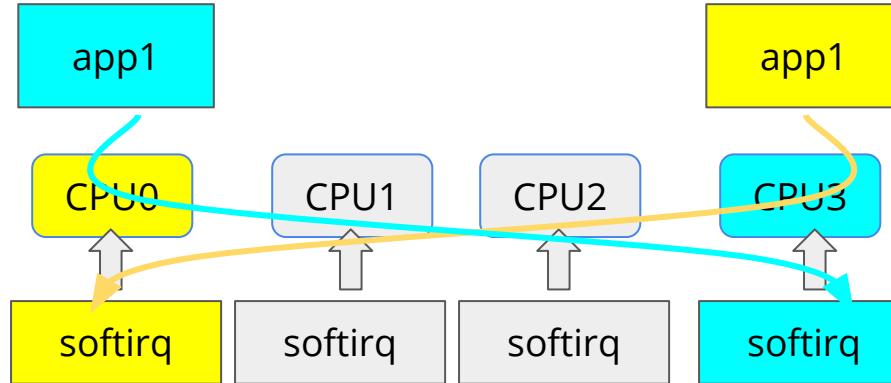


RPS has some advantages over RSS:

1. it can be used with any NIC
2. software filters can easily be added to hash over new protocols
3. Does not increase the interrupt rate

But, it does increase inter-processor interrupts (IPIs) and notifications.

What about Application Processing?

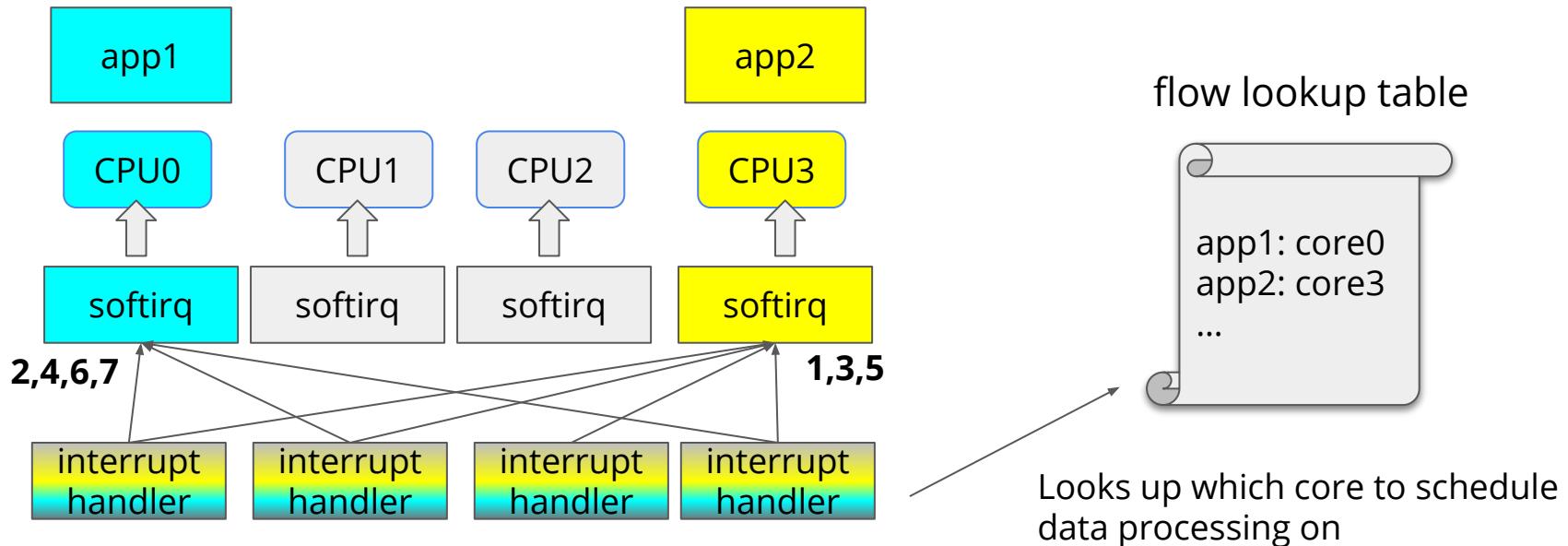


- Applications can be scheduled on any core where they call recv()
- They can be moved around as well
- Then how do we make sure that packet processing also respects "***application locality***"

<https://www.kernel.org/doc/Documentation/networking/scaling.txt>

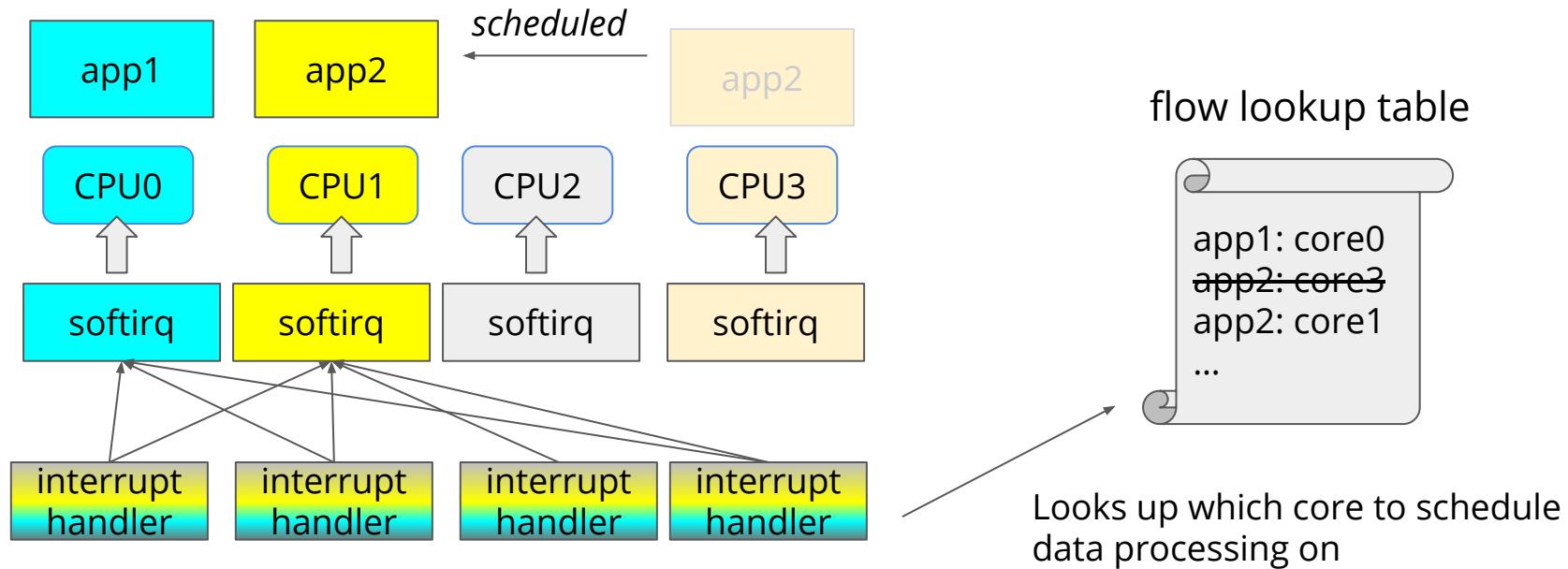
<https://garycplin.blogspot.com/2017/06/linux-network-scaling-receives-packets.html>

RFS: Receive Flow Steering (RFS)



- <https://www.kernel.org/doc/Documentation/networking/scaling.txt>
- <https://garycplin.blogspot.com/2017/06/linux-network-scaling-receives-packets.html>

RFS: Receive Flow Steering



RFS can be implemented in software or hardware (if appropriate NIC supported is there)

<https://www.kernel.org/doc/Documentation/networking/scaling.txt>

<https://garycplin.blogspot.com/2017/06/linux-network-scaling-receives-packets.html>

XPS: Transmit Packet Steering

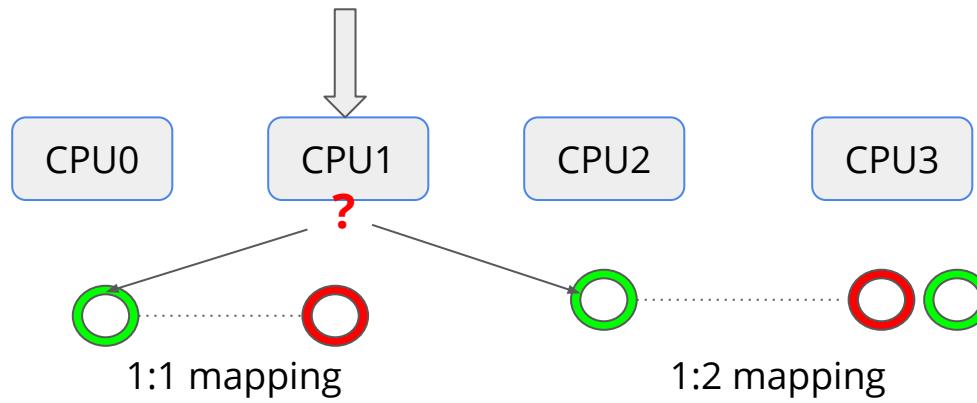
A similar concern arises on the transmit side, which transmit queue to choose to transmit a packet, **why?**

- Often there are n:m mapping between RX queue and TX queues, it makes sense to pick the TX queue, where its associated RX queue is
 - Why is this helpful?
- Data can be transmitted in softirq processing (with qdisc) processing
- General optimization for caching locality

Conceptually it works the same as RPS with a lookup data structure

```
ati@atr-XPS-13:~$ cat /sys/class/net/wlp2s0/queues/tx-2/xps_cpus
00
ati@atr-XPS-13:~$
```

Why you should consider RX:TX mappings



Which core should cpu1 pick for transmission? cpu0 or cpu2

XPS dictate that you should pick cpu0, because it has the TX queue associated with the RX queue which is on CPU1. And often in any network communication if you are transmission you are expecting to receive incoming packets - response, ACKs

- So when you pick CPU0 then the incoming packet will come to cpu1, which has all the connection state
- If you pick cpu2 then the incoming packet will arrive on cpu3, hence missing out on the connection state

In short, it depends upon your system architecture and NIC queue mappings

RSS, RPS, RFS, and XFS

Questions

1. *are they stateful or stateless offloads ?*
2. *do they help with per-packet or per-byte overheads?*

RSS, RPS, RFS, and XFS

Questions

1. *are they stateful or stateless ?*
2. *do they help with per-packet or per-byte overheads?*

Linux tool: ethtool

```
ethtool -N|-U|--config-nfc|--config-ntuple devname rx-flow-hash tcp4|udp4|ah4|esp4|sctp4|tcp6|udp6|ah6|esp6|sctp6 [m|v|t|s|d|f|n|r... |  
flow-type ether|ip4|tcp4|udp4|sctp4|ah4|esp4|ip6|tcp6|udp6|ah6|esp6|sctp6 [src xx:yy:zz:aa:bb:cc] [m xx:yy:zz:aa:bb:cc] [dst xx:yy:zz:aa:bb:cc] [m xx:yy:zz:aa:bb:cc]  
[proto N [m N]] [src-ip ip-address] [m ip-address] [dst-ip ip-address] [m ip-address] [tos N [m N]] [tclass N [m N]] [l4proto N [m N]] [src-port N [m N]]  
[dst-port N [m N]] [spi N [m N]] [l4data N [m N]] [vlan-etype N [m N]] [vlan N [m N]] [user-def N [m N]] [dst-mac xx:yy:zz:aa:bb:cc] [m xx:yy:zz:aa:bb:cc] [action N]  
[loc N]  
delete N
```

Receive flow hash

Which protocol, headers

Which queue?

Specific NIC vendors such Intel also offer their specific tools like Intel Flow Director.

<https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-ethernet-flow-director.pdf>

A bit of System Organization

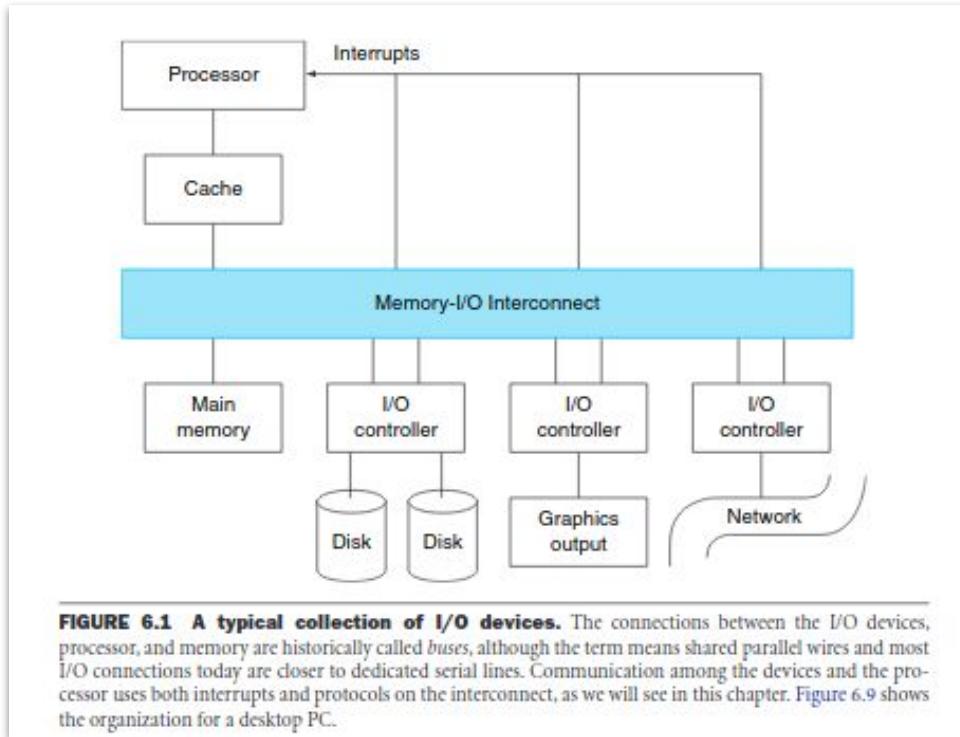
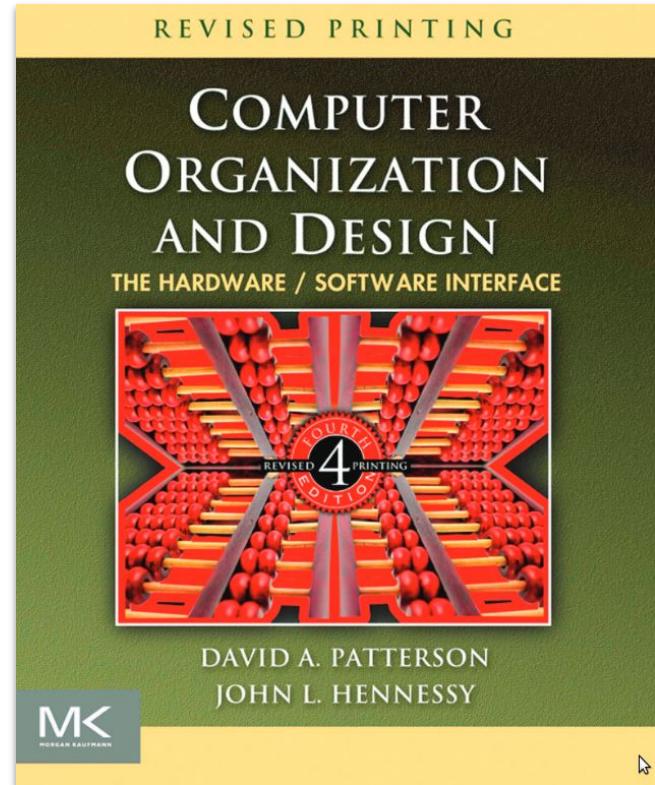


FIGURE 6.1 A typical collection of I/O devices. The connections between the I/O devices, processor, and memory are historically called *buses*, although the term means shared parallel wires and most I/O connections today are closer to dedicated serial lines. Communication among the devices and the processor uses both interrupts and protocols on the interconnect, as we will see in this chapter. Figure 6.9 shows the organization for a desktop PC.

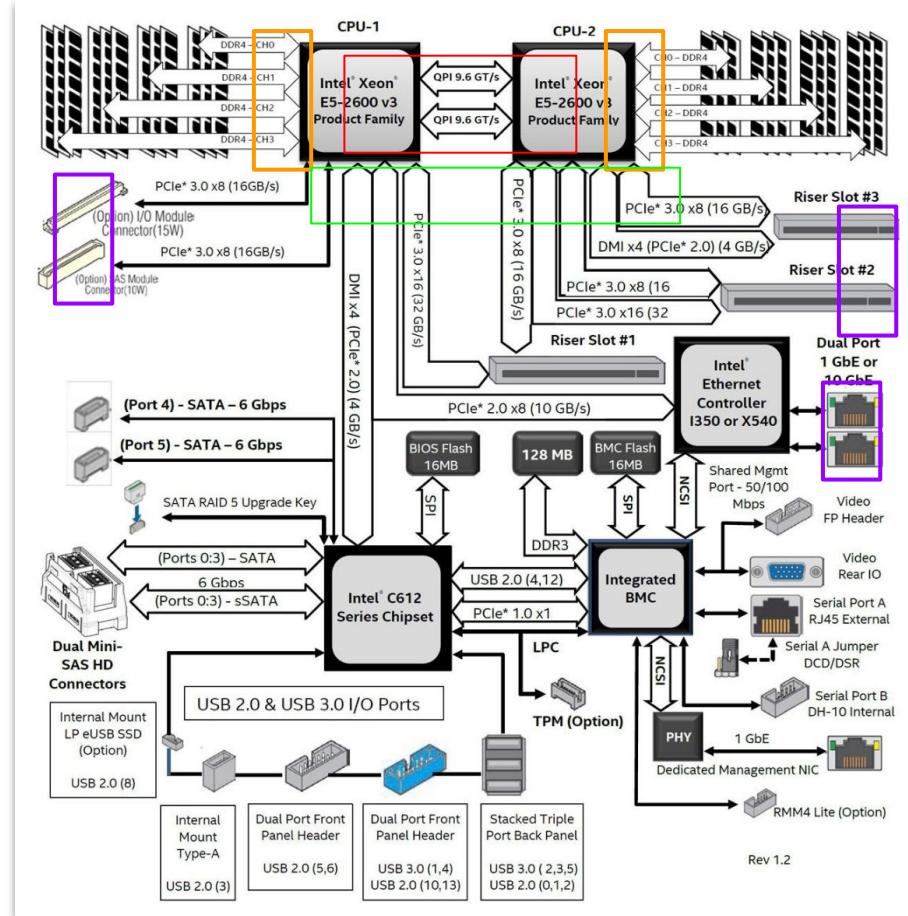


Modern Servers

Important changes

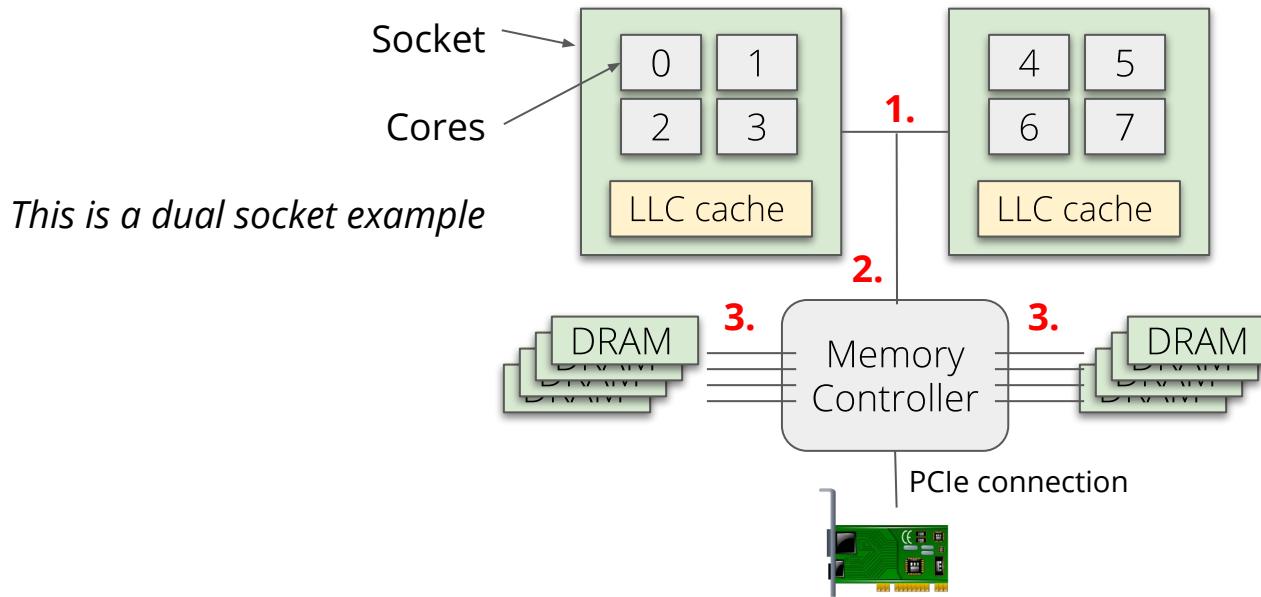
- Multicore systems
- Integrated memory controllers
- Integrated I/O lanes
 - Ethernet locations
- NUMA vs SMP effect
 - Symmetric Multi-Processing
 - Non-Uniform Memory Access

And much more ...



<https://www.tweaktown.com/reviews/7058/intel-server-r2208wt2ys-system-review/index.html>

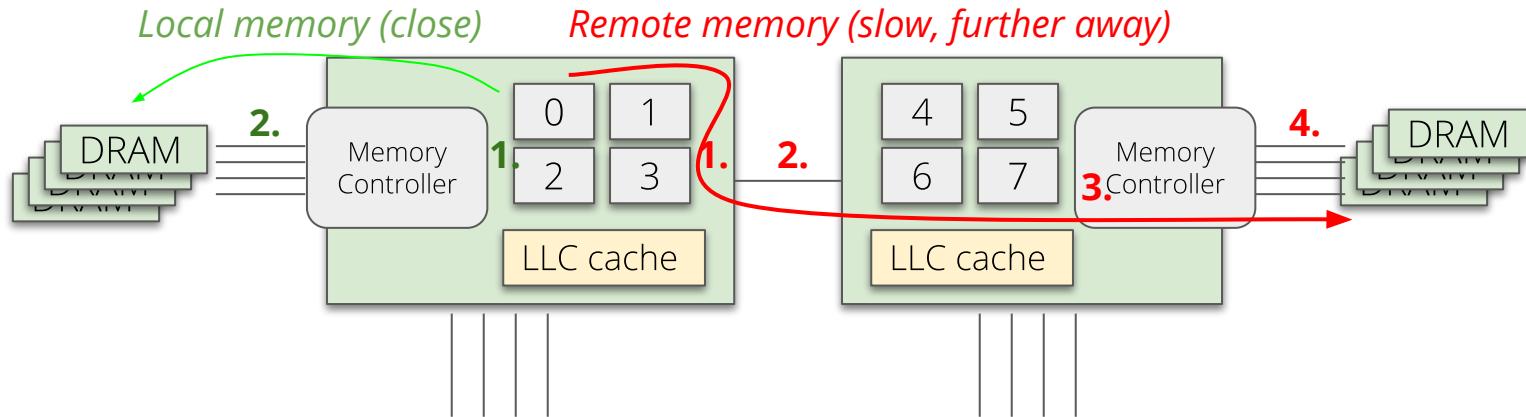
SMP vs NUMA : Example SMP Machine



Key property: memory is equidistant from all CPU cores and the NIC

- *It does not matter which core or memory to choose, because none of them is special (all of them 3 hops away for every core)*

SMP vs NUMA : Example NUMA Machine



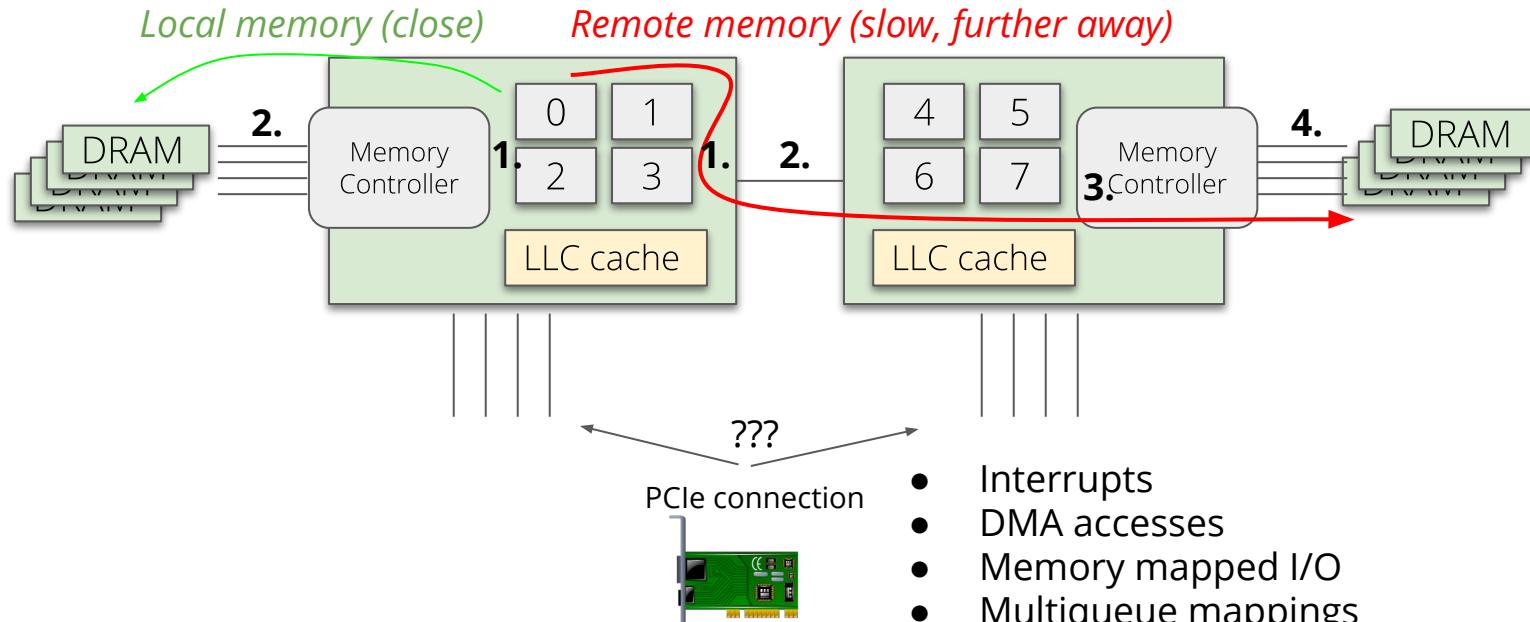
Key property: Memory and cores are not equidistant

- **It does matter** which core or memory to choose, because some of them are closer than others

Distance to local memory for [0-4]: 2 hops, distance to remote memory [0-3]: 4 hops

How does it look for a single socket machine?

SMP vs NUMA



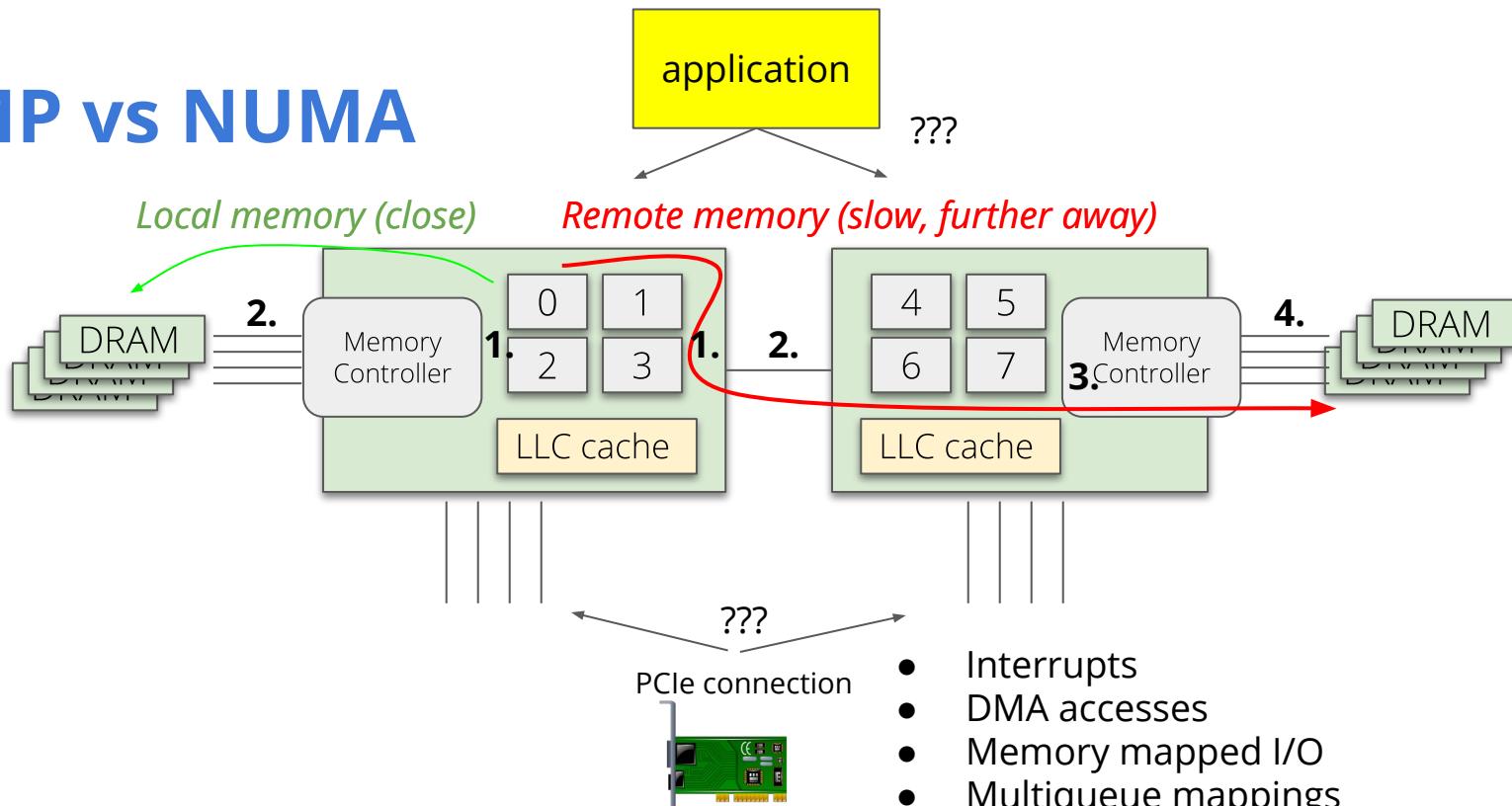
Key property: Memory and cores are not equidistant

- **It does matter** which core or memory to choose, because some of them are closer than others

Distance to local memory for [0-4]: 2 hops, distance to remote memory [0-3]: 4 hops

How does it look for a single socket machine?

SMP vs NUMA



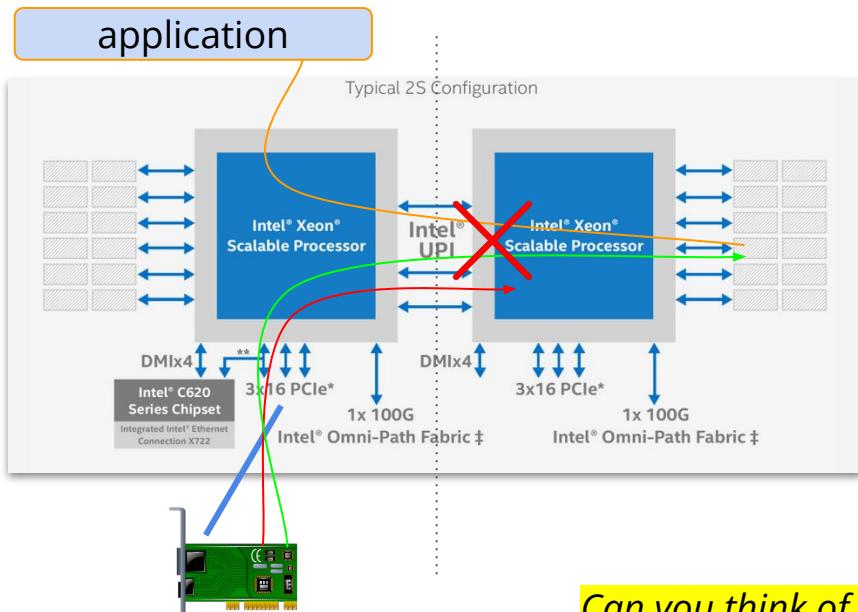
Key property: Memory and cores are not equidistant

- **It does matter** which core or memory to choose, because some of them are closer than others

Distance to local memory for [0-4]: 2 hops, distance to remote memory [0-3]: 4 hops

How does it look for a single socket machine?

Impact of SMP and NUMA Architectures

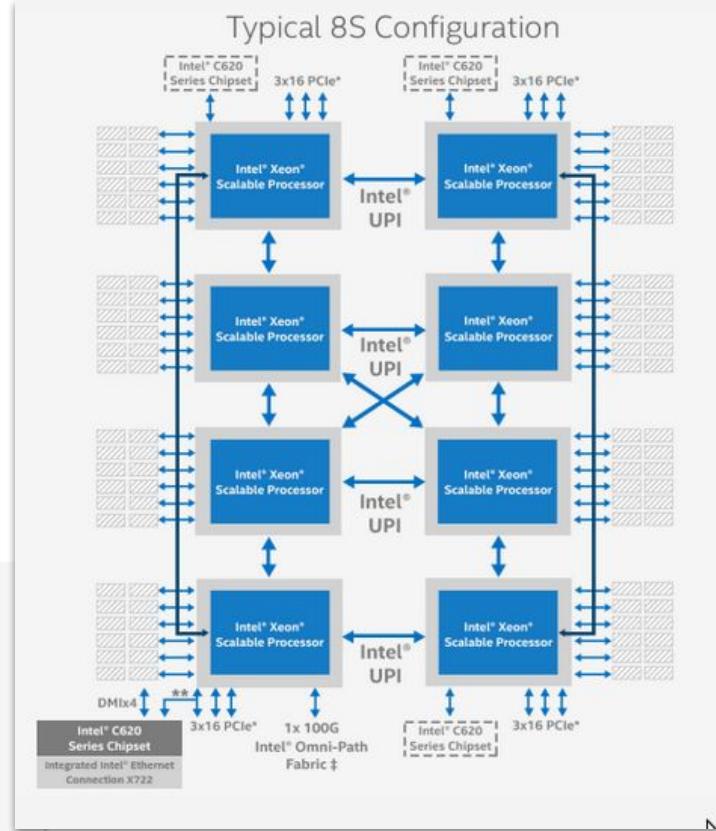
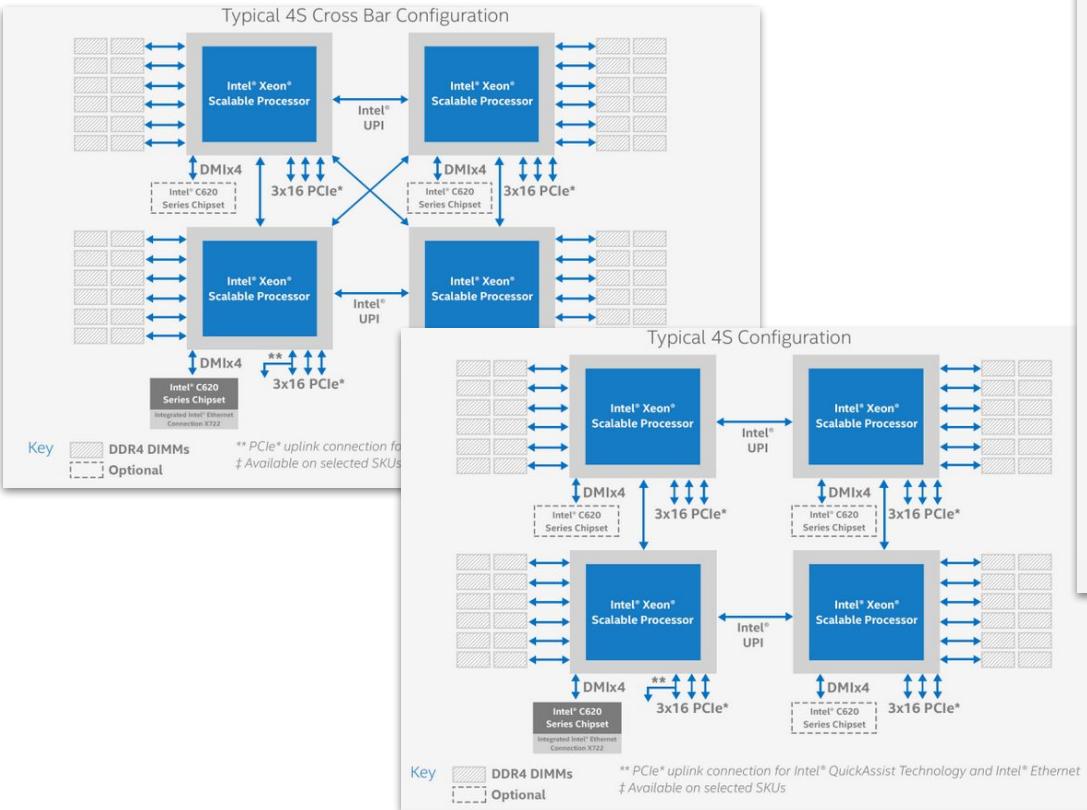


Here are multiple concerns

1. Which CPU (socket) NIC is connected to?
2. Which CPU cores its interrupts and queues mapped to?
3. Where memory for DMA is allocated?
4. Where application is processing networking data?

Can you think of a solution for this dual-socket machine?

Now do for these... ;)



Its an open research problem, to do it

- *Automatically*
- *Efficiently*
- *For all machines → See Barreelfish OS*

Linux Tool: numactl

```
NUMACTL(8)                               Linux Administrator's Manual                               NUMACTL(8)

NAME
    numactl - Control NUMA policy for processes or shared memory

SYNOPSIS
    numactl [ --all ] [ --interleave nodes ] [ --preferred node ] [ --membind nodes ] [ --cpunodebind nodes ] [ --physcpubind cpus ] [ --localalloc ] [--] command {arguments
    ...}
    numactl --show
    numactl --hardware
    numactl [ --huge ] [ --offset offset ] [ --shmmode shmmode ] [ --length length ] [ --strict ]
    [ --shmfd id ] --shm shmkeyfile | --file tmpfsfile
    [ --touch ] [ --dump ] [ --dump-nodes ] memory policy

DESCRIPTION
    numactl runs processes with a specific NUMA scheduling or memory placement policy. The policy is set for command and inherited by all of its children. In addition it can
    set persistent policy for shared memory segments or files.

    Use -- before command if using command options that could be confused with numactl options.

    nodes may be specified as N,N,N or N-N or N,N,N or N-N,N-N and so forth. Relative nodes may be specified as +N,N,N or +N-N or +N,N-N and so forth. The + indicates that
    the node numbers are relative to the process' set of allowed nodes in its current cpuset. A !N-N notation indicates the inverse of N-N, in other words all nodes except N-N.
    If used with + notation, specify !+N-N. When same is specified the previous nodemask specified on the command line is used. all means all nodes in the current cpuset.

    Instead of a number a node can also be:

    netdev:DEV           The node connected to network device DEV.
    file:PATH            The node the block device of PATH.
    ip:HOST              The node of the network device of HOST
    block:PATH           The node of block device PATH
    pci:[seg:]bus:dev[:func]   The node of a PCI device.

    Note that block resolves the kernel block device names only for udev names in /dev use file:

    Policy settings are:

    --all, -a
        Unset default cpuset awareness, so user can use all possible CPUs/nodes for following policy settings.

    --interleave=nodes, -i nodes
        Set a memory interleave policy. Memory will be allocated using round robin on nodes. When memory cannot be allocated on the current interleave target fall back to
        other nodes. Multiple nodes may be specified on --interleave, --membind and --cpunodebind.

    --membind=nodes, -m nodes
        Only allocate memory from nodes. Allocation will fail when there is not enough memory available on these nodes. nodes may be specified as noted above.

    --cpunodebind=nodes, -N nodes
        Only execute command on the CPUs of nodes. Note that nodes may consist of multiple CPUs. nodes may be specified as noted above.

    --physcpubind=cpus, -C cpus
        Only execute command on the CPUs of cpus. Note that cpus may consist of multiple CPUs. cpus may be specified as noted above.
```

Research Paper: MegaPipe (2012)

MegaPipe: A New Programming Interface for Scalable Network I/O

Sangjin Han*, Scott Marshall*, Byung-Gon Chun*, and Sylvia Ratnasamy*

*University of California, Berkeley

*Yahoo! Research

Abstract

We present MegaPipe, a new API for efficient, scalable network I/O for message-oriented workloads. The design of MegaPipe centers around the abstraction of a *channel* – a per-core, bidirectional pipe between the kernel and user space, used to exchange both I/O requests and event notifications. On top of the channel abstraction, we introduce three key concepts of MegaPipe: partitioning, lightweight socket (lwsocket), and batching.

We implement MegaPipe in Linux and adapt memcached and nginx. Our results show that, by embracing a clean-slate design approach, MegaPipe is able to exploit new opportunities for improved performance and ease of programmability. In microbenchmarks on an 8-core server with 64 B messages, MegaPipe outperforms baseline Linux between 29% (for long connections) and 582% (for short connections). MegaPipe improves the performance of a modified version of memcached between 15% and 320%. For a workload based on real-world HTTP traces, MegaPipe boosts the throughput of nginx by 75%.

1 Introduction

Existing network APIs on multi-core systems have difficulties scaling to high connection rates and are inefficient for “message-oriented” workloads, by which we mean workloads with short connections¹ and/or small messages. Such message-oriented workloads include HTTP,

ing and nonblocking communication, asynchronous I/O, event polling, and so forth – limits the extent to which it can be optimized for performance. In contrast, a clean-slate redesign offers the opportunity to present an API that is specialized for high performance network I/O.

An ideal network API must offer not only high performance but also a simple and intuitive programming abstraction. In modern network servers, achieving high performance requires efficient support for *concurrent I/O* so as to enable scaling to large numbers of connections per thread, multiple cores, etc. The original socket API was not designed to support such concurrency. Consequently, a number of new programming abstractions (e.g., epoll, kqueue, etc.) have been introduced to support concurrent operation without overhauling the socket API. Thus, even though the basic socket API is simple and easy to use, programmers face the unavoidable and tedious burden of layering several abstractions for the sake of concurrency. Once again, a clean-slate design of network APIs offers the opportunity to design a network API from the ground up with support for concurrent I/O.

Given the central role of networking in modern applications, we posit that it is worthwhile to explore the benefits of a clean-slate design of network APIs aimed at achieving both high performance and ease of programming. In this paper we present MegaPipe, a new API for efficient, scalable network I/O. The core abstraction MegaPipe introduces is that of a *channel* – a per-core bi-directional

MegaPipe:A New Programming Interface for Scalable Network I/O

What: *is a new networking abstraction for doing high-performance network operations*

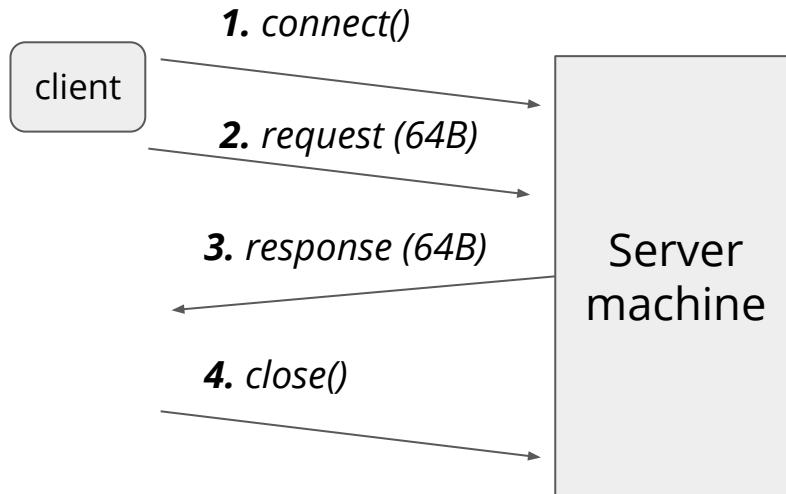
Why:

- High overheads in small packet processing
- Inefficiencies in the Linux kernel networking stack with scalability

What do we mean by “scalable network I/O”:

1. How does the system perform when we increase number of concurrent connection
2. How does the system perform with increasing number of cores in the system

Setup and Challenge

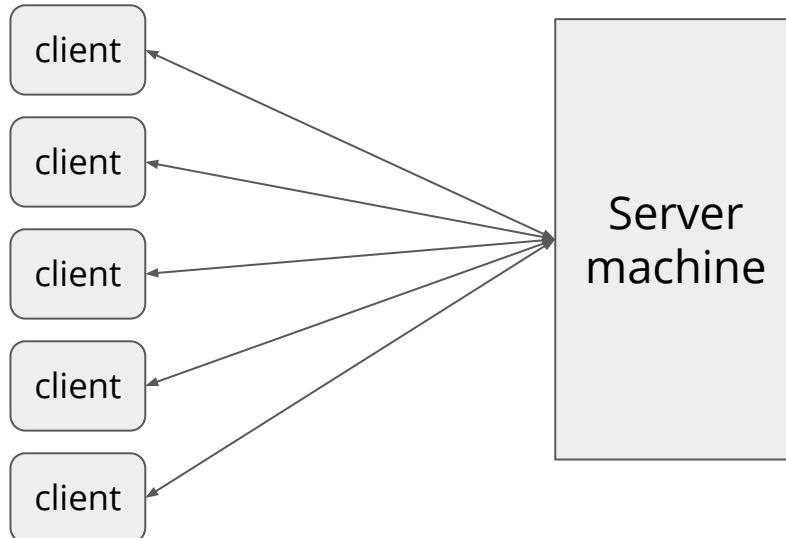


- These clients connect to a server machine
- Request a file, operation, or transaction - get a response back
- “Small” request - response (not a data heavy workload) - a few bytes to kilobytes
- “Short-lived” - quick connect, disconnect

Very common network pattern inside a data center

- On internet as well, but does not matter that much here, why?

Setup and Challenge



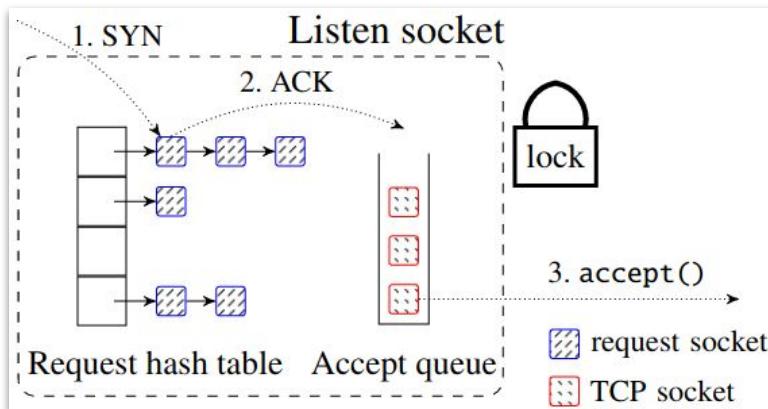
- These clients connect to a server machine
- Request a file, operation, or transaction - get a response back
- **"Small" request - response** (not a data heavy workload) - a few bytes to kilobytes
- **"Short-lived"** quick connect, disconnect

Very stressful to the CPU, system cannot use many of the previously discussed tricks

- System need to process millions packets/sec
- **Per-packet costs dominate**
 - Packet and protocol processing
 - Per packet memory management
 - Scheduling

TSO, LRO, checksum offloading, Jumbo frames are not useful - why?

Specific Problems (1) Global Accept Queue



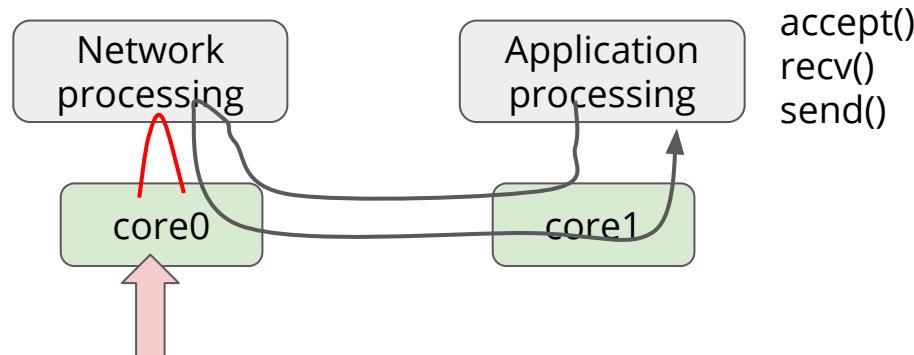
1. Incoming SYN packets are put in a “request hash table”
2. Once SYN + ACK is done, they are moved to a “accept queue”
3. Once, the application calls “accept” they are taken out from the accept queue

Problem: The request hash table and accept queues are shared between all “cores”

- Locking
- Imagine what happens when thousands of new TCP clients connect?

Specific Problems (2) Lack of Connection Affinity

The open problem that we discussed before



The problem becomes more challenging because previous flow-steering mechanisms do flow steering after “sampling” some packets

With short connections - there aren't enough packets to sample

Specific Problems (3) Implementation Details

1. What is a socket
 - a. It is a file descriptor
 - b. Complain to the POSIX standard
 - c. "*The POSIX standard requires that a newly allocated file descriptor be the lowest integer not currently used by the process*" - figuring out minimum requires coordination between all CPUs (not needed)
2. All file descriptors get attached to the Linux VFS (everything is a file)
 - a. The VFS has its own set of file instance, inode, and dentry data structures
 - b. For short connections - lots of global state allocation and de-allocation (not needed)
3. System calls
 - a. Is the way we communicate with the operating system
 - b. But they have performance issues (raises an interrupt/exception, disrupt ongoing flow)

What does MegaPipe propose

1. Partition the Request Hash Table and Accept queue for per-core
 - a. Application dictated partitioning and accept redirection
 - b. Concurrent work with Affinity Accept and Linux 3.19 (SO_REUSEPORT)
 - c. Allows multiple threads to listen on to the same port number
2. A special “lightweight” socket descriptor
 - a. Not a file, but just an identifier. Avoids the VFS overheads
3. A new channel and message based API which allows system call “batching”
 - a. Multiple send/recv requests can be passed in one go, hence, amortizing the overhead of doing a system call
 - b. Readiness vs notification based systems

Understand

stream vs. message based I/O

- TCP is a “**stream**” protocol
- **Stream interfaces or byte-by-byte** interfaces for files and sockets (as they both are treated as files)
- **read()** and **write()** can send/recv any number of bytes
- BSD sockets do not have any idea of what is a message

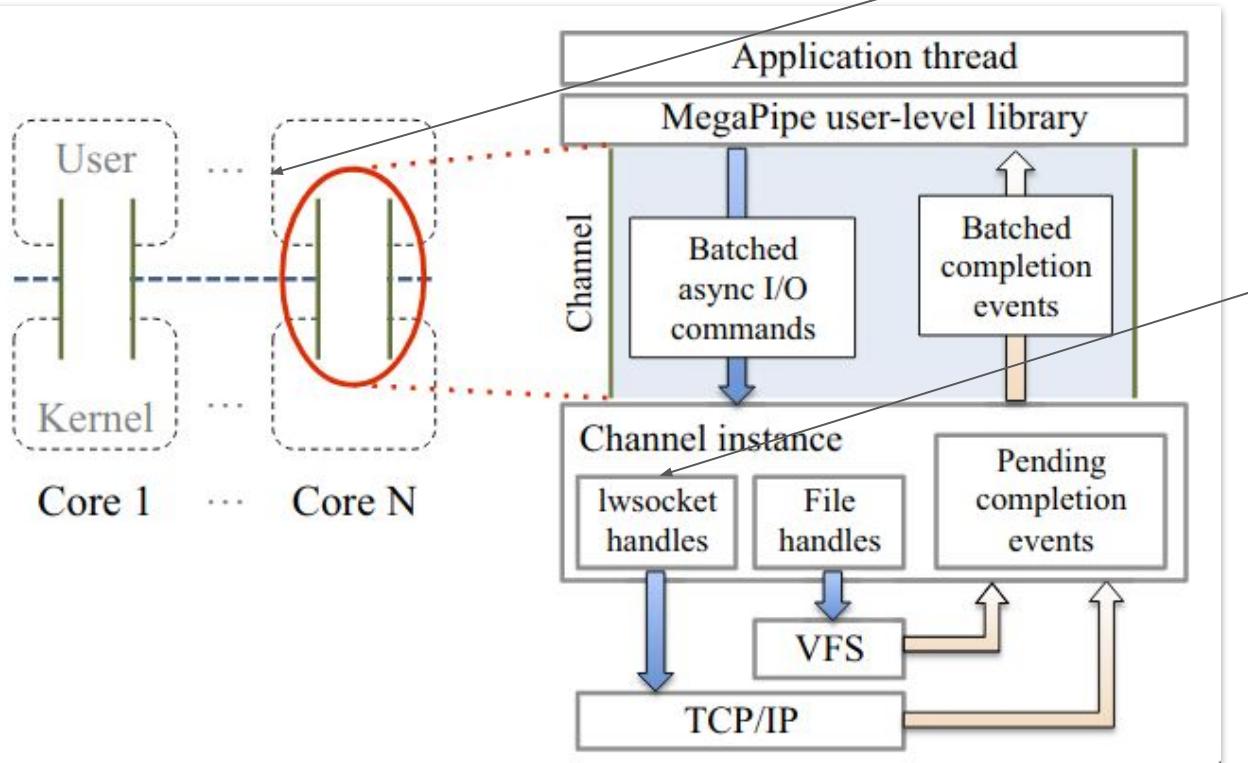
readiness vs. event based I/O

- **Readiness model for sockets:** an application needs to constantly check if sockets are *ready* for more I/O (epoll, select, non-blocking I/O)
- **Event based model:** application posts an I/O operations and get an event in response when the operation completes
 - No constant “readiness” checks
 - But needs how to deliver completion event?

MegaPipe Architecture

Partitioned on each core

- Application provided CPU mask



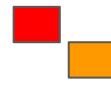
Channel and Notification Based I/O

```
int send (void *, int...)  
  
int recv (void *, int...)
```

Notification based I/O:

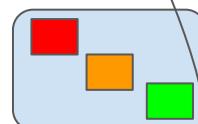
- A simple notification on a separate channel
- Per operation - no need to constantly keep track of
- Efficient, less application involvement

1. messages in one shot



I/O channel
(same idea as RX/TX queues)

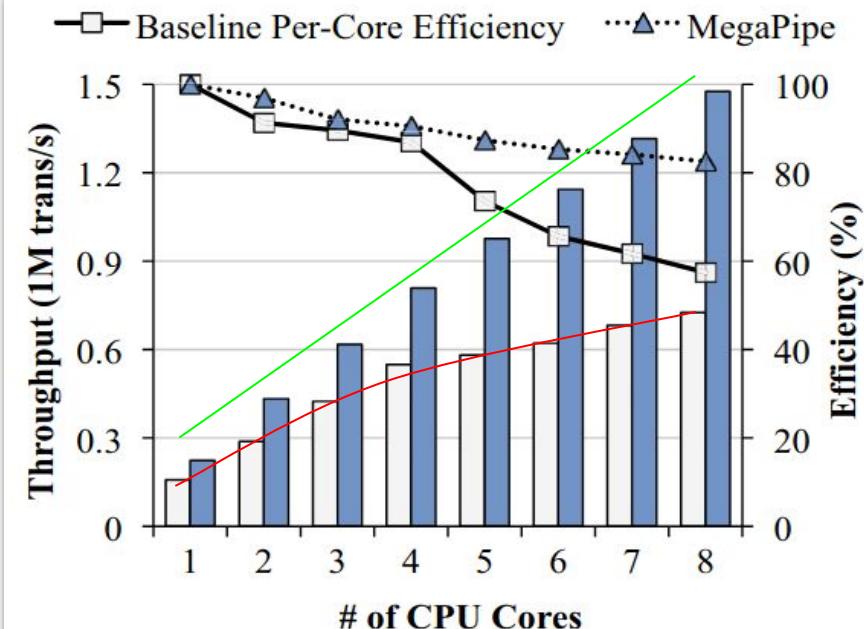
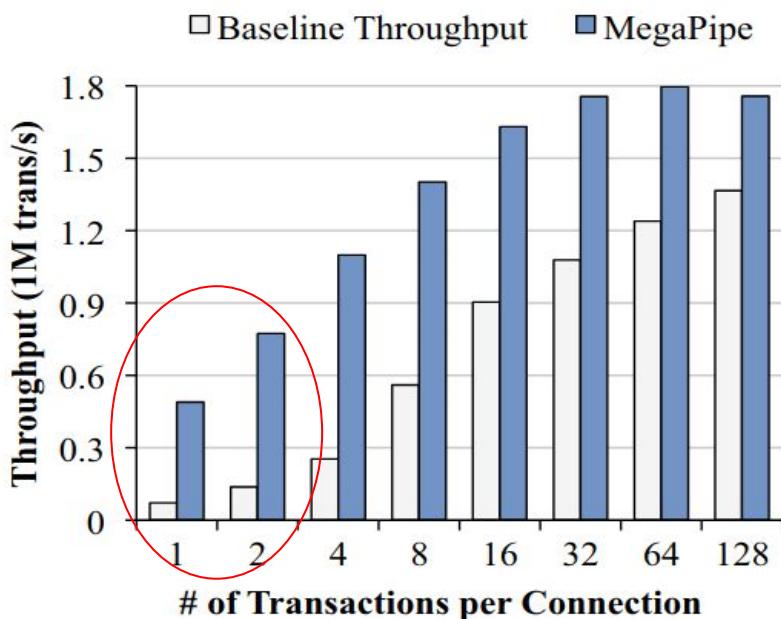
2. batching



3. Syscall to megapipe

4. Batch completion event *notification*

What all of this buys you?



However

1. New non-socket API
 - a. Very hard to convince people to rewrite their networking code
 - b. New semantics
2. Kernel modifications
 - a. Very hard to convince people to modify their kernels
3. Need support from the application
 - a. Very hard to convince people to tell the networking stack how to partition the listening socket and manage accept queues

Some of its proposals (and the prior work, Affinity Accept) are part of the Linux kernel now

Some of these issues you will be facing in the project

1. How do you plan to keep track of outgoing SYN packets?
2. How do you allocate a file descriptor
3. How do you keep track of ANP allocated file descriptors?
4. Are you doing anything special for multicore systems?
5. Anything else you can think of?

Recap So far

1. What is interrupt load balancing
2. How do you find which interrupt(s) a device is using and which CPU core(s) is servicing that interrupt(s)
3. How to map an interrupt to a single/multiple CPU cores
4. What is a multi-queue NIC, what does it help with
5. What is RSS, RPS, RFS, and XFS - and what is the difference between them
6. What impact does NUMA system have on NIC configuration and network processing?
7. What problem(s) MegaPipe solves and how

Do not forget office hours from 3:30 to 4:30

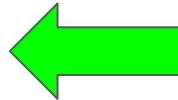
Layout of upcoming lectures - Part 1

Sep 1st, 2020 (today): ~~Introduction and networking concepts~~

Sep 3rd, 2020 (this Tuesday): ~~Networking concepts (continued)~~

Sep 8th, 2020 : ~~Linux networking internals~~

Sep 10th 2020: ~~Multicore scalability~~

Sep 15th 2020: *Userspace networking stacks* 

Sep 17th 2020: *Introduction to RDMA networking*