Network Performance Workshop

Presentors:

Jesper Dangaard Brouer, Red Hat

Hannes Frederic Sowa, Red Hat

Saeed Mahameed, Mellanox

John Fastabend, Intel

Organizer:

Jesper Dangaard Brouer
Principal Engineer, Red Hat

Date: October 2016
Venue: NetDev 1.2, Tokyo, Japan

Introduce purpose and format

- Background for Workshop
  - Status on progress
  - Existing bottlenecks observed in kernel network-stack
  - **Not** about finished / completed work

- Purpose: **discuss**
  - How to address and tackle current bottlenecks
  - Come up with new ideas

Shout-out: Joint work, many contributors!

- Community effort for addressing performance issues
  - **Alexander Duyck**: FIB lookup, page_frag mem alloc, many hotpath fixes
  - **Eric Dumazet**: hotpath guardian, page_frag mem alloc, too much to mention
  - **David Miller**: xmit path rewrite (xmit_more), being most efficient maintainer
  - **Tom Herbert**, **Alexei Starovoitov**, **Brenden Blanco**: Starting XDP
  - **John Fastabend**, **Jamal Hadi Salim**: qdisc layer work
  - Mellanox: Being first XDP driver
    - **Tariq Toukan**, **Saeed Mahameed**, **Rana Shahout**
  - **Florian Westphal**: Netfilter optimizations
  - **Hannes Sowa**, **Paolo Abeni**: Threaded NAPI, softirq, sockets
  - reviewers on netdev@vger.kernel.org

Status: Linux perf improvements summary

- Linux performance, recent improvements
  - approx past 2 years:
- Lowest TX layer (single core, pktgen):
  - Started at: 4 Mpps → 14.8 Mpps (← max 10G wirespeed)
- Lowest RX layer (single core, RX+drop):
  - Started at: 6.4 Mpps → 16 Mpps
  - XDP: drop 20Mpps (looks like HW limit)
- IPv4-forwarding
  - Single core: 1 Mpps → 2 Mpps → (experiment) 2.5Mpps
  - Multi core : 6 Mpps → 12 Mpps (RHEL7.2 benchmark)
  - XDP single core TX-bounce fwd: 10Mpps

Areas of effort

- Focus have been affected by DPDK's focus
- Been focused on the lowest layers
  - TX bottleneck solved
  - RX bottleneck work-in-progress
  - Memory allocator bottleneck
- Why the focus on IPv4 forwarding?
  - On purpose: Been ignoring bottlenecks in socket layer
    - Socket layer do need lot of work!!!
      - As Eric Dumazet constantly point out, (as-always) he it right!
      - Paolo Abeni (Red Hat) looking into socket layer

Longer term target: NetChannels

- Work towards Van Jacobson's NetChannels
- Today: RSS spread flows across RX queues
  - Issue: allows multiplexing into same application
    - Cause need for heavy **locking** (socket queue)
- NetChannels: channel isolation from NIC to application
  - Via lock-free SPSC queue (Single Producer Single Consumer)
- Need to cooperate with NIC HW filters
  - Currently: No uniform way to express HW filters
    - Manual ethtool filters, highly depend on HW support
  - New (common) driver filter API is needed

Basics of NetChannels

- Builds on isolation
  - Single producer and single consumer (SPSC) scheme
    - Implying lock free queue, only a (store) memory barrier
- Current kernel approach
  - *Try* to align, keeping RXq and app “aligned”
    - “Best-effort” RXq and App can “move”
      - Thus, need to handle “worst-case”, thus locking
    - Recent softirq livelock bug, bad keeping app in same CPU
  - Automatic RSS is actually problematic
    - Two RXq's deliver packet into same listener socket

“Channelize” sockets

- Listen() or Bind() init setup of HW filter
  - Deliver into single RXq, SPSC with “listener”
    - More RXq’s with dedicated listener’s, w/HW guarantee
- On Accept() register “signature”
  - Gets back “channel” (new SPSC queue)
  - (maybe) register new HW filter
    - To allow processing on other CPU/Rxq
    - Depend on HW filter update speeds
- Tricky part: Usually fork() after accept (hint: O_CLOEXEC)
  - Make sure parent PID close “accept” socket
  - Transition between “listener” and “established” socket

“Channelize” raw sockets

- Like tcpdump / AF_PACKET
  - XDP likely need to own / consume packets
- XDP program: New return value e.g. XDP_DUMP
  - Deliver “raw” pages into queue
    - Need HW filter and XDP running RXq
      - For achieving “Single Producer” advantage
  - V1: Copy packets to userspace
  - V2: For RX zero-copy to userspace
    - Need separate RXq and page_pool memory safety
    - Issue: vma mapping memory to userspace
      - For speed need pre-VMA mapping (of THP to lower TLB)

Topic: RX bottleneck

- Current focus
  - Bottleneck at lowest RX layer of netstack
- Solving the RX bottleneck is multi-fold
  1) Latency hide cache-miss (in eth_type_trans)
  2) RX ring-buffer bulking/stages in drivers,
  3) Use MM-bulk alloc+free API,
  4) Processing stages (icache optimizations),
  5) New memory alloc strategy on RX

Topic: RX-stages

- A kind of RX bulking
  - Focused on optimizing I-cache usage
  - Working on a vector of packets
    - A lowest RX stage in the driver

RX-stages: Missed driver opportunities

- NAPI already allow a level of RX bulking
  - Drivers (usually) get 64 packet budget (by napi_poll)
  - Drivers don't take advantage of bulk opportunity
- Missed RX opportunities:
  - Drivers process RX-ring 1-packet at the time
    - Call full network stack every time
  - Cause:
    - I-cache likely flushed, when returning to driver code
    - Stall on cache-miss reading packet (ethertype)
    - No knowledge about how many "ready" RX packets

RX-stages: Split driver processing into stages

- If RX ring contains multiple "ready" packets
  - Means kernel was too slow (processing incoming packets)
  - Thus, switch into more efficient mode
    - Bulking or packet “vector” mode
- Controversial idea?
  - Stop seeing multiple RX-ring packets as individual
  - See it as a vector of packets
    - Each driver stage applies actions to packet-vector

RX-stages: What are the driver stages?

- Driver RX-stages “for-loops” over array/vector
  1) Build array of “ready” RX descriptors
     - Start prefetch packet-data in to L2-cache
  2) XDP stage1/2, pass packet-page to XDP hook
     - Mark vector with XDP_ACTIONs
  3) XDP stage2/2: Finish/execute XDP actions
     - Packet-pages left after XDP-stage is XDP_PASS
  4) Each packet-page: Alloc SKB + setup/populate SKB
  5) Call network stack for each packet
     - Optimize more later, when netstack API support bulk RX-call

RX-stages: RX bulking to netstack

- More controversial to deliver a "bundle" to netstack
  - (Driver pre-RX loop is contained inside driver)
  - Split of Driver and netstack code, optimize/split I-cache usage
- RFC proposal by Edward Cree
  - Drivers simply queue RX pkts on SKB list (no-prefetch RX loop)
  - Results very good:
    - First step, 10.2% improvement (simply loop in netstack)
    - Full approach, 25.6% improvement (list'ify upto ip_rcv)
      - Interesting, but upstream was not ready for this step
- More opportunities when netstack know bundle size
  - E.g. caching lookups, flush/free when bundle ends

Not the XDP workshop!

- This is not the XDP workshop
  - Separate own workshop at NetDev 1.2
- This workshop is about
  - the Linux kernel network stack performance!
- But cannot talk about performance
  - Without mentioning XDP
  - Next slides, how XDP relates to netstack

Speaking bluntly about XDP

- Basically a driver RX-code-path benchmark tool
  - eBPF, only thing that makes it usable for real use-cases
    - DDoS use-case is very real!
    - Very powerful: programability at this early stage
- XDP focus: solving driver RX bottleneck
  - E.g: Mlx5 driver, RX drop inside driver (single CPU)
    - 6.3Mpps at NetDev 1.1 (Feb 2016)
    - 12.0Mpps Jesper's PoC hacks
    - 16.5Mpps with XDP and changed MM-model (net-next 86994156c73)
      - (no-cache prefetch, more optimizations coming, expect 23Mpps)

XDP is motivation for NIC vendors

- XDP is motivating drivers developers to:
  - Change memory model to writable-pages
  - Fix RX bottleneck in drivers
- Notice: Current XDP features **secret to performance**:
  - They avoid calling memory layer
  - Local driver page recycle tricks
- Upcoming multi-port TX
  - Cannot hide behind local driver recycling
  - Need more generic solution (like page_pool proposal)

Memory vs. Networking

- Network provoke bottlenecks in memory allocators
  - Lots of work needed in MM-area
- Both in
  - kmem_cache (SLAB/SLUB) allocator
    - (bulk API almost done, more users please!)
  - Page allocator
    - Baseline performance too slow (see later graphs)
    - Drivers: page recycle caches have limits
      - Does not address all areas of problem space

MM: Status on kmem_cache bulk

- Discovered IP-forwarding: hitting slowpath
  - in kmem_cache/SLUB allocator
- Solution: Bulk APIs for kmem_cache (SLAB+SLUB)
  - Status: upstream since kernel 4.6
  - Netstack use bulk *free* of **SKB**s in NAPI-context
    - Use bulking opportunity at DMA-TX completion
    - 4-5% performance improvement for IP forwarding
  - Generic kfree_bulk API
- Rejected: Netstack bulk *alloc* of SKBs
  - As number of RX packets were unknown

MM: kmem_cache bulk, more use-cases

- Network stack – more use-cases
  - Need explicit bulk free use from TCP stack
    - NAPI bulk free, not active for TCP (keep ref too long)
  - Use kfree_bulk() for skb→head
    - (when allocated with kmalloc)
  - Use bulk free API for qdisc delayed free
- RCU use-case
  - Use kfree_bulk() API for delayed RCU free
- Other kernel subsystems?

SKB overhead sources

- Sources of overhead for SKBs (`struct sk_buff`)
  - Memory alloc+free
    - Addressed by kmem_cache bulk API
  - Clearing SKB
    - Need to clear 4 cache-lines!
  - Read-only RX pages
    - Cause more expensive construction the SKB

SKB clearing cost is high

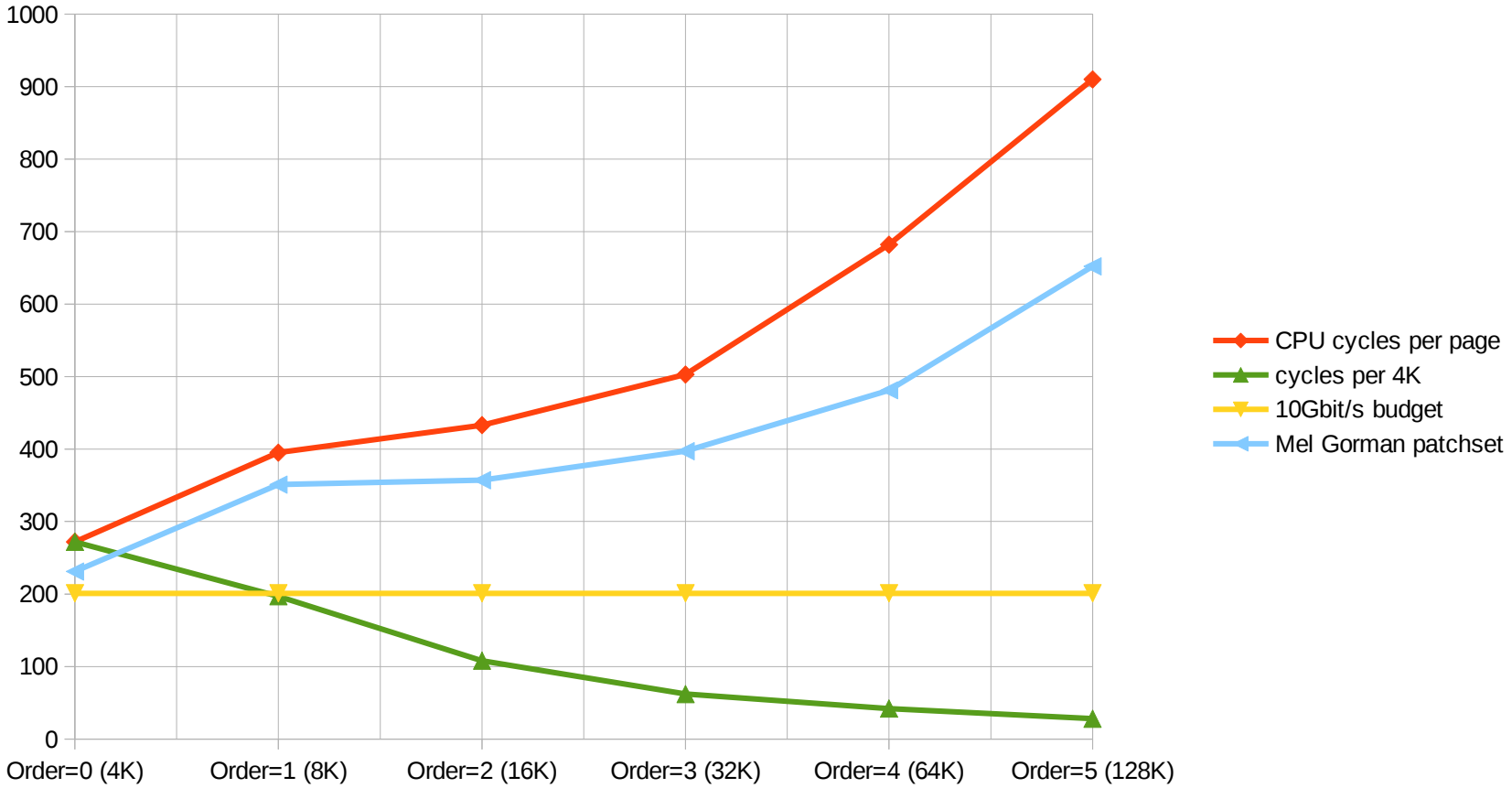
- Options for addressing clearing cost:
  - Smaller/diet SKB (currently 4 cache-lines)
    - Diet too hard!
  - Faster clearing
    - Hand optimized clearing: only save 10 cycles
    - Clear larger contiguous mem (during bulk alloc API)
  - Delay clearing
    - Don't clear on alloc (inside driver)
      - Issue: knowing what fields driver updated
    - Clear sections later, inside netstack RX
      - Mini-SKB overlap struct
    - Allow prefetchw to have effect

SKB allocations: with read-only pages

- Most drivers have read-only RX pages
  - Cause more expensive SKB setup
    1) Alloc separate writable mem area
    2) memcpy over RX packet headers
    3) Store skb_shared_info in writable-area
    4) Setup pointers and offsets, into RX page-"frag"
- Reason: Performance trade off
    A) Page allocator is too slow
    B) DMA-API expensive on some platforms (with IOMMU)
    - Hack: alloc and DMA map larger pages, and “chop-up” page
    - Side-effect: read-only RX page-frames
      - Due to unpredictable DMA unmap time

Benchmark: Page allocator (optimal case, 1 CPU, no congestion)

- Single page (order-0) too slow for 10Gbit/s budget
- Cycles cost increase with page order size
  - But partitioning page into 4K fragments amortize cost

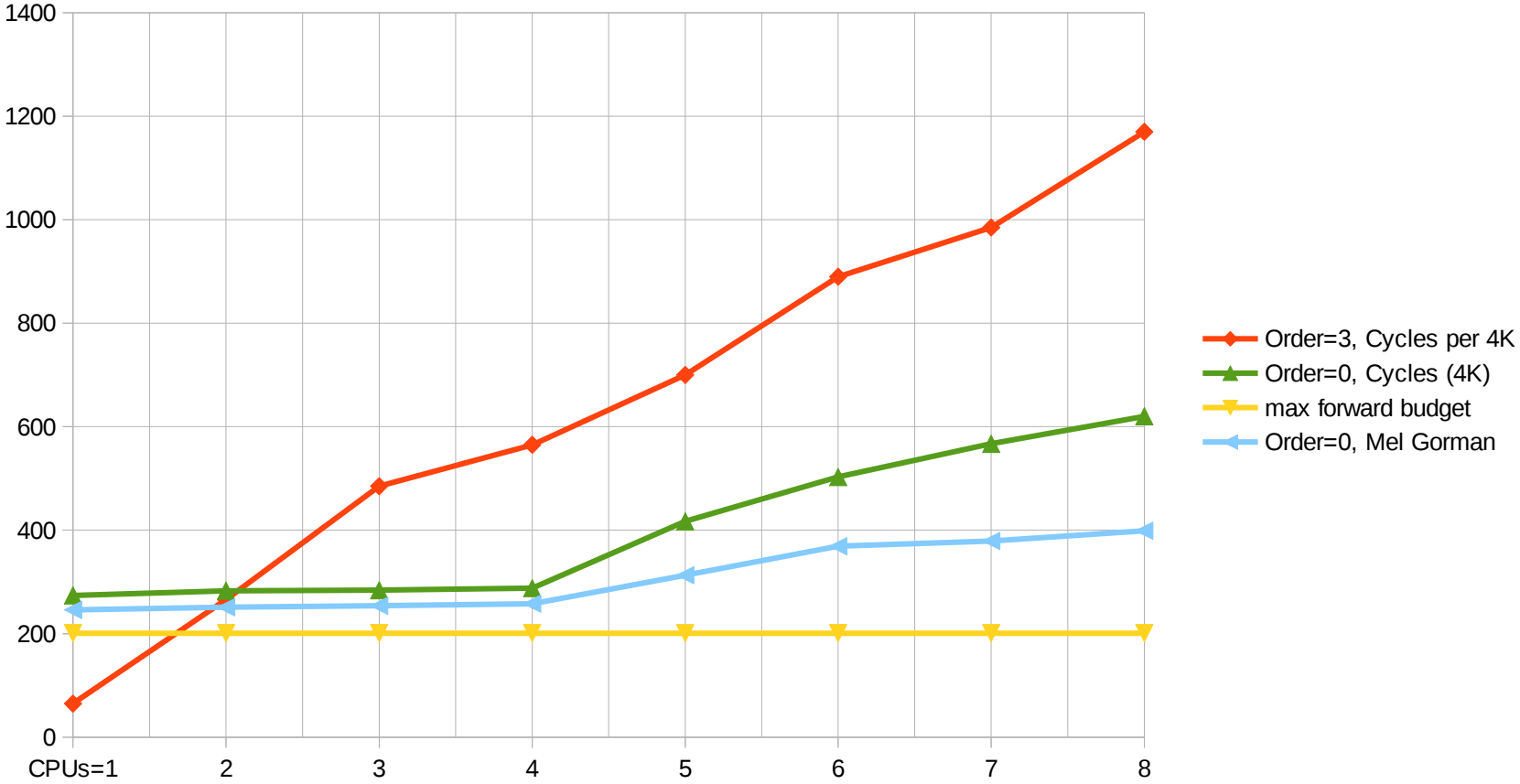


Issues with: Higher order pages

- Performance workaround:
  - Alloc larger order page, handout fragments
    - Amortize alloc cost over several packets
- Troublesome
  - 1. fast sometimes and other times require reclaim/compaction which can stall for prolonged periods of time.
  - 2. clever attacker can pin-down memory
    - Especially relevant for end-host TCP/IP use-case
  - 3. does not scale as well, concurrent workloads

Concurrent CPUs scaling micro-benchmark

- Danger of higher order pages, with parallel workloads
  - Order=0 pages scale well
  - Order=3 pages scale badly, even counting per 4K
    - Already lose advantage with 2 concurrent CPUs



RX-path: Make RX pages writable

- Need to make RX pages writable
- Why is page (considered) read-only?
  - Due to DMA_unmap time
    - Several page fragments (packets) in-flight
    - Last fragment in RX ring queue, call dma_unmap()
    - DMA engine unmap semantics allow overwriting memory
      - (Not a problem on Intel)
- Simple solution: Use one-packet per page
  - And call dma_unmap before using page
- My solution is the page_pool

Page pool: Generic solution, many advantages

- 5 features of a recycling page pool (per device):
  1) Faster than page-allocator speed
     - As a specialized allocator require less checks
  2) DMA IOMMU mapping cost removed
     - Keeping page mapped (credit to Alexei)
  3) Make page writable
     - By predictable DMA unmap point
  4) OOM protection at device level
     - Feedback-loop know #outstanding pages
  5) Zero-copy RX, solving memory early demux
     - Depend on HW filters into RX queues

Page pool: Design

- Idea presented at MM-summit April 2016
- Basic concept for the page_pool
  - Pages are recycled back into originating pool
    - Creates a feedback loop, helps limit pages in pool
  - Drivers still need to handle dma_sync part
  - Page-pool handle dma_map/unmap
    - essentially: constructor and destructor calls
- Page free/return to page-pool, Either:
  1) SKB free knows and call page pool free, **or**
  2) put_page() handle via page flag

Page-pool: opportunity – feedback loop

- Today: Unbounded RX page allocations by drivers
  - Can cause OOM (Out-of-Memory) situations
  - Handled via skb->truesize and queue limits
- Page pool provides a **feedback loop**
  - (Given pages are recycles back to originating pool)
  - Allow bounding pages/memory allowed per RXq
    - Simple solution: configure fixed memory limit
    - Advanced solution, track steady-state
      - Can function as a “Circuit Breaker” (See RFC draft link)

New Topic: TX powers

- I challenge people
  - Solve issue of TX bulking not getting activated

Topic: TX powers – background

- Solved TX bottleneck with xmit_more API
  - See: http://netoptimizer.blogspot.dk/2014/10/unlocked-10gbps-tx-wirespeed-smallest.html
- 10G wirespeed: Pktgen 14.8Mpps single core
  - Spinning same SKB (no mem allocs)
- Primary trick: Bulk packet (descriptors) to HW
  - Delays HW NIC tailptr write
- Interacts with Qdisc bulk dequeue
  - Issue: hard to “activate”

Topic: TX powers – performance gain

- Only artificial benchmarks realize gain
  - like pktgen
- How big is the difference?
  - with pktgen, ixgbe, single core E5-2630 @2.30GHz
  - TX **2.9 Mpps** (clone_skb 0, burst 0) (343 nanosec)

    ↑ Alloc+free SKB+page on for every packet
  - TX **6.6 Mpps** (clone_skb 10000) (151 nanosec)

    ↑ x2 performance: Reuse same SKB 10000 times
  - TX **13.4 Mpps** (pktgen burst 32) (74 nanosec)

    ↑ x2 performance: **Use xmit_more** with 32 packet bursts
  - Faster CPU can reach wirespeed 14.8 Mpps (single core)

Topic: TX powers – Issue

- Only realized for artificial benchmarks, like pktgen
- Issue: For practical use-cases
  - Very hard to "activate" qdisc bulk dequeue
    - Need a queue in qdisc layer
  - Need to hit HW bandwidth limit to “kick-in”
    - Seen TCP hit BW limit, result lower CPU utilization
    - Want to realized gain earlier...

Topic: TX powers – Solutions?

- Solutions for
  - Activating qdisc bulk dequeue / xmit_more
- Idea(1): Change feedback from driver to qdisc/stack
  - If HW have enough pkts in TX ring queue
    - (To keep busy), then qdsic queue instead
  - 1.1 Use BQL numbers, or
  - 1.2 New driver return code
- Idea(2): Allow user-space APIs to bulk send/enqueue
- Idea(3): Connect with RX level SKB bundle abstraction

Topic: TX powers – Experiment BQL push back

- IP-forward performance, single core i7-6700K, mlx5 driver
  - 1.55Mpps (1,554,754 pps) ← much lower than expected
  - Perf report showed: 39.87 % _raw_spin_lock
    - (called by __dev_queue_xmit) => 256.4 ns
    - Something really wrong
      - lock+unlock only cost 6.6ns (26 cycles) on this CPU
      - Clear sign of stalling on TX tailptr write
- Experiment adjust BQL: /sys/class/net/mlx5p1/queues/tx-0/byte_queue_limits/limit_max
    - manually lower until qdisc queue kick in
  - Result: 2.55 Mpps (2,556,346 pps) ← more than expected!
    - +1Mpps and -252 ns

Topic: TC/Qdisc – Background

- Issue: Base overhead too large
  - Qdisc code path takes 6 LOCK operations
    - Even for "direct" xmit case with empty queue
- Measured overhead: between 58ns to 68ns
  - Experiment: 70-82% of cost comes from these locks

Topic: TC/Qdisc – Solutions

- Implement lockless qdisc
  - Still need to support bulk dequeue
  - John Fastabend posted RFC implementation
    - Still locking, but with a skb_array queue
    - Important difference:
      - Separating producer and consumer (locks)
  - Perf improvement numbers?

New Topic: Threaded NAPI

- (by Hannes Sows… own slides?)
- Identified live-lock bug in SoftIRQ processing
  - Fixed by Eric Dumazet

Extra slides

- If unlikely(too_much_time)
  - goto extra_slides;

Topic: RX-MM-allocator – Alternative

- Prerequisite: When page is writable
- Idea: No SKB alloc calls during RX!
  - Don't alloc SKB,
    - Create it inside head or tail-room in data-page
  - skb_shared_info, placed end-of data-page
  - Issues / pitfalls:
    1) Clear SKB section likely expensive
    2) SKB truesize increase(?)
    3) Need full page per packet (ixgbe does page recycle trick)

RX-stages: XDP packet-vector or bulking

- XDP could also benefit from packet-vectors
  - Currently: XDP_TX implicit bulking via tairptr/doorbell
- Some XDP speedup only occur when 100% is XDP
  - Why, implicit get icache benefit, due to small code size
  - Intermixed traffic loose this implicit icache advantage
- General idea: Conceptually build two packet-vector's
  - One related to XDP, one contain XDP_PASS packets
  - XDP_PASS packet-vector proceeds to next RX-stage
  - Either XDP_TX flush before stack or at end of pool loop
    - Moves XDP TX code "outside" critical icache path

Topic: Netfilter Hooks – Background

- Background: Netfilter hook infrastructure
  - iptables uses netfilter hooks (many places in stack)
  - static_key constructs avoid jump/branch, if not used
    - thus, zero cost if not activated
- Issue: Hooks registered on module load time
  - **Empty rulesets still “cost” hook overhead**
  - Every new namespaces inherits the hooks
    - Regardless whether the functionality is needed
  - Loading conntrack is particular expensive
    - Regardless whether any system use it

Topic: Netfilter Hooks – Benchmarks

- Setup, simple IPv4-UDP forward, **no iptables rules!**
  - Single Core, 10G ixgbe, router CPU i7-4790K@4.00GHz
    - Tuned for routing, e.g. ip_early_demux=0, GRO=no
- Step 1: Tune + unload all iptables/netfilter modules
  - 1992996 pps → 502 ns
- Step 2: Load "iptable_**raw**", only 2 hooks "PREROUTING" and "OUTPUT"
  - 1867876 pps → 535 ns → increased cost: +33 ns
- Step 3: Load "iptable_**filter**"
  - 1762888 pps → 566 ns → increased: +64 ns (last +31ns)
- Step 4: Load "nf_**conntrack**_ipv4"
  - 1516940 pps → 659 ns → increased: +157 ns (last +93 ns)

Topic: Netfilter Hooks – Solutions

- Idea: don't activate hooks for empty chains/tables
  - Pitfalls: base counters in empty hook-chains
- Patches posted to address for xtables + conntrack
  - iptables: delay hook register until first ipt set/getsockopt is done
  - conntrack: add explicit dependency on conntrack in modules
    - nf_conntrack_get(struct net*) /_put() needed
- Issue: acceptable way to break backward compat?
  - E.g. drop base counter, if ruleset empty?

Topic: Netfilter Hooks – data structs

- Idea: split structs
  - Into (1) config struct
    - what you hand to netfilter to register your hook
  - and into (2) run time struct
    - what we actually need in packet hot path

- Memory waste in: "`struct net`"
  - 13 families, 8 hooks, 2 pointers per hook -> 1.6k memory per namespace.
    - Conversion to single linked list, save 800 bytes per netns
    - Aaron Conole posted patches!