TC S/W datapath: a performance analysis

Paolo Abeni*, Davide Caratti†, Eelco Chaudron‡, Marcelo Ricardo Leitner§
Red Hat Inc.
Email: *pabeni@redhat.com, †dcaratti@redhat.com, ‡echaudro@redhat.com, §marcelo.leitner@gmail.com

***Abstract*—This paper reports a performance analysis of the Linux Traffic Control(TC) S/W datapath, comparing it to the traditional kernel Open vSwitch(OVS) datapath, examining its recent improvements, and looking at new and future improvement in this area, comprising XDP and eBPF usage.**

***Index Terms*—OVS, TC, performances, XDP**

I. Introduction

The virtual switch is one of the core building blocks for NFV. As such, virtual switch implementations are subject to a large set of functional and performance requirements. The later could be very challenging for carrier grade-usage and current H/W.

Given the above, different implementations of Open vSwitch(OVS), the leading virtual switch project [1] - surfaced both in user-space and in the Linux kernel.

The original user-space and kernel datapaths have been sided by a faster DPDK-based implementation. But even the latter is not able to cope with high-bandwidth line-rate packet speeds and is weighed down by complex setup requirements[2].

To overcome these difficulties, H/W offload solutions for OVS are now spreading. The in-kernel interface to configure such complex offload is offered by the TC APIs via the TC H/W offload hook. This design implicitly gave birth to another in-kernel OVS datapath: the TC S/W datapath, as for each datapath feature supported in the hardware, we need a software path implemented in TC to enable it.

Since we now have 2 fairly non-trivial sub-systems implementing the same functionality inside the kernel, we could consider a de-duplication effort. While the TC datapath is necessary for H/W offload, the plain old OVS kernel datapath is preferred over the TC S/W one as a fallback solution. The main reasons behind such choice are TC not being yet feature-complete and OVS kernel datapath being considered faster.

Is that so? This paper will discuss the TC S/W performances, looking at its recent and current status, as well looking at the upcoming future for kernel related OVS efforts.

II. The testing scenario

In this discussion, we will consider the PVP scenario (Physical to Virtual to Physical) as our reference [3]. The scenario is described in figure 1. A traffic generator sends UDP packets at line rate (64 bytes packets on a 10 Gbps link in our experiments) towards the host under tests. Such host runs an instance of OVS, configured with a variable number of bidirectional flows to forward traffic back and forth a local VM via a tap device. The VM itself runs a DPDK application

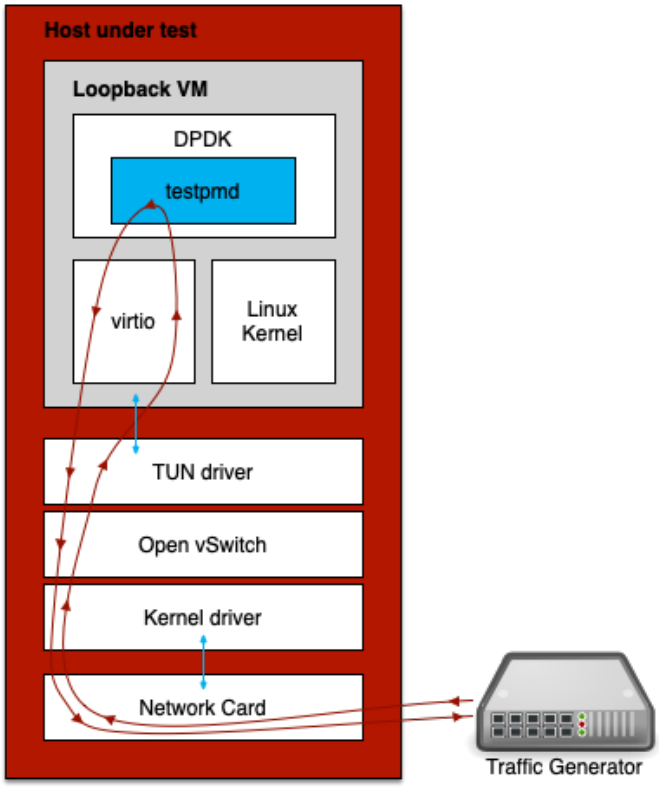

Fig. 1. The setup for PVP testing

(testpmd) to loopback the traffic on the same virtual device. The host OVS instance forwards back the looped traffic to the ingress interface, due to the bi-directional rules above.

The testpmd application has the specific role of stressing as much as possible the forwarding data-plane inside the host. While the above does not describe a real-life usage scenario for OVS it allows us to stress the switch with carrier-grade like traffic with a relatively simple setup.

In our experiments, each OVS flow matches different source IP address - ingress interface pair and has a single action to forward the packet unmodified to the relevant end. Using different src IPs allow different flows to land on different ingress NIC's queues with default RSS configuration.

The H/W under test is an Intel Xeon CPU E5-2690 v3 with 12 core/24 threads, using an Intel 82599ES 10-Gigabit SFI/SFP+ NIC.

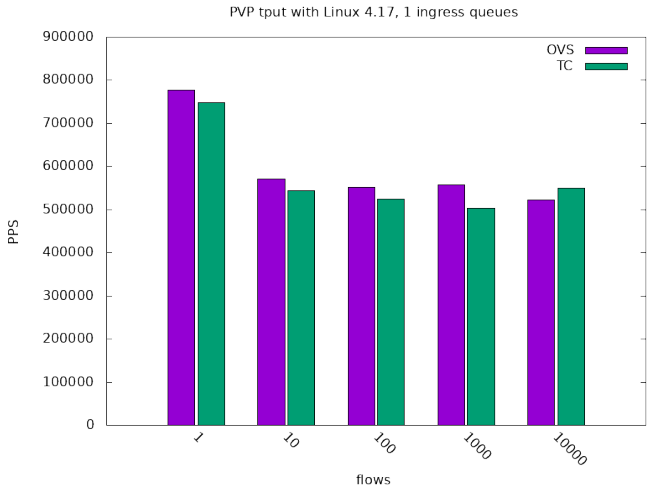

Fig. 2. PVP performances - Linux 4.17

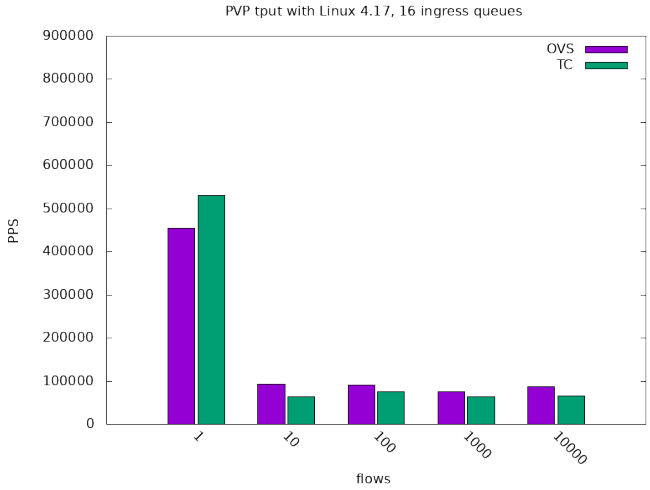

Fig. 3. PVP performances with multiple receive queue - Linux 4.17

III. The TC S/W datapath performances

In this section we look at evolution of the performances of the TC S/W software datapath in the recent pasts and it's current status.

A. Past status

Let's see the figures we collected for the PVP scenario on top of Linux 4.17 (Figure 2).

We see differences between TC and OVS datapath just above noise range and overall figures are not bad: sustaining a packet-rate of an 814Kpps allows line rate on 10Gpbs link with max MTU frames, and we are not far from that.

But this is a for an uncommon configuration, with only a single receive queue enabled on the host ingress NIC. The default number of receive queues for the H/W under test is 16. Let's show the performances change with such configuration (Figure 3).

Topmost perf offender for vhost (1 queue)	Topmost perf offenders for vhost (16 queues)
6.68% _raw_spin_lock	13.76% tun_do_read
5.51% tun_get_user	8.08% __slab_free
5.20% vhost_get_vq_desc	7.79% vhost_get_vq_desc
5.17% masked_flow_lookup	7.26% _copy_to_iter
4.82% ixgbe_xmit_frame_ring	7.23% page_frag_free
3.80% translate_desc	5.95% __check_object_size
3.41% __skb_flow_dissect	5.86% handle_rx
3.34% tun_do_read	4.26% vhost_net_buf_peek
3.33% iov_iter_advance	3.69% translate_desc
2.83% __slab_free	3.42% iov_iter_advance
2.55% _copy_to_iter	3.42% iov_iter_advance
2.46% page_frag_free	2.65% tun_recvmsg

Fig. 4. perf data for Linux 4.17 with OVS backend

```
100.00%           vhost-<pid>
 |
 --     |--87.85%--ret_from_fork
        |           kthread
        |           vhost_worker
        |           |
                    |--79.25%--handle_rx
                              |
                              |--55.45%--tun_recvmsg
                            [...]
                    | --8.60%--handle_tx
                              |
                              |--7.42%--tun_sendmsg
                            [...]
```

Fig. 5. perf data for vhost with call-graph accounting

Again, the performances differences between the TC and the OVS datapath are small, but the aggregate throughput falls to very low values as soon as the number of flows grows.

With our configuration, running multiple flows mean concurrent ksoftirq processes enqueuing packets to the vhost queue. A large performance degradation in such condition looks like a contention problem. We can easily see that in the above test the bottle-neck is the vhost process, keeping a CPU's core fully busy. With some help from the 'perf' tool, we can observe where such process is spending its time.

Figure 4 shows the topmost perf offenders for the vhost process while using the OVS backend, with the ingress NIC configured respectively with 1 and 16 receive queues. The data collected with the TC backend are quite similar.

While the topmost offender's list differs a lot in the two reported scenarios, it's not straightforward to pinpoint the cause for a 6x slowdown. More importantly, there is no apparent display of contention problems in the 16 queues scenario. Which is good, since a lot of effort has been spent recently to let vhost behave well under contention.

We can use again the 'perf' tool to fetch more data. Figure 5 shows a slice of the call-graph accounting for the vhost process in the 16 queue scenario. With call graph accounting 'perf' measures the time spent in a function summing-up also the time required recursively by the nested calls.

The 'handle_rx()' function processes all the packets sent towards the tun device (from OVS, in our scenario), while the

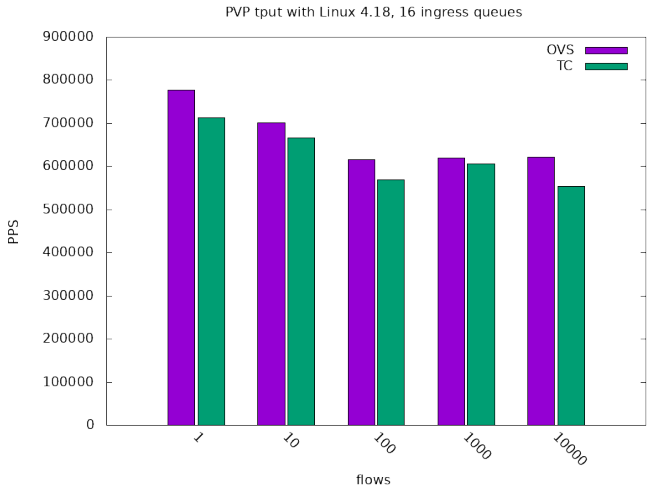

Fig. 6. PVP performances - Linux 4.18

Topmost perf offender (OVS backend)		Topmost perf offendersw (TC backend)	
7.50%	masked_flow_lookup	5.08%	ixgbe_xmit_frame_ring
5.02%	ixgbe_xmit_frame_ring	4.63%	vhost_get_vq_desc
4.20%	vhost_get_vq_desc	4.37%	skb_release_data
3.89%	iov_iter_advance	3.86%	translate_desc
3.18%	translate_desc	3.00%	iov_iter_advance
2.92%	pfifo_fast_dequeue	2.94%	tun_get_user
2.83%	tun_build_skb.isra.57	2.89%	__skb_flow_dissect
2.81%	tun_get_user	2.76%	memcmp
2.09%	__dev_queue_xmit	2.54%	pfifo_fast_dequeue
1.99%	handle_tx	2.17%	rhashtable_jhash2
1.93%	_copy_to_iter	2.14%	tun_do_read
1.88%	key_extract	2.07%	kmem_cache_free

Fig. 7. perf data for Linux 4.18, for vhost process OVS vs TC backend

'handle_tx()' function processes all the packets sent from the tun device (by the VM, in our scenario). We can see that the vhost processing is asymmetric: the receive time is ~10x the transmit time! Looking at the packet counters we can observe that this difference is not due to the receive phase being slower, but due to the 'scheduling' between receiving and transmitting being unfair.

Such scheduling is done by the vhost process itself, limiting the burst of packets processed by each direction to a fixed amount of bytes. In Linux 4.17 vhost also uses a packets-based limit, but only for the TX direction. So a solution for the above asymmetry is actually simple: let's enforce the same packets-limit on both RX and TX. This has been implemented in the 4.18 release cycle[4].

B. Recent improvements

Figure 6 shows the performance for the OVS and TC datapath on top of the 4.18 kernel, using 16 receive queues on the ingress NIC. When moving to multiple flows scenarios, there is still a slightly visible performance degradation compared to single flow, but overall we have a 5x improvement compared to 4.17. Interestingly, there is a measurable gap between TC and OVS datapath, a bit above noise range.

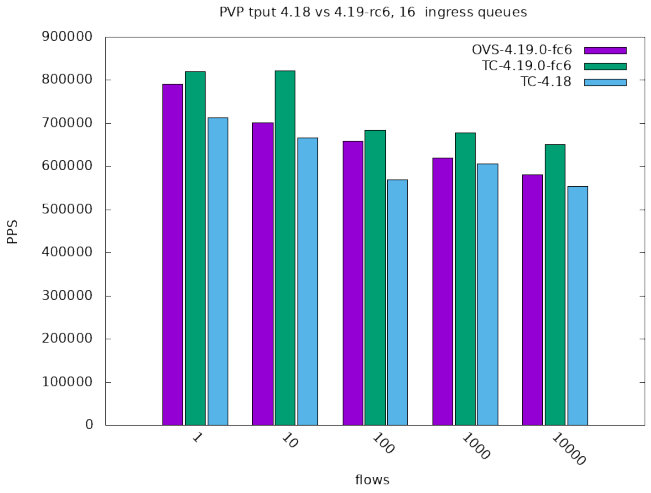

Fig. 8. PVP performances - Linux 4.19.0-rc6

Let's investigate the root cause behind this gap. Figure 7 shows the topmost 'perf' offenders for the vhost process with the TC backend, compared to the ones for the OVS backend. Surprisingly enough, we don't have a clear indication where the TC backend causes vhost to spend more cycles. Instead, we notice a slightly better parsing and lookup time for the TC backend, due to the usage of the highly optimized programmable flow dissector infrastructure.

If we look at functions consuming fewer CPU cycles, we can have instead some hints: with the TC backend, vhost is calling skb_clone(). While the latter does not look like very expensive per se, it adds cost to several other skb-lifecycle-related functions.

Why is the TC backend performing such clone? it uses the act_mirred TC action to forward the skb to the egress device. Such module clones the skb, forward the cloned packet and returns to the caller a controlling action, which is applied by the caller himself. In the TC S/W datapath use-case, such controlling action is always DROP. This somewhat convoluted implementation is necessary as per TC design the caller retains the ownership of the processed skb.

In the given scenario we can optimize this behavior: the mirred module could return to the caller a 'redirect' controlling action, without performing any clone operation. Such a schema has been implemented in the 4.19 release cycle[5].

C. Current status

We can now look at the current status, as reported in figure 8. The kernel under test is 4.19.0-rc6, already comprising the optimization described above. The performance improvement for the TC datapath over 4.18 is quite apparent, and such datapath now performs consistently better than the plain OVS one.

D. Adding more complexity

This discussion does not consider the features list gap between the TC S/W datapath and the OVS one: currently

some features - notably connection tracking - is available only with the OVS datapath. While this gap exists, a complete replacement is not feasible by definition, but such a gap has shrunk over time and support for conntrack H/W offload is in the work. That will likely bring conntrack support for the TC S/W datapath, too.

The scenario used for testing contains several simplifications over real-life use cases; for example, each of our rules contains a single action for forwarding the packet, while real deployments would add at least another one for adding or removing a layer of encapsulation (e.g. vxlan[6]), possibly more to modify in-flight packets. With the TC S/W datapath, each action adds significant overhead, due to the usage of indirect calls and retpolines.

Moreover, the usage of some specific TC actions does not scale well with concurrent flows, due to the usage of a per-action spinlock to protect the data structure access. One relevant example is ack_pedit, used by the TC S/W datapath to perform packet modification.

The solution here is the conversion of TC actions to RCU usage, currently a WIP. The ack_pedit module is actually the only relevant left-over.

The current direction of the Linux kernel networking stack to address retpoline overhead is introducing "listification" support for the relevant hooks: instead of processing a single packet at the time, packets that need similar processing are aggregated into a list and passed to the next layer altogether[7]. While interesting, this approach is hard to adapt to the TC subsystem due to the large number of hooks involved and the stratification which would make such implementation very complex, at best.

With both the OVS and TC datapath, the vhost process has a pivotal role and we hit a bottle-neck there. We could easily almost double the throughput in the PVP scenario, using two separate processes for RX and TX. This latter solution is quite alike at using multiple virtio_net queues, as we trade more CPU power for more throughput.

Finally, we must note that the packet rate reached so-far, while allowing us to process ~10Gbps packets at max MTU size, is nowhere near carrier-grade requirements, which ask for line rate even with min MTU packet size (~14Mpps).

In that respect, even bypass solutions, like OVS-DPDK, are quite far from a perfect match. With these, we measure ~3Mpps in the given scenario, still quite above what the kernel currently offers.

IV. Pulling XDP and eBPF into the play

Currently, support for an XDP/eBPF backend in OVS is under development[8]. XDP is the increasingly recurring solution of choice for high throughput packet processing scenarios and we may wonder if that will change the current situation drastically.

Instead of testing the in-progress XDP/eBPF OVS effort, we opted for a basic custom implementation of an XPP/eBPF forwarding datapath[9], with the main reasoning to avoid the eventual bottle-neck present in this initial phase (and the real reason behind this choice being the need to experiment first hand with this inescapable piece of the Linux networking).

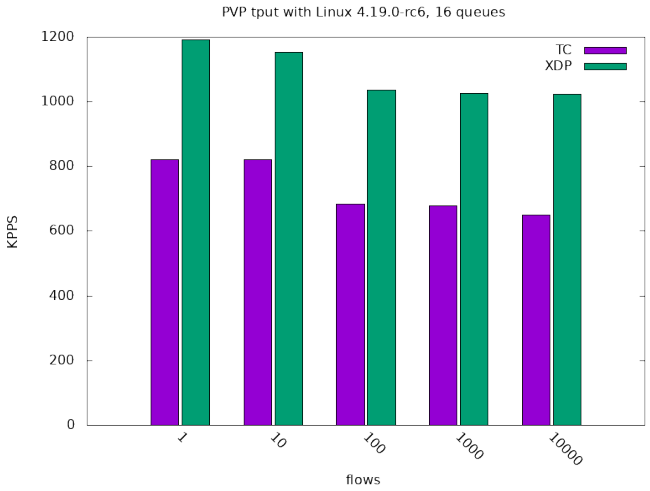

Fig. 9. PVP performances - XDP datapath

With our test implementation, an XDP program is attached to both the ingress NIC and the tun device. It parses the ingress packet up the IP header, extracts the source IP address, looks-up the ingress device id and source address in an user-configured per-CPU map and forward the packet to egress device specified in the map entry. Flow stats are maintained in the same entry. Overall we have a single map lookup and a redirect per packet, with no tail calls. The userspace counterpart allow us to install the needed ruleset and collecting aggregated statistics.

While this is far from being a complete or even usable solution (outside this specific testing scenario), it should give a reasonable upper bound of the performances we can expect with OVS-XDP, as the latter requires several map looks-up and tail-calls per packet.

The performance results are represented in Figure 9, compared to the TC S/W datapath on top of Linux 4.19.0-rc6. The benefit is really apparent, but we are still far line rate with small packets.

V. Future works and conclusion

The forwarding performance for the TC S/W datapath improved in the recent past up to and above the level of the OVS kernel datapath. While in the current status it can sustain reasonably stressing workloads, it's still far from the challenging level required by NFV.

In the near future, we hope that UDP GRO could land in the kernel for forwarded packets, introducing bulking support for some of the scenarios tested here. Specifically, since, each NAPI instance can aggregate at most 8 flows, we can expect performance benefits for test-cases with 100 flows or lest.

References

[1] "Open vSwitch" *https://www.openvswitch.org/*

[2] “Measuring and comparing Open vSwitch performance” *https://developers.redhat.com/blog/2017/06/05/measuring-and-comparing-open-vswitch-performance/*
[3] “Automated Open vSwitch PVP testing” *https://developers.redhat.com/blog/2017/09/28/automated-open-vswitch-pvp-testing/*.
[4] “vhost_net: use packet weight for rx handler, too” *https://github.com/torvalds/linux/commit/db688c24eada63b1efe6d0d7d835e5c3bdd71fd3*
[5] “TC: refactor act_mirred packets re-injection” *https://github.com/torvalds/linux/commit/8f3f6500c74935bfe5a9067e3106b806f336facf*
[6] “OVS flows logic” *https://wiki.openstack.org/wiki/Ovs-flow-logic*
[7] “Handle multiple received packets at each stage” *https://github.com/torvalds/linux/commit/2d1b138505dc29bbd7ac5f82f5a10635ff48bddb*
[8] “OVS eBPF datapath” *https://mail.openvswitch.org/pipermail/ovs-dev/2018-June/348521.html*
[9] “XDP dumb switch” *https://github.com/altoor/xdp_dumb_switch/*