Network stack challenges at increasing speeds

The 100Gbit/s challenge

Jesper Dangaard Brouer
Red Hat inc.

Linux Conf Au, New Zealand, January 2015

Overview

- Intro
  - Understand 100Gbit/s challenge and time budget
  - Measurements: understand the costs in the stack?
- Recent accepted changes
  - TX bulking, xmit_more and qdisc dequeue bulk
- Future work needed
  - RX, qdisc, MM-layer
- Memory allocator limitations
  - Qmempool: Lock-Free bulk alloc and free scheme

Coming soon: 100 Gbit/s

- Increasing network speeds: 10G -> 40G -> 100G
  - challenge the network stack
- Rate increase, time between packets get smaller
  - Frame size 1538 bytes (MTU incl. Ethernet overhead)
    - at **10Gbit/s == 1230.4 ns** between packets (815Kpps)
    - at **40Gbit/s == 307.6 ns** between packets (3.26Mpps)
    - at **100Gbit/s == 123.0 ns** between packets (8.15Mpps)
- Time used in network stack
  - need to be smaller to keep up at these increasing rates

Pour-mans solution to 100Gbit/s

- Don't have 100Gbit/s NICs yet?
  - No problem: use 10Gbit/s NICs with smaller frames
- Smallest frame size 84 bytes (due to Ethernet overhead)
  - at 10Gbit/s == **67.2 ns** between packets (14.88Mpps)
- How much CPU budget is this?
  - Approx **201 CPU cycles** on a 3GHz CPU

Is this possible with hardware?

- Out-of-tree network stack bypass solutions
  - Grown over recent years
    - Like netmap, PF_RING/DNA, DPDK, PacketShader, OpenOnload, RDMA/IBverbs etc.
- Have shown kernel is not using HW optimally
  - On same hardware platform
    - (With artificial network benchmarks)
    - Hardware can forward 10Gbit/s wirespeed smallest packet
    - On a **single CPU !!!**

Single core performance

- Linux kernel have been scaling with number of cores
  - hides regressions for per core efficiency
    - latency sensitive workloads have been affected
- We need to increase/improve efficiency per core
  - IP-forward test, single CPU only 1-2Mpps (1000-500ns)
  - Bypass alternatives handle 14.8Mpps per core (67ns)
    - although this is like comparing apples and bananas

Understand: nanosec time scale

- This time scale is crazy!
  - 67.2ns => 201 cycles (@3GHz)
- Important to understand time scale
  - Relate this to other time measurements
  - Next measurements done on
    - Intel CPU E5-2630
    - Unless explicitly stated otherwise

Time-scale: cache-misses

- A single cache-miss takes: **32 ns**
  - Two misses: 2x32=64ns
  - almost total 67.2 ns budget is gone

- Linux skb (sk_buff) is 4 cache-lines (on 64-bit)
  - writes zeros to these cache-lines, during alloc.
  - usually cache hot, so not full miss

Time-scale: cache-references

- Usually not a full cache-miss
  - memory usually available in L2 or L3 cache
  - SKB usually hot, but likely in L2 or L3 cache.
- CPU E5-xx can map packets directly into L3 cache
  - Intel calls this: Data Direct I/O (DDIO) or DCA
- Measured on E5-2630 (lmbench command "lat_mem_rd 1024 128")
  - L2 access costs **4.3ns**
  - L3 access costs **7.9ns**
  - This is a usable time scale

Time-scale: "LOCK" operation

- Assembler instructions "LOCK" prefix
  - for atomic operations like locks/cmpxchg/atomic_inc
  - some instructions implicit LOCK prefixed, like xchg
- Measured cost
  - atomic "LOCK" operation costs **8.25ns**
- Optimal spinlock usage lock+unlock (same single CPU)
  - Measured spinlock+unlock calls costs **16.1ns**

Time-scale: System call overhead

- Userspace syscall overhead is large
  - (Note measured on E5-2695v2)
  - Default with SELINUX/audit-syscall: 75.34 ns
  - Disabled audit-syscall: **41.85 ns**
- Large chunk of 67.2ns budget
- Some syscalls already exists to amortize cost
  - By sending several packet in a single syscall
    - See: sendmmsg(2) and recvmmsg(2) notice the extra "m"
    - See: sendfile(2) and writev(2)
    - See: mmap(2) tricks and splice(2)

Time-scale: Sync mechanisms

- Knowing the cost of basic sync mechanisms
  - Micro benchmark in tight loop

- Measurements on CPU E5-2695
  - spin_{lock,unlock}: 41 cycles(tsc) 16.091 ns
  - local_BH_{disable,enable}: 18 cycles(tsc) 7.020 ns
  - local_IRQ_{disable,enable}: 7 cycles(tsc) 2.502 ns
  - local_IRQ_{**save,restore}**: 37 cycles(tsc) 14.481 ns

Main tools of the trade

- Out-of-tree network stack bypass solutions
  - Like netmap, PF_RING/DNA, DPDK, PacketShader, OpenOnload, RDMA/Ibverbs, etc.
- How did others manage this in 67.2ns?
  - General tools of the trade is:
    - batching, preallocation, prefetching,
    - staying cpu/numa local, avoid locking,
    - shrink meta data to a minimum, reduce syscalls,
    - faster cache-optimal data structures

Batching is a fundamental tool

- Challenge: Per packet processing cost overhead
  - Use batching/bulking opportunities
    - Where is makes sense
    - Possible at *many different levels*
- Simple example:
  - E.g. working on batch of packets amortize cost
    - Locking per packet, cost 2*8ns=16ns
    - Batch processing while holding lock, amortize cost
    - Batch 16 packets amortized lock cost 1ns

Recent changes

What have been done recently

Unlocked Driver TX potential

- Pktgen 14.8Mpps single core (10G wirespeed)
  - Spinning same SKB (no mem allocs)
- Primary trick: *Bulking packet (descriptors) to HW*
- What is going on:
  - Defer tailptr write, which notifies HW
    - Very expensive write to non-cacheable mem
  - Hard to perf profile
    - Write to device
      - does not showup at MMIO point
      - Next LOCK op is likely “blamed”

API skb->xmit_more

- SKB extended with xmit_more indicator
  - Stack use this to indicate (to driver)
  - another packet will be given immediately
    - After/when ->ndo_start_xmit() returns
- Driver usage
  - Unless TX queue filled
  - Simply add the packet to HW TX ring-queue
  - And defer the expensive indication to the HW

Challenge: Bulking without added latency

- Hard part:
  - **Use bulk API without adding latency**
- Principal: Only bulk when really needed
  - Based on solid indication from stack
- Do NOT speculative delay TX
  - Don't bet on packets arriving shortly
  - Hard to resist...
    - as benchmarking would look good

Use SKB lists for bulking

- Changed: Stack xmit layer
  - Adjusted to work with SKB lists
  - Simply use existing skb->next ptr
- E.g. See dev_hard_start_xmit()
  - skb->next ptr simply used as xmit_more indication
- Lock amortization
  - TXQ lock no-longer per packet cost
  - dev_hard_start_xmit() send entire SKB list
  - while holding TXQ lock (HARD_TX_LOCK)

Existing aggregation in stack GRO/GSO

- Stack already have packet aggregation facilities
  - GRO (Generic Receive Offload)
  - GSO (Generic Segmentation Offload)
  - TSO (TCP Segmentation Offload)
- Allowing bulking of these
  - Introduce no added latency
- Xmit layer adjustments allowed this
  - validate_xmit_skb() handles segmentation if needed

Qdisc layer bulk dequeue

- A queue in a qdisc
  - Very solid opportunity for bulking
    - Already delayed, easy to construct skb-list
- Rare case of reducing latency
  - Decreasing cost of dequeue (locks) and HW TX
    - Before: a per packet cost
    - Now: cost amortized over packets
- Qdisc locking have extra locking cost
  - Due to __QDISC___STATE_RUNNING state
  - Only single CPU run in dequeue (per qdisc)

Qdisc path overhead

- Qdisc code path takes 6 LOCK ops
  - LOCK cost on this arch: approx 8 ns
  - 8 ns * 6 LOCK-ops = 48 ns pure lock overhead
- Measured qdisc overhead: between 58ns to 68ns
    - 58ns: via trafgen –qdisc-path bypass feature
    - 68ns: via ifconfig txlength 0 qdisc NULL hack
  - Thus, using between 70-82% on LOCK ops
- Dequeue side lock cost, now amortized
  - But only in-case of a queue
  - Empty queue, direct_xmit still see this cost
  - Enqueue still per packet locking

Qdisc locking is nasty

- Always **6 LOCK** operations (6 * 8ns = 48ns)
  - **Lock** qdisc(root_lock) (also for direct xmit case)
    - Enqueue + possible Dequeue
      - Enqueue can exit if other CPU is running deq
      - Dequeue takes __QDISC___STATE_RUNNING
  - **Unlock** qdisc(root_lock)
  - **Lock** TXQ
    - Xmit to HW
  - **Unlock** TXQ
  - **Lock** qdisc(root_lock) (can release STATE_RUNNING)
    - Check for more/newly enqueued pkts
      - Softirq reschedule (if quota or need_sched)
  - **Unlock** qdisc(root_lock)

Qdisc TX bulking require BQL

- Only support qdisc bulking for BQL drivers
  - *Implement BQL in your driver now!*
- Needed to avoid overshooting NIC capacity
  - Overshooting cause requeue of packets
- Current qdisc layer requeue cause
  - Head-of-Line blocking
  - Future: better requeue in individual qdiscs?
- Extensive experiments show
  - BQL is very good at limiting requeues

Future work

- What need to be worked on?

- Taking advantage of TX capabilities
  - Limited by
    - RX performance/limitations
    - Userspace syscall overhead
    - FIB route lookup
    - Memory allocator

Future: Lockless qdisc

- Motivation for lockless qdisc (cmpxchg based)
  1) Direct xmit case (qdisc len==0) “fast-path”
     - Still requires taking all 6 locks!
  2) Enqueue cost reduced (qdisc len > 0)
     - from 16ns to 10ns
- Measurement show huge potential for saving
  - (lockless ring queue cmpxchg base implementation)
  - If TCQ_F_CAN_BYPASS saving 58ns
    - Difficult to implement 100% correct
  - Not allowing direct xmit case: saving 48ns

What about RX?

- TX looks good now
  - How do we fix RX?
- Experiments show
  - Forward test, single CPU only 1-2Mpps
  - Highly tuned setup RX max 6.5Mpps (Early drop)
- Alexie started optimizing the RX path
  - from 6.5 Mpps to 9.4 Mpps
    - via build_skb() and skb->data prefetch tuning
    - Early drop, don't show real mem alloc interaction

Memory Allocator limitations

- Artificial RX benchmarking
  - Drop packets early
    - Don't see limitations of mem alloc
- Real network stack usage, hurts allocator

1) RX-poll alloc up-to 64 packets (SKBs)
2) TX put packets into TX ring
3) Wait for TX completion, free up-to 256 SKBs

- IP-forward seems to hit slower-path for SLUB

Micro benchmark: kmem_cache

- Micro benchmarking code execution time
  - kmem_cache with SLUB allocator
- Fast reuse of same element with SLUB allocator
  - Hitting reuse, per CPU lockless fastpath
  - kmem_cache_alloc+kmem_cache_free = 19ns
- Pattern of 256 alloc + 256 free (Based on ixgbe cleanup pattern)
  - Cost increase to: **40ns**

MM: Derived MM-cost via pktgen

- Hack: Implemented SKB recycling in pktgen
  - But touch all usual data+skb areas, incl. zeroing
- Recycling only works for dummy0 device:
  - No recycling: 3,301,677 pkts/sec = 303 ns
  - With recycle: 4,424,828 pkts/sec = 226 ns
- Thus, the derived Memory Manager cost
  - alloc+free overhead is (303 - 226): **77ns**
  - Slower than expected, should have hit slub fast-path
    - SKB->data **page** is likely costing more than SLAB

MM: Memory Manager overhead

- SKB Memory Manager overhead
  - kmem_cache: between 19ns to 40ns
  - pktgen derived: 77ns
  - Larger than our time budget: 67.2ns
- Thus, for our performance needs
  - Either, MM area needs improvements
  - Or need some alternative faster mempool

Qmempool: Faster caching of SKBs

- Implemented qmempool
  - Lock-Free bulk alloc and free scheme
    - Backed by alf_queue
- Practical network measurements show
  - saves 12 ns on "fast-path" drop in iptables "raw" table
  - saves 40 ns with IP-forwarding
    - Forwarding hits slower SLUB use-case

Qmempool: Micro benchmarking

- Micro benchmarked against SLUB
  - Cost of alloc+free (CPU E5-2695)
- Fast-path: reuse-same element in loop
  - kmem_cache(slub): 46 cycles(tsc) 18.599 ns
  - qmempool in softirq: 33 cycles(tsc) 13.287 ns
  - qmempool BH-disable: 47 cycles(tsc) 19.180 ns
- Slower-path: alloc 256-pattern before free:
  - kmem_cache(slub): 100 cycles(tsc) 40.077 ns
  - qmempool BH-disable: 62 cycles(tsc) 24.955 ns

Qmempool what is the secret?

- Why is qmempool so fast?
  - Primarily the bulk support of the Lock-Free queue
  - Sharedq MPMC bulk elems out with a single cmpxchg
    - thus, amortize the per elem cost
- Currently uses per CPU SPSC queue
  - requires no lock/atomic operations
    - could be made faster with a simpler per CPU stack

Alf_queue building block for qmempool

- The ALF (Array based Lock-Free) queue
  - (Basic building for qmempool)
  - Killer feature is bulking
  - Lock-Free ring buffer, but uses cmpxchg ("LOCK" prefixed)
  - Supports Multi/Single-Producer/Consumer combos.
  - Cache-line effect also amortize access cost
    - 8 pointers/elems per cache-line (on 64bit)
  - Pipeline optimized bulk enqueue/dequeue
    - (pipelining currently removed in upstream proposal, due to code size)
- Basically "just" an array of pointer used as a queue
  - with bulk optimized lockless access

Qmempool purpose

- Practical implementation, to find out:
  - if it was possible to be faster than kmem_cache/slub
- Provoke MM-people
  - To come up with something just-as-fast
  - Integrate ideas into MM-layer
  - Perhaps extend MM-layer with bulking
- Next talk by Christoph Lameter on this subject
  - SLUB fastpath improvements
  - and potential booster shots through bulk alloc and free

The End

- Want to discuss MM improvements
  - During Christoph Lameter's talk

- Any input on
  - network related challenges I missed?

Extra

- Extra slides

Extra: Comparing Apples and Bananas?

- Comparing Apples and Bananas?
  - Out-of-tree bypass solution focus/report
    - Layer2 “switch” performance numbers
    - Switching basically only involves:
      - Move page pointer from NIC RX ring to TX ring
  - Linux bridge
    - Involves:
      - Full SKB alloc/free
      - Several look ups
      - Almost as much as L3 forwarding

Using TSQ

- TCP Small Queue (TSQ)
  - Use queue build up in TSQ
    - To send a bulk xmit
      - To take advantage of HW TXQ tail ptr update
    - Should we allow/use
      - Qdisc bulk enqueue
        - Detecting qdisc is empty allowing direct_xmit_bulk?