OVS Offload Using ASAP² Direct

Table of contents

Overview

> **Note**
>
> Supported on ConnectX-5 and above adapter cards.

Open vSwitch (OVS) allows Virtual Machines (VMs) to communicate with each other and with the outside world. OVS traditionally resides in the hypervisor and switching is based on twelve tuple matching on flows. The OVS software based solution is CPU intensive, affecting system performance and preventing full utilization of the available bandwidth. NVIDIA Accelerated Switching And Packet Processing (ASAP^2) technology allows OVS offloading by handling OVS data-plane in ConnectX-5 onwards NIC hardware (Embedded Switch or eSwitch) while maintaining OVS control-plane unmodified. As a result, we observe significantly higher OVS performance without the associated CPU load.

As of v5.0, OVS-DPDK became part ofMLNX_EN package. OVS-DPDK supports ASAP^2 just as the OVS-Kernel (Traffic Control (TC) kernel-based solution) does, yet with a different set of features.

The traditional ASAP^2 hardware data plane is built over SR-IOV virtual functions (VFs), so that the VF is passed through directly to the VM, with the NVIDIA driver running within the VM. An alternate approach that is also supported is vDPA (vhost Data Path Acceleration). vDPA allows the connection to the VM to be established using VirtIO, so that the data-plane is built between the SR-IOV VF and the standard VirtIO driver within the VM, while the control-plane is managed on the host by the vDPA application. Two flavors of vDPA are supported, Software vDPA; and Hardware vDPA. Software vDPA management functionality is embedded into OVS-DPDK, while Hardware vDPA uses a standalone application for management, and can be run with both OVS-Kernel and OVS-DPDK. For further information, please see sections VirtIO Acceleration through VF Relay (Software vDPA) and VirtIO Acceleration through Hardware vDPA.

Installing OVS-Kernel ASAP² Packages

Install the required packages. For the complete solution, you need to install supporting MLNX_EN(v4.4 and above), iproute2, and openvswitch packages.

Installing OVS-DPDK ASAP² Packages

Run:

```
./install --ovs-dpdk –upstream-libs
```

Setting Up SR-IOV

> **Note**
>
> Note that this section applies to both OVS-DPDK and OVS-Kernel similarly.

To set up SR-IOV:

1. Choose the desired card.

   The example below shows a dual-ported ConnectX-5 card (device ID 0x1017) and **a single SR-IOV VF** (Virtual Function, device ID 0x1018).

   In SR-IOV terms, the card itself is referred to as the PF (Physical Function).

   ```
   # lspci -nn | grep Mellanox

   0a:00.0 Ethernet controller [0200]: Mellanox Technologies MT27800 Family [ConnectX-5] [15b3:1017]
   0a:00.1 Ethernet controller [0200]: Mellanox Technologies MT27800 Family [ConnectX-5] [15b3:1017]

   0a:00.2 Ethernet controller [0200]: Mellanox Technologies MT27800 Family [ConnectX-5 Virtual Function] [15b3:1018]
   ```

> **Note**
>
> Enabling SR-IOV and creating VFs is done by the firmware upon admin directive as explained in Step 5 below.

2. Identify the NVIDIA NICs and locate net-devices which are on the NIC PCI BDF.

```
# ls -l /sys/class/net/ | grep 04:00

lrwxrwxrwx 1 root root 0 Mar 27 16:58 enp4s0f0 ->
../../devices/pci0000:00/0000:00:03.0/0000:04:00.0/net/enp4s0f0
lrwxrwxrwx 1 root root 0 Mar 27 16:58 enp4s0f1 ->
../../devices/pci0000:00/0000:00:03.0/0000:04:00.1/net/enp4s0f1
lrwxrwxrwx 1 root root 0 Mar 27 16:58 eth0 ->
../../devices/pci0000:00/0000:00:03.0/0000:04:00.2/net/eth0
lrwxrwxrwx 1 root root 0 Mar 27 16:58 eth1 ->
../../devices/pci0000:00/0000:00:03.0/0000:04:00.3/net/eth1
```

The PF NIC for port #1 is enp4s0f0, and the rest of the commands will be issued on it.

3. Check the firmware version.
Make sure the firmware versions installed are as state in the Release Notes document.

```
# ethtool -i enp4s0f0 | head -5
driver: mlx5_core
version: 5.0-5
firmware-version: 16.21.0338
expansion-rom-version:
bus-info: 0000:04:00.0
```

4. Make sure SR-IOV is enabled on the system (server, card).
Make sure SR-IOV is enabled by the server BIOS, and by the firmware with up to N

VFs, where N is the number of VFs required for your environment. Refer to "NVIDIA Firmware Tools" below for more details.

```
# cat /sys/class/net/enp4s0f0/device/sriov_totalvfs
4
```

5. Turn ON SR-IOV on the PF device.

```
# echo 2 > /sys/class/net/enp4s0f0/device/sriov_numvfs
```

6. Provision the VF MAC addresses using the IP tool.

```
# ip link set enp4s0f0 vf 0 mac e4:11:22:33:44:50
# ip link set enp4s0f0 vf 1 mac e4:11:22:33:44:51
```

7. Verify the VF MAC addresses were provisioned correctly and SR-IOV was turned ON.

```
# cat /sys/class/net/enp4s0f0/device/sriov_numvfs
2

# ip link show dev enp4s0f0
256: enp4s0f0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq
master ovs-system state UP mode DEFAULT group default qlen 1000
link/ether e4:1d:2d:60:95:a0 brd ff:ff:ff:ff:ff:ff
vf 0 MAC e4:11:22:33:44:50, spoof checking off, link-state auto
vf 1 MAC e4:11:22:33:44:51, spoof checking off, link-state auto
```

In the example above, the maximum number of possible VFs supported by the firmware is 4 and only 2 are enabled.

8. Provision the PCI VF devices to VMs using PCI Pass-Through or any other preferred virt tool of choice, e.g virt-manager.

For further information on SR-IOV, refer to HowTo Configure SR-IOV for ConnectX-4/ConnectX-5/ConnectX-6 with KVM (Ethernet).

OVS Hardware Offloads Configuration

OVS-Kernel Hardware Offloads

SwitchDev Configuration

1. Unbind the VFs.

```
echo 0000:04:00.2 > /sys/bus/pci/drivers/mlx5_core/unbind
echo 0000:04:00.3 > /sys/bus/pci/drivers/mlx5_core/unbind
```

> **Note**
>
> VMs with attached VFs must be powered off to be able to unbind the VFs.

2. Change the eSwitch mode from Legacy to SwitchDev on the PF device. This will also create the VF representor netdevices in the host OS.

```
# devlink dev eswitch set pci/0000:3b:00.0 mode switchdev
```

> **Note**
>
> Before changing the mode, make sure that all VFs are unbound.

> **Note**
>
> To go back to SR-IOV legacy mode, run:# devlink dev eswitch set pci/0000:3b:00.0 mode legacyThis will also remove the VF representor netdevices.

On old OSs or kernels that do not support Devlink, moving to SwitchDev mode can be done using sysfs.

```
# echo switchdev > /sys/class/net/enp4s0f0/compat/devlink/mode
```

3. At this stage, VF representors have been created. To map representor to its VF, make sure to obtain the representor's switchid and portname from:

```
# ip -d link show eth4
41: enp0s8f0_1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq
state UP mode DEFAULT group default qlen 1000
link/ether ba:e6:21:37:bc:d4 brd ff:ff:ff:ff:ff:ff promiscuity 0 addrgenmode eui64
numtxqueues 10 numrxqueues 10 gso_max_size 65536 gso_max_segs 65535
portname pf0vf1 switchid f4ab580003a1420c
```

switchid - used to map representor to device, both device PFs have the same switchid.
portname - used to map representor to PF and VF, value returned is pfXvfY, where X is the PF number and Y is the number of VF.
On old kernels, switchid and portname can be acquired through sysfs:

4. Bind the VFs.

```
echo 0000:04:00.2 > /sys/bus/pci/drivers/mlx5_core/bind
echo 0000:04:00.3 > /sys/bus/pci/drivers/mlx5_core/bind
```

SwitchDev Performance Tuning

SwitchDev performance can be further improved by tuning it.

Steering Mode

OVS-kernel supports two steering modes for rules insertion into hardware.

1. **SMFS** – Software Managed Flow Steering (as of MLNX_OFED v5.1, this is the default mode)

   Rules are inserted directly to the hardware by the software (driver). This mode is optimized for rules insertion.

2. **DMFS** – Device Managed Flow Steering

   Rules insertion is done using firmware commands. This mode is optimized for throughput with a small amount of rules in the system.

   The mode can be controlled via sysfs or devlink API in kernels that support it:

   ```
   Sysfs:
   # echo smfs > /sys/class/net/<PF netdev>/compat/devlink/steering_mode

   Devlink:
   # devlink dev param set pci/0000:00:08.0 name flow_steering_mode value "smfs"
   cmode runtime

   Replace smfs param with dmfs for device managed flow steering
   ```

Notes:

- The mode should be set prior to moving to SwitchDev, by echoing to the sysfs or invoking the devlink command.

- Only when moving to SwitchDev will the driver use the mode set by the previous step.
- Mode cannot be changed after moving to SwitchDev.
- The steering mode is applicable for SwitchDev mode only, meaning it does not affect legacy SR-IOV or other configurations.

Troubleshooting SMFS

mlx5 debugfs was extended to support presenting Software Steering resources: dr_domain including it's tables, matchers and rules.
The interface is read-only.

While dump is being created, new steering rules cannot be inserted/deleted.
The steering information is dumped in the CSV form with the following format:
<object_type>,<object_ID>, <object_info>,...,<object_info>

This data can be read at the following path:
/sys/kernel/debug/mlx5/<BDF>/steering/fdb/<domain_handle>

Example:

```
# cat /sys/kernel/debug/mlx5/0000:82:00.0/steering/fdb/dmn_000018644
3100,0x55caa4621c50,0xee802,4,65533
3101,0x55caa4621c50,0xe0100008
```

You can then use the steering dump parser to make the output more human readable.
The parser can be found in the following public GitHub repository:
https://github.com/Mellanox/mlx_steering_dump

vPort Match Mode

OVS-kernel support two modes that define how the rules on match on vport.

1. **Metadata** – rules match on metadata instead of vport number (default mode).

This mode is needed in order to support SR-IOV Live migration and Dual port RoCE features.

Matching on Metadata can have a performance impact.

2. **Legacy** – rules match on vport number.

   In this mode, performance can be higher in comparison to Metadata. It can still be used only if none of the above features (SR-IOV Live migration and Dual port RoCE) is enabled/used.

   The mode can be controlled via sysfs:

   ```
   Set Legacy:
   # echo legacy > /sys/class/net/<PF netdev>/compat/devlink/vport_match_mode

   Set metadata:
   Devlink:
   # echo metadata > /sys/class/net/<PF
   netdev>/compat/devlink/vport_match_mode
   ```

   Note: This mode should be set prior to moving to SwitchDev, by echoing to the sysfs.

Flow Table Large Group Number

Offloaded flows, including Connection Tracking, are added to Virtual Switch Forwarding Data Base (FDB) flow tables. FDB tables have a set of flow groups, where each flow group saves the same traffic pattern flows. E.g, for connection tracking offloaded flow, TCP and UDP are different traffic patterns which will end up in two different flow groups.

A flow group has a limited size to save flow entries. As default, the driver has 15 big FDB flow groups. Each of these big flow groups can save 4M / (15 + 1) = 256k different 5-tuple flow entries at most. For scenarios with more than 15 traffic patterns, the driver provides a module parameter (num_of_groups) to allow customization and performance tuning.

The mode can be controlled via module param or devlink API for kernels that support it:

```
Module param:
# echo <num_of_groups> > /sys/module/mlx5_core/parameters/num_of_groups

Devlink:
# devlink dev param set pci/0000:82:00.0 name fdb_large_groups \
cmode driverinit value 20
```

Notes:

- In MLNX_OFED v5.1, the default value was changed from 4 to 15.
- The change takes effect immediately if there is no flow inside the FDB table (no traffic running and all offloaded flows are aged out). And it can be dynamically changed without reloading the driver.
  If there are still offloaded flows residual when changing this parameter, it will only take effect after all flows have aged out.

Open vSwitch Configuration

Open vSwitch configuration is a simple OVS bridge configuration with SwitchDev.

1. Run the openvswitch service.

   ```
   # systemctl start openvswitch
   ```

2. Create an OVS bridge (here it's named ovs-sriov).

   ```
   # ovs-vsctl add-br ovs-sriov
   ```

3. Enable hardware offload (disabled by default).

   ```
   # ovs-vsctl set Open_vSwitch . other_config:hw-offload=true
   ```

4. Restart the openvswitch service. This step is required for HW offload changes to take effect.

```
# systemctl restart openvswitch
```

> **Note**
>
> HW offload policy can also be changed by setting the tc-policy using one on the following values:
>
> * none - adds a TC rule to both the software and the hardware (default)
>
> * skip_sw - adds a TC rule only to the hardware
>
> * skip_hw - adds a TC rule only to the software
>
> The above change is used for debug purposes.

5. Add the PF and the VF representor netdevices as OVS ports.

```
# ovs-vsctl add-port ovs-sriov enp4s0f0
# ovs-vsctl add-port ovs-sriov enp4s0f0_0
# ovs-vsctl add-port ovs-sriov enp4s0f0_1
```

Make sure to bring up the PF and representor netdevices.

```
# ip link set dev enp4s0f0 up
# ip link set dev enp4s0f0_0 up
# ip link set dev enp4s0f0_1 up
```

The PF represents the uplink (wire).

```
# ovs-dpctl show
system@ovs-system:
lookups: hit:0 missed:192 lost:1
flows: 2
masks: hit:384 total:2 hit/pkt:2.00
port 0: ovs-system (internal)
port 1: ovs-sriov (internal)
port 2: enp4s0f0
port 3: enp4s0f0_0
port 4: enp4s0f0_1
```

6. Run traffic from the VFs and observe the rules added to the OVS data-path.

```
# ovs-dpctl dump-flows

recirc_id(0),in_port(3),eth(src=e4:11:22:33:44:50,dst=e4:1d:2d:a5:f3:9d),
eth_type(0x0800),ipv4(frag=no), packets:33, bytes:3234, used:1.196s, actions:2

recirc_id(0),in_port(2),eth(src=e4:1d:2d:a5:f3:9d,dst=e4:11:22:33:44:50),
eth_type(0x0800),ipv4(frag=no), packets:34, bytes:3332, used:1.196s, actions:3
```

In the example above, the ping was initiated from VF0 (OVS port 3) to the outer node (OVS port 2), where the VF MAC is e4:11:22:33:44:50 and the outer node MAC is e4:1d:2d:a5:f3:9d
As shown above, two OVS rules were added, one in each direction.
Note that you can also verify offloaded packets by adding type=offloaded to the command. For example:

```
# ovs-appctl dpctl/dump-flows type=offloaded
```

Open vSwitch Performance Tuning

Flow Aging

The aging timeout of OVS is given in ms and can be controlled using the following command.

```
# ovs-vsctl set Open_vSwitch . other_config:max-idle=30000
```

TC Policy

Specifies the policy used with HW offloading.

- **none** - adds a TC rule to both the software and the hardware (default)
- **skip_sw** - adds a TC rule only to the hardware
- **skip_hw** - adds a TC rule only to the software

Example:

```
# ovs-vsctl set Open_vSwitch . other_config:tc-policy=skip_sw
```

Note: TC policy should only be used for debugging purposes.

Max-Revalidator

Specifies the maximum time (in ms) that revalidator threads will wait for kernel statistics before executing flow revalidation.

```
# ovs-vsctl set Open_vSwitch . other_config:max-revalidator=10000
```

n-handler-threads

Specifies the number of threads for software datapaths to use for handling new flows. The default value is the number of online CPU cores minus the number of revalidators.

```
# ovs-vsctl set Open_vSwitch . other_config:n-handler-threads=4
```

n-revalidator-threads

Specifies the number of threads for software datapaths to use for revalidating flows in the datapath.

```
# ovs-vsctl set Open_vSwitch . other_config:n-revalidator-threads=4
```

vlan-limit

Limits the number of VLAN headers that can be matched to the specified number.

```
# ovs-vsctl set Open_vSwitch . other_config:vlan-limit=2
```

Basic TC Rules Configuration

Offloading rules can also be added directly, and not only through OVS, using the tc utility.

To create an offloading rule using TC:

1. Create an ingress qdisc (queueing discipline) for each interface that you wish to add rules into.

   ```
   # tc qdisc add dev enp4s0f0 ingress
   # tc qdisc add dev enp4s0f0_0 ingress
   # tc qdisc add dev enp4s0f0_1 ingress
   ```

2. Add TC rules using flower classifier in the following format.

   ```
   # tc filter add dev NETDEVICE ingress protocol PROTOCOL prio PRIORITY \
   ```

```
[chain CHAIN] flower [ MATCH_LIST ] [ action ACTION_SPEC ]
```

Note: List of supported matches (specifications) and actions can be found in Classification Fields (Matches) section.

3. Dump the existing tc rules using flower classifier in the following format.

```
# tc [ -s ] filter show dev NETDEVICE ingress
```

SR-IOV VF LAG

SR-IOV VF LAG allows the NIC's physical functions (PFs) to get the rules that the OVS will try to offload to the bond net-device, and to offload them to the hardware e-switch. Bond modes supported are:

- Active-Backup
- XOR
- LACP

SR-IOV VF LAG enables complete offload of the LAG functionality to the hardware. The bonding creates a single bonded PF port. Packets from up-link can arrive from any of the physical ports, and will be forwarded to the bond device.

When hardware offload is used, packets from both ports can be forwarded to any of the VFs. Traffic from the VF can be forwarded to both ports according to the bonding state. Meaning, when in active-backup mode, only one PF is up, and traffic from any VF will go through this PF. When in XOR or LACP mode, if both PFs are up, traffic from any VF will split between these two PFs.

SR-IOV VF LAG Configuration on ASAP2

To enable SR-IOV VF LAG, both physical functions of the NIC should first be configured to SR-IOV SwitchDev mode, and only afterwards bond the up-link representors.

The example below shows the creation of bond interface on two PFs:

1. Load bonding device and enslave the up-link representor (currently PF) net-device devices.

```
modprobe bonding mode=802.3ad
Ifup bond0 (make sure ifcfg file is present with desired bond configuration)
ip link set enp4s0f0 master bond0
ip link set enp4s0f1 master bond0
```

2. Add the VF representor net-devices as OVS ports. If tunneling is not used, add the bond device as well.

```
ovs-vsctl add-port ovs-sriov bond0
ovs-vsctl add-port ovs-sriov enp4s0f0_0
ovs-vsctl add-port ovs-sriov enp4s0f1_0
```

3. Make sure to bring up the PF and the representor netdevices.

```
ip link set dev bond0 up
ip link set dev enp4s0f0_0 up
ip link set dev enp4s0f1_0 up
```

Note

Once SR-IOV VF LAG is configured, all VFs of the two PFs will become part of the bond, and will behave as described above.

Limitations

- In VF LAG mode, outgoing traffic in load balanced mode is according to the origin ring, thus, half of the rings will be coupled with port 1 and half with port 2. All the traffic on the same ring will be sent from the same port.

- VF LAG configuration is not supported when the NUM_OF_VFS configured in mlxconfig is higher than 64.

Using TC with VF LAG

Both rules can be added using either of the following.

1. Shared block (supported from kernel 4.16 and RHEL/CentOS 7.7 and above).

```
# tc qdisc add dev bond0 ingress_block 22 ingress
# tc qdisc add dev ens4p0 ingress_block 22 ingress
# tc qdisc add dev ens4p1 ingress_block 22 ingress
```

   1. Add drop rule.

```
# tc filter add block 22 protocol arp parent ffff: prio 3 \
flower \
dst_mac e4:11:22:11:4a:51 \
action drop
```

   2. Add redirect rule from bond to representor.

```
# tc filter add block 22 protocol arp parent ffff: prio 3 \
flower \
dst_mac e4:11:22:11:4a:50 \
action mirred egress redirect dev ens4f0_0
```

   3. Add redirect rule from representor to bond.

```
# tc filter add dev ens4f0_0 protocol arp parent ffff: prio 3 \
flower \
dst_mac ec:0d:9a:8a:28:42 \
```

```
action mirred egress redirect dev bond0
```

2. Without shared block (supported from kernel 4.15 and **below**).
    1. Add redirect rule from bond to representor.

```
# tc filter add dev bond0 protocol arp parent ffff: prio 1 \
flower \
dst_mac e4:11:22:11:4a:50 \
action mirred egress redirect dev ens4f0_0
```

    2. Add redirect rule from representor to bond.

```
# tc filter add dev ens4f0_0 protocol arp parent ffff: prio 3 \
flower \
dst_mac ec:0d:9a:8a:28:42 \
action mirred egress redirect dev bond0
```

Classification Fields (Matches)

OVS-Kernel supports multiple classification fields which packets can fully or partially match.

Ethernet Layer 2

- Destination MAC
- Source MAC
- Ethertype

Supported on all kernels.

In OVS dump flows:

```
skb_priority(0/0),skb_mark(0/0),in_port(eth6),eth(src=00:02:10:40:10:0d
,dst=68:54:ed:00:af:de),eth_type(0x8100), packets:1981, bytes:206024, used:0.440s, dp:tc,
actions:eth7
```

Using TC rules:

```
tc filter add dev $rep parent ffff: protocol arp pref 1 \
flower \
dst_mac e4:1d:2d:5d:25:35 \
src_mac e4:1d:2d:5d:25:34 \
action mirred egress redirect dev $NIC
```

IPv4/IPv6

- Source address
- Destination address
- Protocol
    - TCP/UDP/ICMP/ICMPv6
- TOS
- TTL (HLIMIT)

Supported on all kernels.

In OVS dump flows:

```
Ipv4:
ipv4(src=0.0.0.0/0.0.0.0,dst=0.0.0.0/0.0.0.0,proto=17,tos=0/0,ttl=0/0,frag=no)
Ipv6:
ipv6(src=::/::,dst=1:1:1::3:1040:1008,label=0/0,proto=58,tclass=0/0x3,hlimit=64),
```

Using TC rules:

```
IPv4:
tc filter add dev $rep parent ffff: protocol ip pref 1 \
flower \
dst_ip 1.1.1.1 \
src_ip 1.1.1.2 \
ip_proto TCP \
ip_tos 0x3 \
ip_ttl 63 \
action mirred egress redirect dev $NIC


IPv6:
tc filter add dev $rep parent ffff: protocol ipv6 pref 1 \
flower \
dst_ip 1:1:1::3:1040:1009 \
src_ip 1:1:1::3:1040:1008 \
ip_proto TCP \
ip_tos 0x3 \
ip_ttl 63\
action mirred egress redirect dev $NIC
```

TCP/UDP Source and Destination ports & TCP Flags

- TCP/UDP source and destinations ports
- TCP flags

Supported kernels are kernel > 4.13 and RHEL > 7.5

In OVS dump flows:

```
TCP: tcp(src=0/0,dst=32768/0x8000),
UDP: udp(src=0/0,dst=32768/0x8000),
TCP flags: tcp_flags(0/0)
```

Using TC rules:

```
tc filter add dev $rep parent ffff: protocol ip pref 1 \
flower \
ip_proto TCP \
dst_port 100 \
src_port 500 \
tcp_flags 0x4/0x7 \
action mirred egress redirect dev $NIC
```

VLAN

- ID
- Priority
- Inner vlan ID and Priority

Supported kernels: All (QinQ: kernel 4.19 and higher, and RHEL 7.7 and higher)

In OVS dump flows:

```
eth_type(0x8100),vlan(vid=2347,pcp=0),
```

Using TC rules:

```
tc filter add dev $rep parent ffff: protocol 802.1Q pref 1 \
flower \
vlan_ethtype 0x800 \
vlan_id 100 \
vlan_prio 0 \
action mirred egress redirect dev $NIC
QinQ:
tc filter add dev $rep parent ffff: protocol 802.1Q pref 1 \
flower \
```

```
vlan_ethtype 0x8100 \
vlan_id 100 \
vlan_prio 0 \
cvlan_id 20 \
cvlan_prio 0 \
cvlan_ethtype 0x800 \
action mirred egress redirect dev $NIC
```

Tunnel

- ID (Key)
- Source IP address
- Destination IP address
- Destination port
- TOS (supported from kernel 4.19 and above & RHEL 7.7 and above)
- TTL (support from kernel 4.19 and above & RHEL 7.7 and above)
- Tunnel options (Geneve)

Supported kernels:

- VXLAN: All
- GRE: Kernel > 5.0, RHEL 7.7 and above
- Geneve: Kernel > 5.0, RHEL 7.7 and above

In OVS dump flows:

```
tunnel(tun_id=0x5,src=121.9.1.1,dst=131.10.1.1,ttl=0/0,tp_dst=4789,flags(+key))
```

Using TC rules:

```
# tc filter add dev $rep protocol 802.1Q parent ffff: pref 1
flower \
vlan_ethtype 0x800 \
vlan_id 100 \
vlan_prio 0 \
action mirred egress redirect dev $NIC
QinQ:
# tc filter add dev vxlan100 protocol ip parent ffff: \
flower \
skip_sw \
dst_mac e4:11:22:11:4a:51 \
src_mac e4+:11:22:11:4a:50 \
enc_src_ip 20.1.11.1 \
enc_dst_ip 20.1.12.1 \
enc_key_id 100 \
enc_dst_port 4789 \
action tunnel_key unset \
action mirred egress redirect dev ens4f0_0
```

Supported Actions

Forward

Forward action allows for packet redirection:

- From VF to wire
- Wire to VF
- VF to VF

Supported on all kernels.

In OVS dump flows:

```
skb_priority(0/0),skb_mark(0/0),in_port(eth6),eth(src=00:02:10:40:10:0d
,dst=68:54:ed:00:af:de),eth_type(0x8100), packets:1981, bytes:206024, used:0.440s, dp:tc,
actions:eth7
```

Using TC rules:

```
tc filter add dev $rep parent ffff: protocol arp pref 1 \
flower \
dst_mac e4:1d:2d:5d:25:35 \
src_mac e4:1d:2d:5d:25:34 \
action mirred egress redirect dev $NIC
```

Drop

Drop action allows to drop incoming packets.

Supported on all kernels.

In OVS dump flows:

```
skb_priority(0/0),skb_mark(0/0),in_port(eth6),eth(src=00:02:10:40:10:0d
,dst=68:54:ed:00:af:de),eth_type(0x8100), packets:1981, bytes:206024, used:0.440s, dp:tc,
actions:drop
```

Using TC rules:

```
tc filter add dev $rep parent ffff: protocol arp pref 1 \
flower \
dst_mac e4:1d:2d:5d:25:35 \
src_mac e4:1d:2d:5d:25:34 \
action drop
```

Statistics

By default, each flow collects the following statistics:

- Packets – number of packets which hit the flow
- Bytes – total number of bytes which hit the flow
- Last used – the amount of time passed since last packet hit the flow

Supported on all kernels.

In OVS dump flows:

```
skb_priority(0/0),skb_mark(0/0),in_port(eth6),eth(src=00:02:10:40:10:0d
,dst=68:54:ed:00:af:de),eth_type(0x8100), packets:1981, bytes:206024, used:0.440s, dp:tc,
actions:drop
```

Using TC rules:

```
#tc -s filter show dev $rep ingress

filter protocol ip pref 2 flower chain 0
filter protocol ip pref 2 flower chain 0 handle 0x2
eth_type ipv4
ip_proto tcp
src_ip 192.168.140.100
src_port 80
skip_sw
in_hw
action order 1: mirred (Egress Redirect to device p0v11_r) stolen
index 34 ref 1 bind 1 installed 144 sec used 0 sec
Action statistics:
Sent 388344 bytes 2942 pkt (dropped 0, overlimits 0 requeues 0)
backlog 0b 0p requeues 0
```

Tunnels (Encapsulation/Decapsulation)

OVS-kernel supports offload of tunnels using encapsulation and decapsulation actions.

- Encapsulation – pushing of tunnel header is supported on Tx
- Decapsulation – popping of tunnel header is supported on Rx

Supported Tunnels:

- VXLAN (IPv4/IPv6) – supported on all Kernels
- GRE (IPv4/IPv6) – supported on kernel 5.0 and above & RHEL 7.6 and above
- Geneve (IPv4/IPv6) - supported on kernel 5.0 and above & RHEL 7.6 and above

OVS configuration:
In case of offloading tunnel, the PF/bond should not be added as a port in the OVS datapath. It should rather be assigned with the IP address to be used for encapsulation. The example below shows two hosts (PFs) with IPs 1.1.1.177 and 1.1.1.75, where the PF device on both hosts is enp4s0f0, and the VXLAN tunnel is set with VNID 98:

- On the first host:

```
# ip addr add 1.1.1.177/24 dev enp4s0f1

# ovs-vsctl add-port ovs-sriov vxlan0 -- set interface vxlan0 type=vxlan
options:local_ip=1.1.1.177 options:remote_ip=1.1.1.75 options:key=98
```

- On the second host:

```
# ip addr add 1.1.1.75/24 dev enp4s0f1

# ovs-vsctl add-port ovs-sriov vxlan0 -- set interface vxlan0 type=vxlan
options:local_ip=1.1.1.75 options:remote_ip=1.1.1.177 options:key=98
```

- for GRE IPv4 tunnel need use type=gre
- for GRE IPv6 tunnel need use type=ip6gre
- for GENEVE tunnel need use type=geneve

Note

When encapsulating guest traffic, the VF's device MTU must be reduced to allow the host/HW to add the encap headers without fragmenting the resulted packet. As such, the VF's MTU must be lowered by 50 bytes from the uplink MTU for IPv4 and 70 bytes for IPv6.

Tunnel offload using TC rules:

```
Encapsulation:
# tc filter add dev ens4f0_0 protocol 0x806 parent ffff: \
flower \
skip_sw \
dst_mac e4:11:22:11:4a:51 \
src_mac e4:11:22:11:4a:50 \
action tunnel_key set \
src_ip 20.1.12.1 \
dst_ip 20.1.11.1 \
id 100 \
action mirred egress redirect dev vxlan100

Decapsulation:
# tc filter add dev vxlan100 protocol 0x806 parent ffff: \
flower \
skip_sw \
dst_mac e4:11:22:11:4a:51 \
src_mac e4:11:22:11:4a:50 \
```

```
enc_src_ip 20.1.11.1 \
enc_dst_ip 20.1.12.1 \
enc_key_id 100 \
enc_dst_port 4789 \
action tunnel_key unset \
action mirred egress redirect dev ens4f0_0
```

VLAN Push/Pop

OVS-kernel supports offload of vlan header push/pop actions.

- Push—pushing of VLAN header is supported on Tx
- Pop—popping of tunnel header is supported on Rx

> **Note**
>
> Starting with ConnectX-6 Dx hardware models and above, pushing of VLAN header is also supported on Rx, and popping of VLAN header is also supported on Tx.

OVS Configuration

Add a tag=$TAG section for the OVS command line that adds the representor ports. For example, VLAN ID 52 is being used here.

```
# ovs-vsctl add-port ovs-sriov enp4s0f0
# ovs-vsctl add-port ovs-sriov enp4s0f0_0 tag=52
# ovs-vsctl add-port ovs-sriov enp4s0f0_1 tag=52
```

The PF port should not have a VLAN attached. This will cause OVS to add VLAN push/pop actions when managing traffic for these VFs.

Dump Flow Example

```
recirc_id(0),in_port(3),eth(src=e4:11:22:33:44:50,dst=00:02:c9:e9:bb:b2),eth_type(0x0800),ip
\
packets:0, bytes:0, used:never, actions:push_vlan(vid=52,pcp=0),2

recirc_id(0),in_port(2),eth(src=00:02:c9:e9:bb:b2,dst=e4:11:22:33:44:50),eth_type(0x8100),
\
vlan(vid=52,pcp=0),encap(eth_type(0x0800),ipv4(frag=no)), packets:0, bytes:0,
used:never, actions:pop_vlan,3
```

VLAN Offload using TC Rules Example

```
# tc filter add dev ens4f0_0 protocol ip parent ffff: \
flower \
skip_sw \
dst_mac e4:11:22:11:4a:51 \
src_mac e4:11:22:11:4a:50 \
action vlan push id 100 \
action mirred egress redirect dev ens4f0

# tc filter add dev ens4f0 protocol 802.1Q parent ffff: \
flower \
skip_sw \
dst_mac e4:11:22:11:4a:51 \
src_mac e4:11:22:11:4a:50 \
vlan_ethtype 0x800 \
vlan_id 100 \
vlan_prio 0 \
action vlan pop \
action mirred egress redirect dev ens4f0_0
```

TC Configuration for ConnectX-6 Dx and Above

Example of VLAN Offloading with popping header on Tx and pushing on Rx using TC Rules:

```
# tc filter add dev ens4f0_0 ingress protocol 802.1Q parent ffff: \
      flower \
            vlan_id 100 \
      action vlan pop \
      action tunnel_key set \
            src_ip 4.4.4.1 \
            dst_ip 4.4.4.2 \
            dst_port 4789 \
            id 42 \
      action mirred egress redirect dev vxlan0

# tc filter add dev vxlan0 ingress protocol all parent ffff: \
      flower \
            enc_dst_ip 4.4.4.1 \
            enc_src_ip 4.4.4.2 \
            enc_dst_port 4789 \
            enc_key_id 42 \
      action tunnel_key unset \
      action vlan push id 100 \
       action mirred egress redirect dev ens4f0_0
```

Header Rewrite

This action allows for modifying packet fields.

Ethernet Layer 2

- Destination MAC
- Source MAC

Supported kernels: Kernel 4.14 and above & RHEL 7.5 and above

In OVS dump flows:

```
skb_priority(0/0),skb_mark(0/0),in_port(eth6),eth(src=00:02:10:40:10:0d
,dst=68:54:ed:00:af:de),eth_type(0x8100), packets:1981, bytes:206024, used:0.440s, dp:tc,
actions: set(eth(src=68:54:ed:00:f4:ab,dst=fa:16:3e:dd:69:c4)),eth7
```

Using TC rules:

```
tc filter add dev $rep parent ffff: protocol arp pref 1 \
flower \
dst_mac e4:1d:2d:5d:25:35 \
src_mac e4:1d:2d:5d:25:34 \
action pedit ex \
munge eth dst set 20:22:33:44:55:66 \
munge eth src set aa:ba:cc:dd:ee:fe \
action mirred egress redirect dev $NIC
```

IPv4/IPv6

- Source address
- Destination address
- Protocol
- TOS
- TTL (HLIMIT)

Supported kernels: Kernel 4.14 and above & RHEL 7.5 and above

In OVS dump flows:

```
Ipv4:
set(eth(src=de:e8:ef:27:5e:45,dst=00:00:01:01:01:01)),
set(ipv4(src=10.10.0.111,dst=10.20.0.122,ttl=63))
```

```
Ipv6:
set(ipv6(dst=2001:1:6::92eb:fcbe:f1c8,hlimit=63)),
```

Using TC rules:

```
IPv4:
tc filter add dev $rep parent ffff: protocol ip pref 1 \
flower \
dst_ip 1.1.1.1 \
src_ip 1.1.1.2 \
ip_proto TCP \
ip_tos 0x3 \
ip_ttl 63 \
pedit ex \
munge ip src set 2.2.2.1 \
munge ip dst set 2.2.2.2 \
munge ip tos set 0 \
munge ip ttl dec \
action mirred egress redirect dev $NIC


IPv6:
tc filter add dev $rep parent ffff: protocol ipv6 pref 1 \
flower \
dst_ip 1:1:1::3:1040:1009 \
src_ip 1:1:1::3:1040:1008 \
ip_proto tcp \
ip_tos 0x3 \
ip_ttl 63\
pedit ex \
munge ipv6 src set 2:2:2::3:1040:1009 \
munge ipv6 dst set 2:2:2::3:1040:1008 \
munge ipv6 hlimit dec \
action mirred egress redirect dev $NIC
```

> **Note**
>
> IPv4 and IPv6 header rewrite is only supported with match on UDP/TCP/ICMP protocols.

TCP/UDP Source and Destination Ports

- TCP/UDP source and destinations ports

Supported kernels: kernel > 4.16 & RHEL > 7.6

In OVS dump flows:

```
TCP:
set(tcp(src= 32768/0xffff,dst=32768/0xffff)),
UDP:

set(udp(src= 32768/0xffff,dst=32768/0xffff)),
```

Using TC rules:

```
TCP:

tc filter add dev $rep parent ffff: protocol ip pref 1 \
flower \
dst_ip 1.1.1.1 \
src_ip 1.1.1.2 \
ip_proto tcp \
ip_tos 0x3 \
ip_ttl 63 \
```

```
pedit ex \
pedit ex munge ip tcp sport set 200
pedit ex munge ip tcp dport set 200
action mirred egress redirect dev $NIC

UDP:
tc filter add dev $rep parent ffff: protocol ip pref 1 \
flower \
dst_ip 1.1.1.1 \
src_ip 1.1.1.2 \
ip_proto udp \
ip_tos 0x3 \
ip_ttl 63 \
pedit ex \
pedit ex munge ip udp sport set 200
pedit ex munge ip udp dport set 200
action mirred egress redirect dev $NIC
```

VLAN

- ID

Supported on all kernels.

In OVS dump flows:

```
Set(vlan(vid=2347,pcp=0/0)),
```

Using TC rules:

```
tc filter add dev $rep parent ffff: protocol 802.1Q pref 1 \
flower \
vlan_ethtype 0x800 \
vlan_id 100 \
vlan_prio 0 \
```

```
action vlan modify id 11 pipe
action mirred egress redirect dev $NIC
```

Connection Tracking

The TC connection tracking action performs connection tracking lookup by sending the packet to netfilter conntrack module. Newly added connections may be associated, via the ct commit action, with a 32 bit mark, 128 bit label and source/destination NAT values.

The following example allows ingress tcp traffic from the uplink representor to vf1_rep, while assuring that egress traffic from vf1_rep is only allowed on established connections. In addition, mark and source IP NAT is applied.

In OVS dump flows:

```
ct(zone=2,nat)
ct_state(+est+trk)
actions:ct(commit,zone=2,mark=0x4/0xffffffff,nat(src=5.5.5.5))
```

Using TC rules:

```
# tc filter add dev $uplink_rep ingress chain 0 prio 1 proto ip \
flower \
ip_proto tcp \
ct_state -trk \
action ct zone 2 nat pipe
action goto chain 2
# tc filter add dev $uplink_rep ingress chain 2 prio 1 proto ip \
flower \
ct_state +trk+new \
action ct zone 2 commit mark 0xbb nat src addr 5.5.5.7 pipe \
action mirred egress redirect dev $vf1_rep
# tc filter add dev $uplink_rep ingress chain 2 prio 1 proto ip \
flower \
ct_zone 2 \
```

```
ct_mark 0xbb \
ct_state +trk+est \
action mirred egress redirect dev $vf1_rep

#Setup filters on $vf1_rep, allowing only established connections of zone 2 through, and reverse nat (dst nat in this case)
# tc filter add dev $vf1_rep ingress chain 0 prio 1 proto ip \
flower \
ip_proto tcp \
ct_state -trk \
action ct zone 2 nat pipe \
action goto chain 1
# tc filter add dev $vf1_rep ingress chain 1 prio 1 proto ip \
flower \
ct_zone 2 \
ct_mark 0xbb \
ct_state +trk+est \
action mirred egress redirect dev eth0
```

Connection Tracking Performance Tuning

- Max offloaded connections—specifies the limit on the number of offloaded connections.
  Example:

  ```
  # devlink dev param set pci/${pci_dev} name ct_max_offloaded_conns value $max cmode runtime
  ```

- Allow mixed NAT/non-NAT CT—allows offloading of the following scenario:

  ```
  • cookie=0x0, duration=21.843s, table=0, n_packets=4838718, n_bytes=241958846, ct_state=-trk,ip,in_port=enp8s0f0 actions=ct(table=1,zone=2)
  • cookie=0x0, duration=21.823s, table=1, n_packets=15363, n_bytes=773526, ct_state=+new+trk,ip,in_port=enp8s0f0 actions=ct(commit,zone=2,nat(dst=11.11.11.11)),output:"enp8s0f0_1"
  ```

```
• cookie=0x0, duration=21.806s, table=1, n_packets=4767594, n_bytes=238401190,
ct_state=+est+trk,ip,in_port=enp8s0f0
actions=ct(zone=2,nat),output:"enp8s0f0_1"
```

Example:

```
# echo enable >
/sys/class/net/<device>/compat/devlink/ct_action_on_nat_conns
```

Forward to Chain (TC Only)

TC interface supports adding flows on different chains. Only chain 0 is accessed by default. Access to the other chains requires usafe of the goto action.

In this example, a flow is created on chain 1 without any match and redirect to wire.

The second flow is created on chain 0 and match on source MAC and action goto chain 1.

This example simulates simple MAC spoofing.

```
#tc filter add dev $rep parent ffff: protocol all chain 1 pref 1 \
flower \
action mirred egress redirect dev $NIC

#tc filter add dev $rep parent ffff: protocol all chain 1 pref 1 \
flower \
src_mac aa:bb:cc:aa:bb:cc
action goto chain 1
```

Port Mirroring (Flow Based VF Traffic Mirroring for ASAP²)

Unlike para-virtual configurations, when the VM traffic is offloaded to the hardware via SR-IOV VF, the host side Admin cannot snoop the traffic (e.g. for monitoring).

ASAP² uses the existing mirroring support in OVS and TC along with the enhancement to the offloading logic in the driver to allow mirroring the VF traffic to another VF.

The mirrored VF can be used to run traffic analyzer (tcpdump, wireshark, etc) and observe the traffic of the VF being mirrored.

The example below shows the creation of port mirror on the following configuration:

```
# ovs-vsctl show
09d8a574-9c39-465c-9f16-47d81c12f88a
Bridge br-vxlan
Port "enp4s0f0_1"
Interface "enp4s0f0_1"
Port "vxlan0"
Interface "vxlan0"
type: vxlan
options: {key="100", remote_ip="192.168.1.14"}
Port "enp4s0f0_0"
Interface "enp4s0f0_0"
Port "enp4s0f0_2"
Interface "enp4s0f0_2"
Port br-vxlan
Interface br-vxlan
type: internal
ovs_version: "2.14.1"
```

- To set **enp4s0f0_0** as the mirror port, and mirror all of the traffic:

```
# ovs-vsctl -- --id=@p get port enp4s0f0_0 \
-- --id=@m create mirror name=m0 select-all=true output-port=@p \
-- set bridge br-vxlan mirrors=@m
```

- To set **enp4s0f0_0** as the mirror port, and only mirror the traffic, the destination is enp4s0f0_1:

```
# ovs-vsctl -- --id=@p1 get port enp4s0f0_0 \
-- --id=@p2 get port enp4s0f0_1 \
-- --id=@m create mirror name=m0 select-dst-port=@p2 output-port=@p1 \
-- set bridge br-vxlan mirrors=@m
```

- To set enp4s0f0_0 as the mirror port, and only mirror the traffic the source is enp4s0f0_1:

```
# ovs-vsctl -- --id=@p1 get port enp4s0f0_0 \
-- --id=@p2 get port enp4s0f0_1 \
-- --id=@m create mirror name=m0 select-src-port=@p2 output-port=@p1 \
-- set bridge br-vxlan mirrors=@m
```

- To set enp4s0f0_0 as the mirror port and mirror, all the traffic on enp4s0f0_1:

```
# ovs-vsctl -- --id=@p1 get port enp4s0f0_0 \
-- --id=@p2 get port enp4s0f0_1 \
-- --id=@m create mirror name=m0 select-dst-port=@p2 select-src-port=@p2
output-port=@p1 \
-- set bridge br-vxlan mirrors=@m
```

To clear the mirror port:

```
# ovs-vsctl clear bridge br-vxlan mirrors
```

Mirroring using TC:

```
Mirror to VF
tc filter add dev $rep parent ffff: protocol arp pref 1 \
flower \
dst_mac e4:1d:2d:5d:25:35 \
```

```
src_mac e4:1d:2d:5d:25:34 \
action mirred egress mirror dev $mirror_rep pipe \
action mirred egress redirect dev $NIC

Mirror to tunnel:
tc filter add dev $rep parent ffff: protocol arp pref 1 \
flower \
dst_mac e4:1d:2d:5d:25:35 \
src_mac e4:1d:2d:5d:25:34 \
action tunnel_key set \
src_ip 1.1.1.1 \
dst_ip 1.1.1.2 \
dst_port 4789 \
id 768 \
pipe \
action mirred egress mirror dev vxlan100 pipe \
action mirred egress redirect dev $NIC
```

Forward to Multiple Destinations

Forward to up 32 destinations (representors and tunnels) is supported using TC.

Example 1: forward to 32 VFs.

```
tc filter add dev $NIC parent ffff: protocol arp pref 1 \
                flower \
                dst_mac e4:1d:2d:5d:25:35 \
                src_mac e4:1d:2d:5d:25:34 \
                action mirred egress mirror dev $rep0 pipe \
                action mirred egress mirror dev $rep1 pipe \
...
                action mirred egress mirror dev $rep30 pipe \
```

```
                action mirred egress redirect dev $rep31
```

Example 2: forward to 16 tunnels.

```
tc filter add dev $rep parent ffff: protocol arp pref 1 \
                flower \
                dst_mac e4:1d:2d:5d:25:35 \
                src_mac e4:1d:2d:5d:25:34 \
                action tunnel_key set src_ip $ip_src dst_ip $ip_dst \
dst_port 4789 id 0 nocsum \
pipe action mirred egress mirror dev vxlan0 pipe \
                action tunnel_key set src_ip $ip_src dst_ip $ip_dst \
dst_port 4789 id 1 nocsum \
pipe action mirred egress mirror dev vxlan0 pipe \
...
                action tunnel_key set src_ip $ip_src dst_ip $ip_dst \
dst_port 4789 id 15 nocsum \
pipe action mirred egress redirect dev vxlan0
```

Note

- TC supports up to 32 actions
- If header rewrite is used, then all destinations should have the same header rewrite
- If VLAN push/pop is used, then all destinations should have the same VLAN ID and actions

sFLOW

This feature allows for monitoring traffic sent between two VMs on the same host using an sFlow collector.

The example below assumes the environment is configured as described below.

```
# ovs-vsctl show
09d8a574-9c39-465c-9f16-47d81c12f88a
Bridge br-vxlan
Port "enp4s0f0_1"
Interface "enp4s0f0_1"
Port "vxlan0"
Interface "vxlan0"
type: vxlan
options: {key="100", remote_ip="192.168.1.14"}
Port "enp4s0f0_0"
Interface "enp4s0f0_0"
Port "enp4s0f0_2"
Interface "enp4s0f0_2"
Port br-vxlan
Interface br-vxlan
type: internal
ovs_version: "2.14.1"
```

To sample all traffic over the OVS bridge:

```
# ovs-vsctl -- --id=@sflow create sflow agent=\"$SFLOW_AGENT\" \
target=\"$SFLOW_TARGET:$SFLOW_PORT\" header=$SFLOW_HEADER \
sampling=$SFLOW_SAMPLING polling=10 \
-- set bridge br-vxlan sflow=@sflow
```

Parameter	Description
SFLOW_AGENT	Indicates that the sFlow agent should send traffic from SFLOW_AGENT's IP address

Parameter	Description
SFLOW_TARGET	Remote IP address of the sFLOW collector
SFLOW_HEADER	Size of packet header to sample (in bytes)
SFLOW_SAMPLING	Sample rate

To clear the sFLOW configuration:

```
# ovs-vsctl clear bridge br-vxlan sflow
```

To list the sFLOW configuration:

```
# ovs-vsctl list sflow
```

sFLOW using TC:

```
Sample to VF
tc filter add dev $rep parent ffff: protocol arp pref 1 \
flower \
dst_mac e4:1d:2d:5d:25:35 \
src_mac e4:1d:2d:5d:25:34 \
action sample rate 10 group 5 trunc 96 \
action mirred egress redirect dev $NIC
```

Note

Userspace application is needed in order to process to sampled packet from the kernel. Example: https://github.com/Mellanox/libpsample

Rate Limit

OVS-kernel supports offload of VF rate limit using OVS configuration and TC.

The example below sets a rate limit to the VF related to representor eth0 to 10Mbps.

```
OVS:
# ovs-vsctl set interface eth0 ingress_policing_rate=10000

tc:
# tc_filter add dev eth0 root prio 1 protocol ip matchall skip_sw action police rate
10mbit burst 20k
```

Kernel Requirements

This kernel config should be enabled in order to support switchdev offload.

- CONFIG_NET_ACT_CSUM – needed for action csum
- CONFIG_NET_ACT_PEDIT – needed for header rewrite
- CONFIG_NET_ACT_MIRRED – needed for basic forward
- CONFIG_NET_ACT_CT – needed for connection tracking (supported from kernel 5.6)
- CONFIG_NET_ACT_VLAN - needed for action vlan push/pop
- CONFIG_NET_ACT_GACT
- CONFIG_NET_CLS_FLOWER
- CONFIG_NET_CLS_ACT
- CONFIG_NET_SWITCHDEV
- CONFIG_NET_TC_SKB_EXT - needed for connection tracking (supported from kernel 5.6)

- CONFIG_NET_ACT_CT - needed for connection tracking (supported from kernel 5.6)
- CONFIG_NFT_FLOW_OFFLOAD
- CONFIG_NET_ACT_TUNNEL_KEY
- CONFIG_NF_FLOW_TABLE - needed for connection tracking (supported from kernel 5.6)
- CONFIG_SKB_EXTENSIONS - needed for connection tracking (supported from kernel 5.6)
- CONFIG_NET_CLS_MATCHALL
- CONFIG_NET_ACT_POLICE
- CONFIG_MLX5_ESWITCH

VF Metering

OVS-kernel supports offloading of VF metering (TX and RX) using sysfs. Metering of number of packets per second (PPS) and bytes per second (BPS) is supported.

The example bellow sets Rx meter on VF 0 with value 10Mbps BPS.

```
echo 10000000 > /sys/class/net/enp4s0f0/device/sriov/0/meters/rx/bps/rate
echo 65536 > /sys/class/net/enp4s0f0/device/sriov/0/meters/rx/bps/burst
```

The example bellow sets Tx meter on VF 0 with value 1000 PPS.

```
echo 1000 > /sys/class/net/enp4s0f0/device/sriov/0/meters/tx/pps/rate
echo 100 > /sys/class/net/enp4s0f0/device/sriov/0/meters/tx/pps/burst
```

Note

Both rate and burst must not be zero and burst may need to be adjusted according to the requirements.

The following counters can be used to query the number dropped packet/bytes:

```
#cat /sys/class/net/enp8s0f0/device/sriov/0/meters/rx/pps/packets_dropped
#cat /sys/class/net/enp8s0f0/device/sriov/0/meters/rx/pps/bytes_dropped
#cat /sys/class/net/enp8s0f0/device/sriov/0/meters/rx/bps/packets_dropped
#cat /sys/class/net/enp8s0f0/device/sriov/0/meters/rx/bps/bytes_dropped
#cat /sys/class/net/enp8s0f0/device/sriov/0/meters/tx/pps/packets_dropped
#cat /sys/class/net/enp8s0f0/device/sriov/0/meters/tx/pps/bytes_dropped
#cat /sys/class/net/enp8s0f0/device/sriov/0/meters/tx/bps/packets_dropped
#cat /sys/class/net/enp8s0f0/device/sriov/0/meters/tx/bps/bytes_dropped
```

Representor Metering

Metering for uplink and VF representors traffic support has been added.

Traffic going to a representor device can be a result of a miss in the embedded switch (eSwitch) FDB tables. This means that a packet which arrived from that representor into the eSwitch was not matched against the existing rules in the hardware FDB tables and needs to be forwarded to software to be handled there and is, therefore, forwarded to the originating representor device driver.

The meter allows to configure the max rate [packets/sec] and max burst [packets] for traffic going to the representor driver. Any traffic exceeding values provided by the user will be dropped in hardware. There are statistics that show number of dropped packets.

The configuration of a representors metering is done via a new sysfs called `miss_rl_cfg`.

- Full path of the miss_rl_cfg parameter: /sys/class/net//rep_config/miss_rl_cfg

- Usage: echo "<rate> <burst>" > /sys/class/net//rep_config/miss_rl_cfg. Rate is the max rate of packets allowed for this representor (in packets/sec units) and burst is the max burst size allowed for this representor (in packets units). Both values must be specified. The default is 0 for both, meaning unlimited rate and burst.

To view the amount of packets and bytes that were dropped due to traffic exceeding the user-provided rate and burst, two read-only sysfs for statistics are exposed.

- /sys/class/net//rep_config/miss_rl_dropped_bytes counts how many FDB-miss bytes were dropped due to reaching the miss limits

- /sys/class/net//rep_config/miss_rl_dropped_packets counts how many FDB-miss packets were dropped due to reaching the miss limits

Open vSwitch Metering

There are two types of meters, kpps (kilobits per second) and pktps(packets per second), which are described in Meter Syntax of OpenFlow 1.3+ Switch Meter Table Commands. OVS-Kernel supports offloading them both.

The example below is to offload a kpps meter. Please follow the steps after doing basic configurations as described in section 5.1.3.

1. Create OVS meter with a target rate.

```
ovs-ofctl -O OpenFlow13 add-meter ovs-sriov
meter=1,kbps,band=type=drop,rate=204800
```

2. Delete the default rule.

```
ovs-ofctl del-flows ovs-sriov
```

3. Configure OpenFlow rules. Here VF bandwidth on the receiving side will be limited by the rate configured in step 1.

```
ovs-ofctl -O OpenFlow13 add-flow ovs-sriov 'ip,dl_dst=e4:11:22:33:44:50,actions=
meter:1,output:enp4s0f0_0'
```

```
ovs-ofctl -O OpenFlow13 add-flow ovs-sriov 'ip,dl_src=e4:11:22:33:44:50,actions=
output:enp4s0f0'
ovs-ofctl -O OpenFlow13 add-flow ovs-sriov 'arp,actions=normal'
```

4. Run iperf server and be ready to receive UDP traffic. On the outer node, run iperf client to send UDP traffic to this VF. After traffic starts, check the offloaded meter rule.

```
ovs-appctl dpctl/dump-flows --names type=offloaded

recirc_id(0),in_port(enp4s0f0),eth(dst=e4:11:22:33:44:50),eth_type(0x0800),ipv4(frag=
packets:11626587, bytes:17625889188, used:0.470s, actions:meter(0),enp4s0f0_0
```

In order to verify metering, iperf client should set the target bandwidth with a number which is larger than the meter rate configured. Then it will be visible that packets are received with the limited rate on the server side and the extra packets are dropped by hardware.

Multiport eSwitch Mode

The multiport eSwitch mode allows to add rules on a VF representor with an action forwarding the packet to the physical port of the physical function. This can be used to implement failover or forward packets based on external information such the cost of the route.

1. To configure this more, the nvconig parameter LAG_RESOURCE_ALLOCATION must be set.

2. After the driver loads, configure multiport eSwitch for each PF where enp8s0f0 and enp8s0f1 represent the netdevices for the PFs.

```
echo multiport_esw >
/sys/class/net/enp8s0f0/compat/devlink/lag_port_select_mode
echo multiport_esw >
/sys/class/net/enp8s0f1/compat/devlink/lag_port_select_mode
```

The mode becomes operational after entering switchdev mode on both PFs.

Rule example:

```
tc filter add dev enp8s0f0_0 prot ip root flower dst_ip 7.7.7.7 action mirred egress redirect dev enp8s0f1
```

OVS-DPDK Hardware Offloads

OVS-DPDK Hardware Offloads Configuration

To configure OVS-DPDK HW offloads:

1. Unbind the VFs.

   ```
   echo 0000:04:00.2 > /sys/bus/pci/drivers/mlx5_core/unbind
   echo 0000:04:00.3 > /sys/bus/pci/drivers/mlx5_core/unbind
   ```

   Note: VMs with attached VFs must be powered off to be able to unbind the VFs.

2. Change the e-switch mode from Legacy to SwitchDev on the PF device (make sure all VFs are unbound). This will also create the VF representor netdevices in the host OS.

   ```
   echo switchdev > /sys/class/net/enp4s0f0/compat/devlink/mode
   ```

   To revert to SR-IOV Legacy mode:

   ```
   echo legacy > /sys/class/net/enp4s0f0/compat/devlink/mode
   ```

   Note that running this command will also result in the removal of the VF representor netdevices.

3. Bind the VFs.

```
echo 0000:04:00.2 > /sys/bus/pci/drivers/mlx5_core/bind
echo 0000:04:00.3 > /sys/bus/pci/drivers/mlx5_core/bind
```

4. Run the Open vSwitch service.

```
systemctl start openvswitch
```

5. Enable hardware offload (disabled by default).

```
ovs-vsctl --no-wait set Open_vSwitch . other_config:dpdk-init=true
ovs-vsctl set Open_vSwitch . other_config:hw-offload=true
```

6. Configure the DPDK white list.

```
ovs-vsctl --no-wait set Open_vSwitch . other_config:dpdk-extra="-a
0000:01:00.0,representor=[0],dv_flow_en=1,dv_esw_en=1,dv_xmeta_en=1"
```

> **Note**
>
> Representor=[0-N]

7. Restart the Open vSwitch service. This step is required for HW offload changes to take effect.

```
systemctl restart openvswitch
```

8. Create OVS-DPDK bridge.

```
ovs-vsctl --no-wait add-br br0-ovs -- set bridge br0-ovs datapath_type=netdev
```

9. Add PF to OVS.

```
ovs-vsctl add-port br0-ovs pf -- set Interface pf type=dpdk options:dpdk-devargs=0000:88:00.0
```

10. Add representor to OVS.

```
ovs-vsctl add-port br0-ovs representor -- set Interface representor type=dpdk options:dpdk-devargs=0000:88:00.0,representor=[0]
```

> **Note**
>
> Representor=[0-N]

Offloading VXLAN Encapsulation/Decapsulation Actions

vSwitch in userspace rather than kernel-based Open vSwitch requires an additional bridge. The purpose of this bridge is to allow use of the kernel network stack for routing and ARP resolution.

The datapath needs to look-up the routing table and ARP table to prepare the tunnel header and transmit data to the output port.

Configuring VXLAN Encap/Decap Offloads

> **Note**
>
> The configuration is done with:
>
> - PF on 0000:03:00.0 PCI and MAC 98:03:9b:cc:21:e8
> - Local IP 56.56.67.1 - br-phy interface will be configured to this IP

- Remote IP 56.56.68.1

To configure OVS-DPDK VXLAN:

1. Create a br-phy bridge.

```
ovs-vsctl add-br br-phy -- set Bridge br-phy datapath_type=netdev -- br-set-external-id br-phy bridge-id br-phy -- set bridge br-phy fail-mode=standalone other_config:hwaddr=98:03:9b:cc:21:e8
```

2. Attach PF interface to br-phy bridge.

```
ovs-vsctl add-port br-phy p0 -- set Interface p0 type=dpdk options:dpdk-devargs=0000:03:00.0
```

3. Configure IP to the bridge.

```
ip addr add 56.56.67.1/24 dev br-phy
```

4. Create a br-ovs bridge.

```
ovs-vsctl add-br br-ovs -- set Bridge br-ovs datapath_type=netdev -- br-set-external-id br-ovs bridge-id br-ovs -- set bridge br-ovs fail-mode=standalone
```

5. Attach representor to br-ovs.

```
ovs-vsctl add-port br-ovs pf0vf0 -- set Interface pf0vf0 type=dpdk options:dpdk-devargs=0000:03:00.0,representor=[0]
```

6. Add a port for the VXLAN tunnel.

```
ovs-vsctl add-port ovs-sriov vxlan0 -- set interface vxlan0 type=vxlan options:local_ip=56.56.67.1 options:remote_ip=56.56.68.1 options:key=45
```

```
options:dst_port=4789
```

Connection Tracking Offload

Connection tracking enables stateful packet processing by keeping a record of currently open connections.
OVS flows using connection tracking can be accelerated using advanced Network Interface Cards (NICs) by offloading established connections.

To view offloaded connections, run:

```
ovs-appctl dpctl/offload-stats-show
```

SR-IOV VF LAG

To configure OVS-DPDK SR-IOV VF LAG:

1. Enable SR-IOV on the NICs.

   ```
   mlxconfig -d <PCI> set SRIOV_EN=1
   ```

2. Allocate the desired number of VFs per port.

   ```
   echo $n > /sys/class/net/<net name>/device/sriov_numvfs
   ```

3. Unbind all VFs.

   ```
   echo <VF PCI> >/sys/bus/pci/drivers/mlx5_core/unbind
   ```

4. Change both NICs' mode to SwitchDev.

   ```
   devlink dev eswitch set pci/<PCI> mode switchdev
   ```

5. Create Linux bonding using kernel modules.

```
modprobe bonding mode=<desired mode>
```

Note: Other bonding parameters can be added here. The supported Bond modes are: Active-Backup, XOR and LACP.

6. Bring all PFs and VFs down.

```
ip link set <PF/VF> down
```

7. Attach both PFs to the bond.

```
ip link set <PF> master bond0
```

8. To work with VF-LAG with OVS-DPDK, add the bond master (PF) to the bridge.

```
ovs-vsctl add-port br-phy p0 -- set Interface p0 type=dpdk options:dpdk-devargs=0000:03:00.0 options:dpdk-lsc-interrupt=true
```

9. Add representor $N of PF0 or PF1 to a bridge.

```
ovs-vsctl add-port br-phy rep$N -- set Interface rep$N type=dpdk options:dpdk-devargs=<PF0 PCI>,representor=pf0vf$N

OR

ovs-vsctl add-port br-phy rep$N -- set Interface rep$N type=dpdk options:dpdk-devargs=<PF0 PCI>,representor=pf1vf$N
```

VirtIO Acceleration through VF Relay (Software & Hardware vDPA)

Note

Hardware vDPA is supported on ConnectX-6 Dx, ConnectX-6 Lx & BlueField-2 cards and above only.

Note

Hardware vDPA is enabled by default. In case your hardware does not support vDPA, the driver will fall back to Software vDPA.

To check which vDPA mode is activated on your driver, run: `ovs-ofctl -O OpenFlow14 dump-ports br0-ovs` and look for `hw-mode` flag.

Note

This feature has not been accepted to the OVS-DPDK Upstream yet, making its API subject to change.

In user space, there are two main approaches for communicating with a guest (VM), either through SR-IOV, or through virtIO.

Phy ports (SR-IOV) allow working with port representor, which is attached to the OVS and a matching VF is given with pass-through to the guest. HW rules can process packets from up-link and direct them to the VF without going through SW (OVS). Therefore, using SR-IOV achieves the best performance.

However, SR-IOV architecture requires the guest to use a driver specific to the underlying HW. Specific HW driver has two main drawbacks:

1. Breaks virtualization in some sense (guest is aware of the HW). It can also limit the type of images supported.

2. Gives less natural support for live migration.

Using virtIO port solves both problems. However, it reduces performance and causes loss of some functionalities, such as, for some HW offloads, working directly with virtIO. To solve this conflict, a new netdev type- dpdkvdpa has been created. The new netdev is similar to the regular DPDK netdev, yet introduces several additional functionalities.

dpdkvdpa translates between phy port to virtIO port. It takes packets from the Rx queue and sends them to the suitable Tx queue, and allows transfer of packets from virtIO guest (VM) to a VF, and vice-versa, benefitting from both SR-IOV and virtIO.

To add vDPA port:

```
ovs-vsctl add-port br0 vdpa0 -- set Interface vdpa0 type=dpdkvdpa \
options:vdpa-socket-path=<sock path> \
options:vdpa-accelerator-devargs=<vf pci id> \
options:dpdk-devargs=<pf pci id>,representor=[id] \
options: vdpa-max-queues =<num queues> \
options: vdpa-sw=<true/false>
```

Note: vdpa-max-queues is an optional field. When the user wants to configure 32 vDPA ports, the maximum queues number is limited to 8.

vDPA Configuration in OVS-DPDK Mode

Prior to configuring vDPA in OVS-DPDK mode, follow the steps below.

1. Generate the VF.

```
echo 0 > /sys/class/net/enp175s0f0/device/sriov_numvfs
echo 4 > /sys/class/net/enp175s0f0/device/sriov_numvfs
```

2. Unbind each VF.

```
echo <pci> > /sys/bus/pci/drivers/mlx5_core/unbind
```

3. Switch to SwitchDev mode.

```
echo switchdev >> /sys/class/net/enp175s0f0/compat/devlink/mode
```

4. Bind each VF.

```
echo <pci> > /sys/bus/pci/drivers/mlx5_core/bind
```

5. Initialize OVS with:

```
ovs-vsctl --no-wait set Open_vSwitch . other_config:dpdk-init=true
ovs-vsctl --no-wait set Open_vSwitch . other_config:hw-offload=true
```

To configure vDPA in OVS-DPDK mode on ConnectX-5 cards and above:

1. Open vSwitch configuration.

```
ovs-vsctl --no-wait set Open_vSwitch . other_config:dpdk-extra="-a
0000:01:00.0,representor=[0],dv_flow_en=1,dv_esw_en=1,dv_xmeta_en=1"
/usr/share/openvswitch/scripts/ovs-ctl restart
```

2. Create OVS-DPDK bridge.

```
ovs-vsctl add-br br0-ovs -- set bridge br0-ovs datapath_type=netdev
ovs-vsctl add-port br0-ovs pf -- set Interface pf type=dpdk options:dpdk-
devargs=0000:01:00.0
```

3. Create vDPA port as part of the OVS-DPDK bridge.

```
ovs-vsctl add-port br0-ovs vdpa0 -- set Interface vdpa0 type=dpdkvdpa
options:vdpa-socket-path=/var/run/virtio-forwarder/sock0 options:vdpa-
accelerator-devargs=0000:01:00.2 options:dpdk-
devargs=0000:01:00.0,representor=[0] options: vdpa-max-queues=8
```

To configure vDPA in OVS-DPDK mode on BlueField cards:

Set the bridge with the software or hardware vDPA port:

- **On the ARM side**:

  Create the OVS-DPDK bridge.

  ```
  ovs-vsctl add-br br0-ovs -- set bridge br0-ovs datapath_type=netdev
  ovs-vsctl add-port br0-ovs pf -- set Interface pf type=dpdk options:dpdk-devargs=0000:af:00.0
  ovs-vsctl add-port br0-ovs rep-- set Interface rep type=dpdk options:dpdk-devargs=0000:af:00.0,representor=[0]
  ```

- **On the host side**:

  Create the OVS-DPDK bridge.

  ```
  ovs-vsctl add-br br1-ovs -- set bridge br1-ovs datapath_type=netdev protocols=OpenFlow14
  ovs-vsctl add-port br0-ovs vdpa0 -- set Interface vdpa0 type=dpdkvdpa options:vdpa-socket-path=/var/run/virtio-forwarder/sock0 options:vdpa-accelerator-devargs=0000:af:00.2
  ```

  Note: To configure **SW** vDPA, add "**options:vdpa-sw=true"** to the end of the command.

Software vDPA Configuration in OVS-Kernel Mode

SW vDPA can also be used in configurations where the HW offload is done through TC and not DPDK.

1. Open vSwitch configuration.

   ```
   ovs-vsctl set Open_vSwitch . other_config:dpdk-extra="-a 0000:01:00.0,representor=[0],dv_flow_en=1,dv_esw_en=0,idv_xmeta_en=0,isolated_mode=1"
   /usr/share/openvswitch/scripts/ovs-ctl restart
   ```

2. Create OVS-DPDK bridge.

```
ovs-vsctl add-br br0-ovs -- set bridge br0-ovs datapath_type=netdev
```

3. Create vDPA port as part of the OVS-DPDK bridge.

```
ovs-vsctl add-port br0-ovs vdpa0 -- set Interface vdpa0 type=dpdkvdpa
options:vdpa-socket-path=/var/run/virtio-forwarder/sock0 options:vdpa-
accelerator-devargs=0000:01:00.2 options:dpdk-
devargs=0000:01:00.0,representor=[0] options: vdpa-max-queues=8
```

4. Create Kernel bridge.

```
ovs-vsctl add-br br-kernel
```

5. Add representors to Kernel bridge.

```
ovs-vsctl add-port br-kernel enp1s0f0_0
ovs-vsctl add-port br-kernel enp1s0f0
```

Large MTU/Jumbo Frame Configuration

To configure MTU/jumbo frames:

1. Verify that the Kernel version on the VM is 4.14 or above.

```
cat /etc/redhat-release
```

2. Set the MTU on both physical interfaces in the host.

```
ifconfig ens4f0 mtu 9216
```

3. Send a large size packet and verify that it is sent and received correctly.

```
tcpdump -i ens4f0 -nev icmp &
ping 11.100.126.1 -s 9188 -M do -c 1
```

4. Enable host_mtu in xml, and add the following values to xml.

```
host_mtu=9216,csum=on,guest_csum=on,host_tso4=on,host_tso6=on
```

Example:

```
<qemu:commandline>
<qemu:arg value='-chardev'/>
<qemu:arg value='socket,id=charnet1,path=/tmp/sock0,server'/>
<qemu:arg value='-netdev'/>
<qemu:arg value='vhost-user,chardev=charnet1,queues=16,id=hostnet1'/>
<qemu:arg value='-device'/>
<qemu:arg value='virtio-net-
pci,mq=on,vectors=34,netdev=hostnet1,id=net1,mac=00:21:21:24:02:01,bus=pci.0,addr=0xC,page-
per-
vq=on,rx_queue_size=1024,tx_queue_size=1024,host_mtu=9216,csum=on,guest_csum=on,host_tso
</qemu:commandline>
```

5. Add mtu_request=9216 option to the OvS ports inside the container and restart the OVS:

```
ovs-vsctl add-port br0-ovs pf -- set Interface pf type=dpdk options:dpdk-
devargs=0000:c4:00.0 mtu_request=9216
```

OR:

```
ovs-vsctl add-port br0-ovs vdpa0 -- set Interface vdpa0 type=dpdkvdpa
options:vdpa-socket-path=/tmp/sock0 options:vdpa-accelerator-
devargs=0000:c4:00.2 options:dpdk-devargs=0000:c4:00.0,representor=[0]
mtu_request=9216
/usr/share/openvswitch/scripts/ovs-ctl restart
```

6. Start the VM and configure the MTU on the VM.

```
ifconfig eth0 11.100.124.2/16 up
ifconfig eth0 mtu 9216
ping 11.100.126.1 -s 9188 -M do -c1
```

E2E Cache

> **Note**
>
> This feature is at beta level.

OVS offload rules are based on a multi-table architecture. E2E cache feature enables merging the multi-table flow matches and actions into one joint flow.

This improves connection tracking performance by using a single-table when exact match is detected.

- ***To set the E2E cache size (default = 4k):***

```
ovs-vsctl set open_vswitch . other_config:e2e-size=<size>
systemctl restart openvswitch
```

Note: Make sure to restart the openvswitch service in order for the configuration to take effect.

- ***To enable/disable E2E cache (default = disabled) :***

```
ovs-vsctl set open_vswitch . other_config:e2e-enable=<true/false>
systemctl restart openvswitch
```

Note: Make sure to restart the openvswitch service in order for the configuration to take effect.

- ***To run E2E cache statistics:***

```
ovs-appctl dpctl/dump-e2e-stats
```

- ***To run E2E cache flows:***

```
ovs-appctl dpctl/dump-e2e-flows
```

Geneve Encapsulation/Decapsulation

Geneve tunneling offload feature support includes matching on extension header.

To configure OVS-DPDK Geneve encap/decap:

1. Create a br-phy bridge.

```
ovs-vsctl --may-exist add-br br-phy -- set Bridge br-phy datapath_type=netdev -- br-set-external-id br-phy bridge-id br-phy -- set bridge br-phy fail-mode=standalone
```

2. Attach PF interface to br-phy bridge.

```
ovs-vsctl add-port br-phy pf -- set Interface pf type=dpdk options:dpdk-devargs=<PF PCI>
```

3. Configure IP to the bridge.

```
ifconfig br-phy <$local_ip_1> up
```

4. Create a br-int bridge.

```
ovs-vsctl --may-exist add-br br-int -- set Bridge br-int datapath_type=netdev -- br-set-external-id br-int bridge-id br-int -- set bridge br-int fail-mode=standalone
```

5. Attach representor to br-int.

```
ovs-vsctl add-port br-int rep$x -- set Interface rep$x type=dpdk options:dpdk-devargs=<PF PCI>,representor=[$x]
```

6. Add a port for the GENEVE tunnel.

```
ovs-vsctl add-port br-int geneve0 -- set interface geneve0 type=geneve options:key=<VNI> options:remote_ip=<$remote_ip_1> options:local_ip=<$local_ip_1>
```

Parallel Offloads

OVS-DPDK supports parallel insertion and deletion of offloads (flow & CT). While multiple threads are supported, by default only one is used.

To configure multiple threads:

```
ovs-vsctl set Open_vSwitch . other_config:n-offload-threads=3
```

Make sure to restart the openvswitch service in order for the configuration to take effect.

```
systemctl restart openvswitch
```

Note

For more information, see the OvS user manual.

sFlow

This feature allows for monitoring traffic sent between two VMs on the same host using an sFlow collector.

To sample all traffic over the OVS bridge, run the following:

```
# ovs-vsctl -- --id=@sflow create sflow agent=\"$SFLOW_AGENT\" \
target=\"$SFLOW_TARGET:$SFLOW_HEADER\" header=$SFLOW_HEADER \
sampling=$SFLOW_SAMPLING polling=10 \
-- set bridge sflow=@sflow
```

Parameter	Description
SFLOW_AGENT	Indicates that the sFlow agent should send traffic from SFLOW_AGENT's IP address
SFLOW_TARGET	Remote IP address of the sFLOW collector
SFLOW_PORT	Remote IP destination port of the sFlow collector
SFLOW_HEADER	Size of packet header to sample (in bytes)
SFLOW_SAMPLING	Sample rate

To clear the sFLOW configuration, runt he following:

```
# ovs-vsctl clear bridge br-vxlan mirrors
```

> **Note**
>
> Currently sFlow for OVS-DPDK is supported without CT.

CT CT NAT

To enable ct-ct-nat offloads in OvS-DPDK, execute the following command (deafult value is false):

```
ovs-vsctl set open_vswitch . other_config:ct-action-on-nat-conns=true
```

If disabled, ct-ct-nat configurations will not be fully offloaded, improving connection offloading rate for
other cases (ct and ct-nat).

If enabled, ct-ct-nat configurations will be fully offloaded but ct and ct-nat offloading will be slower to be created.

OpenFlow Meters (OpenFlow13+):

OpenFlow meters in OVS are implemented according to RFC 2697 (Single Rate Three Color Marker—srTCM).

- The srTCM meters an IP packet stream and marks its packets either green, yellow, or red. The color is decided on a Committed Information Rate (CIR) and two associated burst sizes, Committed Burst Size (CBS), and Excess Burst Size (EBS).

- A packet is marked green if it does not exceed the CBS, yellow if it exceeds the CBS but not the EBS, and red otherwise.

- The volume of green packets should never be smaller than the CIR.

To configure a meter in OVS:

1. Create a meter over a certain bridge:

    1. ```
       ovs-ofctl -O openflow13 add-meter $bridge
       meter=$id,$pktps/$kbps,band=type=drop,rate=$rate,
       [burst,burst_size=$burst_size]
       ```

    2. Parameters:

Parameter	Description
bridge	Name of the bridge on which the meter will be applied.
id	Unique meter ID (32 bits) which will be used as an identifier for the meter.
pktps/kbps	Indication if the meter should work according to packets-per-second or kilobits-per-second.
rate	Rate of pktps/kbps of allowed data transmission.
burst	If set, enables burst support for meter bands through the “burst_size” parameter.
burst_size	If burst is specified for the meter entry, configures the maximum burst allowed for the band in kilobits/packets, depending on whether kbps or pktps was specified. If unspecified, the switch is free to select some reasonable value depending on its configuration. Currently, if burst was not specified, the burst_size parameter is set as the “rate”.

2. Add the meter to a certain OpenFlow rule. For example:
       ```

```
ovs-ofctl -O openflow13 add-flow $bridge "table=0,actions=meter:$id,normal“
```

3. View the meter statistics:

```
ovs-ofctl -O openflow13 meter-stats $bridge meter=$id
```

4. For more information, refer to openvswitch documentation http://www.openvswitch.org/support/dist-docs/ovs-ofctl.8.txt

VirtIO Acceleration through Hardware vDPA

Hardware vDPA Installation

Hardware vDPA requires QEMU v2.12 (or with upstream 6.1.0) and DPDK v20.11 as minimal versions.

To install QEMU:

1. Clone the sources:

```
git clone https://git.qemu.org/git/qemu.git
cd qemu
git checkout v2.12
```

2. Build QEMU:

```
mkdir bin
cd bin
../configure --target-list=x86_64-softmmu --enable-kvm
make -j24
```

To install DPDK:

1. Clone the sources:

```
git clone git://dpdk.org/dpdk
cd dpdk
git checkout v20.11
```

2. Install dependencies (if needed):

```
yum install cmake gcc libnl3-devel libudev-devel make pkgconfig valgrind-devel
pandoc libibverbs libmlx5 libmnl-devel -y
```

3. Configure DPDK:

```
export RTE_SDK=$PWD
make config T=x86_64-native-linuxapp-gcc
cd build
sed -i 's/\(CONFIG_RTE_LIBRTE_MLX5_PMD=\)n/\1y/g' .config
sed -i 's/\(CONFIG_RTE_LIBRTE_MLX5_VDPA_PMD=\)n/\1y/g' .config
```

4. Build DPDK:

```
make -j
```

5. Build the vDPA application:

```
cd $RTE_SDK/examples/vdpa/
make -j
```

Hardware vDPA Configuration

To configure huge pages:

```
mkdir -p /hugepages
mount -t hugetlbfs hugetlbfs /hugepages
echo <more> > /sys/devices/system/node/node0/hugepages/hugepages-1048576kB/nr_hugepages
echo <more> > /sys/devices/system/node/node1/hugepages/hugepages-1048576kB/nr_hugepages
```

To configure a vDPA VirtIO interface in an existing VM's xml file (using libvirt):

1. Open the VM's configuration xml for editing:

   ```
   virsh edit <domain name>
   ```

2. Modify/add the following:

   1. Change the top line to:

      ```
      <domain type='kvm'
      xmlns:qemu='http://libvirt.org/schemas/domain/qemu/1.0'>
      ```

   2. Assign a memory amount and use 1GB page size for hugepages (size must be the same as used for the vDPA application), so that the memory configuration looks like the following.

      ```
      <memory unit='KiB'>4194304</memory>
      <currentMemory unit='KiB'>4194304</currentMemory>
      <memoryBacking>
      <hugepages>
      <page size='1048576' unit='KiB'/>
      </hugepages>
      </memoryBacking>
      ```

   3. Assign an amount of CPUs for the VM CPU configuration, so that the `vcpu` and `cputune` configuration looks like the following.

```
<vcpu placement='static'>5</vcpu>
<cputune>
<vcpupin vcpu='0' cpuset='14'/>
<vcpupin vcpu='1' cpuset='16'/>
<vcpupin vcpu='2' cpuset='18'/>
<vcpupin vcpu='3' cpuset='20'/>
<vcpupin vcpu='4' cpuset='22'/>
</cputune>
```

4. Set the memory access for the CPUs to be shared, so that the `cpu` configuration looks like the following.

```
<cpu mode='custom' match='exact' check='partial'>
<model fallback='allow'>Skylake-Server-IBRS</model>
<numa>
<cell id='0' cpus='0-4' memory='8388608' unit='KiB' memAccess='shared'/>
</numa>
</cpu>
```

5. Set the emulator in use to be the one built in step "2. Build QEMU" above, so that the emulator configuration looks as follows.

```
<emulator><path to qemu executable></emulator>
```

6. Add a virtio interface using qemu command line argument entries, so that the new interface snippet looks as follows.

```
<qemu:commandline>
<qemu:arg value='-chardev'/>
<qemu:arg value='socket,id=charnet1,path=/tmp/sock-virtio0'/>
<qemu:arg value='-netdev'/>
<qemu:arg value='vhost-user,chardev=charnet1,queues=16,id=hostnet1'/>
<qemu:arg value='-device'/>
<qemu:arg value='virtio-net-
pci,mq=on,vectors=6,netdev=hostnet1,id=net1,mac=e4:11:c6:d3:45:f2,bus=p
```

```
page-per-vq=on,rx_queue_size=1024,tx_queue_size=1024'/>
</qemu:commandline>
```

Note: In this snippet, the vhostuser socket file path, the amount of queues, the MAC and the PCI slot of the VirtIO device can be configured.

Running Hardware vDPA

> **Note**
>
> Hardware vDPA supports SwitchDev mode only.

Create the ASAP^2 environment:

1. Create the VFs.
2. Enter switchdev mode.
3. Set up OVS.

Run the vDPA application.

```
cd $RTE_SDK/examples/vdpa/build
./vdpa -w <VF PCI BDF>,class=vdpa --log-level=pmd,info -- -i
```

Create a vDPA port via the vDPA application CLI.

```
create /tmp/sock-virtio0 <PCI DEVICE BDF>
```

Note: The vhostuser socket file path must be the one used when configuring the VM.

Start the VM.

```
virsh start <domain name>
```

For further information on the vDPA application, please visit: https://doc.dpdk.org/guides/sample_app_ug/vdpa.html.

Bridge Offload

Note

- Bridge offload is supported on ConnectX-6 Dx NIC
- Bridge offload is supported SwitchDev mode only
- Bridge offload is supported from kernel version 5.15

A Linux bridge is in-kernel software network switch (based on and implements subset of IEEE 802.1D standard) that is used to connect Ethernet segments together in a protocol-independent way. Packets are forwarded based on L2 Ethernet header addresses.

mlx5 provides capabilities to offload bridge data-plane unicast packet forwarding and VLAN management to hardware.

Basic Configuration

1. Initialize the ASAP^2 environment:
    1. Create the VFs.
    2. Enter switchdev mode.
2. Create a bridge and add mlx5 representors to bridge:

```
ip link add name bridge0 type bridge
ip link set enp8s0f0_0 master bridge0
```

Configuring VLAN

1. Enable VLAN filtering on the bridge.

   ```
   ip link set bridge0 type bridge vlan_filtering 1
   ```

2. Configure port VLAN matching (trunk mode). In this configuration, only packets with specified VID are allowed.

   ```
   bridge vlan add dev enp8s0f0_0 vid 2
   ```

3. Configure port VLAN tagging (access mode). In this configuration VLAN header in pushed/popped on reception/transmission on port.

   ```
   bridge vlan add dev enp8s0f0_0 vid 2 pvid untagged
   ```

VF LAG Support

Bridge supports offloading on bond net device that is fully initialized with mlx5 uplink representors and is in single (shared) FDB LAG mode. Details about initialization of LAG are provided in SR-IOV VF LAG section, above.

Add bonding net device to bridge.

```
ip link set bond0 master bridge0
```

For further information on interacting with Linux bridge via iproute2 bridge tool, please consult man page (man 8 bridge).

Appendix: NVIDIA Firmware Tools

Download and install the MFT package corresponding to your computer's operating system. You would need the kernel-devel or kernel-headers RPM before the tools are built and installed.
The package is available at nvidia.com/en-us/networking/ Products Software Firmware Tools.

1. Start the mst driver.

```
# mst start
Starting MST (Mellanox Software Tools) driver set
Loading MST PCI module - Success
Loading MST PCI configuration module - Success
Create devices
```

2. Show the devices status.

```
ST modules:
------------
MST PCI module loaded
MST PCI configuration module loaded

PCI devices:
------------
DEVICE_TYPE MST PCI RDMA NET NUMA
ConnectX4lx(rev:0) /dev/mst/mt4117_pciconf0.1 04:00.1 net-enp4s0f1 NA
ConnectX4lx(rev:0) /dev/mst/mt4117_pciconf0 04:00.0 net-enp4s0f0 NA



# mlxconfig -d /dev/mst/mt4117_pciconf0 q | head -16

Device #1:
----------
```

```
Device type: ConnectX4lx
PCI device: /dev/mst/mt4117_pciconf0

Configurations: Current
SRIOV_EN True(1)
NUM_OF_VFS 8
PF_LOG_BAR_SIZE 5
VF_LOG_BAR_SIZE 5
NUM_PF_MSIX 63
NUM_VF_MSIX 11
LINK_TYPE_P1 ETH(2)
LINK_TYPE_P2 ETH(2)
```

3. Make sure your configuration is as follows:

   * SR-IOV is enabled (SRIOV_EN=1)
   * The number of enabled VFs is enough for your environment (NUM_OF_VFS=N)
   * The port's link type is Ethernet (LINK_TYPE_P1/2=2) when applicable

   If this is not the case, use mlxconfig to enable that, as follows:

   1. Enable SR-IOV.

      ```
      # mlxconfig -d /dev/mst/mt4115_pciconf0 s SRIOV_EN=1
      ```

   2. Set the number of required VFs.

      ```
      # mlxconfig -d /dev/mst/mt4115_pciconf0 s NUM_OF_VFS=8
      ```

   3. Set the link type to Ethernet.

      ```
      # mlxconfig -d /dev/mst/mt4115_pciconf0 s LINK_TYPE_P1=2
      # mlxconfig -d /dev/mst/mt4115_pciconf0 s LINK_TYPE_P2=2
      ```

4. Conduct a cold reboot (or a firmware reset).

```
# mlxfwreset -d /dev/mst/mt4115_pciconf0 reset
```

5. Query the firmware to make sure everything is set correctly.

```
# mlxconfig -d /dev/mst/mt4115_pciconf0 q
```