fast linux networking: DPDK, XDP, and Garuda

nick black <niblack> for azure orbital 2023-02-03

PART I

NETWORKING IS A HOUSE DIVIDED

A DICHOTOMY OF NETWORKED ENTITIES

Some networked devices exist almost entirely to serve exactly one function. Performance is maximized by allowing that one application full use of the machine and network.

Other devices offer unpredictable, diverse functionality. They provide fairness among applications, but introduce overhead such that it is impossible for any one task to reach this peak performance.

THE MINIMAL, IDEAL NETWORK

Two nodes.

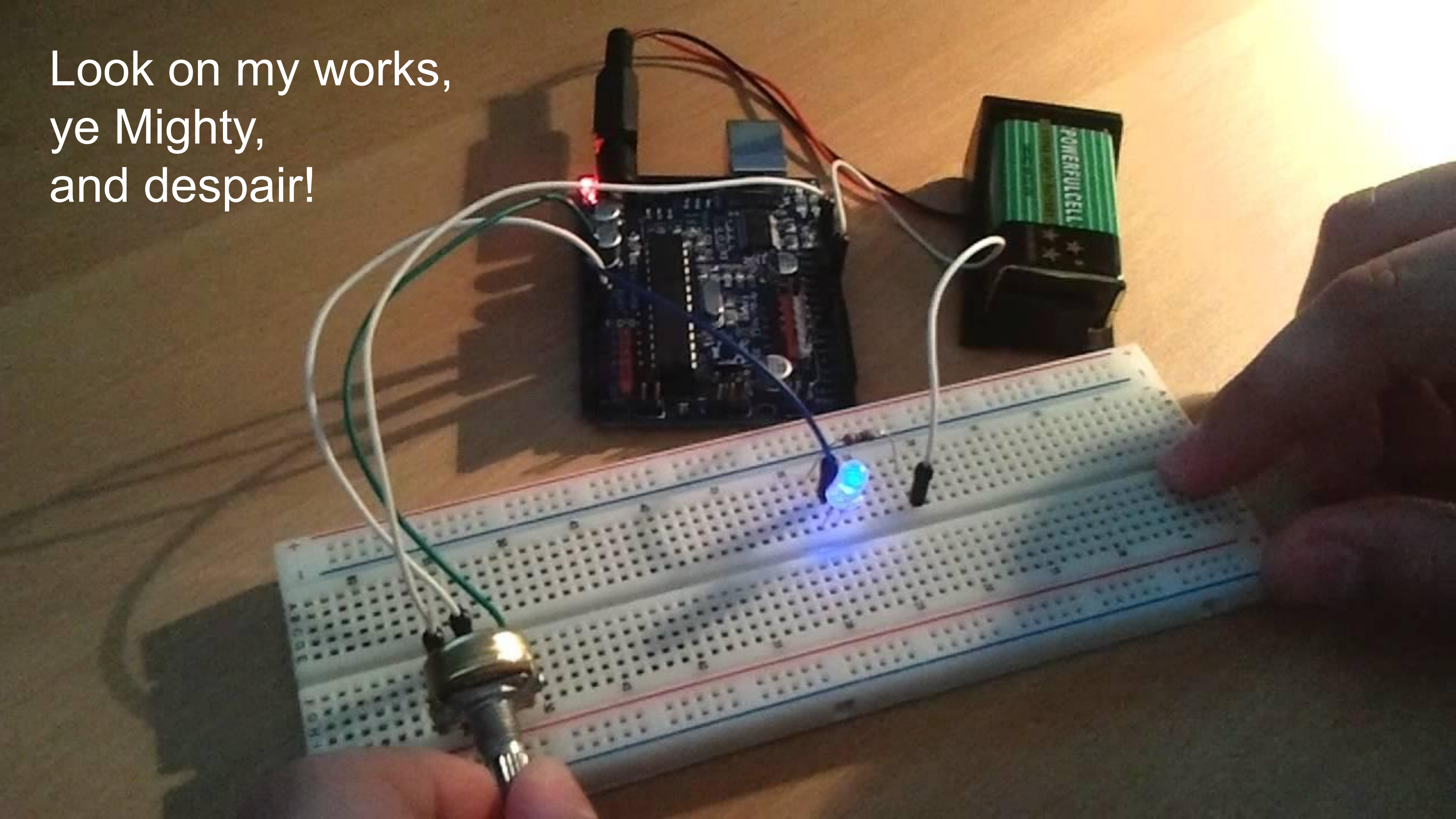
One application per node.

Infinite buffering.

Reliable transport.

In this happy model, there is no need for a protocol.

No addresses. No ports. No headers. No packets. Full utilization of the channel.



Look on my works,
ye Mighty,
and despair!

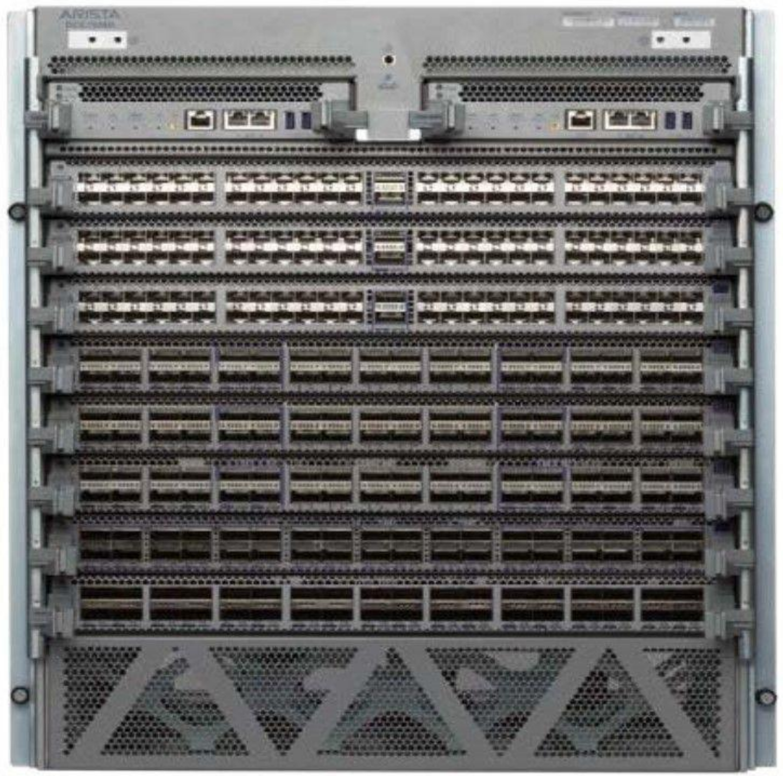
ARISTA 7508r3

230Tbps switching capacity

9.6Tbps per line card

288x 400Gbps ports

at half a megadollar USD, sadly
unsuitable for personal projects.



HISTORY IS A NIGHTMARE FROM WHICH I AM TRYING TO AWAKE

Traffic grows to fill the network. Application programmers convert the wins of systems programmers and hardware designers into raytraced cat pictures.

Performance is always paramount. Every cycle matters.

The history of networking is an unceasing struggle to fully utilize hardware, *seeking both to maximize a given flow's performance, and to maximize the overall performance of many flows*, while dodging javascript cats yeeted from userspace and being DDoSed by teenage botnet operators.

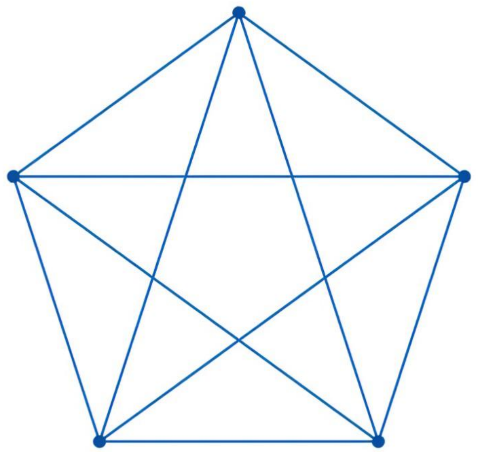
ADDRESSING: BEYOND THE CROSSBAR

As nodes are added to the network, a complete graph quickly becomes impractical.

Remove underutilized edges, *routing* that traffic instead through large primary connections. Now you've got trunks, making up a backbone.

Less total capacity, better utilization, lower cost.

Possibility of indirect transfer mandates addresses (a one-time header), our first source of overhead.

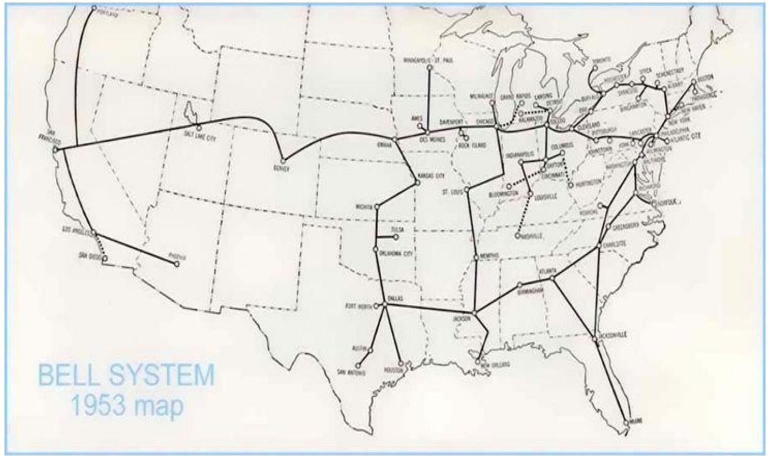

K_5

PACKET SWITCHING: SCREW KANSAS

Baran 1960, Davies 1965: we can better utilize this network by switching packets rather than preallocating channels, and stop caring about wichita eating a soviet nuke.

packets reproduce the per-channel overhead N times, while adding framing overhead.

maximum packet length is itself, in part, a question of fairness.



RFC 1 §1, AND THE UR-NETWORK

I. A Summary of the IMP Software

Messages

Information is transmitted from HOST to HOST in bundles called messages. A message is any stream of not more than 8080 bits, together with its header. The header is 16 bits and contains the following information:

Destination	5 bits
Link	8 bits
Trace	1 bit
Spare	2 bits

The destination is the numerical code for the HOST to which the message should be sent. The trace bit signals the IMPs to record status information about the message and send the information back to the NMC (Network Measurement Center, i.e., UCLA). The spare bits are unused.

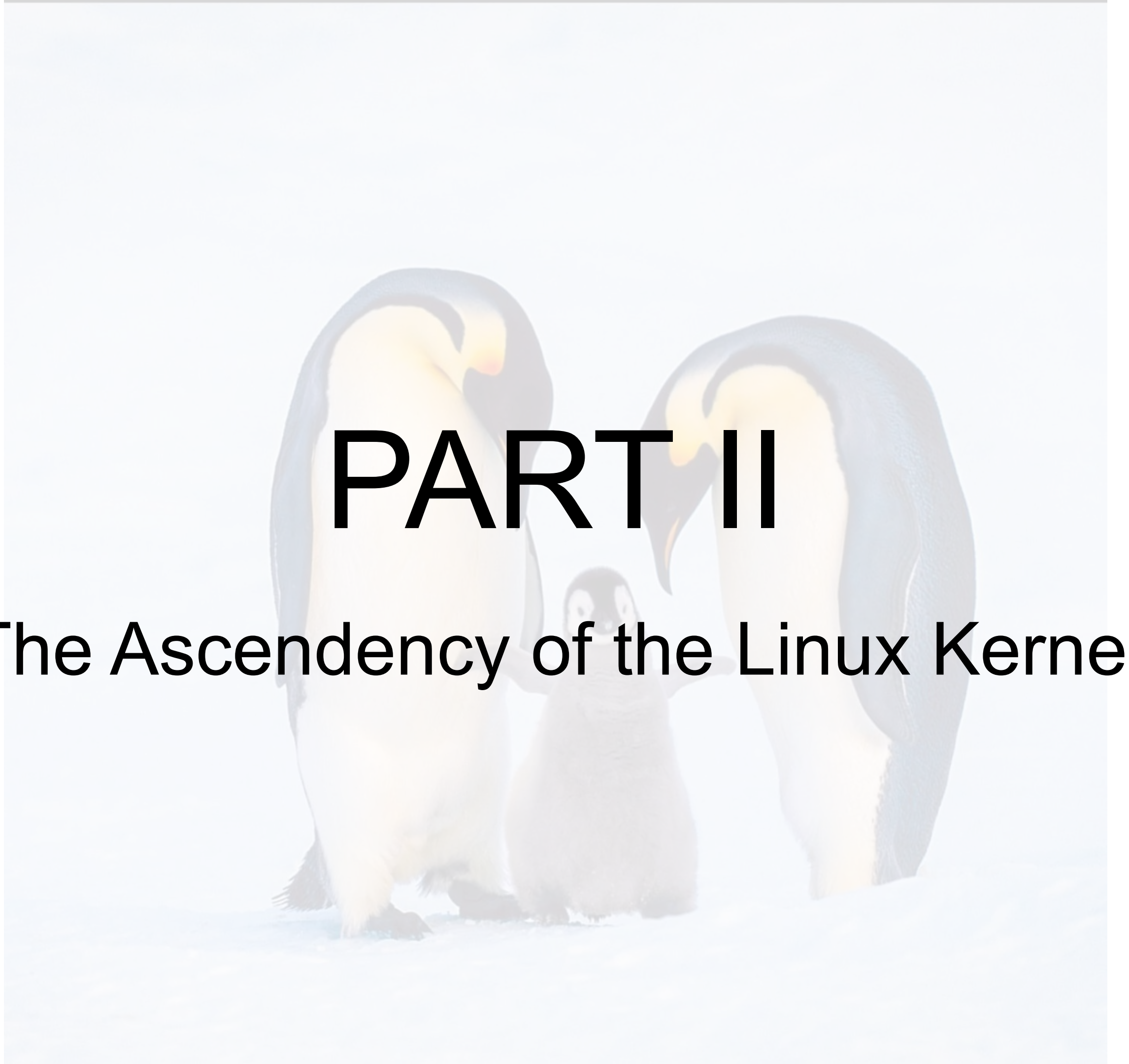
ADDRESSING AND FRAMING ARE SIGNIFICANT OVERHEADS

Ethernet (sans 802.1q): 256 bit-times of 304 (84.2%)

- IPv4: 10.5 octets of minimum 20 (52.5%)
- IPv6: 34 octets of minimum 40 (85%)
  - UDP: 4 octets of 8 (50%)
  - TCP: 4.5 octets of minimum 20 (22.5%)

potentially 81.4% of minimum 672 bit-time Ethernet frame (212% overhead)!

...yet we pay them in the name of flexibility.



PART II

The Ascendency of the Linux Kernel

FORTY YEARS OF MICROSOFT DROPPING THE BALL

1983-01-01: Flag Day. ARPANet moves from NCP to TCP/IP.

1983-08: BSD4.2 released with Berkeley Sockets.

1985: Microsoft implements subset of unroutable NetBIOS for MS-Net

1987-11: RFCs 1034 and 1035 codify modern DNS

1987: Microsoft implements LAN Manager atop SMB1 atop NBF

1993: RFC 1510 specifies Kerberos 5

1993: Microsoft adds NTLMv1 to LAN Manager, releases it on NT3.1 + WfW 3.1

1995: Windows 95 emerges with TCP/IP and Winsock...over a decade late

2023: Windows kernel still closed-source, lol

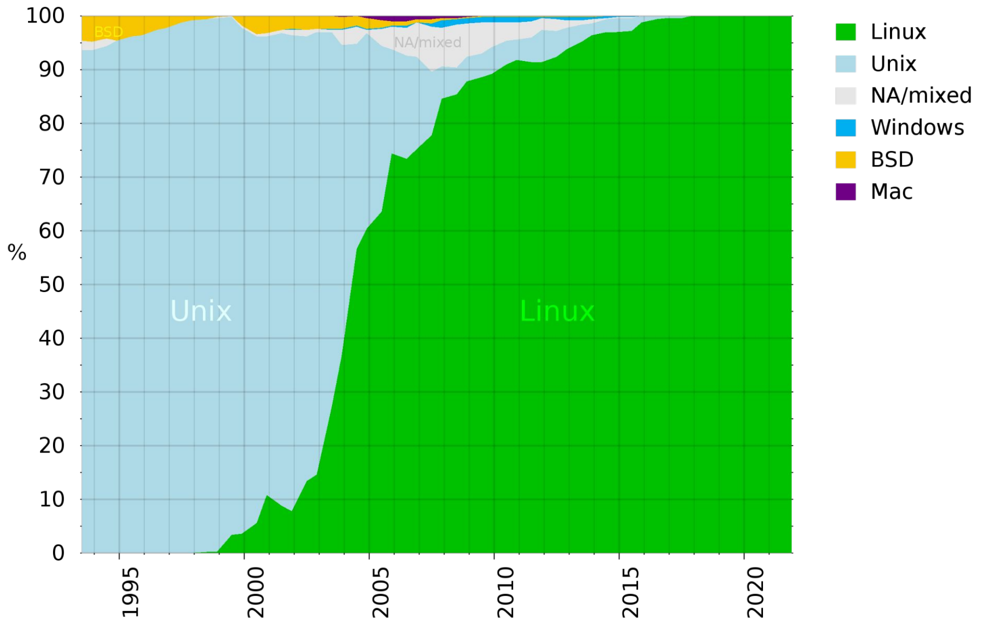
[redacted]

LINUX: THE INTERNET'S OPERATING SYSTEM

As late as 2000, there was significant variety in deployed operating systems.

In 2023, if you're working on a datacenter machine, a cloud VM, a supercomputer, or a network appliance, you're using Linux.

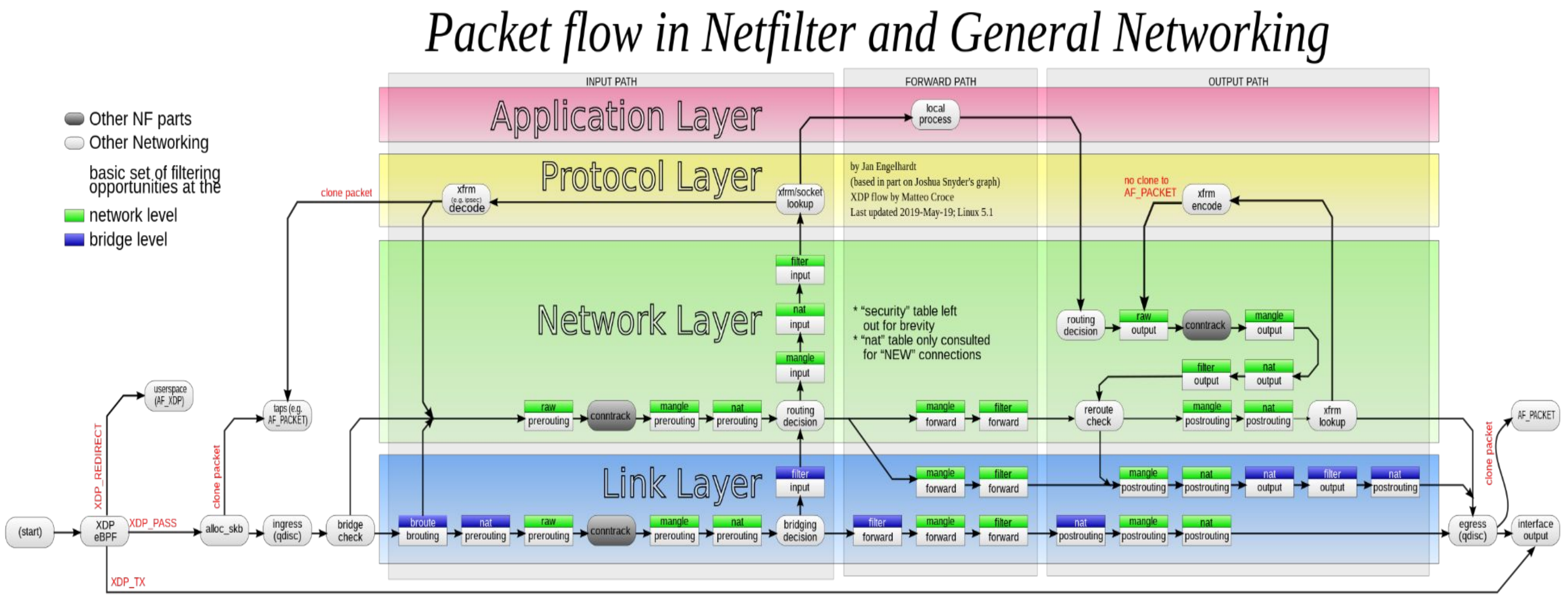
(or some freak RTOS)



LINUX NETWORKING EMBIGGENS

- nftables hooks: packet classification, mangling, or even queuing to userspace
- connection-tracking (exposed to userspace)
- rule-based routing
- bridging with vlan integration and brtables hooks
- unified netlink for neighbors, routing, addresses, devices
- pluggable tcp congestion algorithms and queuing disciplines
- network namespaces and cgroup-based resource allocation
- xfrm for pluggable accelerators
- zerocopy, hugepages, scatter-gather, receive flow steering, gro/lro/tso/gso
- tun/tap, tunneling, packet sockets, and good ol' berkeley sockets

THE DIAGRAM


Packet flow in Netfilter and General Networking
INPUT PATH
FORWARD PATH
OUTPUT PATH
Application Layer
Protocol Layer
Network Layer
Link Layer
Other NF parts
Other Networking
basic set of filtering opportunities at the
network level
bridge level
by Jan Engelhardt
(based in part on Joshua Snyder's graph)
XDP flow by Matteo Croce
Last updated 2019-May-19; Linux 5.1
* "security" table left out for brevity
* "nat" table only consulted for "NEW" connections
local process
xfrm (e.g. ipsec) decode
xfrm/socket lookup
no clone to AF_PACKET
xfrm encode
clone packet
filter input
nat input
mangle input
routing decision
raw output
conntrack
mangle output
filter output
nat output
raw prerouting
mangle prerouting
nat prerouting
mangle forward
filter forward
reroute check
mangle postrouting
nat postrouting
xfrm lookup
AF_PACKET
userspace (AF_XDP)
taps (e.g. AF_PACKET)
XDP_REDIRECT
XDP_PASS
XDP_TX
(start)
XDP eBPF
alloc_skb
ingress (qdisc)
bridge check
broute brouting
nat prerouting
bridging decision
filter forward
nat postrouting
nat output
filter output
egress (qdisc)
interface output

...BUT THE PACKETS COME FASTER AND FASTER

UDPv4 (no VLAN, no IPv4 options) over 10GE:

- 84B frames: 67ns, 18B payload (21.4%), 14.88Mpps
- 1538B frames: 1231ns, 1472B payload (95.7%), 812.7Kpps
- 3110B frames: 2489ns, 3044B payload (97.9%), 401.9Kpps

On a 3GHz core:

- Cycles per packet: 201, 3693, and 7467
- Cycles per 64-byte cache load: ~52
- Cycles per 64-bit word: ~6.5
- Cycles per delivered frame byte: 3.14 max (converges to ~0.8)

"ZERO-COST ABSTRACTIONS"

If there exists an nftables hook, but there are no nftables hooked to it, is it a performance problem?

- nf hooks are registered as callbacks (`nf_register_net_hooks()`):
    - load a pointer (hopefully from cache)
    - push frame
    - indirect branch (hopefully correctly predicted, hopefully to code in cache)
- nf hooks are dynamic:
    - execute memory fence (hopefully fast)
    - atomic read (hopefully from cache)
    - branch (hopefully correctly predicted)
    - pop frame
    - return branch

Call it ten cycles in the best case. That's 5% of our budget for minimum-size frames.

The real cost comes in increased pressure on our cache and TLB: a miss anywhere kills us.

MULTICORE: THE SCHEDULER STRIKES BACK

Multisocket ccNUMA machines can only help so much: I/O devices are local to some package and its memory; other sockets have reduced bandwidth and greater latency to the resources of the zone.

Multicore within the zone is very helpful, subject to some conditions:

- Need scale shared cache and memory with the cores
- Need make certain resources per-core to avoid synchronization
- Need synchronize packet rx/tx if packets mustn't be reordered
- Need fairly distribute load among cores for full utilization

I GET BY WITH A LITTLE HELP FROM DDIO

Hardware does what it can to help us:

- Bigger caches, more memory controllers, more bandwidth to caches and RAM, better microarchitecture, more cores, wider units, higher frequencies, huge pages, DMA, MSI-X, non-temporal loads, prefetching, more cores.
- DCA: snoop PCIe writes to memory and copy them to LLC
- DDIO: PCIe writes directly to LLC

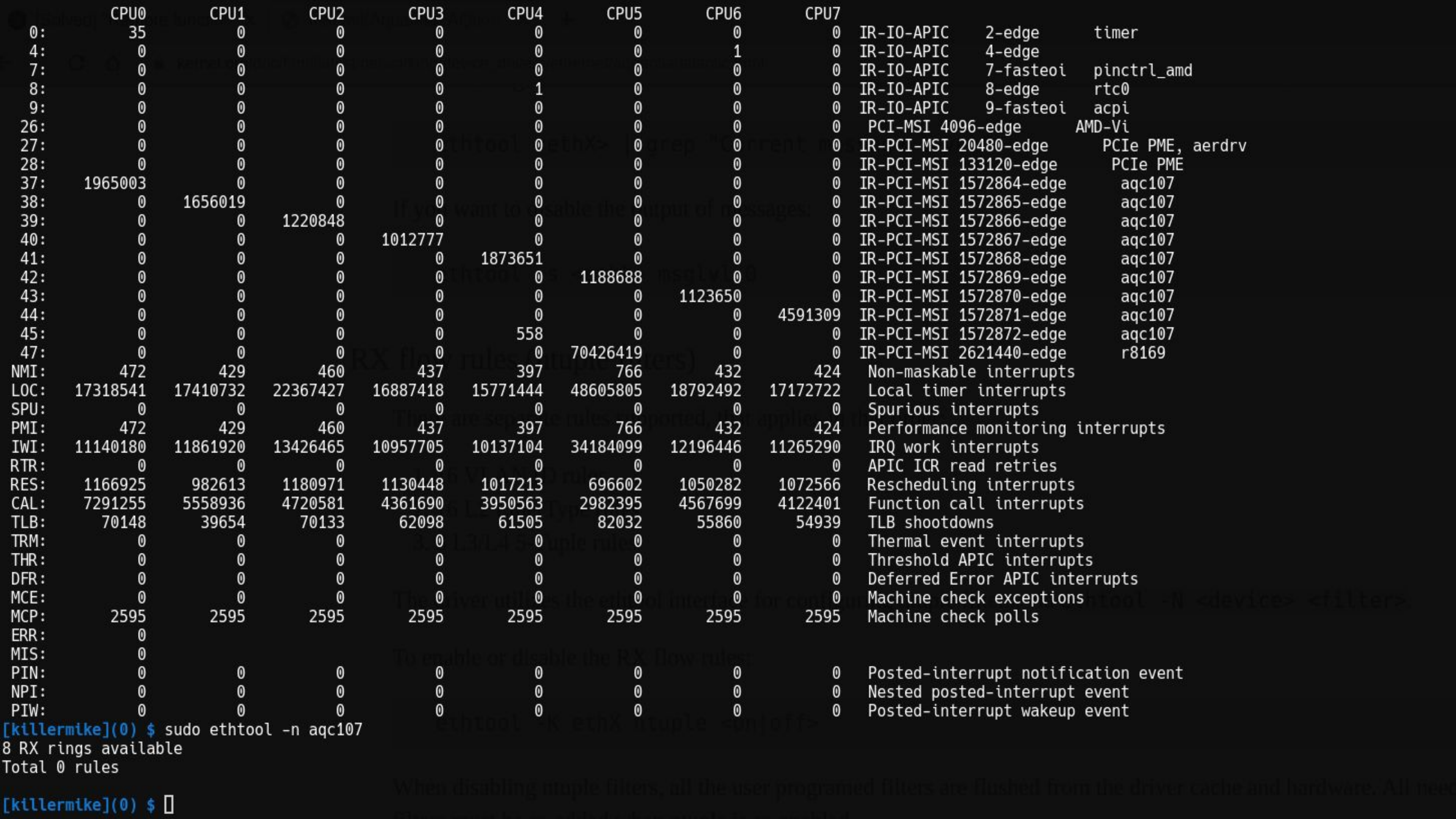
Yet 10Gbps on commodity hardware remained frustratingly out of reach.

PART III

Polly Packet's Adventures in Socketland

FROM THE ADC TO LAYER 3

- Frame streams into NIC buffer, possibly undergoing hardware LRO/GRO
- NIC checks L2 destination address and FCS, discarding unwanted frames
- NIC runs Receiver Side Scaling classification, selects target queue
- NIC DMAs frame into ringbuffer if there is space (and maybe if there isn't)
    - PCIe -> PCIe root complex -> memory (IOMMU possibly involved)
- NIC sends MSI-X in-band interrupt (which cannot pass DMA)
- LAPIC translates MSI-X to core, raises hardware interrupt
- Hard irqh marks dev, raises NET_RX_SOFTIRQ
- Soft irqh (net_rx_action()) cycles through marked devices calling napi_poll()
- napi_poll() calls XDP program if attached, or RPSes into per-cpu backlog



```
            CPU0       CPU1       CPU2       CPU3       CPU4       CPU5       CPU6       CPU7
   0:         35          0          0          0          0          0          0          0  IR-IO-APIC    2-edge      timer
   4:          0          0          0          0          0          0          1          0  IR-IO-APIC    4-edge
   7:          0          0          0          0          0          0          0          0  IR-IO-APIC    7-fasteoi   pinctrl_amd
   8:          0          0          0          0          1          0          0          0  IR-IO-APIC    8-edge      rtc0
   9:          0          0          0          0          0          0          0          0  IR-IO-APIC    9-fasteoi   acpi
  26:          0          0          0          0          0          0          0          0   PCI-MSI 4096-edge      AMD-Vi
  27:          0          0          0          0          0          0          0          0  IR-PCI-MSI 20480-edge      PCIe PME, aerdrv
  28:          0          0          0          0          0          0          0          0  IR-PCI-MSI 133120-edge      PCIe PME
  37:    1965003          0          0          0          0          0          0          0  IR-PCI-MSI 1572864-edge      aqc107
  38:          0    1656019          0          0          0          0          0          0  IR-PCI-MSI 1572865-edge      aqc107
  39:          0          0    1220848          0          0          0          0          0  IR-PCI-MSI 1572866-edge      aqc107
  40:          0          0          0    1012777          0          0          0          0  IR-PCI-MSI 1572867-edge      aqc107
  41:          0          0          0          0    1873651          0          0          0  IR-PCI-MSI 1572868-edge      aqc107
  42:          0          0          0          0          0    1188688          0          0  IR-PCI-MSI 1572869-edge      aqc107
  43:          0          0          0          0          0          0    1123650          0  IR-PCI-MSI 1572870-edge      aqc107
  44:          0          0          0          0          0          0          0    4591309  IR-PCI-MSI 1572871-edge      aqc107
  45:          0          0          0          0        558          0          0          0  IR-PCI-MSI 1572872-edge      aqc107
  47:          0          0          0          0          0   70426419          0          0  IR-PCI-MSI 2621440-edge      r8169
 NMI:        472        429        460        437        397        766        432        424   Non-maskable interrupts
 LOC:   17318541   17410732   22367427   16887418   15771444   48605805   18792492   17172722   Local timer interrupts
 SPU:          0          0          0          0          0          0          0          0   Spurious interrupts
 PMI:        472        429        460        437        397        766        432        424   Performance monitoring interrupts
 IWI:   11140180   11861920   13426465   10957705   10137104   34184099   12196446   11265290   IRQ work interrupts
 RTR:          0          0          0          0          0          0          0          0   APIC ICR read retries
 RES:    1166925     982613    1180971    1130448    1017213     696602    1050282    1072566   Rescheduling interrupts
 CAL:    7291255    5558936    4720581    4361690    3950563    2982395    4567699    4122401   Function call interrupts
 TLB:      70148      39654      70133      62098      61505      82032      55860      54939   TLB shootdowns
 TRM:          0          0          0          0          0          0          0          0   Thermal event interrupts
 THR:          0          0          0          0          0          0          0          0   Threshold APIC interrupts
 DFR:          0          0          0          0          0          0          0          0   Deferred Error APIC interrupts
 MCE:          0          0          0          0          0          0          0          0   Machine check exceptions
 MCP:       2595       2595       2595       2595       2595       2595       2595       2595   Machine check polls
 ERR:          0
 MIS:          0
 PIN:          0          0          0          0          0          0          0          0   Posted-interrupt notification event
 NPI:          0          0          0          0          0          0          0          0   Nested posted-interrupt event
 PIW:          0          0          0          0          0          0          0          0   Posted-interrupt wakeup event
[killermike](0) $ sudo ethtool -n aqc107
8 RX rings available
Total 0 rules

[killermike](0) $ 
```

COMMON RX HARDWARE INTERRUPT HANDLER

```
static inline void ____napi_schedule(struct softnet_data *sd, struct napi_struct *napi) {
      list_add_tail(&napi->poll_list, &sd->poll_list);
      __raise_softirq_irqoff(NET_RX_SOFTIRQ);
}

void __napi_schedule(struct napi_struct *n) {
      unsigned long flags;
      local_irq_save(flags);
      ____napi_schedule(&__get_cpu_var(softnet_data), n);
      local_irq_restore(flags);
}
```

RX SOFTWARE INTERRUPT HANDLER (ABRIDGED)

```
static __latent_entropy void net_rx_action(struct softirq_action *h) {
        for (;;) { // cycle through all devices on poll/repoll lists
                struct napi_struct *n = list_first_entry(&list, struct napi_struct, poll_list);
                skb_defer_free_flush(sd);
                if (list_empty(&list)) {
                        if (!sd_has_rps_ipi_waiting(sd) && list_empty(&repoll)) goto end;
                        break;
                }
                budget -= napi_poll(n, &repoll);
                if (unlikely(budget <= 0 || time_after_eq(jiffies, time_limit))) {
                        sd->time_squeeze++; break;
                }
        }
  }
```

LIFE AS A SKBUFF

`skbuff`: the fundamental packet wrapper of the kernel. From now on, we work with skbuffs, not raw packets. We're still operating in softirq context!

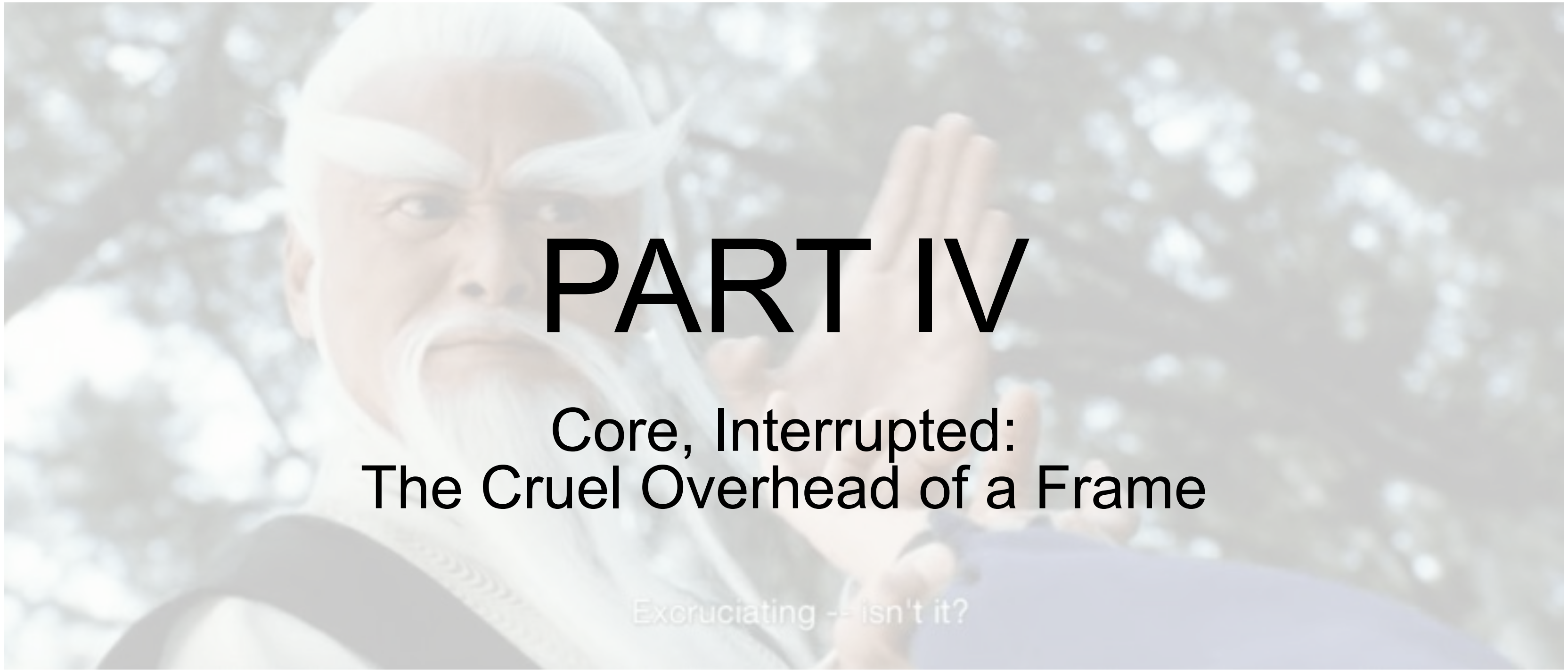
- skb is dequeued from per-cpu list, skb data area from per-cpu arena
- data is copied from DMA ringbuffer into (refcounted) skb data
- vlan, timestamp, pkt_type, protocol, csum skb fields set (eth_type_trans())
- netif_receive_skb() called by process_backlog() or driver NAPI poll()
- invoke generic XDP programs, detag VLAN and dispatch to viface
- check for attached AF_PACKET taps and *clone* skb to each (if room)
- ingress qdisc (default stateless fifo), might drop
- if device is in a bridge, put skb on bridge interface's backlog, otherwise
- deliver to L3 protocol handler via deliver_skb()

BRIDGING, BROUTING, ROUTING

- nftables provide prerouting, input, forward, output, postrouting hooks for inet and bridges (for bridges, read "routing" as "forwarding"), NAT on L2, L3, L4
- nftables furthermore provides ingress and egress hooks on devices
- bridge has a forwarding database, mcast group database, vlan/vni filters
- if destination MAC is local, bridge delivers to device L3 protocol handler
- ip_rcv() / ipv6_rcv() check validity and set up L3 skb elements
- ip_rcv_finish() is called via netfilter PREROUTING hook:
  - `return NF_HOOK(NFPROTO_IPV4, NF_INET_PRE_ROUTING, skb, dev, NULL, ip_rcv_finish);`
- ip_rcv_finish() checks for cached route decision, or performs one:
- ip_route_input_slow() -> fib_lookup() -> fib_table_lookup()
- dispatch to one of ip_local_deliver(), ip_forward(), multicast/tunnel input

FROM THE ROUTING DECISION TO USERSPACE

- ip_local_deliver() reassembles fragments and then...INPUT netfilter hook
    - `return NF_HOOK(NFPROTO_IPV4, NF_INET_LOCAL_IN, skb, skb->dev, NULL, ip_local_deliver_finish);`
- ip_local_deliver_finish() delivers to raw sockets, or
- ip_local_deliver_finish() checks for an xfrm policy, or
- ip_local_deliver_finish() invokes L4 callback, or
- ip_local_deliver_finish() invokes icmp_send() with Dest Unreach (Proto)
- L4 callbacks: tcp_v4_rcv(), udp_rcv()
- tcp_v4_rcv() implements TCP stream reassembly, kicks ARQ machine
- udp_rcv() validates checksum, if used
- find socket, call sock_queue_rcv(), set waiting thread(s) runnable
- continue running softirq handler until done or descheduled



PART IV

Core, Interrupted:
The Cruel Overhead of a Frame

THE COST OF A PACKET

"Interrupt overhead" is a misnomer: Interrupts are fast. What's slow is context switching (low microseconds), rebuilding the TLB, and refilling evicted cache.

Equally slow is the copying of the frame from its device-accessible memory to more general memory as an "skbuff". The Linux networking stack works almost entirely on skbuffs, with substantial overhead. **"Zero-copy" generally means avoiding a copy between kernel- and userspace, *not* this copy.** Allocation of skbuffs is fast: a per-cpu stack.

Packet output, either to a local socket or a device, generally involves another copy, a system call, and a context switch.

THEORETICALLY AVOIDABLE SOURCES OF DELAY/OVERHEAD

- Latency due to DMA into non-local memory
- Scheduler overhead
- Latency waiting for the soft irqh to be scheduled
- Delays warming cache/TLBs for soft irqh
- Read of packet data into cache (eliminated by DCA/DDIO)
- Write of packet data into skbuff data area
- Cloning into taps
- Waiting for RPS-dispatched soft irqhs to be scheduled
- Delays warming cache/TLBs for stack traversals
- Stack traversals, including nftables and queuing disciplines
- Latency due to any non-local reads/writes
- Locking/lookup in socket maps
- Scheduler overhead (reprise)
- Latency waiting for userspace be scheduled
- System call overhead
- Delays warming cache/TLBs for userspace
- Copy of data to userspace

ELIMINATING DELAYS VIA SYSTEM ADMINISTRATION

- Latency due to DMA into non-local memory
- Scheduler overhead + context switch
- Latency waiting for the soft irqh to be scheduled
- Delays warming cache/TLBs for soft irqh
- Read of packet data into cache
- Write of packet data into skbuff data area
- Cloning into taps
- Waiting for RPS-dispatched soft irqhs to be scheduled
- Delays warming cache/TLBs for stack traversals
- Stack traversals, including nftables and queuing disciplines
- Latency due to any non-local reads/writes
- Locking/lookup in socket maps
- Scheduler overhead + context switch (reprise)
- Latency waiting for userspace be scheduled
- System call overhead
- Delays warming cache/TLBs for userspace
- Copy of data to userspace

LOCALITY AND AFFINITY

- Avoid non-local DMA by directing all queues to cores on local package
- Avoid non-local accesses by restricting userspace to the local package AND
- Only allocating for these userspace threads from the local memory

If we can guarantee these three things, we'll never perform a suboptimal access.

The costs are complexity, reduced total cycles, reduced total memory, and reduced total memory bandwidth.

The savings are cycles and bandwidth per unit of load.

Packet arrival time vs (increasing) non-local access time mean *adding computational power can reduce IPC more quickly than we can add cycles.*

ESPECIALLY when one considers that DCA/DDIO only work with the local memory.

ISOLATION AND POPULATION MANAGEMENT

- Keep remaining userspace, soft irqhs off our cores via isolation or cpusets
- Keep unrelated hard irqhs off our cores via irq affinity
- Keep memory traffic off our memory by cpusets + migration

If we can guarantee these three things, we'll never be blocked by unrelated processes, and we'll have full use of our local memory (and its bandwidth).

- Launch only as many threads as we have physical cores on the package
- Place them in an "isolated" cgroup to eliminate scheduler domain

This minimizes useless context switching, and keeps the scheduler out of our face. Of course, other applications can no longer make use of our cores, or our memory, even when they're going unused. They're also far away from a shared network card.

TRADING AWAY FLEXIBILITY FOR PERFORMANCE

Originally, our system dynamically responded to changing loads, keeping active threads apart so that they might make full use of the machine, migrating load freely from where it is high to where it is low. Applications were treated fairly.

We have designated some application as important, breaking that symmetry. The system has minimized overhead due to context switches, cache eviction, and non-local memory access for that application. Other applications are disadvantaged, even when our application is lightly loaded. Minimal changes to the application were required.

We have begun to claw back the ideal network, but we are nowhere near done.

ELIMINATING TLB MISSES WITH HUGE PAGES

4KiB pages cover 64 cache reads, but only two 1500B packets.

327,680 such pages in 10Gib. A typical TLB has between 1Ki and 4Ki entries.

Every memory access we make must go through the TLB for virtual-to-physical translation. A miss means a page table walk with numerous memory accesses.

So long as we're willing to preallocate and lock the memory (making it unavailable to other processes, forcing contiguous physical allocations, suffering initial delay for page table population, and rendering the memory unswappable), we can make use of huge pages. 2MiB pages cover 2^9 4KiB pages. 1GiB pages cover 2^{18}!

PART V

HFT, DPDK, XDP

FURTHER IMPROVEMENTS REQUIRE STACK UNIFICATION

If we want to get much further, it becomes clear that we must eliminate the switch from kernel- to userspace, and eliminate traversal of the kernel's networking stack.

We can do this in two ways: run (almost) everything in kernelspace, or run (almost) everything in userspace.

Complex code is a poor fit for kernelspace, where faults can easily bring down the entire machine (possibly permanently!), standard libraries are unavailable, debugging is difficult, iteration time is high, and one must consider numerous execution contexts and memory access types.

Userspace is the place. Of course, the kernel's primary role is arbitrating userspace access to hardware. Other applications lose access entirely! We furthermore lose kernel functionality: routing, taps, bridging, tunneling, security, traffic shaping...

ELIMINATING DELAYS VIA STACK BYPASS *IN TOTO*

- Latency due to DMA into non-local memory
- Scheduler overhead + context switch
- Latency waiting for the soft irqh to be scheduled
- Delays warming cache/TLBs for soft irqh
- Read of packet data into cache
- Write of packet data into skbuff data area
- Cloning into taps
- Waiting for RPS-dispatched soft irqhs to be scheduled
- Delays warming cache/TLBs for stack traversals
- Stack traversals, including nftables and queuing disciplines
- Latency due to any non-local reads/writes
- Locking/lookup in socket maps
- Scheduler overhead + context switch (reprise)
- Latency waiting for userspace be scheduled
- System call overhead
- Delays warming cache/TLBs for userspace
- Copy of data to userspace

Now we're cooking with thermonuclear fire! This is the approach of DPDK.

DPDK: VAN JACOBSON CHANNELS TAKEN TO THE EXTREME

HFT and HPC programmers had been doing *ad hoc* userspace networking stacks for two decades.

FIXME FIXME FIXME describe accursed DPDK in detail

MAYBE A LITTLE TOO EXTREME TBH

DPDK is a PITA:

- We oftentimes want to keep a lot of that kernel functionality.
- We oftentimes want other applications to be able to use the card.
- We can already minimize the frontend (everything through soft irqh) using the affinity and isolation technique described earlier; NAPI and polling further reduce the per-packet impact.

ELIMINATING DELAYS VIA STACK BYPASS *IN PARALLEL*

- Latency due to DMA into non-local memory
- Scheduler overhead + context switch
- Latency waiting for the soft irqh to be scheduled
- Delays warming cache/TLBs for soft irqh
- Read of packet data into cache
- Write of packet data into skbuff data area
- Cloning into taps
- Waiting for RPS-dispatched soft irqhs to be scheduled
- Delays warming cache/TLBs for stack traversals
- Stack traversals, including nftables and queuing disciplines
- Latency due to any non-local reads/writes
- Locking/lookup in socket maps
- Scheduler overhead + context switch (reprise)
- Latency waiting for userspace be scheduled
- System call overhead
- Delays warming cache/TLBs for userspace
- Copy of data to userspace

This can be accomplished with packet sockets, which can support zero-copy. Interesting, but messy, and not so fast.

BROKE: EXTREME WOKE: eXpress

Some further observations regarding DPDK:

- The TX path is much simpler and faster than RX.
  - What if we just accelerated the RX path, and called it a day?
- We don't want our application to have to deal with unrelated packets.
- Some apps *are* small and simple enough to easily run in the kernel.
- The copy from RX ringbuffer to skbuff dominates the total RX path delay.

If we could just elide this copy (together with our other optimizations), and bypass the stack *selectively*, we'd win back the majority of our performance. We'd only lose kernel functionality for the selected traffic, and lose it only on the RX path.

ELIMINATING DELAYS VIA STACK BYPASS *IN PARTES*

- Latency due to DMA into non-local memory
- Scheduler overhead + context switch
- Latency waiting for the soft irqh to be scheduled
- Delays warming cache/TLBs for soft irqh
- Read of packet data into cache
- Write of packet data into skbuff data area
- Cloning into taps
- Waiting for RPS-dispatched soft irqhs to be scheduled
- Delays warming cache/TLBs for stack traversals
- Stack traversals, including nftables and queuing disciplines
- Latency due to any non-local reads/writes
- Locking/lookup in socket maps
- Scheduler overhead + context switch (reprise)
- Latency waiting for userspace be scheduled
- System call overhead
- Delays warming cache/TLBs for userspace
- Copy of data to userspace

This, combined with eBPF programs run from the soft irqh, is the Linux kernel's AF_XDP socket family.

THE MECHANICS OF XDP

FIXME FIXME FIXME talk about rings and such

XDP vs DPDK: THE STRUGGLE CONTINUES

We initially looked at the tradeoff between functionality and performance.

Here we see that tension once more, in miniature. Both of these methods can be far, far more performant than Berkeley sockets. Both greatly complicate deployment and require per-device support. DPDK appears, on the surface, to have a small advantage. It is more established, and can easily hit its peak.

Any FAANG-approved engineer (or ChatGPT) can chomp some adderall, write a DPDK program, expense an extra NIC, and shuffle packets around at high rates. If you have the source to your vendor's application, you can maybe coax it into integrating with your larger network. Good luck composing it with other code.

XDP is the hackers' response, driven by the community with less commercial fanfare. It coexists with and leverages the rich complexity master programmers have learned to tame, bypassing only what it needs.

An elegant weapon, for a more civilized age.

I'm far too l33t for DPDK, and I hope you are, too.