HowTo Guides

Release 17.02.1

June 02, 2017

CHAPTER

ONE

LIVE MIGRATION OF VM WITH SR-IOV VF

1.1 Overview

It is not possible to migrate a Virtual Machine which has an SR-IOV Virtual Function (VF).

To get around this problem the bonding PMD is used.

The following sections show an example of how to do this.

1.2 Test Setup

A bonded device is created in the VM. The virtio and VF PMD's are added as slaves to the bonded device. The VF is set as the primary slave of the bonded device.

A bridge must be set up on the Host connecting the tap device, which is the backend of the Virtio device and the Physical Function (PF) device.

To test the Live Migration two servers with identical operating systems installed are used. KVM and Qemu 2.3 is also required on the servers.

In this example, the servers have Niantic and or Fortville NIC's installed. The NIC's on both servers are connected to a switch which is also connected to the traffic generator.

The switch is configured to broadcast traffic on all the NIC ports. A *Sample switch configuration* can be found in this section.

The host is running the Kernel PF driver (ixgbe or i40e).

The ip address of host_server_1 is 10.237.212.46

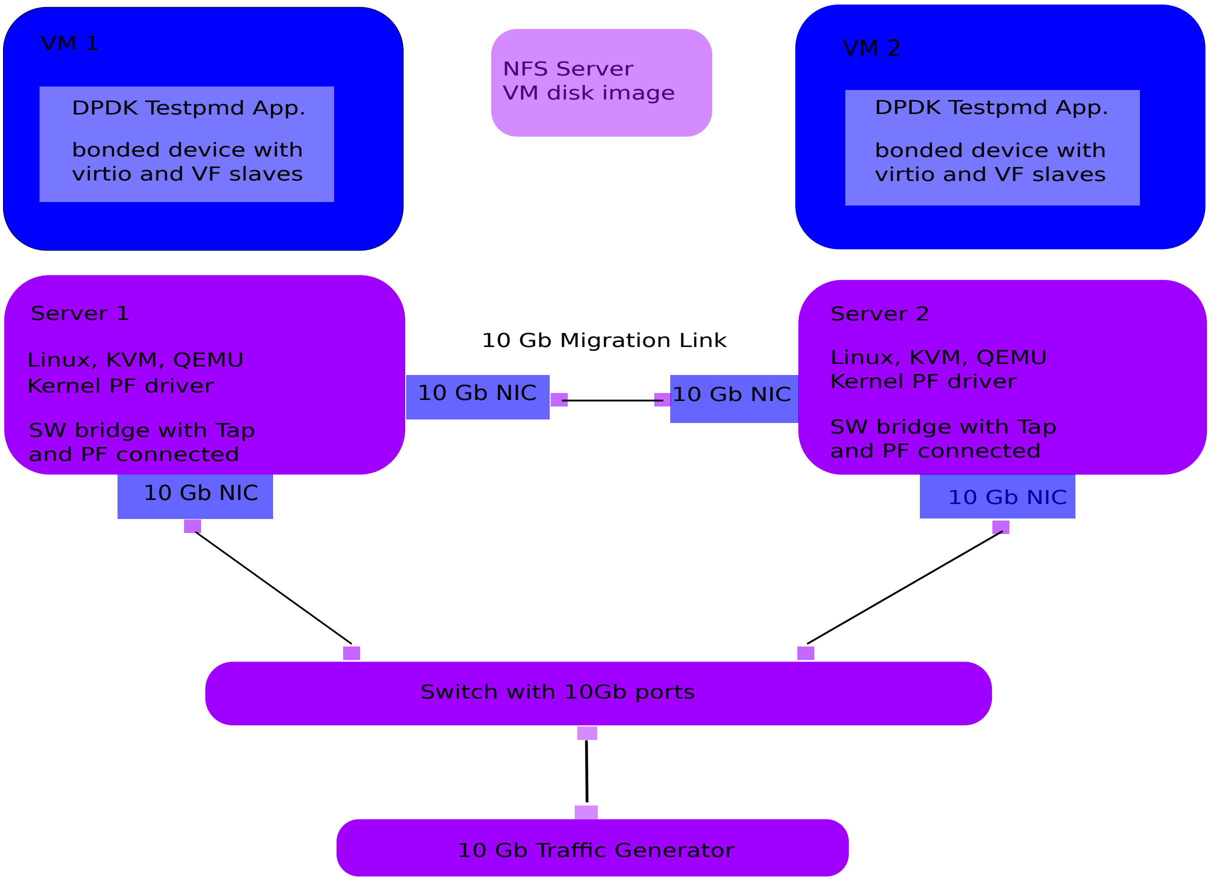
The ip address of host_server_2 is 10.237.212.131

1.3 Live Migration steps

The sample scripts mentioned in the steps below can be found in the *Sample host scripts* and *Sample VM scripts* sections.

1.3.1 On host_server_1: Terminal 1

```
cd /root/dpdk/host_scripts
./setup_vf_on_212_46.sh
```



For Fortville NIC

```
./vm_virtio_vf_i40e_212_46.sh
```

For Niantic NIC

```
./vm_virtio_vf_one_212_46.sh
```

1.3.2 On host_server_1: Terminal 2

```
cd /root/dpdk/host_scripts
./setup_bridge_on_212_46.sh
./connect_to_qemu_mon_on_host.sh
(qemu)
```

1.3.3 On host_server_1: Terminal 1

In VM on host_server_1:

```
cd /root/dpdk/vm_scripts
./setup_dpdk_in_vm.sh
./run_testpmd_bonding_in_vm.sh

testpmd> show port info all
```

The `mac_addr` command only works with kernel PF for Niantic

```
testpmd> mac_addr add port 1 vf 0 AA:BB:CC:DD:EE:FF
```

The syntax of the `testpmd` command is:

Create bonded device (mode) (socket).

Mode 1 is active backup.

Virtio is port 0 (P0).

VF is port 1 (P1).

Bonding is port 2 (P2).

```
testpmd> create bonded device 1 0
Created new bonded device net_bond_testpmd_0 on (port 2).
testpmd> add bonding slave 0 2
testpmd> add bonding slave 1 2
testpmd> show bonding config 2
```

The syntax of the `testpmd` command is:

set bonding primary (slave id) (port id)

Set primary to P1 before starting bonding port.

```
testpmd> set bonding primary 1 2
testpmd> show bonding config 2
testpmd> port start 2
Port 2: 02:09:C0:68:99:A5
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Port 1 Link Up - speed 10000 Mbps - full-duplex
Port 2 Link Up - speed 10000 Mbps - full-duplex

testpmd> show bonding config 2
```

Primary is now P1. There are 2 active slaves.

Use P2 only for forwarding.

```
testpmd> set portlist 2
testpmd> show config fwd
testpmd> set fwd mac
testpmd> start
testpmd> show bonding config 2
```

Primary is now P1. There are 2 active slaves.

```
testpmd> show port stats all
```

VF traffic is seen at P1 and P2.

```
testpmd> clear port stats all
testpmd> set bonding primary 0 2
testpmd> remove bonding slave 1 2
testpmd> show bonding config 2
```

Primary is now P0. There is 1 active slave.

```
testpmd> clear port stats all
testpmd> show port stats all
```

No VF traffic is seen at P0 and P2, VF MAC address still present.

```
testpmd> port stop 1
testpmd> port close 1
```

Port close should remove VF MAC address, it does not remove perm_addr.

The `mac_addr` command only works with the kernel PF for Niantic.

```
testpmd> mac_addr remove 1 AA:BB:CC:DD:EE:FF
testpmd> port detach 1
Port '0000:00:04.0' is detached. Now total ports is 2
testpmd> show port stats all
```

No VF traffic is seen at P0 and P2.

1.3.4 On host_server_1: Terminal 2

```
(qemu) device_del vf1
```

1.3.5 On host_server_1: Terminal 1

In VM on host_server_1:

```
testpmd> show bonding config 2
```

Primary is now P0. There is 1 active slave.

```
testpmd> show port info all
testpmd> show port stats all
```

1.3.6 On host_server_2: Terminal 1

```
cd /root/dpdk/host_scripts
./setup_vf_on_212_131.sh
./vm_virtio_one_migrate.sh
```

1.3.7 On host_server_2: Terminal 2

```
./setup_bridge_on_212_131.sh
./connect_to_qemu_mon_on_host.sh
(qemu) info status
VM status: paused (inmigrate)
(qemu)
```

1.3.8 On host_server_1: Terminal 2

Check that the switch is up before migrating.

```
(qemu) migrate tcp:10.237.212.131:5555
(qemu) info status
VM status: paused (postmigrate)
```

For the Niantic NIC.

```
(qemu) info migrate
capabilities: xbzrle: off rdma-pin-all: off auto-converge: off zero-blocks: off
Migration status: completed
total time: 11834 milliseconds
downtime: 18 milliseconds
setup: 3 milliseconds
transferred ram: 389137 kbytes
throughput: 269.49 mbps
remaining ram: 0 kbytes
total ram: 1590088 kbytes
duplicate: 301620 pages
```

```
skipped: 0 pages
normal: 96433 pages
normal bytes: 385732 kbytes
dirty sync count: 2
(qemu) quit
```

For the Fortville NIC.

```
(qemu) info migrate
capabilities: xbzrle: off rdma-pin-all: off auto-converge: off zero-blocks: off
Migration status: completed
total time: 11619 milliseconds
downtime: 5 milliseconds
setup: 7 milliseconds
transferred ram: 379699 kbytes
throughput: 267.82 mbps
remaining ram: 0 kbytes
total ram: 1590088 kbytes
duplicate: 303985 pages
skipped: 0 pages
normal: 94073 pages
normal bytes: 376292 kbytes
dirty sync count: 2
(qemu) quit
```

1.3.9 On host_server_2: Terminal 1

In VM on host_server_2:

Hit Enter key. This brings the user to the testpmd prompt.

```
testpmd>
```

1.3.10 On host_server_2: Terminal 2

```
(qemu) info status
VM status: running
```

For the Niantic NIC.

```
(qemu) device_add pci-assign,host=06:10.0,id=vf1
```

For the Fortville NIC.

```
(qemu) device_add pci-assign,host=03:02.0,id=vf1
```

1.3.11 On host_server_2: Terminal 1

In VM on host_server_2:

```
testomd> show port info all
testpmd> show port stats all
testpmd> show bonding config 2
testpmd> port attach 0000:00:04.0
Port 1 is attached.
Now total ports is 3
Done

testpmd> port start 1
```

The `mac_addr` command only works with the Kernel PF for Niantic.

```
testpmd> mac_addr add port 1 vf 0 AA:BB:CC:DD:EE:FF
testpmd> show port stats all.
testpmd> show config fwd
testpmd> show bonding config 2
testpmd> add bonding slave 1 2
testpmd> set bonding primary 1 2
testpmd> show bonding config 2
testpmd> show port stats all
```

VF traffic is seen at P1 (VF) and P2 (Bonded device).

```
testpmd> remove bonding slave 0 2
testpmd> show bonding config 2
testpmd> port stop 0
testpmd> port close 0
testpmd> port detach 0
Port '0000:00:03.0' is detached. Now total ports is 2

testpmd> show port info all
testpmd> show config fwd
testpmd> show port stats all
```

VF traffic is seen at P1 (VF) and P2 (Bonded device).

1.4 Sample host scripts

1.4.1 setup_vf_on_212_46.sh

Set up Virtual Functions on host_server_1

```
#!/bin/sh
# This script is run on the host 10.237.212.46 to setup the VF

# set up Niantic VF
cat /sys/bus/pci/devices/0000\:09\:00.0/sriov_numvfs
echo 1 > /sys/bus/pci/devices/0000\:09\:00.0/sriov_numvfs
cat /sys/bus/pci/devices/0000\:09\:00.0/sriov_numvfs
rmmod ixgbevf

# set up Fortville VF
cat /sys/bus/pci/devices/0000\:02\:00.0/sriov_numvfs
echo 1 > /sys/bus/pci/devices/0000\:02\:00.0/sriov_numvfs
cat /sys/bus/pci/devices/0000\:02\:00.0/sriov_numvfs
rmmod i40evf
```

1.4.2 vm_virtio_vf_one_212_46.sh

Setup Virtual Machine on host_server_1

```
#!/bin/sh

# Path to KVM tool
KVM_PATH="/usr/bin/qemu-system-x86_64"

# Guest Disk image
DISK_IMG="/home/username/disk_image/virt1_sml.disk"

# Number of guest cpus
VCPUS_NR="4"
```

```
# Memory
MEM=1536

taskset -c 1-5 $KVM_PATH \
 -enable-kvm \
 -m $MEM \
 -smp $VCPUS_NR \
 -cpu host \
 -name VM1 \
 -no-reboot \
 -net none \
 -vnc none -nographic \
 -hda $DISK_IMG \
 -netdev type=tap,id=net1,script=no,downscript=no,ifname=tap1 \
 -device virtio-net-pci,netdev=net1,mac=CC:BB:BB:BB:BB:BB \
 -device pci-assign,host=09:10.0,id=vf1 \
 -monitor telnet::3333,server,nowait
```

1.4.3 setup_bridge_on_212_46.sh

Setup bridge on host_server_1

```
#!/bin/sh
# This script is run on the host 10.237.212.46 to setup the bridge
# for the Tap device and the PF device.
# This enables traffic to go from the PF to the Tap to the Virtio PMD in the VM.

# ens3f0 is the Niantic NIC
# ens6f0 is the Fortville NIC

ifconfig ens3f0 down
ifconfig tap1 down
ifconfig ens6f0 down
ifconfig virbr0 down

brctl show virbr0
brctl addif virbr0 ens3f0
brctl addif virbr0 ens6f0
brctl addif virbr0 tap1
brctl show virbr0

ifconfig ens3f0 up
ifconfig tap1 up
ifconfig ens6f0 up
ifconfig virbr0 up
```

1.4.4 connect_to_qemu_mon_on_host.sh

```
#!/bin/sh
# This script is run on both hosts when the VM is up,
# to connect to the Qemu Monitor.

telnet 0 3333
```

1.4.5 setup_vf_on_212_131.sh

Set up Virtual Functions on host_server_2

```
#!/bin/sh
# This script is run on the host 10.237.212.131 to setup the VF

# set up Niantic VF
cat /sys/bus/pci/devices/0000\:06\:00.0/sriov_numvfs
echo 1 > /sys/bus/pci/devices/0000\:06\:00.0/sriov_numvfs
cat /sys/bus/pci/devices/0000\:06\:00.0/sriov_numvfs
rmmod ixgbevf

# set up Fortville VF
cat /sys/bus/pci/devices/0000\:03\:00.0/sriov_numvfs
echo 1 > /sys/bus/pci/devices/0000\:03\:00.0/sriov_numvfs
cat /sys/bus/pci/devices/0000\:03\:00.0/sriov_numvfs
rmmod i40evf
```

1.4.6 vm_virtio_one_migrate.sh

Setup Virtual Machine on host_server_2

```
#!/bin/sh
# Start the VM on host_server_2 with the same parameters except without the VF
# parameters, as the VM on host_server_1, in migration-listen mode
# (-incoming tcp:0:5555)

# Path to KVM tool
KVM_PATH="/usr/bin/qemu-system-x86_64"

# Guest Disk image
DISK_IMG="/home/username/disk_image/virt1_sml.disk"

# Number of guest cpus
VCPUS_NR="4"

# Memory
MEM=1536

taskset -c 1-5 $KVM_PATH \
 -enable-kvm \
 -m $MEM \
 -smp $VCPUS_NR \
 -cpu host \
 -name VM1 \
 -no-reboot \
 -net none \
 -vnc none -nographic \
 -hda $DISK_IMG \
 -netdev type=tap,id=net1,script=no,downscript=no,ifname=tap1 \
 -device virtio-net-pci,netdev=net1,mac=CC:BB:BB:BB:BB:BB \
 -incoming tcp:0:5555 \
 -monitor telnet::3333,server,nowait
```

1.4.7 setup_bridge_on_212_131.sh

Setup bridge on host_server_2

```
#!/bin/sh
# This script is run on the host to setup the bridge
# for the Tap device and the PF device.
# This enables traffic to go from the PF to the Tap to the Virtio PMD in the VM.
```

```
# ens4f0 is the Niantic NIC
# ens5f0 is the Fortville NIC

ifconfig ens4f0 down
ifconfig tap1 down
ifconfig ens5f0 down
ifconfig virbr0 down

brctl show virbr0
brctl addif virbr0 ens4f0
brctl addif virbr0 ens5f0
brctl addif virbr0 tap1
brctl show virbr0

ifconfig ens4f0 up
ifconfig tap1 up
ifconfig ens5f0 up
ifconfig virbr0 up
```

1.5 Sample VM scripts

1.5.1 setup_dpdk_in_vm.sh

Set up DPDK in the Virtual Machine

```
#!/bin/sh
# this script matches the vm_virtio_vf_one script
# virtio port is 03
# vf port is 04

cat  /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
echo 1024 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
cat  /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages

ifconfig -a
/root/dpdk/usertools/dpdk-devbind.py --status

rmmod virtio-pci ixgbevf

modprobe uio
insmod /root/dpdk/x86_64-default-linuxapp-gcc/kmod/igb_uio.ko

/root/dpdk/usertools/dpdk-devbind.py -b igb_uio 0000:00:03.0
/root/dpdk/usertools/dpdk-devbind.py -b igb_uio 0000:00:04.0

/root/dpdk/usertools/dpdk-devbind.py --status
```

1.5.2 run_testpmd_bonding_in_vm.sh

Run testpmd in the Virtual Machine.

```
#!/bin/sh
# Run testpmd in the VM

# The test system has 8 cpus (0-7), use cpus 2-7 for VM
# Use taskset -pc <core number> <thread_id>

# use for bonding of virtio and vf tests in VM
```

```
/root/dpdk/x86_64-default-linuxapp-gcc/app/testpmd \
-c f -n 4 --socket-mem 350 --  --i --port-topology=chained
```

1.6 Sample switch configuration

The Intel switch is used to connect the traffic generator to the NIC's on host_server_1 and host_server_2.

In order to run the switch configuration two console windows are required.

Log in as root in both windows.

TestPointShared, run_switch.sh and load /root/switch_config must be executed in the sequence below.

1.6.1 On Switch: Terminal 1

run TestPointShared

```
/usr/bin/TestPointShared
```

1.6.2 On Switch: Terminal 2

execute run_switch.sh

```
/root/run_switch.sh
```

1.6.3 On Switch: Terminal 1

load switch configuration

```
load /root/switch_config
```

1.6.4 Sample switch configuration script

The `/root/switch_config` script:

```
# TestPoint History
show port 1,5,9,13,17,21,25
set port 1,5,9,13,17,21,25 up
show port 1,5,9,13,17,21,25
del acl 1
create acl 1
create acl-port-set
create acl-port-set
add port port-set 1 0
add port port-set 5,9,13,17,21,25 1
create acl-rule 1 1
add acl-rule condition 1 1 port-set 1
add acl-rule action 1 1 redirect 1
apply acl
create vlan 1000
add vlan port 1000 1,5,9,13,17,21,25
set vlan tagging 1000 1,5,9,13,17,21,25 tag
```

```
set switch config flood_ucast fwd
show port stats all 1,5,9,13,17,21,25
```

CHAPTER

TWO

LIVE MIGRATION OF VM WITH VIRTIO ON HOST RUNNING VHOST_USER

2.1 Overview

Live Migration of a VM with DPDK Virtio PMD on a host which is running the Vhost sample application (vhost-switch) and using the DPDK PMD (ixgbe or i40e).

The Vhost sample application uses VMDQ so SRIOV must be disabled on the NIC's.

The following sections show an example of how to do this migration.

2.2 Test Setup

To test the Live Migration two servers with identical operating systems installed are used. KVM and QEMU is also required on the servers.

QEMU 2.5 is required for Live Migration of a VM with vhost_user running on the hosts.

In this example, the servers have Niantic and or Fortville NIC's installed. The NIC's on both servers are connected to a switch which is also connected to the traffic generator.

The switch is configured to broadcast traffic on all the NIC ports.

The ip address of host_server_1 is 10.237.212.46

The ip address of host_server_2 is 10.237.212.131

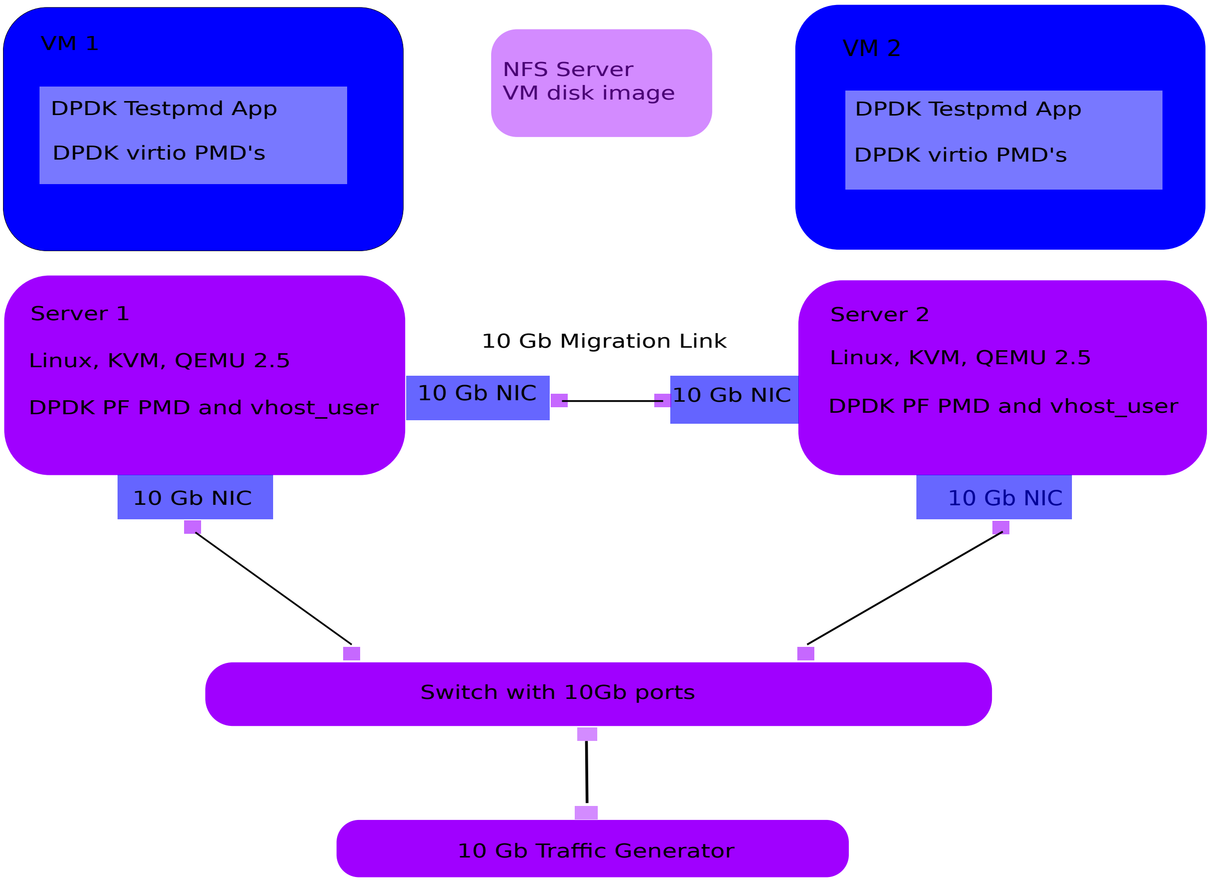
2.3 Live Migration steps

The sample scripts mentioned in the steps below can be found in the *Sample host scripts* and *Sample VM scripts* sections.

2.3.1 On host_server_1: Terminal 1

Setup DPDK on host_server_1

```
cd /root/dpdk/host_scripts
./setup_dpdk_on_host.sh
```



2.3.2 On host_server_1: Terminal 2

Bind the Niantic or Fortville NIC to igb_uio on host_server_1.

For Fortville NIC.

```
cd /root/dpdk/usertools
./dpdk-devbind.py -b igb_uio 0000:02:00.0
```

For Niantic NIC.

```
cd /root/dpdk/usertools
./dpdk-devbind.py -b igb_uio 0000:09:00.0
```

2.3.3 On host_server_1: Terminal 3

For Fortville and Niantic NIC's reset SRIOV and run the vhost_user sample application (vhost-switch) on host_server_1.

```
cd /root/dpdk/host_scripts
./reset_vf_on_212_46.sh
./run_vhost_switch_on_host.sh
```

2.3.4 On host_server_1: Terminal 1

Start the VM on host_server_1

```
./vm_virtio_vhost_user.sh
```

2.3.5 On host_server_1: Terminal 4

Connect to the QEMU monitor on host_server_1.

```
cd /root/dpdk/host_scripts
./connect_to_qemu_mon_on_host.sh
(qemu)
```

2.3.6 On host_server_1: Terminal 1

In VM on host_server_1:

Setup DPDK in the VM and run testpmd in the VM.

```
cd /root/dpdk/vm_scripts
./setup_dpdk_in_vm.sh
./run_testpmd_in_vm.sh

testpmd> show port info all
testpmd> set fwd mac retry
testpmd> start tx_first
testpmd> show port stats all
```

Virtio traffic is seen at P1 and P2.

2.3.7 On host_server_2: Terminal 1

Set up DPDK on the host_server_2.

```
cd /root/dpdk/host_scripts
./setup_dpdk_on_host.sh
```

2.3.8 On host_server_2: Terminal 2

Bind the Niantic or Fortville NIC to igb_uio on host_server_2.

For Fortville NIC.

```
cd /root/dpdk/usertools
./dpdk-devbind.py -b igb_uio 0000:03:00.0
```

For Niantic NIC.

```
cd /root/dpdk/usertools
./dpdk-devbind.py -b igb_uio 0000:06:00.0
```

2.3.9 On host_server_2: Terminal 3

For Fortville and Niantic NIC's reset SRIOV, and run the vhost_user sample application on host_server_2.

```
cd /root/dpdk/host_scripts
./reset_vf_on_212_131.sh
./run_vhost_switch_on_host.sh
```

2.3.10 On host_server_2: Terminal 1

Start the VM on host_server_2.

```
./vm_virtio_vhost_user_migrate.sh
```

2.3.11 On host_server_2: Terminal 4

Connect to the QEMU monitor on host_server_2.

```
cd /root/dpdk/host_scripts
./connect_to_qemu_mon_on_host.sh
(qemu) info status
VM status: paused (inmigrate)
(qemu)
```

2.3.12 On host_server_1: Terminal 4

Check that switch is up before migrating the VM.

```
(qemu) migrate tcp:10.237.212.131:5555
(qemu) info status
VM status: paused (postmigrate)

(qemu) info migrate
capabilities: xbzrle: off rdma-pin-all: off auto-converge: off zero-blocks: off
Migration status: completed
total time: 11619 milliseconds
downtime: 5 milliseconds
setup: 7 milliseconds
transferred ram: 379699 kbytes
throughput: 267.82 mbps
remaining ram: 0 kbytes
total ram: 1590088 kbytes
duplicate: 303985 pages
skipped: 0 pages
normal: 94073 pages
normal bytes: 376292 kbytes
dirty sync count: 2
(qemu) quit
```

2.3.13 On host_server_2: Terminal 1

In VM on host_server_2:

Hit Enter key. This brings the user to the testpmd prompt.

```
testpmd>
```

2.3.14 On host_server_2: Terminal 4

In QEMU monitor on host_server_2

```
(qemu) info status
VM status: running
```

2.3.15 On host_server_2: Terminal 1

In VM on host_server_2:

```
testomd> show port info all
testpmd> show port stats all
```

Virtio traffic is seen at P0 and P1.

2.4 Sample host scripts

2.4.1 reset_vf_on_212_46.sh

```
#!/bin/sh
# This script is run on the host 10.237.212.46 to reset SRIOV

# BDF for Fortville NIC is 0000:02:00.0
cat /sys/bus/pci/devices/0000\:02\:00.0/max_vfs
echo 0 > /sys/bus/pci/devices/0000\:02\:00.0/max_vfs
cat /sys/bus/pci/devices/0000\:02\:00.0/max_vfs

# BDF for Niantic NIC is 0000:09:00.0
cat /sys/bus/pci/devices/0000\:09\:00.0/max_vfs
echo 0 > /sys/bus/pci/devices/0000\:09\:00.0/max_vfs
cat /sys/bus/pci/devices/0000\:09\:00.0/max_vfs
```

2.4.2 vm_virtio_vhost_user.sh

```
#/bin/sh
# Script for use with vhost_user sample application
# The host system has 8 cpu's (0-7)

# Path to KVM tool
KVM_PATH="/usr/bin/qemu-system-x86_64"

# Guest Disk image
DISK_IMG="/home/user/disk_image/virt1_sml.disk"

# Number of guest cpus
VCPUS_NR="6"

# Memory
MEM=1024

VIRTIO_OPTIONS="csum=off,gso=off,guest_tso4=off,guest_tso6=off,guest_ecn=off"

# Socket Path
SOCKET_PATH="/root/dpdk/host_scripts/usvhost"

taskset -c 2-7 $KVM_PATH \
 -enable-kvm \
 -m $MEM \
 -smp $VCPUS_NR \
 -object memory-backend-file,id=mem,size=1024M,mem-path=/mnt/huge,share=on \
 -numa node,memdev=mem,nodeid=0 \
 -cpu host \
 -name VM1 \
 -no-reboot \
 -net none \
```

```
 -vnc none \
 -nographic \
 -hda $DISK_IMG \
 -chardev socket,id=chr0,path=$SOCKET_PATH \
 -netdev type=vhost-user,id=net1,chardev=chr0,vhostforce \
 -device virtio-net-pci,netdev=net1,mac=CC:BB:BB:BB:BB:BB,$VIRTIO_OPTIONS \
 -chardev socket,id=chr1,path=$SOCKET_PATH \
 -netdev type=vhost-user,id=net2,chardev=chr1,vhostforce \
 -device virtio-net-pci,netdev=net2,mac=DD:BB:BB:BB:BB:BB,$VIRTIO_OPTIONS \
 -monitor telnet::3333,server,nowait
```

2.4.3 connect_to_qemu_mon_on_host.sh

```
#!/bin/sh
# This script is run on both hosts when the VM is up,
# to connect to the Qemu Monitor.

telnet 0 3333
```

2.4.4 reset_vf_on_212_131.sh

```
#!/bin/sh
# This script is run on the host 10.237.212.131 to reset SRIOV

# BDF for Ninatic NIC is 0000:06:00.0
cat /sys/bus/pci/devices/0000\:06\:00.0/max_vfs
echo 0 > /sys/bus/pci/devices/0000\:06\:00.0/max_vfs
cat /sys/bus/pci/devices/0000\:06\:00.0/max_vfs

# BDF for Fortville NIC is 0000:03:00.0
cat /sys/bus/pci/devices/0000\:03\:00.0/max_vfs
echo 0 > /sys/bus/pci/devices/0000\:03\:00.0/max_vfs
cat /sys/bus/pci/devices/0000\:03\:00.0/max_vfs
```

2.4.5 vm_virtio_vhost_user_migrate.sh

```
#/bin/sh
# Script for use with vhost user sample application
# The host system has 8 cpu's (0-7)

# Path to KVM tool
KVM_PATH="/usr/bin/qemu-system-x86_64"

# Guest Disk image
DISK_IMG="/home/user/disk_image/virt1_sml.disk"

# Number of guest cpus
VCPUS_NR="6"

# Memory
MEM=1024

VIRTIO_OPTIONS="csum=off,gso=off,guest_tso4=off,guest_tso6=off,guest_ecn=off"

# Socket Path
SOCKET_PATH="/root/dpdk/host_scripts/usvhost"

taskset -c 2-7 $KVM_PATH \
 -enable-kvm \
```

```
-m $MEM \
-smp $VCPUS_NR \
-object memory-backend-file,id=mem,size=1024M,mem-path=/mnt/huge,share=on \
-numa node,memdev=mem,nodeid=0 \
-cpu host \
-name VM1 \
-no-reboot \
-net none \
-vnc none \
-nographic \
-hda $DISK_IMG \
-chardev socket,id=chr0,path=$SOCKET_PATH \
-netdev type=vhost-user,id=net1,chardev=chr0,vhostforce \
-device virtio-net-pci,netdev=net1,mac=CC:BB:BB:BB:BB:BB,$VIRTIO_OPTIONS \
-chardev socket,id=chr1,path=$SOCKET_PATH \
-netdev type=vhost-user,id=net2,chardev=chr1,vhostforce \
-device virtio-net-pci,netdev=net2,mac=DD:BB:BB:BB:BB:BB,$VIRTIO_OPTIONS \
-incoming tcp:0:5555 \
-monitor telnet::3333,server,nowait
```

2.5 Sample VM scripts

2.5.1 setup_dpdk_virtio_in_vm.sh

```
#!/bin/sh
# this script matches the vm_virtio_vhost_user script
# virtio port is 03
# virtio port is 04

cat  /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
echo 1024 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
cat  /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages

ifconfig -a
/root/dpdk/usertools/dpdk-devbind.py --status

rmmod virtio-pci

modprobe uio
insmod /root/dpdk/x86_64-default-linuxapp-gcc/kmod/igb_uio.ko

/root/dpdk/usertools/dpdk-devbind.py -b igb_uio 0000:00:03.0
/root/dpdk/usertools/dpdk-devbind.py -b igb_uio 0000:00:04.0

/root/dpdk/usertools/dpdk-devbind.py --status
```

2.5.2 run_testpmd_in_vm.sh

```
#!/bin/sh
# Run testpmd for use with vhost_user sample app.
# test system has 8 cpus (0-7), use cpus 2-7 for VM

/root/dpdk/x86_64-default-linuxapp-gcc/app/testpmd \
-c 3f -n 4 --socket-mem 350 -- --burst=64 --i --disable-hw-vlan-filter
```

CHAPTER

THREE

FLOW BIFURCATION HOW-TO GUIDE

Flow Bifurcation is a mechanism which uses hardware capable Ethernet devices to split traffic between Linux user space and kernel space. Since it is a hardware assisted feature this approach can provide line rate processing capability. Other than KNI, the software is just required to enable device configuration, there is no need to take care of the packet movement during the traffic split. This can yield better performance with less CPU overhead.

The Flow Bifurcation splits the incoming data traffic to user space applications (such as DPDK applications) and/or kernel space programs (such as the Linux kernel stack). It can direct some traffic, for example data plane traffic, to DPDK, while directing some other traffic, for example control plane traffic, to the traditional Linux networking stack.

There are a number of technical options to achieve this. A typical example is to combine the technology of SR-IOV and packet classification filtering.

SR-IOV is a PCI standard that allows the same physical adapter to be split as multiple virtual functions. Each virtual function (VF) has separated queues with physical functions (PF). The network adapter will direct traffic to a virtual function with a matching destination MAC address. In a sense, SR-IOV has the capability for queue division.

Packet classification filtering is a hardware capability available on most network adapters. Filters can be configured to direct specific flows to a given receive queue by hardware. Different NICs may have different filter types to direct flows to a Virtual Function or a queue that belong to it.

In this way the Linux networking stack can receive specific traffic through the kernel driver while a DPDK application can receive specific traffic bypassing the Linux kernel by using drivers like VFIO or the DPDK `igb_uio` module.

3.1 Using Flow Bifurcation on IXGBE in Linux

On Intel 82599 10 Gigabit Ethernet Controller series NICs Flow Bifurcation can be achieved by SR-IOV and Intel Flow Director technologies. Traffic can be directed to queues by the Flow Director capability, typically by matching 5-tuple of UDP/TCP packets.

The typical procedure to achieve this is as follows:

1. Boot the system without iommu, or with `iommu=pt`.
2. Create Virtual Functions:

   ```
   echo 2 > /sys/bus/pci/devices/0000:01:00.0/sriov_numvfs
   ```

3. Enable and set flow filters:

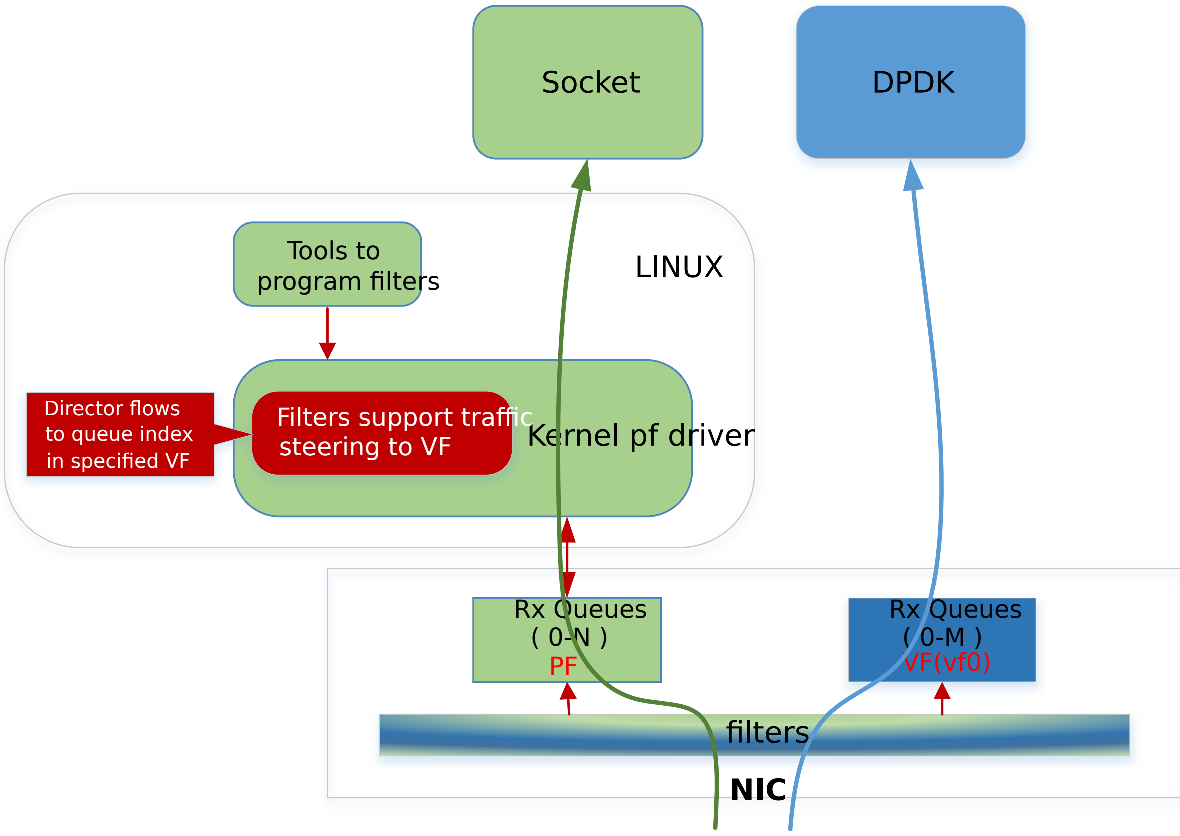
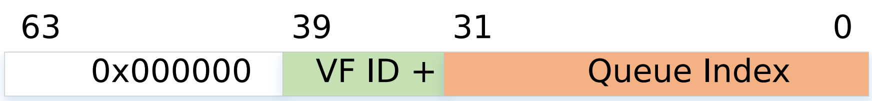


Fig. 3.1: Flow Bifurcation Overview

```
ethtool -K eth1 ntuple on
ethtool -N eth1 flow-type udp4 src-ip 192.0.2.2 dst-ip 198.51.100.2 \
        action $queue_index_in_VF0
ethtool -N eth1 flow-type udp4 src-ip 198.51.100.2 dst-ip 192.0.2.2 \
        action $queue_index_in_VF1
```

Where:

- `$queue_index_in_VFn`: Bits 39:32 of the variable defines VF id + 1; the lower 32 bits indicates the queue index of the VF. Thus:
  - `$queue_index_in_VF0` = `(0x1 & 0xFF) << 32 + [queue index]`.
  - `$queue_index_in_VF1` = `(0x2 & 0xFF) << 32 + [queue index]`.



4. Compile the DPDK application and insert `igb_uio` or probe the `vfio-pci` kernel modules as normal.
5. Bind the virtual functions:

```
modprobe vfio-pci
dpdk-devbind.py -b vfio-pci 01:10.0
dpdk-devbind.py -b vfio-pci 01:10.1
```

6. Run a DPDK application on the VFs:

```
testpmd -c 0xff -n 4 -- -i -w 01:10.0 -w 01:10.1 --forward-mode=mac
```

In this example, traffic matching the rules will go through the VF by matching the filter rule. All other traffic, not matching the rules, will go through the default queue or scaling on queues in the PF. That is to say UDP packets with the specified IP source and destination addresses will go through the DPDK application. All other traffic, with different hosts or different protocols, will go through the Linux networking stack.

Note:

- The above steps work on the Linux kernel v4.2.
- The Flow Bifurcation is implemented in Linux kernel and ixgbe kernel driver using the following patches:
  - ethtool: Add helper routines to pass vf to rx_flow_spec
  - ixgbe: Allow flow director to use entire queue space
- The Ethtool version used in this example is 3.18.

3.2 Using Flow Bifurcation on I40E in Linux

On Intel X710/XL710 series Ethernet Controllers Flow Bifurcation can be achieved by SR-IOV, Cloud Filter and L3 VEB switch. The traffic can be directed to queues by the Cloud Filter and L3 VEB switch's matching rule.

- L3 VEB filters work for non-tunneled packets. It can direct a packet just by the Destination IP address to a queue in a VF.
- Cloud filters work for the following types of tunneled packets.
  - Inner mac.
  - Inner mac + VNI.
  - Outer mac + Inner mac + VNI.
  - Inner mac + Inner vlan + VNI.
  - Inner mac + Inner vlan.

The typical procedure to achieve this is as follows:

1. Boot the system without iommu, or with `iommu=pt`.
2. Build and insert the `i40e.ko` module.
3. Create Virtual Functions:

   ```
   echo 2 > /sys/bus/pci/devices/0000:01:00.0/sriov_numvfs
   ```

4. Add udp port offload to the NIC if using cloud filter:

   ```
   ip li add vxlan0 type vxlan id 42 group 239.1.1.1 local 10.16.43.214 dev <name>
   ifconfig vxlan0 up
   ip -d li show vxlan0
   ```

   Note: Output such as `add vxlan port 8472, index 0 success` should be found in the system log.

5. Examples of enabling and setting flow filters:

- L3 VEB filter, for a route whose destination IP is 192.168.50.108 to VF 0's queue 2.

  ```
  ethtool -N <dev_name> flow-type ip4 dst-ip 192.168.50.108 \
          user-def 0xffffffff00000000 action 2 loc 8
  ```

- Inner mac, for a route whose inner destination mac is 0:0:0:0:9:0 to PF's queue 6.

  ```
  ethtool -N <dev_name> flow-type ether dst 00:00:00:00:00:00 \
          m ff:ff:ff:ff:ff:ff src 00:00:00:00:09:00 m 00:00:00:00:00:00 \
          user-def 0xffffffff00000003 action 6 loc 1
  ```

- Inner mac + VNI, for a route whose inner destination mac is 0:0:0:0:9:0 and VNI is 8 to PF's queue 4.

  ```
  ethtool -N <dev_name> flow-type ether dst 00:00:00:00:00:00 \
          m ff:ff:ff:ff:ff:ff src 00:00:00:00:09:00 m 00:00:00:00:00:00 \
          user-def 0x800000003 action 4 loc 4
  ```

- Outer mac + Inner mac + VNI, for a route whose outer mac is 68:05:ca:24:03:8b, inner destination mac is c2:1a:e1:53:bc:57, and VNI is 8 to PF's queue 2.

  ```
  ethtool -N <dev_name> flow-type ether dst 68:05:ca:24:03:8b \
          m 00:00:00:00:00:00 src c2:1a:e1:53:bc:57 m 00:00:00:00:00:00 \
          user-def 0x800000003 action 2 loc 2
  ```

- Inner mac + Inner vlan + VNI, for a route whose inner destination mac is 00:00:00:00:20:00, inner vlan is 10, and VNI is 8 to VF 0's queue 1.

  ```
  ethtool -N <dev_name> flow-type ether dst 00:00:00:00:01:00 \
          m ff:ff:ff:ff:ff:ff src 00:00:00:00:20:00 m 00:00:00:00:00:00 \
          vlan 10 user-def 0x800000000 action 1 loc 5
  ```

- Inner mac + Inner vlan, for a route whose inner destination mac is 00:00:00:00:20:00, and inner vlan is 10 to VF 0's queue 1.

  ```
  ethtool -N <dev_name> flow-type ether dst 00:00:00:00:01:00 \
          m ff:ff:ff:ff:ff:ff src 00:00:00:00:20:00 m 00:00:00:00:00:00 \
          vlan 10 user-def 0xffffffff00000000 action 1 loc 5
  ```

Note:

- If the upper 32 bits of 'user-def' are `0xffffffff`, then the filter can be used for programming an L3 VEB filter, otherwise the upper 32 bits of 'user-def' can carry the tenant ID/VNI if specified/required.
- Cloud filters can be defined with inner mac, outer mac, inner ip, inner vlan and VNI as part of the cloud tuple. It is always the destination (not source) mac/ip that these filters use. For all these examples dst and src mac address fields are overloaded dst == outer, src == inner.
- The filter will direct a packet matching the rule to a vf id specified in the lower 32 bit of user-def to the queue specified by 'action'.
- If the vf id specified by the lower 32 bit of user-def is greater than or equal to `max_vfs`, then the filter is for the PF queues.

6. Compile the DPDK application and insert `igb_uio` or probe the `vfio-pci` kernel modules as normal.
7. Bind the virtual function:

```
modprobe vfio-pci
dpdk-devbind.py -b vfio-pci 01:10.0
dpdk-devbind.py -b vfio-pci 01:10.1
```

8. run DPDK application on VFs:

```
testpmd -c 0xff -n 4 -- -i -w 01:10.0 -w 01:10.1 --forward-mode=mac
```

Note:

- The above steps work on the i40e Linux kernel driver v1.5.16.
- The Ethtool version used in this example is 3.18. The mask `ff` means 'not involved', while `00` or no mask means 'involved'.
- For more details of the configuration, refer to the cloud filter test plan

CHAPTER

FOUR

PVP REFERENCE BENCHMARK SETUP USING TESTPMD

This guide lists the steps required to setup a PVP benchmark using testpmd as a simple forwarder between NICs and Vhost interfaces. The goal of this setup is to have a reference PVP benchmark without using external vSwitches (OVS, VPP, ...) to make it easier to obtain reproducible results and to facilitate continuous integration testing.

The guide covers two ways of launching the VM, either by directly calling the QEMU command line, or by relying on libvirt. It has been tested with DPDK v16.11 using RHEL7 for both host and guest.

4.1 Setup overview

In this diagram, each red arrow represents one logical core. This use-case requires 6 dedicated logical cores. A forwarding configuration with a single NIC is also possible, requiring 3 logical cores.

4.2 Host setup

In this setup, we isolate 6 cores (from CPU2 to CPU7) on the same NUMA node. Two cores are assigned to the VM vCPUs running testpmd and four are assigned to testpmd on the host.

4.2.1 Host tuning

1. On BIOS, disable turbo-boost and hyper-threads.
2. Append these options to Kernel command line:

```
intel_pstate=disable mce=ignore_ce default_hugepagesz=1G hugepagesz=1G hugepages=6 isolcpu
```

3. Disable hyper-threads at runtime if necessary or if BIOS is not accessible:

```
cat /sys/devices/system/cpu/cpu*[0-9]/topology/thread_siblings_list \
    | sort | uniq \
    | awk -F, '{system("echo 0 > /sys/devices/system/cpu/cpu"$2"/online")}'
```

4. Disable NMIs:

```
echo 0 > /proc/sys/kernel/nmi_watchdog
```

5. Exclude isolated CPUs from the writeback cpumask:

```
echo ffffff03 > /sys/bus/workqueue/devices/writeback/cpumask
```

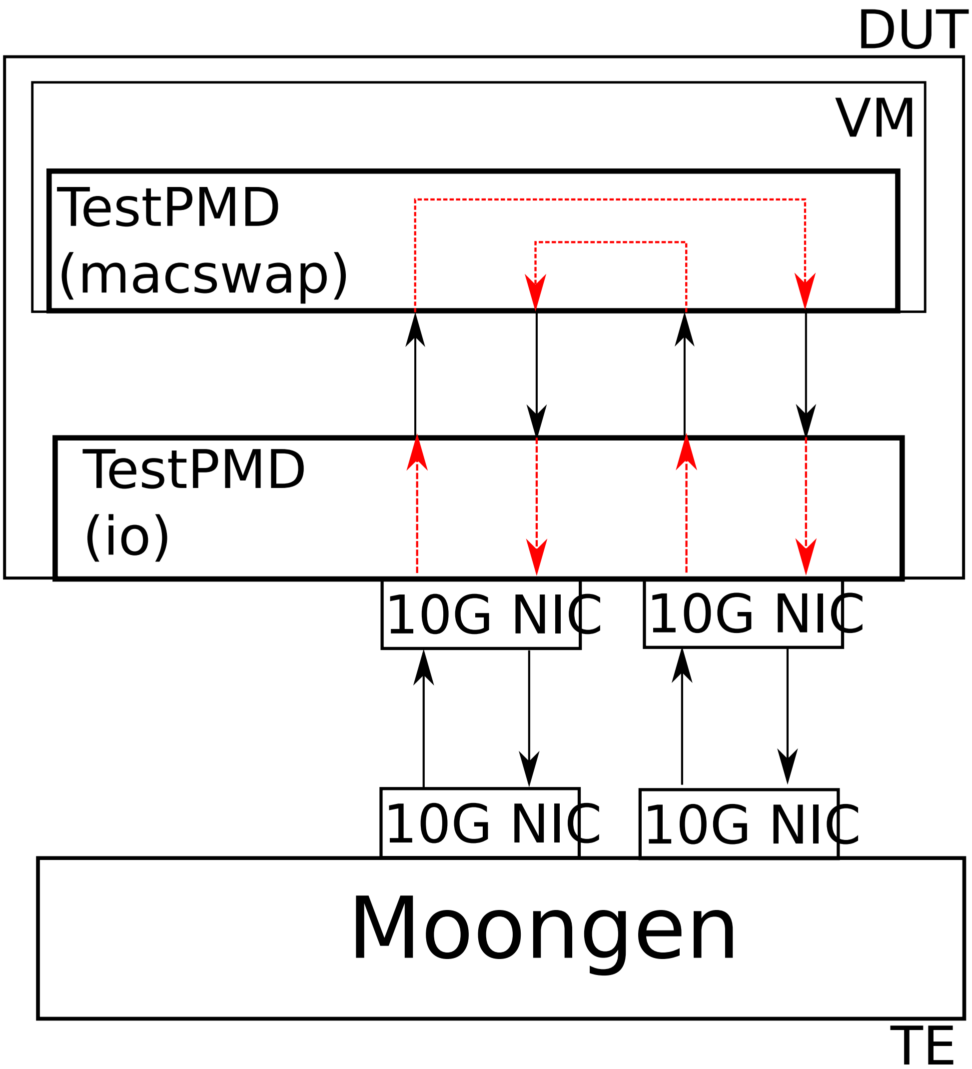


Fig. 4.1: PVP setup using 2 NICs

6. Isolate CPUs from IRQs:

```
clear_mask=0xfc #Isolate CPU2 to CPU7 from IRQs
for i in /proc/irq/*/smp_affinity
do
  echo "obase=16;$(( 0x$(cat $i) & ~$clear_mask ))" | bc > $i
done
```

4.2.2 Qemu build

Build Qemu:

```
git clone git://git.qemu.org/qemu.git
cd qemu
mkdir bin
cd bin
../configure --target-list=x86_64-softmmu
```

4.2.3 DPDK build

Build DPDK:

```
git clone git://dpdk.org/dpdk
cd dpdk
export RTE_SDK=$PWD
make install T=x86_64-native-linuxapp-gcc DESTDIR=install
```

4.2.4 Testpmd launch

1. Assign NICs to DPDK:

```
modprobe vfio-pci
$RTE_SDK/install/sbin/dpdk-devbind -b vfio-pci 0000:11:00.0 0000:11:00.1
```

Note: The Sandy Bridge family seems to have some IOMMU limitations giving poor performance results. To achieve good performance on these machines consider using UIO instead.

2. Launch the testpmd application:

```
$RTE_SDK/install/bin/testpmd -l 0,2,3,4,5 --socket-mem=1024 -n 4 \
    --vdev 'net_vhost0,iface=/tmp/vhost-user1' \
    --vdev 'net_vhost1,iface=/tmp/vhost-user2' -- \
    --portmask=f --disable-hw-vlan -i --rxq=1 --txq=1
    --nb-cores=4 --forward-mode=io
```

With this command, isolated CPUs 2 to 5 will be used as lcores for PMD threads.

3. In testpmd interactive mode, set the portlist to obtain the correct port chaining:

```
set portlist 0,2,1,3
start
```

4.2.5 VM launch

The VM may be launched either by calling QEMU directly, or by using libvirt.

Qemu way

Launch QEMU with two Virtio-net devices paired to the vhost-user sockets created by testpmd. Below example uses default Virtio-net options, but options may be specified, for example to disable mergeable buffers or indirect descriptors.

```
<QEMU path>/bin/x86_64-softmmu/qemu-system-x86_64 \
    -enable-kvm -cpu host -m 3072 -smp 3 \
    -chardev socket,id=char0,path=/tmp/vhost-user1 \
    -netdev type=vhost-user,id=mynet1,chardev=char0,vhostforce \
    -device virtio-net-pci,netdev=mynet1,mac=52:54:00:02:d9:01,addr=0x10 \
    -chardev socket,id=char1,path=/tmp/vhost-user2 \
    -netdev type=vhost-user,id=mynet2,chardev=char1,vhostforce \
    -device virtio-net-pci,netdev=mynet2,mac=52:54:00:02:d9:02,addr=0x11 \
    -object memory-backend-file,id=mem,size=3072M,mem-path=/dev/hugepages,share=on \
    -numa node,memdev=mem -mem-prealloc \
    -net user,hostfwd=tcp::1002$1-:22 -net nic \
    -qmp unix:/tmp/qmp.socket,server,nowait \
    -monitor stdio <vm_image>.qcow2
```

You can use this qmp-vcpu-pin script to pin vCPUs.

It can be used as follows, for example to pin 3 vCPUs to CPUs 1, 6 and 7, where isolated CPUs 6 and 7 will be used as lcores for Virtio PMDs:

```
export PYTHONPATH=$PYTHONPATH:<QEMU path>/scripts/qmp
./qmp-vcpu-pin -s /tmp/qmp.socket 1 6 7
```

Libvirt way

Some initial steps are required for libvirt to be able to connect to testpmd's sockets.

First, SELinux policy needs to be set to permissive, since testpmd is generally run as root (note, as reboot is required):

```
cat /etc/selinux/config

# This file controls the state of SELinux on the system.
# SELINUX= can take one of these three values:
#     enforcing  - SELinux security policy is enforced.
#     permissive - SELinux prints warnings instead of enforcing.
#     disabled   - No SELinux policy is loaded.
SELINUX=permissive

# SELINUXTYPE= can take one of three two values:
#     targeted - Targeted processes are protected,
#     minimum  - Modification of targeted policy.
#                Only selected processes are protected.
#     mls      - Multi Level Security protection.
SELINUXTYPE=targeted
```

Also, Qemu needs to be run as root, which has to be specified in `/etc/libvirt/qemu.conf`:

```
user = "root"
```

Once the domain created, the following snippet is an extract of he most important information (hugepages, vCPU pinning, Virtio PCI devices):

```
<domain type='kvm'>
  <memory unit='KiB'>3145728</memory>
  <currentMemory unit='KiB'>3145728</currentMemory>
```

```
    <memoryBacking>
      <hugepages>
        <page size='1048576' unit='KiB' nodeset='0'/>
      </hugepages>
      <locked/>
    </memoryBacking>
    <vcpu placement='static'>3</vcpu>
    <cputune>
      <vcpupin vcpu='0' cpuset='1'/>
      <vcpupin vcpu='1' cpuset='6'/>
      <vcpupin vcpu='2' cpuset='7'/>
      <emulatorpin cpuset='0'/>
    </cputune>
    <numatune>
      <memory mode='strict' nodeset='0'/>
    </numatune>
    <os>
      <type arch='x86_64' machine='pc-i440fx-rhel7.0.0'>hvm</type>
      <boot dev='hd'/>
    </os>
    <cpu mode='host-passthrough'>
      <topology sockets='1' cores='3' threads='1'/>
      <numa>
        <cell id='0' cpus='0-2' memory='3145728' unit='KiB' memAccess='shared'/>
      </numa>
    </cpu>
    <devices>
      <interface type='vhostuser'>
        <mac address='56:48:4f:53:54:01'/>
        <source type='unix' path='/tmp/vhost-user1' mode='client'/>
        <model type='virtio'/>
        <driver name='vhost' rx_queue_size='256' />
        <address type='pci' domain='0x0000' bus='0x00' slot='0x10' function='0x0'/>
      </interface>
      <interface type='vhostuser'>
        <mac address='56:48:4f:53:54:02'/>
        <source type='unix' path='/tmp/vhost-user2' mode='client'/>
        <model type='virtio'/>
        <driver name='vhost' rx_queue_size='256' />
        <address type='pci' domain='0x0000' bus='0x00' slot='0x11' function='0x0'/>
      </interface>
    </devices>
  </domain>
```

4.3 Guest setup

4.3.1 Guest tuning

1. Append these options to the Kernel command line:

```
default_hugepagesz=1G hugepagesz=1G hugepages=1 intel_iommu=on iommu=pt isolcpus=1,2 rcu_n
```

2. Disable NMIs:

```
echo 0 > /proc/sys/kernel/nmi_watchdog
```

3. Exclude isolated CPU1 and CPU2 from the writeback cpumask:

```
echo 1 > /sys/bus/workqueue/devices/writeback/cpumask
```

4. Isolate CPUs from IRQs:

```
clear_mask=0x6 #Isolate CPU1 and CPU2 from IRQs
for i in /proc/irq/*/smp_affinity
do
  echo "obase=16;$(( 0x$(cat $i) & ~$clear_mask ))" | bc > $i
done
```

4.3.2 DPDK build

Build DPDK:

```
git clone git://dpdk.org/dpdk
cd dpdk
export RTE_SDK=$PWD
make install T=x86_64-native-linuxapp-gcc DESTDIR=install
```

4.3.3 Testpmd launch

Probe vfio module without iommu:

```
modprobe -r vfio_iommu_type1
modprobe -r vfio
modprobe  vfio enable_unsafe_noiommu_mode=1
cat /sys/module/vfio/parameters/enable_unsafe_noiommu_mode
modprobe vfio-pci
```

Bind the virtio-net devices to DPDK:

```
$RTE_SDK/tools/dpdk-devbind.py -b vfio-pci 0000:00:10.0 0000:00:11.0
```

Start testpmd:

```
$RTE_SDK/install/bin/testpmd -l 0,1,2 --socket-mem 1024 -n 4 \
    --proc-type auto --file-prefix pg -- \
    --portmask=3 --forward-mode=macswap --port-topology=chained \
    --disable-hw-vlan --disable-rss -i --rxq=1 --txq=1 \
    --rxd=256 --txd=256 --nb-cores=2 --auto-start
```

4.4 Results template

Below template should be used when sharing results:

```
Traffic Generator: <Test equipment (e.g. IXIA, Moongen, ...)>
Acceptable Loss: <n>%
Validation run time: <n>min
Host DPDK version/commit: <version, SHA-1>
Guest DPDK version/commit: <version, SHA-1>
Patches applied: <link to patchwork>
QEMU version/commit: <version>
Virtio features: <features (e.g. mrg_rxbuf='off', leave empty if default)>
CPU: <CPU model>, <CPU frequency>
NIC: <NIC model>
Result: <n> Mpps
```

VIRTIO_USER FOR CONTAINER NETWORKING

Container becomes more and more popular for strengths, like low overhead, fast boot-up time, and easy to deploy, etc. How to use DPDK to accelerate container networking becomes a common question for users. There are two use models of running DPDK inside containers, as shown in Fig. 5.1.

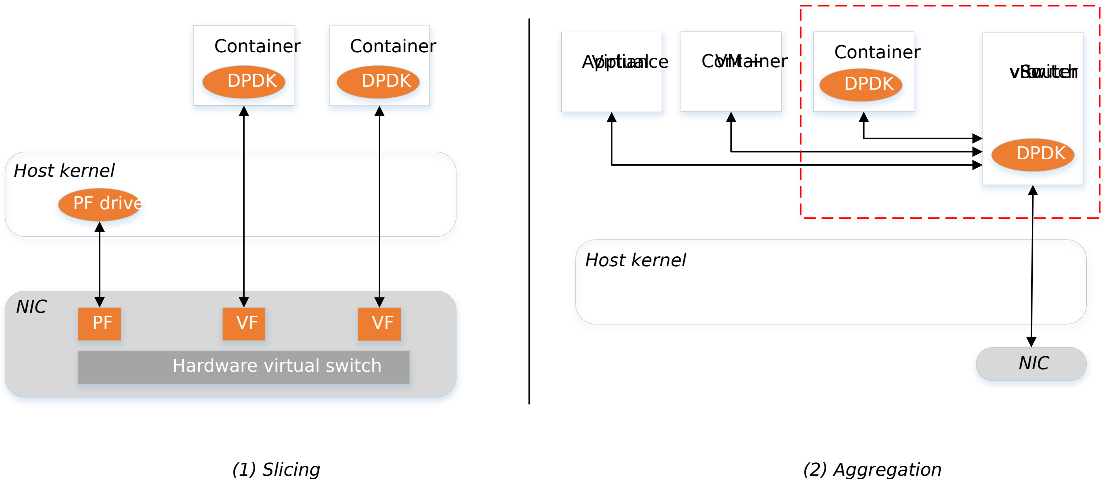


Fig. 5.1: Use models of running DPDK inside container

This page will only cover aggregation model.

5.1 Overview

The virtual device, virtio-user, with unmodified vhost-user backend, is designed for high performance user space container networking or inter-process communication (IPC).

The overview of accelerating container networking by virtio-user is shown in Fig. 5.2.

Different virtio PCI devices we usually use as a para-virtualization I/O in the context of QEMU/VM, the basic idea here is to present a kind of virtual devices, which can be attached and initialized by DPDK. The device emulation layer by QEMU in VM's context is saved by just registering a new kind of virtual device in DPDK's ether layer. And to minimize the change, we reuse already-existing virtio PMD code (driver/net/virtio/).

Virtio, in essence, is a shm-based solution to transmit/receive packets. How is memory shared? In VM's case, qemu always shares the whole physical layout of VM to vhost backend. But it's not feasible for a container, as a process, to share all virtual memory regions to

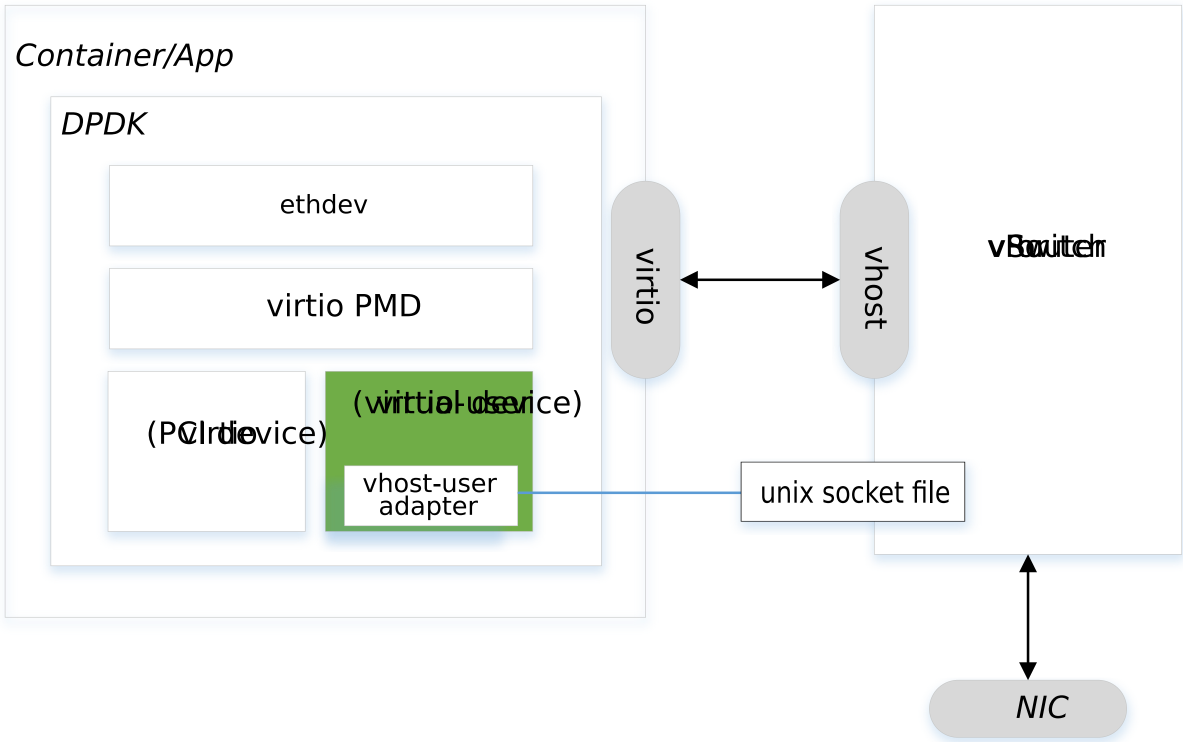


Fig. 5.2: Overview of accelerating container networking by virtio-user

backend. So only those virtual memory regions (aka, hugepages initialized in DPDK) are sent to backend. It restricts that only addresses in these areas can be used to transmit or receive packets.

5.2 Sample Usage

Here we use Docker as container engine. It also applies to LXC, Rocket with some minor changes.

1. Compile DPDK.

```
make install RTE_SDK=`pwd` T=x86_64-native-linuxapp-gcc
```

2. Write a Dockerfile like below.

```
cat <<EOT >> Dockerfile
FROM ubuntu:latest
WORKDIR /usr/src/dpdk
COPY . /usr/src/dpdk
ENV PATH "$PATH:/usr/src/dpdk/x86_64-native-linuxapp-gcc/app/"
EOT
```

3. Build a Docker image.

```
docker build -t dpdk-app-testpmd .
```

4. Start a testpmd on the host with a vhost-user port.

```
$(testpmd) -c 0x3 -n 4 --socket-mem 1024,1024 \
    --vdev 'eth_vhost0,iface=/tmp/sock0' --no-pci -- -i
```

5. Start a container instance with a virtio-user port.

```
docker run -i -t -v /tmp/sock0:/var/run/usvhost \
    -v /dev/hugepages:/dev/hugepages \
    dpdk-app-testpmd testpmd -c 0xc -n 4 -m 1024 --no-pci \
    --vdev=virtio_user0,path=/var/run/usvhost \
    -- -i --txqflags=0xf00 --disable-hw-vlan
```

Note: If we run all above setup on the host, it's a shm-based IPC.

5.3 Limitations

We have below limitations in this solution:

- Cannot work with –huge-unlink option. As we need to reopen the hugepage file to share with vhost backend.
- Cannot work with –no-huge option. Currently, DPDK uses anonymous mapping under this option which cannot be reopened to share with vhost backend.
- Cannot work when there are more than VHOST_MEMORY_MAX_NREGIONS(8) hugepages. In another word, do not use 2MB hugepage so far.
- Applications should not use file name like HUGEFILE_FMT ("%smap_%d"). That will bring confusion when sharing hugepage files with backend by name.
- Root privilege is a must. DPDK resolves physical addresses of hugepages which seems not necessary, and some discussions are going on to remove this restriction.

CHAPTER

SIX

VIRTIO_USER AS EXCEPTIONAL PATH

The virtual device, virtio-user, was originally introduced with vhost-user backend, as a high performance solution for IPC (Inter-Process Communication) and user space container networking.

Virtio_user with vhost-kernel backend is a solution for exceptional path, such as KNI which exchanges packets with kernel networking stack. This solution is very promising in:

- Maintenance

  All kernel modules needed by this solution, vhost and vhost-net (kernel), are upstreamed and extensively used kernel module.

- Features

  vhost-net is born to be a networking solution, which has lots of networking related featuers, like multi queue, tso, multi-seg mbuf, etc.

- Performance

  similar to KNI, this solution would use one or more kthreads to send/receive packets from user space DPDK applications, which has little impact on user space polling thread (except that it might enter into kernel space to wake up those kthreads if necessary).

The overview of an application using virtio-user as exceptional path is shown in Fig. 6.1.

6.1 Sample Usage

As a prerequisite, the vhost/vhost-net kernel CONFIG should be chosen before compiling the kernel and those kernel modules should be inserted.

1. Compile DPDK and bind a physical NIC to igb_uio/uio_pci_generic/vfio-pci.

   This physical NIC is for communicating with outside.

2. Run testpmd.

   ```
   $(testpmd) -c 0xc -n 4 \
           --vdev=virtio_user0,path=/dev/vhost-net,queue_size=1024 \
           -- -i --txqflags=0x0 --disable-hw-vlan --enable-lro --crc-strip \
           --enable-rx-cksum --rxd=1024 --txd=1024
   ```

   This command runs testpmd with two ports, one physical NIC to communicate with outside, and one virtio-user to communicate with kernel.

- `--enable-lro`

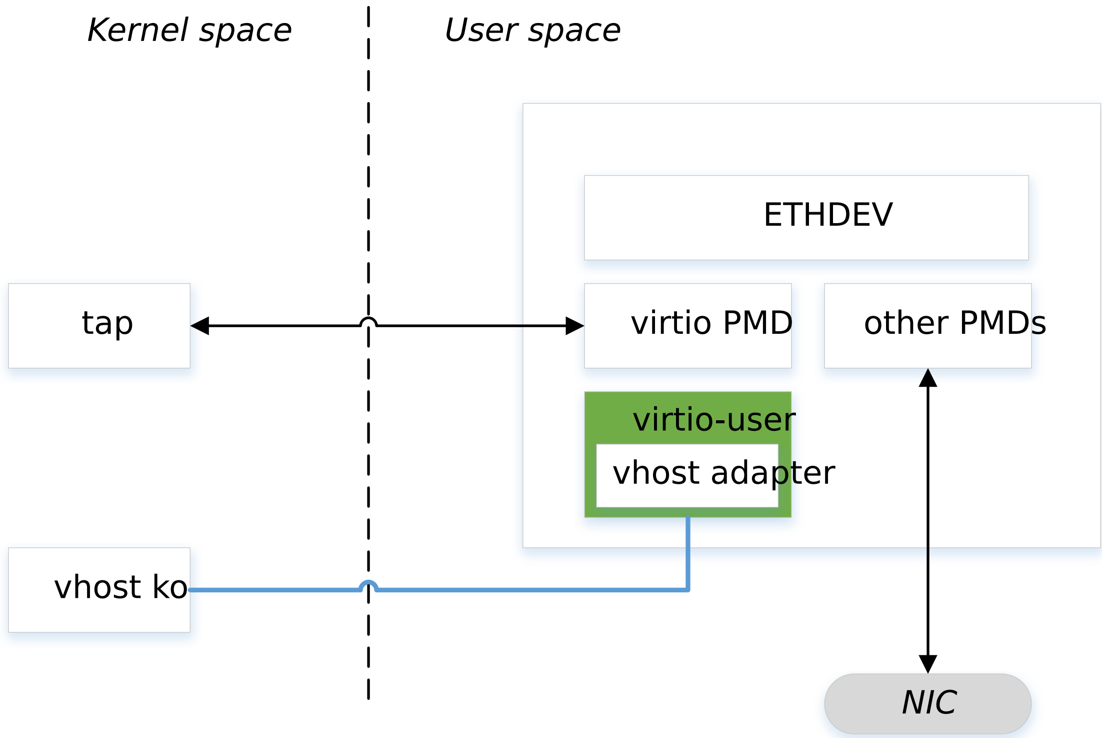


Fig. 6.1: Overview of a DPDK app using virtio-user as excpetional path

This is used to negotiate VIRTIO_NET_F_GUEST_TSO4 and VIRTIO_NET_F_GUEST_TSO6 feature so that large packets from kernel can be transmitted DPDK application and further TSOed by physical NIC.

- `--enable-rx-cksum`

  This is used to negotiate VIRTIO_NET_F_GUEST_CSUM so that packets from kernel can be deemed as valid Rx checksumed.

- `queue_size`

  256 by default. To avoid shortage of descriptors, we can increase it to 1024.

- `queues`

  Number of multi-queues. Each qeueue will be served by a kthread. For example:

```
$(testpmd) -c 0xc -n 4 \
        --vdev=virtio_user0,path=/dev/vhost-net,queues=2,queue_size=1024 \
        -- -i --txqflags=0x0 --disable-hw-vlan --enable-lro \
        --crc-strip --enable-rx-cksum --txq=2 --rxq=2 --rxd=1024 \
        --txd=1024
```

1. Start testpmd:

```
(testpmd) start
```

2. Configure IP address and start tap:

```
ifconfig tap0 1.1.1.1/24 up
```

Note: The tap device will be named tap0, tap1, etc, by kernel.

Then, all traffic from physical NIC can be forwarded into kernel stack, and all traffic on the tap0 can be sent out from physical NIC.

6.2 Limitations

This solution is only available on Linux systems.