Challenge 10Gbit/s

Wirespeed smallest frame

Single CPU core

Jesper Dangaard Brouer
Hannes Frederic Sowa
Daniel Borkmann
Florian Westphal

Network-Services-Team, Red Hat inc.

Netfilter Workshop, July 2014

Overview

- Part 1: Intro
  - Understanding 10Gbit/s time budget
- Part 2: Measurements
  - Where is the cost hiding in the Linux stack?
- Part 3: Tools of the trade used by others
  - How did others manage this?
- Part 4: Realistically goals
  - What is realistically doable in our stack?

Part 1: Faster alternatives

- Out-of-tree network stack bypass solutions
  - Grown over recent years
    - Like netmap, PF_RING/DNA, DPDK, PacketShader, OpenOnload etc.
- Have shown kernel is not using HW optimally
  - On same hardware platform
    - They handle 10Gbit/s wirespeed smallest frame
    - On a **single CPU**

Part 1: General purpose network platform

- Linux Kernel is best platform available
  - For general purpose networking
- But needs to rethink its architecture
  - To also better adapt for current/future high-end network performance challenges

Part 1: Understand wirespeed challenge

- First step: Understand engineering challenge
  - Of processing 10Gbit/s wirespeed
    - At the smallest Ethernet frame size
    - On a single CPU

- The peak packet rate is:
  - 14.88 Mpps (million packets per sec)
    - Uni-directional on 10Gbit/s with smallest frame size

Pert 1: What is the smallest Ethernet frame

- Ethernet specific (20 bytes)
  - 12 bytes = inter-frame gap
  - 8 bytes = MAC preamble
- Ethernet frame (64 bytes)
  - 14 bytes = MAC header
  - 46 bytes = Minimum payload size
  - 4 bytes = Ethernet CRC
- Minimum size Ethernet frame is: 84 bytes (20 + 64)

Packet Per Sec (pps) range

- PPS range of 10Gbit/s
  - Peak packet rate: 14,880,952 pps
    - calculated as: (10*10^9) bits/sec / (84 bytes * 8)
  - 1500 MTU packet rate: 812,744 pps
    - calculated as: (10*10^9) bits/sec / (1538 bytes * 8)

Part 1: IMPORTANT: Time budget

- **Important part to wrap-your-head around**
  - 14.88 Mpps time budget
    - For processing a single packet is:
  - **67.2 ns** (*nanoseconds*) (calc as: 1/14880952*10^9 ns)

- This correspond to approx:
  - 201 CPU cycles on a 3GHz CPU
    - This is a very small time/cycles budget!

Part 1: Understand: nanosec time scale

- Next: Important to understand time scale
  - Need to relate this to other time measurements
  - Next measurements done on
    - Intel CPU E5-2630
    - Unless explicitly stated otherwise

Time: 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: 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)
- 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: "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)
  - two LOCK calls costs **16.5ns**

Time: 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)

Part 2: Measurement results

- Where is the cost hiding in the Linux stack?
  - Can point us at hotspots
  - But architectural changes are likely need
    - For larger performance wins
  - Next measurements done on
    - Intel CPU E5-2630
    - Unless explicitly stated otherwise

Part 2: 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 = 22.4ns
- Pattern of 128 alloc + 128 free (Based on ixgbe cleanup pattern)
  - Cost increase to: **40.2ns**
- The SKB data/page also have similar MM-cost
  - Thus, approx **80ns** alloc+free overhead

Part 2: 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**

Part 2: Memory Manager overhead

- SKB Memory Manager overhead
  - kmem_cache: approx 80ns
  - 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

Part 2: Qdisc path overhead

- Optimal "fast-path" qdisc is empty, 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
- Pure qdisc overhead
  - As large as our 67.2 ns time budget

Part 2: HW level TX batching

- Batch packets into ixgbe drivers HW TX ring buffer
  - update TDT (Transmit Descriptor Tail) every N packets
- Pktgen showed significant perf improvement
  - Tuning pktgen to send 7Mpps (single CPU)
  - Update TDT every 32 packets, result **11Mpps**
    - (2x=9Mpps, 4x9.6Mpps, 8x=10.5Mpps, 16x=10.9Mpps)
- Thus, ixgbe hardware level batching worth doing
  - Solution that does not increase latency, hard part

Part 3: main tools of the trade

- Out-of-tree network stack bypass solutions
  - Like netmap, PF_RING/DNA, DPDK, PacketShader, OpenOnload, 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

Part 3: Batching is essential

- Challenge: Per packet processing cost overhead
  - Do massive use of batching/bulking
  - Working on batch of packets amortize cost

- General don't do locking, but easy example:
  - e.g. locking per packet, cost 2*8ns=16ns
    - Batch processing while holding lock, amortize cost
    - Batch 16 packets amortized lock cost 1ns

Part 3: (Almost) no alloc/free memory cost

- Most fundamental difference:
  - **not** trying to save memory
- Preallocate huge amounts of memory
  - In huge-pages to avoid TLB lookups
  - Removes cost of per packet dynamic memory alloc
  - Free is simple, e.g mark “free” and return to mempool
  - Never zero memory, can only contain old pkt data

Part 3: Packet metadata one cache-line

- Shrink "skb" packet metadata structure
  - Only one single cache-line "small"
  - Linux skb is 4 cache-lines (on 64 bit)
- Common case no atomic refcnt
  - DPDK have some support, but default off

Part 3: No syscall overhead

- Direct userspace delivery
  - NIC driver basically in userspace
    - Does expose complexity and driver mem to user
- netmap does have a syscall to shield driver
  - But amortize cost with packet bulking

Part 3: No linked-lists

- Use faster data structures
  - Specifically cache-line optimized data structures
    - For each cache-line fetch, get several elements
  - E.g. avoid using linked-lists
- Ideas from research on cache efficient data structures
  - like: "An efficient unbounded Lock-free queue for Multi-core systems"

Part 3: Example from DPDK

- Efficient data structure for FIFO queues
  - (DPDK's based on FreeBSD's bufring.h)
  - Lockless ring buffer, but uses cmpxchg ("LOCK" prefixed)
  - Supports Multi/Single-Producer/Consumer combos.
  - Cache-line effect amortize access cost
  - Pipeline optimized bulk enqueue/dequeue

- Basically "just" an array of pointer used as a queue
  - with pipeline optimized lockless access

Part 3: Numbers L2-switching

- 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

Part 4: Realistically goals

- What is realistically doable in our stack?

Part 4: batching

- Introduce batching where it makes sense

  Given time scale of performing action within critical region

  - might be natural to process batch of packets

- Getting the API right is difficult
  - Danger of introducing latency
    - when "waiting" for another packet (before updating tail ptr)

Part 4: Batching in qdisc layer

- Packet queueing (should) naturally occur in qdisc layer
  - Playing with bulking, to find right API
- Qdisc layers "own" overhead
  - can be the bottleneck, feeding driver fast-enough
- Testing: difficult to "overload" qdisc layer, single CPU
  - Userspace tools min: syscall+alloc+free overhead
  - Forwarding testing or pktgen into qdisc layer?

Part 4: Faster mem alloc for SKB

- Memory Manager overhead
  - Approx 80ns per SKB+data
  - Can SLUB code be improved?
    - Measure it!
- Play with mempool ideas
  - Practical measurements will show

Part 4: L2 forward plane

- Only realistically to compete
  - at same level: Layer2 forwarding
- Linux problem
  - Alloc full SKB for bridging/switching
- To compete:
  - Needs a L2 forwarding plane/layer
  - Architectural changes likely required

The End

- Open discussion
  - Do we need an architectural change?
  - Where should we start?
- We/Red Hat
  - Will allocate resources/persons to project
  - Open to collaborate