

# XDP vs OVS

August 17, 2020

Long overdue blog post on XDP; so many details uncovered during testing causing tests to be redone.

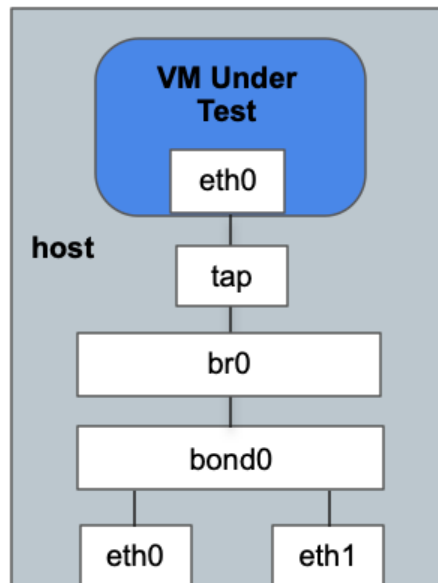
This post focuses on a comparison of XDP and OVS in delivering packets **to** a VM from the perspective of CPU cycles spent by the host in processing those packets. There are a lot of variables at play, and changing any one of them radically affects the outcome, though it should be no surprise XDP is always lighter and faster.

## Setup

I believe I am covering all of the settings here that I discovered over the past few months that caused variations in the data.

## Host

The host is a standard, modern server (Dell PowerEdge R640) with an Intel® Xeon® Platinum 8168 CPU @ 2.70GHz with 96 hardware threads (48 cores + hyper threading to yield 96 logical cpus in the host). The server is running Ubuntu 18.04 with the recently released 5.8.0 kernel. It has a Mellanox Connectx4-LX ethernet card with 2 25G ports into an 802.3ad (LACP) bond, and the bond is connected to an OVS bridge.



As discussed in [1] to properly compare the CPU costs of the 2 networking solutions, we need to consolidate packet processing to a

single CPU. Handling all packets destined to the same VM on the same CPU avoids lock contention on the tun ring, so consolidating packets to a single CPU is actually best case performance.

Ensure RPS is disabled in the host:

```
for d in eth0 eth1; do
    find /sys/class/net/${d}/queues -name rps_cpus |
    while read f; do
        echo 0 | sudo tee ${f}
    done
done
```

and add flow rules in the NIC to push packets for the VM under test to a single CPU:

```
sudo ethtool -N eth0 flow-type ether dst 12:34:de:ad:ca:fe action 2
sudo ethtool -N eth1 flow-type ether dst 12:34:de:ad:ca:fe action 2
```

For this host and ethernet card, packets for queue 2 are handled on CPU 5 (consult `/proc/interrupts` for the mapping on your host).

XDP bypasses the qdisc layer, so to have a fair comparison make **noqueue** the default qdisc *before* starting the VM:

```
sudo sysctl -w net.core.default_qdisc=noqueue
```

(or add a udev rule [2]).

Finally, the host is fairly quiet with only one VM running (the one under test) and very little network traffic outside of the VM under test and a few, low traffic ssh sessions used to run commands to collect data about the tests.

## Virtual Machine

The VM has 8 cpus and is also running Ubuntu 18.04 with a 5.8.0 kernel. It uses tap+vhost for networking with the tap device a port in the OVS bridge as shown in the picture above. The tap device has a single queue, and RPS is also disabled in the guest:

```
echo 00 | sudo tee /sys/class/net/eth0/queues -name rps_cpus
```

The VM is also quiet with no load running in the guest OS.

The point of this comparison is host side processing of packets, so packets are dropped in the guest as soon as possible using a bpf program [3] attached to eth0 as a tc filter. (Note: Theoretically, XDP should be used to drop the packets in the guest OS since it truly is the fewest cycles per packet. However, XDP in the VM requires a multi-queue NIC[5], and adding queues to the guest NIC has a **huge** affect on the results.)

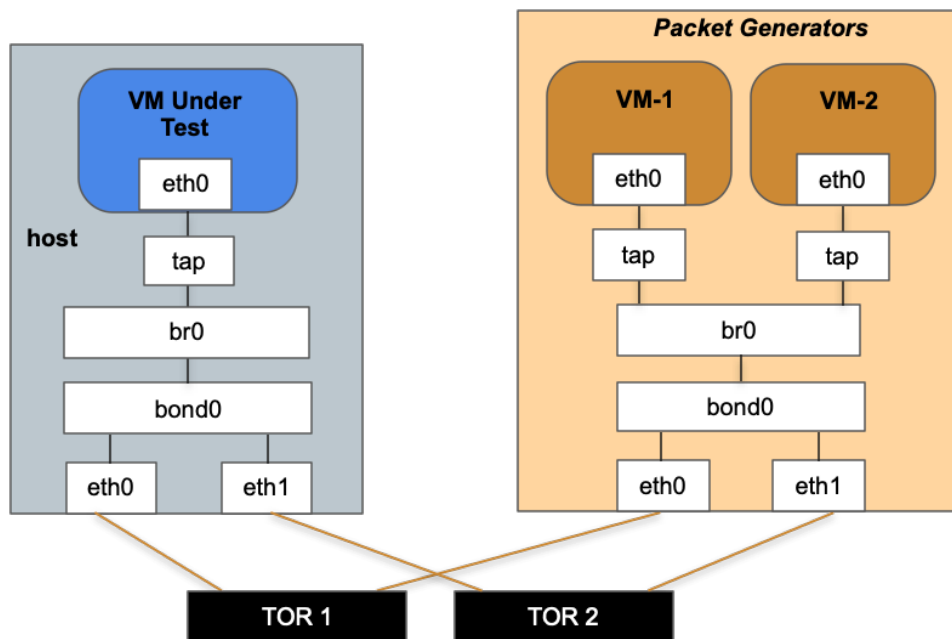
In the host, the qemu threads corresponding to the guest CPUs (vcpus) are affined (as a set) to 8 hardware threads in the same NUMA node as CPU 5 (the host CPU processing packets per the RSS rules mentioned earlier). The vhost thread for the VM's tap device is also affined to a small set of host CPUs in the same NUMA node to avoid scheduling collisions with the vcpu threads, the CPU processing packets (5) and its sibling hardware thread (CPU 53 in my case) – all of which add variability to the results.

## Forwarding with XDP

Packet forwarding with XDP is done by attaching an L2 forwarding program [4] to eth0 and eth1. The program pulls the VLAN and destination mac from the ethernet header, and uses the pair as a key for a lookup in a hash map. The lookup returns the next device index for the packet which for packets destined to the VM is the index of its tap device. If an entry is found, the packet is redirected to the device via XDP\_REDIRECT. The use case was presented in depth at netdevconf 0x14 [5].

## Packet generator

Packets are generated using 2 VMs on a server that is directly connected to the same TOR switches as the hypervisor running the VM under test. The point of the investigation is to measure the overhead of delivering packets to a VM, so memcpy is kept to a minimum by having the packet generator [6] in the VMs send 1-byte UDP packets.



Each VM can generate a little over 1 million packets per sec (1M pps), for a maximum load of 2.2M pps based on 2 separate source addresses.

## CPU Measurement

As discussed in [1] a fair number of packets are processed in the context of some interrupted, victim process or when handled on an idle CPU the cycles are not fully accounted in the softirq time shown in tools like mpstat.

This test binds `openssl speed`, a purely userspace command[1], to the CPU handling packets to fully consume 100% of all CPU cycles which makes the division of CPU time between user, system and softirq more transparent. In this case, the output of `mpstat -P 5` shows how all of the cycles for CPU 5 were spent (within the resolution of system accounting):

\* %softirq is the time spent handling packets. This data is shown in the graphs below.

\* %usr represents the usable CPU time for processes to make progress on their workload. In this test, it shows the percentage of CPU consumed by openssl and compares to the times shown by openssl within 1-2%.

\* %sys is the percentage of kernel time and for the data shown below was always <0.2%.

As an example, in this `mpstat` output openssl is only getting 14.2% of the CPU while 85.8% was spent handling the packet load:

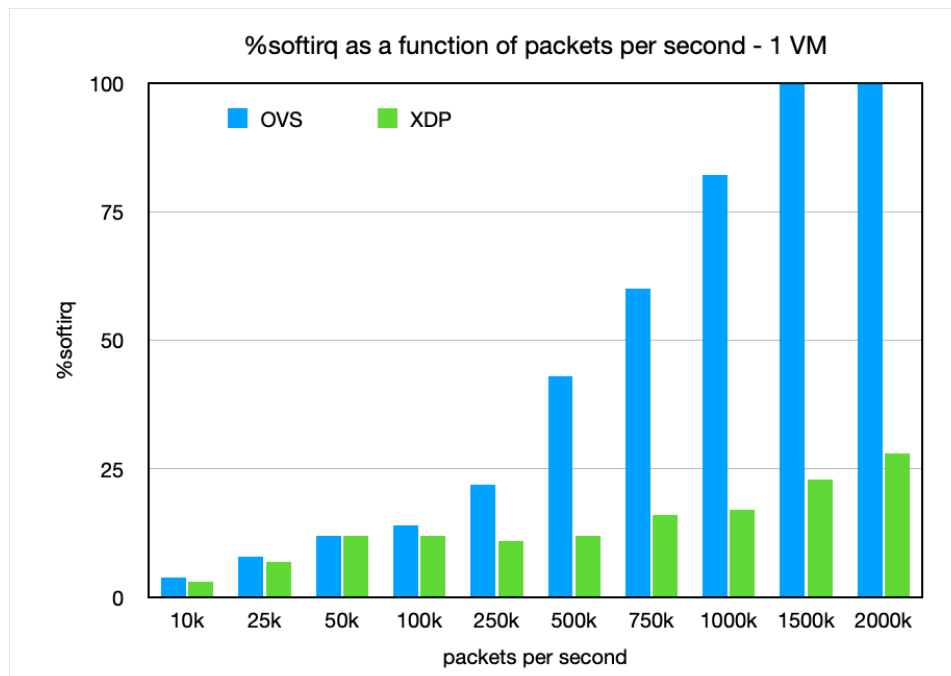
CPU	%usr	%nice	%sys	%iowait	%irq	%soft	%idle
5	14.20	0.00	0.00	0.00	0.00	85.80	0.00

(%steal, %guest and %gnice dropped were always 0 and dropped for conciseness.)

Let's get to the data.

## CPU Comparison

This chart shows a comparison of the %softirq required to handle various PPS rates for both OVS and XDP. Lower numbers are better (higher percentages mean more CPU cycles).



There is 1-2% variability in ksoftirqd percentages despite the 5-second averaging, but the variability does not really affect the important points of this comparison.

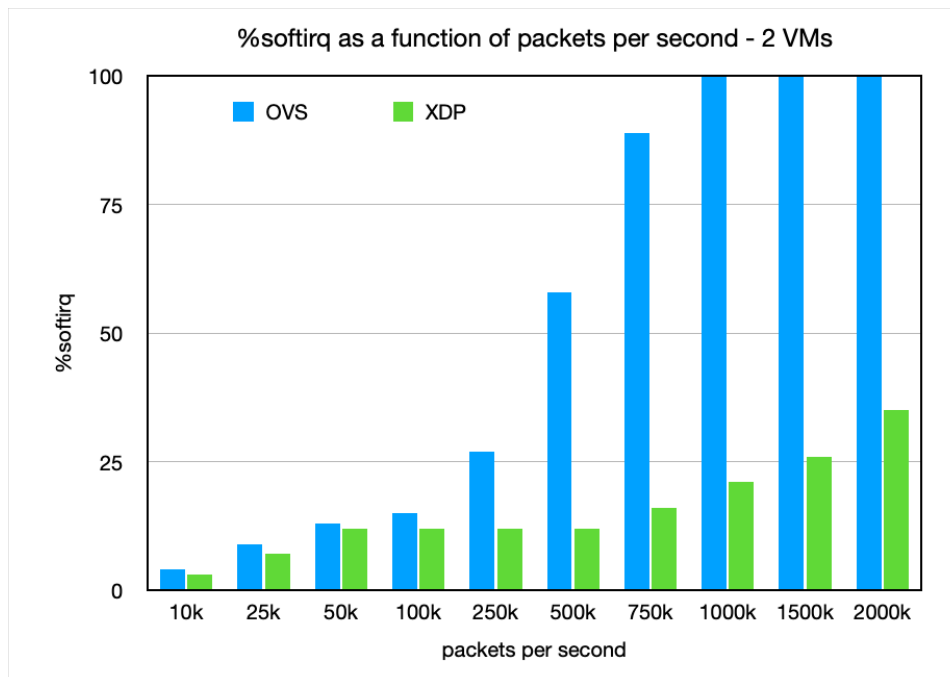
The results should not be that surprising. OVS has well established scaling problems and the chart shows that as packet rates increase. In my tests it was not hard to saturate a CPU with OVS, reaching a maximum packet rate to the VM of 1.2M pps. The 100% softirq at 1.5M pps and up is saturation of ksoftirqd alone with nothing else running on that CPU. Running another process on CPU 5 immediately affects the throughput rate as the CPU splits time between processing packets and running that process. With openssl, the packet rate to the

VM is cut in half with packet drops at the host ingress as it can no longer keep up with the packet rate given the overhead of OVS.

XDP on the other hand could push 2M pps to the VM before the **guest** could no longer keep up with packet drops at the tap device (ie., no room in the tun ring meaning the guest has not processed the previous packets). As shown above, the host still has plenty of CPU to handle more packets or run workloads (preferred condition for a cloud host).

One thing to notice about the chart above is the apparent flat lining of CPU usage between 50k pps and 500k pps. That is not a typo, and the results are very repeatable. This needs more investigation, but I believe it shows the efficiencies kicking in from a combination of more packets getting handled per napi poll cycle (closer to maximum of the netdev budget) and the kernel side bulking in XDP before a flush is required.

Hosts typically run more than 1 VM, so let's see the effect of adding a second VM to the mix. For this case a second VM is started with the same setup as mentioned earlier, but now the traffic load is split equally between 2 VMs. The key point here is a single CPU processing interleaved network traffic for 2 different destinations.



For OVS, CPU saturation with ksoftirqd happens with a maximum packet rate to each VM of 800k pps (compared to 1.2M with only a single VM). The saturation is in the host with packet drops shown at host ingress, and again any competition for the CPU processing packets cuts the rate in half.

Meanwhile, XDP is barely affected by the second VM with a modest increase of 3-4% in softirq at the upper packet rates. In this case, the redirected packets are just hitting separate bulking queues in the kernel. The two packet generators are not able to hit 4+M pps to find the maximum per-VM rate.

## Final Thoughts

CPU cycles are only the beginning for comparing network solutions. A full OVS-vs-XDP comparison needs to consider all the resources consumed – e.g., memory as well as CPU. For example, OVS has ovs-vswitchd which consumes a high amount of memory (>750MB RSS on this server with only the 2 VMs) and additional CPU cycles to handle upcalls (flow misses) and revalidate flow entries in the kernel which on an active hypervisor can easily consume 50+% cpu (not counting increased usage from various bugs[7]).

Meanwhile, XDP is still early in its lifecycle. Right now, using XDP for this setup requires VLAN acceleration in the NIC [5] to be disabled meaning the VLAN header has to be removed by the ebpf program before forwarding to the VM. Using the proposed hardware hints solution reduces the softirq time by another 1-2% meaning 1-2% more usable CPU by leveraging hardware acceleration with XDP. This is just an example of how XDP will continue to get faster as it works better with hardware offloads.

## Acronyms

LACP Link Aggregation Control Protocol

NIC Network Interface Card

NUMA Non-Uniform Memory Access

OVS Open VSwitch

PPS Packets per Second

RPS Receive Packet Steering

RSS Receive Side Scaling

TOR Top-of-Rack

VM Virtual Machine

XDP Express Data Path in Linux

## References

[1] <https://people.kernel.org/dsahern/the-cpu-cost-of-networking-on-a-host>

- [2] <https://people.kernel.org/dsahern/rss-rps-locking-qdisc>
- [3] [https://github.com/dsahern/bpf-progs/blob/master/ksrc/rx\\_acl.c](https://github.com/dsahern/bpf-progs/blob/master/ksrc/rx_acl.c)
- [4] [https://github.com/dsahern/bpf-progs/blob/master/ksrc/xdp\\_l2fwd.c](https://github.com/dsahern/bpf-progs/blob/master/ksrc/xdp_l2fwd.c)
- [5] <https://netdevconf.info/0x14/session.html?tutorial-XDP-and-the-cloud>
- [6] <https://github.com/dsahern/random-cmds/blob/master/src/pktgen.c>
- [7] <https://www.mail-archive.com/ovs-dev@openvswitch.org/msg39266.html>