Optimizing network packet processing on modern CPUs: VPP
Who am I?

- Guillaume /gi.jom/
- Software engineer at Cisco Meraki for ~3 years
- Designing and implementing the MX firmware
  - Main focus on data plane & performance
Meraki MX

- Cloud-managed security & routing appliance
  - 2M+ online
- SD-WAN
  - Automated VPN topology
  - Smart uplink usage
- IDS/IPS
- L3/L7 Firewall
- BGP/OSPF support
Routers 101
What’s inside a router?

Traditionally divided in three main components

- **Data plane**
  - Forwards *data* packets

- **Control plane**
  - Routing daemons
  - Configures the dataplane (e.g. installs forwarding tables)

- **Management plane**
  - Handles configuration
  - CLI/SSH/Yang/…
Data plane architectures

- Hardware data planes
  - Dedicated ASICs
  - Maximize performance
  - Energy efficient

Why bother with a software data plane then?

Software data planes

- Use off-the-shelf processors
  - Economy of scale
  - Performance can be good
  - Energy efficiency can follow as well

- Cloud deployments
  - Can’t use ASICs in the cloud

- Deep packet inspection (DPI) is control flow heavy
  - Regular expressions/Reassembly/series of lookups
  - Hard to do in hardware

- Updates are simpler

Examples: Linux, DPDK, VPP, Click…
Example: L2 forwarding in Linux

```c
int netdev_tx_t br_dev_xmit(struct sk_buff *skb, struct net_device *dev) {
    dest = eth_hdr(skb)->h_dest;
    if (is_broadcast_ether_addr(dest)) {
        br_flood(br, skb, BR_PKT_BROADCAST, false, true);
    } else if (is_multicast_ether_addr(dest)) {
        [...]
    } else if ((dst = br_fdb_find_rcu(br, dest, vid)) != NULL) {
        br_forward(dst->dst, skb, false, true);
    } else {
        br_flood(br, skb, BR_PKT_UNICAST, false, true);
    }

out:
    rcu_read_unlock();
    return NETDEV_TX_OK;
}
```

- Series of function calls: run-to-completion
- DPDK/Click is similar (though DPDK offers another programming model)
- Barebones
Example: L2 forwarding in VPP

```c
static_always_inline uword
l2fwd_node_inline (vlib_main_t * vm, vlib_node_runtime_t * node,
    vlib_frame_t * frame, int do_trace) {
    from = vlib_frame_vector_args (frame);
    n_left = frame->n_vectors; /* number of packets to process */
    vlib_get_buffers (vm, from, bufs, n_left);
    while (n_left > 0)
    {
        u32 sw_if_index0;
        ethernet_header_t *h0;
        l2fib_entry_key_t key0;
        l2fib_entry_result_t result0;

        sw_if_index0 = vnet_buffer (b[0])->sw_if_index[VLIB_RX];

        h0 = vlib_buffer_get_current (b[0]);

        /* process 1 pkt */
        l2fib_lookup_1 (msm->mac_table, &cached_key, &cached_result,
            h0->dst_address, vnet_buffer (b[0])->l2.bd_index, &key0,
            /* not used */ &result0);
        l2fwd_process (vm, node, msm, em, b[0], sw_if_index0, &result0, next);
        next += 1; b += 1; n_left -= 1;
    }
    vlib_buffer_enqueue_to_next (vm, node, from, nexts, frame->n_vectors);
    return frame->n_vectors;
}
```

- Series of nodes processing up to 256 buffers
- Explicit batch support in nodes: enables multiple optimizations (prefetch/unrolling/SIMD...)

First steps in VPP
What is VPP?

- Vector packet processing
- Software dataplane for common architectures (x86-64/aarch64)
- Originally developed by Cisco
  - Open-sourced in 2016, now managed by the Linux Foundation (fd.io)
  - Rumor says it was designed to be the slow-path for a hardware dataplane...
- Performance as 1st class citizen
  - Multithreaded scalability
  - Optimized data-structures (vector/bitmap/pool/hash table/prefix tree)
  - Lots of tooling to gain insight on what’s going on during execution
- Used by Intel to showcase performance gains of new architectures
  - Unusual payload as it tends to be backend bound

[https://fd.io/technology/](https://fd.io/technology/)
Example: dummy IP-in-IP tunnel

- Input: L2 frame
- Destination IP lookup to retrieve a tuple of IPs
  - Encapsulate the IP packet inside an IP packet with the retrieved IPs
  - Drop if not present in the table
- Send back to original interface
Example: test setup

VPP

device-input

pg-input  demoperf  interface-output

Internal packet generator  Demo node  Send to interface
Walkthrough

- VPP CLI
- Packet generator config
- show runtime
Optimizing a node
Software data planes (cont.)

- **Performance metrics:**
  - Throughput: usually packets per seconds (pps), or bits per second (bps)
  - Latency (time spent in router)
  - Multithreading scalability

- **Some orders of magnitudes**
  - Current CPUs are running around 2-3GHz (so as much cycles per second)
  - Typical workloads: non-crypto: 10Mpps per core; crypto workloads: 1Mpps per core
  - → CPU budget is roughly 300 cycles per packet for non-crypto, 3000 for crypto
  - Cache latencies (for Intel haswell) L1: 4 cycles, L2: 12 cycles, L3 ~40-50 cycles, RAM 200 cycles
    - Not much room for cache misses!

https://docs.fd.io/csit/master/report/vpp_performance_tests/throughput_speedup_multi_core/ip4-2n-icx-e810cq.html#b-ip4routing-base-scale-avf
VPP routing performance results

https://www.7-cpu.com/cpu/Haswell.html
Intel Haswell latencies
1st step: batching

- VPP’s signature move (it’s in the name !)
Results:

1 buffer:

```
vpp# show runtime
Time .8, 10 sec internal node vector rate 1.00 loops/sec 2119944.14
   vector rates in 2.716e06, out 2.716e06, drop 0.00000e0, punt 0.00000e0
      Name     State   Calls       Vectors     Suspends    Clocks    Vectors/Call
           active  10000000  10000000    0           2.05e2     1.00
```

```
vpp# show perfmon statistics
   instructions/packet, cycles/packet and IPC
   Calls      Packets   Packets/Call   Clocks  Packet Instructions/ Packet   IPC
vpp_main (0)  demoperf  4357275  4357275     1.00        310.20       411.00  1.32
```

252 buffers:

```
vpp# show runtime
Time .8, 10 sec internal node vector rate 252.00 loops/sec 51738.07
   vector rates in 1.3071e7, out 1.3071e7, drop 0.00000e0, punt 0.00000e0
      Name     State   Calls       Vectors     Suspends    Clocks    Vectors/Call
           active  44875      11160000     0           7.15e1   252.00
```

```
vpp# show perfmon statistics
   instructions/packet, cycles/packet and IPC
   Calls      Packets   Packets/Call   Clocks  Packet Instructions/ Packet   IPC
vpp_main (0)  demoperf  194750  49077000    252.00       71.39     108.39  1.52
```
1st step: batching

- VPP’s signature move (it’s in the name !)
  - Increased IPCs
  - Instruction cache is hot
  - Hardware prefetcher can already start kicking in
Processor pipeline

Hot L1 Instruction Cache + Hardware prefetcher + ⇒ Less memory stalls and easier for the branch predictor ⇒ More instructions in parallel
1 step further

- What’s the cache behaviour like?
  - Perfmon (uses CPU performance counters – PMU)

Count:
- Cycles
- Cache hits/misses
- Instruction issued/retired
- Branch mispredictions
- ...

Figure 2-3. The Intel® Core® microarchitecture Pipeline Functionality
1 step further

- What’s the cache behaviour like?
  - 2 L1 misses/pkt
  - Non negligible amount of L2 misses per packet

```
vpp# show perfmon statistics

   cache hits and misses
                     L1 hit/pkt L1 miss/pkt L2 hit/pkt L2 miss/pkt L3 hit/pkt L3 miss/pkt
vpp_main (0)         demoperf 17.08   1.98   1.51  .47   .47   0.00
```

Quiz

- What are the memory accesses that trigger this?
Proper methodology

- Use a profiler to detect hot paths
Proper methodology

- Use a profiler to detect hot paths
  - Vlib_buffer metadata, data and (data - 20)
Fix: help the prefetcher

- Hardware instructions to trigger prefetch
  - _mm_prefetch
- Prefetch the memory locations one buffer before they are used
Results

- Much better!

```
vpp# show runtime
Time 7.1, 10 sec internal node vector rate 252.00 loops/sec 59835.79
  vector rates in 1.4849e7, out 1.4849e7, drop 0.0000e0, punt 0.0000e0

   Name   State  Calls  Vectors  Suspends  Clocks  Vectors/Call
   demoperf active  418200  105386400   0   4.19e1   252.00
```

```
vpp# show perfmon statistics

   cache hits and misses
   L1 hit/pkt L1 miss/pkt L2 hit/pkt L2 miss/pkt L3 hit/pkt L3 miss/pkt

vpp_main (0)
   demoperf          29.08   .11   .09   .02   .02   0.00
```
Wait a minute...

● Only one address was looked up. What happens if there’s much more?
  ○ 10k addresses with ~uniform distribution
  ○ ~half of the addresses are on the table
Result

- Cache misses for bihash lookup
  - Let’s prefetch!
- ⇒ Important to benchmark with expected workloads
Fix: unroll + help the prefetcher

- Use loop iterations to give time to prefetch
- Opens opportunity for SIMD
Result

- Reduce L2 misses

```
vpp# show runtime
Time 1.5, 10 sec internal node vector rate 150.97 loops/sec 56040.99
  vector rates in 1.4897e7, out 7.8156e5, drop 7.0813e6, punt 0.0000e0
  Name  State  Calls  Vectors  Suspend  Clocks  Vectors/Call
demoperf  active  8200  28664000  0  5.53e1  252.00
demoperf  active  2000  10209147  0  1.42e1  196.50
```

```
vpp# show perfmon statistics
  cache hits and misses
  L1 hit/pkt L1 miss/pkt L2 hit/pkt L2 miss/pkt L3 hit/pkt L3 miss/pkt
  vpp_main (0)
    demoperf  28.08  .99  .93  .06  .06  0.00
    demoperf  46.06  .79  .51  .00  .00  0.00
```
Result

- Also lots of branch misprediction!

```
vpp# show perfmon statistics

Branches, branches taken and mis-predictions
Branches/call Branches/pkt Taken/call Taken/pkt % MisPred

vpp_main (0)

demoperf   2054.36   8.15   1055.52   4.19   06.76
pe-input   8343.94  33.11  3813.38  15.12  03.15
```
Fix: branchless code

- Unpredictable branches trigger lots of rollbacks: important to avoid those
- In our case, lookup fails half of the time: impossible to predict
- Can remove branches with conditional moves:
  - Ternary operator usually gets compiled to those\[\text{int } \text{res} = \text{cond } ? \text{a} : \text{b};\]
  - Avoids rollback (but can stall it if data is far)

⇒ Can use conditional moves to split the buffers between those who have an entry and those who don’t!
Additional topics

● Multi-threading:
  ● Avoid contention ⇒ per thread data when possible/fine-grained locking if not
    ○ Counters
    ○ Bihash

● Vector instructions (SIMD)
  ○ Single instruction can perform multiple operations in a reduced number of cycles
  ○ 128/256/512bits
  ○ Accelerated crypto: block size (AES-NI/Armv8 crypto extension)
Recap/Lessons
Common processor pipeline bottlenecks

Typical causes for stall/inefficiency for VPP
- Bad speculation
- Memory latency

Figure 2-3. The Intel® Core™ Microarchitecture Pipeline Functionality
Disclaimer

*Premature optimization is the root of evil*

Optimizing is pretty cool, but it’s easy to do it wrong:

- Optimize cold paths → limited gains
- Not benchmarking → No idea if you are actually optimizing
- Pipeline optimization is not everything: e.g. choosing the right data structures
- …
Resources

https://www.brendangregg.com/perf.html


Questions ?