Network Performance BoF

BoF organizer:

Jesper Dangaard Brouer
Principal Engineer, Red Hat

Date: 10th to 12th February 2016
Venue: NetDev 1.1, Seville, Spain

Presentors:

Jesper Dangaard Brouer, Red Hat

Florian Westphal, Red Hat

Felix Fietkau, OpenWRT

Gilberto Bertin, CloudFlare

John Fastabend, Intel

Jamal Hadi Salim, Mojatatu

Hannes Frederic Sowa, Red Hat

Introduce BoF purpose and format

- Background for BoF
  - Existing bottlenecks observed in kernel network-stack
  - **Not** about finished / completed work
- The presentation format
  - Each topic 2-5 slides
- Purpose: **discuss**
  - How to address and tackle current bottlenecks
  - Come up with new ideas

Overview: Topics and presenters

- Topic: Cloudflare (Gilberto)
- Topic: RX bottleneck (Jesper)
- Topic: TX powers (Jesper)
- Topic: Small devices (Felix)
- Topic: Netfilter hooks (Florian)
- Topic: icache, stage processing (Jesper)
- Topic: TC/qdisc (Jamal/John)
- Topic: packet-page (Jesper/Hannes)
- Topic: RX-MM-allocator (Jesper)
- Topic: MM-bulk (Jesper)
- Topic: Bind namespace (Hannes)

Topic: Linux at CloudFlare – Background

- CloudFlare hits bottlenecks in Linux
  - Packets floods can really stress our Linux boxes
- Issue: using just the Linux kernel it would be much harder to mitigate all the DDoS traffic we see everyday
  - Even with not-so-big packets floods (2M UDP PPS)
  - Even with Iptables drop rules in the raw table
  - RX queue saturated
    - Traffic sharing that RX queue is dropped… :-(

Topic: Linux at CloudFlare – Solution

- Userspace **offloading** with Netmap or ef_vi
  - Flow Steering to redirect bad traffic to a RX queue
  - The queue is detached from the network stack
  - A userspace program poll()s the queue, inspects the packets and reinjects the good ones
  - It's fast! (And so maybe we can learn something)
    - **Circular buffers**: no need to kmalloc and free sk_buffs
    - **BPF**: no need to fully parse the packet if we are likely going to discard it

Topic: CloudFlare – Idea (Jesper)

- Idea: Use RPS (Recv Packet Steering)
- Evaluate potential: approx 4Mpps at RPS level
  - After mlx5 optimizations (next slides)
    - Measured: 7 Mpps for RPS → remote CPU drop 4Mpps
  - RPS bulk enqueue to backlog
    - Measured (PoC code): 9 Mpps
- Solution: 1 CPU handle RX level
  - Multiple remote CPUs handle filtering (less-than 4Mpps each)
  - RX CPU handle (PoC) 9Mpps
    - Still not handle full 14.8Mpps DoS

Topic: RX bottleneck – measurements

- Is lower RX levels a bottleneck? (test: drop as early as possible)
  - IPv4-Forwarding speed, (*all **single core** tests*)
    - Ixgbe: 2Mpps – Mlx5: 1.6Mpps
  - Early drop in iptables RAW table
    - Ixgbe: 5.8Mpps – Mlx5: 4.5Mpps
  - Drop in driver (call dev_kfree_skb_any, instead of napi_gro_receive)
    - Ixgbe: 9.6 Mpps – Mlx5: 6.3Mpps
- Shows early drop:
  - Not fast-enough for DDoS use-case
  - And still gap to DPDK
  - Need to fix lower RX layers

Topic: RX bottleneck – drop in driver(ixgbe)

- **ixgbe** drop with dev_kfree_skb_any()
  - 9,620,713 pps → 104 ns
- Perf report:
  - 43.19% memcpy (cache-miss, copy headers, to “page_frag”)
  - 20.29% Memory related
  - 14.78% ixgbe_clean_rx_irq (ok: driver routine)
  - 11.78% __build_skb (60% spend on memset 0 skb)
  - 2.02% DMA sync calls
  - 1.83% eth_type_trans (no cache-miss due to memcpy)
- See: later topic: RX-MM-allocator
  - Explains why this happens, and propose:
    - Implementing a new allocator for RX

Topic: RX bottleneck – drop in driver(mlx5)

- **mlx5** drop with dev_kfree_skb_any()
  - 6,253,970 pps → 159.9 ns
- Perf report:
  - 29.85% Memory related (Bad case of MM slow-path)
  - 29.67% eth_type_trans (cache-miss)
  - 16.71% mlx5e_{poll_rx_cq,post_rx_wqes,get_cqe}
  - 9.96% __build_skb (memset 0 skb)
- This driver need: use MM-layer better: Prime candidate for MM-bulk API
- Jesper's experiment: 12,088,767 → 82.7 ns
  1) Avoid cache-miss on eth_type_trans,
  2) and (icache) loop calling napi_consume_skb (replaced: napi_gro_receive())
  3) Use SLUB/SKB bulk alloc+free API (with tuned SLUB)

Topic: RX bottleneck – Solutions?

- Solving the RX bottleneck is multi-fold
  1) Latency hide cache-miss (in eth_type_trans)
  2) RX ring-buffer bulking in drivers,
  3) Use MM-bulk alloc+free API,
  4) icache optimizations (processing stages),
  5) New memory alloc strategy on RX?

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 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: Small devices – Background

- Optimizing too much for high-end Intel CPUs?!
  - Low-end OpenWRT router boxes is large market
  - ARM based Android devices also run our network stack
- Smaller devices characteristics
  - I-cache size comparable to Intel 32KiB,
    - but no smart prefetchers, and slower access
  - D-cache sizes significantly smaller
    - e.g. avoid large prefetch loops
  - Smaller cache-**line** sizes (Typical: 16, 32 or 64 bytes)
    - some of our cacheline optimization might be wrong?

Topic: Small devices – Benchmarks(1)

- Benchmarks on QCA9558 SoC (MIPS 74Kc, 720 MHz)
- 64 KiB icache, 32 KiB dcache, linesize: 32 bytes
- Example: Routing/NAT speed, base: 268 Mbit/s
  - After insmod nf_conntrack_rtcache: 360 Mbit/s
  - After rmmod iptable_mangle: 390 Mbit/s
  - After rmmod iptable_raw: 400 Mbit/s
- Optimization approaches:
  - remove (or conditionally disable) unnecessary hooks
  - eliminate redundant access to kernel or packet data

Topic: Small devices – Benchmarks(2)

```
    10.13%  [ip_tables]     [k] ipt_do_table
     6.21%  [kernel]        [k] __netif_receive_skb_core
     4.19%  [kernel]        [k] __dev_queue_xmit
     3.07%  [kernel]        [k] ag71xx_hard_start_xmit
     2.99%  [nf_conntrack]  [k] nf_conntrack_in
     2.93%  [kernel]        [k] ip_rcv
     2.81%  [kernel]        [k] ag71xx_poll
     2.49%  [kernel]        [k] nf_iterate
     2.02%  [kernel]        [k] eth_type_trans
     1.96%  [kernel]        [k] r4k_dma_cache_inv
     1.95%  [nf_conntrack]  [k] __nf_conntrack_find_get
     1.71%  [nf_conntrack]  [k] tcp_error
     1.66%  [kernel]        [k] inet_proto_csum_replace4
     1.61%  [kernel]        [k] dev_hard_start_xmit
     1.59%  [nf_conntrack]  [k] tcp_packet
     1.45%  perf            [.] _ftext
     1.43%  [xt_tcpudp]     [k] tcp_mt
     1.43%  [kernel]        [k] br_pass_frame_up
     1.42%  [kernel]        [k] ip_forward
     1.41%  [kernel]        [k] __local_bh_enable_ip
```

Iptables related: 22.29%

Topic: Small devices – Out-of-tree hacks

- Lightweight SKB structures
  - Used for forwarding, allocate "meta" bookkeeping SKBs
    - dedicated kmem_cache pool for predictable latency
    - or recycle tricks
- D-cache savings by "dirty pointer" tricks
  - Useful trick for forwarding
    - Avoid invalidate D-cache, entire 1500 bytes Ethernet frame
    - change NIC driver DMA-API calls
    - packet contents are "valid" up until a dirty pointer
    - forwarding don't need to touch most of data section
- (e.g. see https://code.google.com/p/gfiber-gflt100/ meta types nbuff/fkbuff/skbuff)

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

Topic: icache – Background

- Issue: Network stack, poor util of instruction-cache
  - Code path size, a packet travel, larger than icache
  - Every packet travel individually,
    - experiencing same icache misses (as the previous packet)

Topic: icache – Solution

- Idea: process several packets at each “stage”
  - **Step 1:** Driver bundle pkts towards stack
  - RX-poll routine already process many (eg. budget 64)
    - But calls "full" stack for every packet, effect “flushing-icache”
  - View pkts avail in the RX ring, as arrived same time
    - Thus, process them at the same time.
    - This RX bulking, amortize cost in a scalable manor
- Side-effect: Cache-miss latency hiding
  - (next slide)

Topic: cache – eth_type_trans()

- Issue: First cache-miss happen too soon for prefetch
  - In eth_type_trans()
- Use icache RX loop for cache-miss hiding
  - Avoid touching pkt-data page, in RX loop, but prefetch
    - By delay calling eth_type_trans(),
      - Call it just before calling stack (via napi_gro_receive)
  - Then, prefetch have time hide cache-miss on data
- One step further: don't call eth_type_trans
  - Get this info, via HW RX descriptor
    - Or Gerlitz had idea how HW can support this! :-)

Topic: icache – RPS (Recv Packet Steering)

- **Step 2:** Bundle/stage at GRO and RPS layer
- GRO does this already, just get little faster
- Potential for optimizing RPS
  - With packet bundle from driver RX layer
- Issue: RPS takes cross CPU locks per packet
  - Solution: RPS bulk enqueue for remote CPUs
    - Eric Dumazet points out, we already have:
      - RPS and RFS defer sending the IPI (Inter-Processor Interrupt)
      - Thus, cross CPU calls (cost ~133 ns) is already amorized
    - Can still save the per packet cost of locking RPS
      - When enqueuing packets, PoC 7Mpps → 9Mpps

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
    - Locking reduced to: two cmpxchg (enq+deq).
      - What about clear/set_bit operations?
  - TODO: Perf improvement numbers?

Topic: packet-page – Warning crazy idea

- Idea: Pickup packet-page before alloc SKB
  - very early at RX, only “extract” page from RX ring
    - send it on alternative “bypass” path
- Use-cases:
  - Transfer "packet-page" to kernel bypass solutions
    - e.g. hook point for DPDK, netmap and RAW af_packet
  - Outgoing device, just move pkt-page directly to TX ring
  - Guest OS'es, forward/map pkt-page directly
- Filtering: Need HW supported filtering
  - Mark packets by HW in RX descriptor
    - Software filter too slow, will cause cache miss

Topic: packet-page – eval perf gain

- Need to measure perf gain this will give us
- Eval with Mlx5 (100G), crazy tuning, skylake i7-6700K
  - Not easy to disconnect early RX code from SKB alloc
    - Instead use MM-bulk API to lower SKB overhead, +tune SLUB
  - Avoid cache miss on eth_trans_type() + icache RX loop
  - Optimize driver to RX drop frames inside driver (single core)
    - RX driver drop: 12Mpps → 82.7 ns
      - (p.s. started at 6.4Mpps)
  - Subtract, SLUB (7.3 ns) and SKB (22.9 ns) related =>
      - (aside-note: 12ns or 52% of SKB cost is memset(0))
    - 52.5 ns → extrapolate 19 Mpps max performance

Topic: RX-MM-allocator – Background

- Idea: Implementing a new allocator for RX
- Issue: (ixgbe) DMA-**sync** on RX ring pkt-data page
  - Side-effect (of DMA-sync) cannot write into page
    - Faster on some archs (PowerPC)
- Cause overhead, e.g. these allocs and steps:
  - 1) alloc: SKB
  - 2) skb_shared_info, end-of data-page, but cannot write
  - 3) alloc: "page-frag" (page_frag_cache), for skb_shared_info
  - 4) memcpy header, into "page-frag"

Topic: RX-MM-allocator – Alternative

- Instead use DMA-**unmap**:
  - allows writing in pkt data-page
- Idea: No alloc calls during RX!
  - Don't alloc SKB, make head-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)

Topic: MM-bulk – Background

- Reason behind needing MM bulk API
  - Discovered IP-forwarding: hitting slowpath
    - in kmem_cache/SLUB allocator
  - Caused by DMA completion happens "later"
    - Causing more outstanding memory objects that fastpath
- Status: net-stack DMA use-case, soon completed
  - 4-5% performance improvement for IP forwarding
  - SLUB changes stable in kernel 4.4
  - SLAB changes soon accepted in AKPMs tree

Topic: MM-bulk – Issues

- Bulk free, works great for IP-forward + UDP
- Issue: Does not “kick-in” for TCP
  - TCP keeping objects longer than DMA completion
  - How to use this bulk free for TCP?

- Future: Generic kfree_bulk() proposed upstream
  - Use-case for freeing skb→head
    - In case skb_free_head() → kfree()

Status: Linux perf improvements

- 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):
  - Started at: 6.4 Mpps → 12 Mpps (still experimental)
- IPv4-forwarding
  - Single core: 1 Mpps → 2 Mpps → (experiment) 2.5Mpps
  - Multi core : 6 Mpps → 12 Mpps (RHEL7.2 benchmark)