Performance of VPP Linux Control Plane

Pim van Pelt <pim@ipng.ch> • 2022-02-06 • FOSDEM ‘22
Introduction

Pim van Pelt (PBVP1-RIPE)

- Member of the RIPE community since 1999 (RIPE #34)
  - Has used pim@ipng.nl for 22 years
  - And also pim@ipng.ch for 15 years
  - Incorporated ipng.ch in Switzerland in 2021
Introduction

IPng Networks GmbH

- Developer of Software Routers - VPP and DPDK [ref]
- Tiny operator from Brüttisellen (ZH), Switzerland [ref]
- Twelve VPP/Bird2 routers [ref] (UN/LOCODE names)
- European ring: peering on the FLAP* [ref] ~1850 adjacencies
- Acquired AS8298 from SixXS [ref]
Scalar packet processing

A typical ingress Ethernet Rx (or batch of them):

1. Causes an interrupt: kernel stops what it was doing
2. Looks up ethertype; was this ARP, IPv4, IPv6, MPLS, etc?
3. Was it destined for me? If so, handoff to UDP/TCP/ICMP stack
4. Should I forward? If so, look up the L3 nexthop
5. Should I route the packet? If so, decr TTL, NAT, calc checksum
6. Which interface? Look up the L2 nexthop
7. Rewrite the L2 header (src is my MAC, dst is L2 nexthop)
8. Enqueue the Ethernet egress transmission
9. Kernel resumes what it was doing

Packets go through this one-by-one and your CPU cache hates you.
DPDK architecture

- Runs in Linux userspace (either host or VM guest)
- Kernel bypass (eg. SR-IOV, UIO, VFIO) for device access
- Fully consumes CPU (one pthread per logical core)
  - Implements various Poll Mode Drivers (PMD)
  - PMDs offer hardware offload capabilities
  - Lockless queues, buffers, hash tables, timers, mempools
  - Run-To-Completion model: Rx ⇒ process ⇒ Tx
- DPDK threads (lcores) subscribe to queue(s) of port(s)
  - Receive Side Scaling (RSS): n threads on one NIC
  - Hashing on IPv4, IPv6, MPLS, VXLAN, I2-payload, ...
  - Tx Queues: Each thread has an output to each NIC
int main (int argc, char **argv) {
    rte_eal_init(argc, argv);
    rte_eal_mp_remote_launch(lcore_main, NULL, CALL_MASTER);
    RTE_LCORE_FOREACH_SLAVE(lcore_id)
        if (rte_eal_wait_lcore(lcore_id) < 0) return -1;
    return 0;
}

struct rte_mbuf *pkts[32]; int port = 0, queue = 0;
void lcore_main(void) {
    while (running) {
        n = rte_eth_rx_burst(port, queue, pkts, 32);
        process_pkts(pkts, n);
        rte_eth_tx_burst(port, queue, pkts, n);
    }
}
Good to know: Rules of Thumb

0.5 ns - CPU L1 CACHE reference
1 ns - A photon traveling ~300mm distance
3.5 ns - CPU L2 CACHE reference
10 ns - CPU L3 CACHE reference
70 ns - DDR MEMORY reference
135 ns - CPU cross-QPI/NUMA best case on XEON E7-*
202 ns - CPU cross-QPI/NUMA worst case on XEON E7-*

Takeaway: RAM is hideously slow
DPDK reads a vector of up to 256 packets from its interfaces:

1. Packets are prefetched (or directly written) into CPU d-cache
2. All packets go through a directed graph
   a. First packet: graph node’s code loaded into CPU i-cache
   b. All additional packets: fully in d/i-cache: 7-20x faster
3. Packets then traverse as a vector into the next node(s)
   a. Optimized with SIMD (SSE, AVX, AVX512, ...)
   b. No context switches, good TLB hit rate due to hugepages
   c. Lockless: multi-threading gives linear scaling
4. Hardware offload: use silicon if available
5. Plugins: rearrange the graph nodes and add functionality
Vector Packet Processing - example

```bash
pim@hippo:~$ vppctl
vpp# set interface state TenGigabitEthernet3/0/0 up
vpp# set interface mtu packet 9000 TenGigabitEthernet3/0/0
vpp# set interface ip address TenGigabitEthernet3/0/0 2001:db8:0:1::2/64
vpp# set interface ip address TenGigabitEthernet3/0/0 192.0.2.2/24
vpp# ip route add 2000::/3 via 2001:db8:0:1::1
vpp# ip route add 0.0.0.0/0 via 192.0.2.1
pim@hippo:~$ vppctl show interface TenGigabitEthernet3/0/0
TenGigabitEthernet3/0/0  up  9000/0/0/0     rx packets            5969930253
rx bytes           2139798549228
tx packets           14517083897
tx bytes           6864831067486
drops                        945
ip4                   3862409855
ip6                   2107502378
```

Always hungry for packets…
Wrote a VPP Plugin [github] that:

1. Creates tun/tap interface in Linux for a given VPP interface
   a. Linux->VPP: packets into TAP are inserted into `virtio-input`
   b. VPP->Linux: Traffic to `ip4-local` and `ip6-local` is punted into TAP
2. Syncs interface changes in VPP into Linux
3. Listens to `Netlink messages` and syncs Linux changes into VPP
4. Allows operators to use VPP almost exactly as if it were Linux
   - Configure interfaces, addresses, routes by hand, or ...
   - ...using common tools like `ip(1)`, FRR, or Bird/Bird2

⇒ VPP is Linux’s software equivalent of an ASIC dataplane ⇐
VPP: Linux Controlplane

For the curious ...

**Part 1** - Create sub-interface (.1q, .1ad, q-in-q, q-in-ad) in Linux

**Part 2** - Sync link state, MTU and IP addresses in Linux

**Part 3** - Automatically create sub-interfaces in Linux

**Part 4** - Netlink: Sync link state, MTU, neighbor, IP addresses, create new sub-interface (.1q, .1ad, q-in-*) in VPP

**Part 5** - Netlink: Sync routes in VPP

**Part 6** - Expose interface stats from VPP in SNMP

**Part 7** - HOWTO: Installation and Configuration in Production

*) Thanks to Neale Ranns, Matt Smith and Jon Loeliger for the collaboration
VPP: Linux Controlplane - ip

```bash
pim@hippo:~$ vppctl lcp create TenGigabitEthernet3/0/0 host-if xe0

pim@hippo:~$ sudo ip link set xe0 up mtu 9000
pim@hippo:~$ sudo ip address add 2001:db8:0:1::2/64 dev xe0
pim@hippo:~$ sudo ip address add 192.0.2.2/24 dev xe0

pim@hippo:~$ sudo ip link add link xe0 name servers type vlan id 101
pim@hippo:~$ sudo ip link set servers mtu 1500 up
pim@hippo:~$ sudo ip addr add 2001:678:d78:3::86/64 dev servers
pim@hippo:~$ sudo ip addr add 194.1.163.86/27 dev servers
pim@hippo:~$ sudo ip route add default via 2001:678:d78:3::1
pim@hippo:~$ sudo ip route add default via 194.1.163.65
pim@hippo:~$ ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: icmp_seq=0 ttl=121 time=1.348 ms
```

ping 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: icmp_seq=0 ttl=121 time=1.348 ms
...
VPP: Linux Controlplane - Bird2

```
pim@frggh0:~$ birdc show route count
BIRD 2.0.7 ready.
5935108 of 5935108 routes for 867667 networks in table master4
994480 of 994480 routes for 142326 networks in table master6
245091 of 245091 routes for 245091 networks in table t_roa4
48925 of 48925 routes for 48925 networks in table t_roa6
Total: 7223604 of 7223604 routes for 1304009 networks in 4 tables
```

```
pim@frggh0:~$ birdc show ospf neighbor ospf6
BIRD 2.0.7 ready.
Router ID      Pri        State   DTime   Interface  Router IP
194.1.163.33     1   Full/PtP     31.868  xe1-2.100  fe80::6a05:caff:fe32:3e38
194.1.163.32     1   Full/PtP     38.641  xe1-3.200  fe80::6a05:caff:fe32:3cdb
194.1.163.140    1   Full/DR      37.944  xe1-1.2006 fe80::5054:ff:feb0:442c
Converges full BGP table in 9s
```
VPP: SNMP and NMS

1. Wrote an SNMP Agent [github]
2. Added logo to LibreNMS [ref]
3. Added distro to LibreNMS Agent [ref]
There is another ...

... super-useful DPDK app

Cisco T-Rex traffic load tester [link]

● Stateless/Stateful load testing
● Python API - fully programmable
  ○ Traffic streams w/ scapy [api]
  ○ Traffic ramp up/down [api]
  ○ Runtime statistics [api]
  ○ Interactive / CLI [docs]
● DPDK Integration
  ○ ~10-15Mpps per core
  ○ Linear scaling with cores
  ○ 1G/10G/25G/40G/100G
Simple configuration:
- version: 2
  interfaces: ['5:00.0', '5:00.1']
  port_info:
    - src_mac: [0x0A,0x0B,0x0C,0x01,0x02,0xAA] # Mac A
    - dest_mac: [0x0A,0x0B,0x0C,0x01,0x02,0xBB] # Mac B
    - src_mac: [0x0A,0x0B,0x0C,0x01,0x02,0xBB] # Mac B
    - dest_mac: [0x0A,0x0B,0x0C,0x01,0x02,0xAA] # Mac A

Startup:
$ sudo ./t-rex-64 -i -c 6
$ ./trex-console
Stateless Traffic Profiles

Assemble packet streams with scapy:

- IPv4/IPv6 src/dst; proto; port src/dst; size; ratios; timings

```python
self.ip_range = {'src': {'start': "16.0.0.1", 'end': "16.0.0.254"},
                 'dst': {'start': "48.0.0.1", 'end': "48.0.0.254"}}
```

- Streams are applied on one or more ports
- Ports are configured to send a rate of traffic (bps, pps or % of line)

```python
# default IMIX properties
self.imix_table = [ {'size': 60,  'pps': 28,  'isg':0 },
                    {'size': 590, 'pps': 16,  'isg':0.1 },
                    {'size': 1514, 'pps': 4,   'isg':0.2 } ]
```
Load Testing Methodology

Method 1: VPP has one worker thread, one Rx/Tx queue
- Send unidirectional traffic
- Measure cycles/packet for 1kpps, 1Mpps, 10Mpps, ...
  ⇒ Report max packets/sec for one CPU

Method 2: VPP has n-1 worker threads with [1, 2, 3, ...] Rx queues
- Send unidirectional, or bidirectional (!) traffic
  - Warmup at 1kpps (30sec)
  - Ramp up to 100% line rate (in 600sec)
  - Keep at 100% (30sec)
- Measure point at which packet forwarding loss > 0.1%
  ⇒ Report bits/sec, packets/sec and % of line rate.
Method 1 - Single Thread Saturation

TUI - TRex Console UI

- start -f st1/udp_1pkt_simple.py -p 0 -m 1kpps
- pause
- resume
- update -m 1Mpps
- update -m 10Mpps
- update -m 100%
- stop
Method 1 - TRex Console

Legend:

1. NIC Info, T-Rex CPU utilization
2. Sent traffic (L1, L2, packets/sec)
3. Received traffic (L2, packets/sec)
4. Detailed packet/byte counters

Shown:
19.1Gbit and 6.24Mpps of imix at 16.2% CPU

T-Rex Stateless Loadtester -- pim@ipng.nl

optional arguments:
-h, --help            show this help message and exit
-s SERVER, --server SERVER
    Remote trex address (default: 127.0.0.1)
-p PROFILE_FILE, --profile PROFILE_FILE
    STL profile file to replay (default: imix.py)
-o OUTPUT_FILE, --output OUTPUT_FILE
    File to write results into, use "-" for stdout (default: -)
-wm WARMUP_MULT, --warmup_mult WARMUP_MULT
    During warmup, send this "mult" (default: 1kpps)
-wd WARMUP_DURATION, --warmup_duration WARMUP_DURATION
    Duration of warmup, in seconds (default: 30)
-rt RAMPUP_TARGET, --rampup_target RAMPUP_TARGET
    Target percentage of line rate to ramp up to (default: 100)
-rd RAMPUP_DURATION, --rampup_duration RAMPUP_DURATION
    Time to take to ramp up to target percentage of line rate, in seconds (default: 600)
-hd HOLD_DURATION, --hold_duration HOLD_DURATION
    Time to hold the loadtest at target percentage, in seconds (default: 30)
Method 2 - Interesting profiles

From easier to more challenging (*):

1. **bench-var2-1514b**: 1514b UDP, multiple flows (random src/dst IP)
   \[ \Rightarrow 810\text{Kpps at } 10\text{Gbps} \]

2. **bench-var2-imix**: Mix of 60, 590, 1514 byte UDP, multiple flows
   \[ \Rightarrow 3.2\text{Mpps at } 10\text{Gbps} \]

3. **bench-var2-64b**: 64 byte UDP, multiple flows
   \[ \Rightarrow 14.88\text{Mpps at } 10\text{Gbps}, \text{RSS with multiple Rx queues} \]

4. **bench**: 64 byte UDP, single flow (constant src/dst IP:port)
   \[ \Rightarrow 14.88\text{Mpps at } 10\text{Gbps}, \text{only one Rx queue (= one } lcore) \]

*) numbers are unidirectional
Selection of DUTs

1. Netgate 6100
   CPU: Atom C3558 • 2.2GHz / 3.8GHz
   Network: 2x 10GbE SFP+
   RAM: 224kB L1 • 4MB L2 • 8GB DDR4
   Network: 2x 1GbE SFP/RJ45, 4x 2.5GbE
   Disk: 16G eMMC
   Price: CHF 650,-

2. Supermicro 5018D-FN8T
   CPU: Xeon D1518 • 2.2GHz / 4.0 GHz
   Network: 6x 10GbE SFP+
   RAM: 256kB L1 • 1MB L2 • 6MB L3 • 32GB DDR4
   Network: 4x 1GbE i350, 2x 1GbE i210
   Disk: 128GB mSATA
   Price: CHF 1'350,-

3. ASRock Taichi B550 / Ryzen 5950X
   CPU: Ryzen 5950X • 3.4GHz / 5.05 GHz
   Network: 2x 100GbE QSFP28
   RAM: 1MB L1 • 8MB L2 • 64MB L3 • 128GB DDR4
   Network: 4x 10GbE SFP+, 4x 1GbE i350
   Disk: 2TB NVME
   Price: CHF 2'850,-
## Method 1: Results

<table>
<thead>
<tr>
<th></th>
<th>cycles/packet @ 1kpps</th>
<th>cycles/packet @ 1Mpps</th>
<th>cycles/packet @ 10Mpps</th>
<th>Max PPS per core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atom C3558</td>
<td>4943</td>
<td>620</td>
<td>358</td>
<td>5.01Mpps</td>
</tr>
<tr>
<td>Xeon D1518</td>
<td>2037</td>
<td>341</td>
<td>179</td>
<td>10.20Mpps</td>
</tr>
<tr>
<td>Ryzen 5950X</td>
<td>1112</td>
<td>245</td>
<td>178</td>
<td>22.28Mpps</td>
</tr>
</tbody>
</table>

- **CPU cycles/packet:** lower is better
- **Max PPS per core:** higher is better

vpp v22.02-rc0~492-gf715b33f1 built by pim on rhino at 2022-01-12T11:45:02
Legend:

1. X-axis: time 0 .. 640 seconds
2. Y-axis(left): packets/sec
3. Y-axis(right): fraction of line rate

Method 2: Graphs explained

- Device handles 35% of line rate
- Device handles 100% of line rate
- Rampup 0 - 100%
- Warmup
- Hold
Method 2: Baseline: Kernel 64b

Profile: 14.88Mpps at 10Gbps

TL/DR: Kernel routing is not efficient

<table>
<thead>
<tr>
<th></th>
<th>Linux</th>
<th>FreeBSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atom C3558</td>
<td>632kpps</td>
<td>626kpps</td>
</tr>
<tr>
<td>Xeon D1518</td>
<td>603kpps</td>
<td>597kpps</td>
</tr>
<tr>
<td>Ryzen 5950X</td>
<td>881kpps</td>
<td>847kpps</td>
</tr>
</tbody>
</table>

Yeah, we're all down here

pfSense 21.05.2; Ubuntu 20.04.3
Method 2: Results VPP 1514b

Profile: 810kpps at 10Gbps *

TL/DR: Everybody can be a winner ;-)  
Atom C3558, Xeon D1518, Ryzen 5950X  
1. Uni- or bidirectional: doesn’t matter  
2. 3, 2, or 1 Rx Queue: doesn’t matter  
3. Unsurprising, each CPU can forward >800kpps.  
So far so good...

*) also often called iperf test. Not very useful.
Method 2: Results VPP imix

Profile: 3.2Mpps at 10Gbps

Everybody can be a winner ;-)  

Atom C3558, Xeon D1518, Ryzen 5950X  
1. Uni- or bidirectional: doesn’t matter  
2. 3, 2, or 1 Rx Queue: doesn’t matter  
3. Bidirectional is 6.4Mpps:  
   a. Single Atom core: 5Mpps  
   b. Two directions uses 2 CPUs -  
      Rx-Queue0-nic0 on lcore1  
      Rx-Queue0-nic1 on lcore2

Still good...
Method 2: Results VPP 64b (multi flows)

Profile: 14.88Mpps at 10Gbps

Differences become visible
1. Atom C3558
2. Xeon D1518
3. Ryzen 5950X

Observe linear scaling:
- Adding Rx queue goes from 5Mpps → 10Mpps → 14.88Mpps
Method 2: Results VPP 64b (multi flows)

Profile: 14.88Mpps at 10Gbps

Differences become visible
1. Atom C3558
2. Xeon D1518
3. Ryzen 5950X

Observe linear scaling:
- Adding Rx queue goes from 10Mpps → 14.88Mpps
Method 2: Results VPP 64b (multi flows)

Profile: 14.88Mpps at 10Gbps

Differences become visible
1. Atom C3558
2. Xeon D1518
3. Ryzen 5950X

Who needs scaling anyway:
Ryzen single core throughput: 22.3Mpps
Can easily handle line rate with 1 CPU.
Method 2: Results VPP 64b (single flow)

Profile: 14.88Mpps at 10Gbps, 1 flow

Most difficult case possible

1. Ryzen 5950X: 14.88Mpps
2. Xeon D1518: 10.20Mpps
3. Atom C3558: 5.01Mpps
Additional considerations

Raw forwarding horsepower isn't everything

Netgate 6100 -
- CPU is 16W TDP, has QAT → crypto acceleration
- At 3 cores: ~15.3Mpps forwarding at 19W → 1.24\(\mu\)J per packet

Supermicro 5018D-FN8T -
- CPU is 35W TDP, is hyperthreaded (4C/8T)
- Hyperthreading reduces from 10.2Mpps/core to 6.3Mpps/core (but 8 cores!)
- At 3 threads: 37.8Mpps forwarding at 48W → 1.56\(\mu\)J per packet

ASRock Taichi B550 / Ryzen 5950X -
- CPU is 105W TDP, is hyperthreaded, has 16 cores (16C/32T);
- Hyperthreading is a wash: 10.2Mpps/core to 5.15Mpps/core
- At 15 threads: 330Mpps forwarding at 265W → 0.81\(\mu\)J per packet

PCIe v4.0 bandwidth bottleneck (24 lanes ~ 768Gbps)
Questions, Discussion

If you peer with IPng Networks, thanks!
If you don’t: please peer with AS8298 <peering@ipng.ch>

Useful Resources

- VPP: fd.io
- VPP Linux CP: Github
- Articles: ipng.ch
- Twitter: @IPngNetworks

Also: thanks for listening!