DPDK

DATA PLANE DEVELOPMENT KIT

Network Interface Controller Drivers

Release 2.2.0

January 16, 2016

CHAPTER

ONE

BNX2X POLL MODE DRIVER

The BNX2X poll mode driver library (**librte_pmd_bnx2x**) implements support for **QLogic 578xx** 10/20 Gbps family of adapters as well as their virtual functions (VF) in SR-IOV context. It is supported on several standard Linux distros like Red Hat 7.x and SLES12 OS. It is compile-tested under FreeBSD OS.

More information can be found at QLogic Corporation's Official Website.

1.1 Supported Features

BNX2X PMD has support for:

- Base L2 features
- Unicast/multicast filtering
- Promiscuous mode
- Port hardware statistics
- SR-IOV VF

1.2 Non-supported Features

The features not yet supported include:

- TSS (Transmit Side Scaling)
- RSS (Receive Side Scaling)
- LRO/TSO offload
- Checksum offload
- SR-IOV PF

1.3 Co-existence considerations

- BCM578xx being a CNA can have both NIC and Storage personalities. However, coexistence with storage protocol drivers (cnic, bnx2fc and bnx2fi) is not supported on the same adapter. So storage personality has to be disabled on that adapter when used in DPDK applications.

- For SR-IOV case, bnx2x PMD will be used to bind to SR-IOV VF device and Linux native kernel driver (bnx2x) will be attached to SR-IOV PF.

1.4 Supported QLogic NICs

- 578xx

1.5 Prerequisites

- Requires firmware version **7.2.51.0**. It is included in most of the standard Linux distros. If it is not available visit QLogic Driver Download Center to get the required firmware.

1.6 Pre-Installation Configuration

1.6.1 Config File Options

The following options can be modified in the `.config` file. Please note that enabling debugging options may affect system performance.

- `CONFIG_RTE_LIBRTE_BNX2X_PMD` (default **y**)

  Toggle compilation of bnx2x driver.

- `CONFIG_RTE_LIBRTE_BNX2X_DEBUG` (default **n**)

  Toggle display of generic debugging messages.

- `CONFIG_RTE_LIBRTE_BNX2X_DEBUG_INIT` (default **n**)

  Toggle display of initialization related messages.

- `CONFIG_RTE_LIBRTE_BNX2X_DEBUG_TX` (default **n**)

  Toggle display of transmit fast path run-time messages.

- `CONFIG_RTE_LIBRTE_BNX2X_DEBUG_RX` (default **n**)

  Toggle display of receive fast path run-time messages.

- `CONFIG_RTE_LIBRTE_BNX2X_DEBUG_PERIODIC` (default **n**)

  Toggle display of register reads and writes.

1.6.2 Driver Compilation

BNX2X PMD for Linux x86_64 gcc target, run the following "make" command:

```
cd <DPDK-source-directory>
make config T=x86_64-native-linuxapp-gcc install
```

To compile BNX2X PMD for Linux x86_64 clang target, run the following "make" command:

```
cd <DPDK-source-directory>
make config T=x86_64-native-linuxapp-clang install
```

To compile BNX2X PMD for Linux i686 gcc target, run the following "make" command:

```
cd <DPDK-source-directory>
make config T=i686-native-linuxapp-gcc install
```

To compile BNX2X PMD for Linux i686 gcc target, run the following "make" command:

```
cd <DPDK-source-directory>
make config T=i686-native-linuxapp-gcc install
```

To compile BNX2X PMD for FreeBSD x86_64 clang target, run the following "gmake" command:

```
cd <DPDK-source-directory>
gmake config T=x86_64-native-bsdapp-clang install
```

To compile BNX2X PMD for FreeBSD x86_64 gcc target, run the following "gmake" command:

```
cd <DPDK-source-directory>
gmake config T=x86_64-native-bsdapp-gcc install -Wl,-rpath=/usr/local/lib/gcc48 CC=gcc48
```

To compile BNX2X PMD for FreeBSD x86_64 gcc target, run the following "gmake" command:

```
cd <DPDK-source-directory>
gmake config T=x86_64-native-bsdapp-gcc install -Wl,-rpath=/usr/local/lib/gcc48 CC=gcc48
```

1.7 Linux

1.7.1 Linux Installation

1.7.2 Sample Application Notes

This section demonstrates how to launch `testpmd` with QLogic 578xx devices managed by `librte_pmd_bnx2x` in Linux operating system.

1. Request huge pages:

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

2. Load `igb_uio` or `vfio-pci` driver:

   ```
   insmod ./x86_64-native-linuxapp-gcc/kmod/igb_uio.ko
   ```

   or

   ```
   modprobe vfio-pci
   ```

3. Bind the QLogic adapters to `igb_uio` or `vfio-pci` loaded in the previous step:

   ```
   ./tools/dpdk_nic_bind.py --bind igb_uio 0000:84:00.0 0000:84:00.1
   ```

   or

   Setup VFIO permissions for regular users and then bind to `vfio-pci`:

   ```
   sudo chmod a+x /dev/vfio

   sudo chmod 0666 /dev/vfio/*

   ./tools/dpdk_nic_bind.py --bind vfio-pci 0000:84:00.0 0000:84:00.1
   ```

4. Start `testpmd` with basic parameters:

   ```
   ./x86_64-native-linuxapp-gcc/app/testpmd -c 0xf -n 4 -- -i
   ```

Example output:

```
[...]
EAL: PCI device 0000:84:00.0 on NUMA socket 1
EAL:   probe driver: 14e4:168e rte_bnx2x_pmd
EAL:   PCI memory mapped at 0x7f14f6fe5000
EAL:   PCI memory mapped at 0x7f14f67e5000
EAL:   PCI memory mapped at 0x7f15fbd9b000
EAL: PCI device 0000:84:00.1 on NUMA socket 1
EAL:   probe driver: 14e4:168e rte_bnx2x_pmd
EAL:   PCI memory mapped at 0x7f14f5fe5000
EAL:   PCI memory mapped at 0x7f14f57e5000
EAL:   PCI memory mapped at 0x7f15fbd4f000
Interactive-mode selected
Configuring Port 0 (socket 0)
PMD: bnx2x_dev_tx_queue_setup(): fp[00] req_bd=512, thresh=512,
             usable_bd=1020, total_bd=1024,
                          tx_pages=4
PMD: bnx2x_dev_rx_queue_setup(): fp[00] req_bd=128, thresh=0,
             usable_bd=510, total_bd=512,
                          rx_pages=1, cq_pages=8
PMD: bnx2x_print_adapter_info():
[...]
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Port 1 Link Up - speed 10000 Mbps - full-duplex
Done
testpmd>
```

1.7.3 SR-IOV: Prerequisites and sample Application Notes

This section provides instructions to configure SR-IOV with Linux OS.

1. Verify SR-IOV and ARI capabilities are enabled on the adapter using `lspci`:

   ```
   lspci -s <slot> -vvv
   ```

   Example output:

   ```
   [...]
   Capabilities: [1b8 v1] Alternative Routing-ID Interpretation (ARI)
   [...]
   Capabilities: [1c0 v1] Single Root I/O Virtualization (SR-IOV)
   [...]
   Kernel driver in use: igb_uio
   ```

2. Load the kernel module:

   ```
   modprobe bnx2x
   ```

   Example output:

   ```
   systemd-udevd[4848]: renamed network interface eth0 to ens5f0
   systemd-udevd[4848]: renamed network interface eth1 to ens5f1
   ```

3. Bring up the PF ports:

   ```
   ifconfig ens5f0 up
   ifconfig ens5f1 up
   ```

4. Create VF device(s):

   Echo the number of VFs to be created into “sriov_numvfs” sysfs entry of the parent PF.

   Example output:

```
echo 2 > /sys/devices/pci0000:00/0000:00:03.0/0000:81:00.0/sriov_numvfs
```

5. Assign VF MAC address:

   Assign MAC address to the VF using iproute2 utility. The syntax is: ip link set <PF iface> vf <VF id> mac <macaddr>

   Example output:

   ```
   ip link set ens5f0 vf 0 mac 52:54:00:2f:9d:e8
   ```

6. PCI Passthrough:

   The VF devices may be passed through to the guest VM using virt-manager or virsh etc. bnx2x PMD should be used to bind the VF devices in the guest VM using the instructions outlined in the Application notes below.

CHAPTER

TWO

CXGBE POLL MODE DRIVER

The CXGBE PMD (**librte_pmd_cxgbe**) provides poll mode driver support for **Chelsio T5** 10/40 Gbps family of adapters. CXGBE PMD has support for the latest Linux and FreeBSD operating systems.

More information can be found at Chelsio Communications Official Website.

2.1 Features

CXGBE PMD has support for:

- Multiple queues for TX and RX
- Receiver Side Steering (RSS)
- VLAN filtering
- Checksum offload
- Promiscuous mode
- All multicast mode
- Port hardware statistics
- Jumbo frames

2.2 Limitations

The Chelsio T5 devices provide two/four ports but expose a single PCI bus address, thus, librte_pmd_cxgbe registers itself as a PCI driver that allocates one Ethernet device per detected port.

For this reason, one cannot whitelist/blacklist a single port without whitelisting/blacklisting the other ports on the same device.

2.3 Supported Chelsio T5 NICs

- 1G NICs: T502-BT
- 10G NICs: T520-BT, T520-CR, T520-LL-CR, T520-SO-CR, T540-CR

- 40G NICs: T580-CR, T580-LP-CR, T580-SO-CR
- Other T5 NICs: T522-CR

2.4 Prerequisites

- Requires firmware version **1.13.32.0** and higher. Visit Chelsio Download Center to get latest firmware bundled with the latest Chelsio Unified Wire package.

  For Linux, installing and loading the latest cxgb4 kernel driver from the Chelsio Unified Wire package should get you the latest firmware. More information can be obtained from the User Guide that is bundled with the Chelsio Unified Wire package.

  For FreeBSD, the latest firmware obtained from the Chelsio Unified Wire package must be manually flashed via cxgbetool available in FreeBSD source repository.

  Instructions on how to manually flash the firmware are given in section *Linux Installation* for Linux and section *FreeBSD Installation* for FreeBSD.

2.5 Pre-Installation Configuration

2.5.1 Config File Options

The following options can be modified in the `.config` file. Please note that enabling debugging options may affect system performance.

- `CONFIG_RTE_LIBRTE_CXGBE_PMD` (default **y**)

  Toggle compilation of librte_pmd_cxgbe driver.

- `CONFIG_RTE_LIBRTE_CXGBE_DEBUG` (default **n**)

  Toggle display of generic debugging messages.

- `CONFIG_RTE_LIBRTE_CXGBE_DEBUG_REG` (default **n**)

  Toggle display of registers related run-time check messages.

- `CONFIG_RTE_LIBRTE_CXGBE_DEBUG_MBOX` (default **n**)

  Toggle display of firmware mailbox related run-time check messages.

- `CONFIG_RTE_LIBRTE_CXGBE_DEBUG_TX` (default **n**)

  Toggle display of transmission data path run-time check messages.

- `CONFIG_RTE_LIBRTE_CXGBE_DEBUG_RX` (default **n**)

  Toggle display of receiving data path run-time check messages.

2.5.2 Driver Compilation

To compile CXGBE PMD for Linux x86_64 gcc target, run the following “make” command:

```
cd <DPDK-source-directory>
make config T=x86_64-native-linuxapp-gcc install
```

To compile CXGBE PMD for FreeBSD x86_64 clang target, run the following "gmake" command:

```
cd <DPDK-source-directory>
gmake config T=x86_64-native-bsdapp-clang install
```

2.6 Linux

2.6.1 Linux Installation

Steps to manually install the latest firmware from the downloaded Chelsio Unified Wire package for Linux operating system are as follows:

1. Load the kernel module:

   ```
   modprobe cxgb4
   ```

2. Use ifconfig to get the interface name assigned to Chelsio card:

   ```
   ifconfig -a | grep "00:07:43"
   ```

   Example output:

   ```
   p1p1      Link encap:Ethernet  HWaddr 00:07:43:2D:EA:C0
   p1p2      Link encap:Ethernet  HWaddr 00:07:43:2D:EA:C8
   ```

3. Install cxgbtool:

   ```
   cd <path_to_uwire>/tools/cxgbtool
   make install
   ```

4. Use cxgbtool to load the firmware config file onto the card:

   ```
   cxgbtool p1p1 loadcfg <path_to_uwire>/src/network/firmware/t5-config.txt
   ```

5. Use cxgbtool to load the firmware image onto the card:

   ```
   cxgbtool p1p1 loadfw <path_to_uwire>/src/network/firmware/t5fw-*.bin
   ```

6. Unload and reload the kernel module:

   ```
   modprobe -r cxgb4
   modprobe cxgb4
   ```

7. Verify with ethtool:

   ```
   ethtool -i p1p1 | grep "firmware"
   ```

   Example output:

   ```
   firmware-version: 1.13.32.0, TP 0.1.4.8
   ```

2.6.2 Running testpmd

This section demonstrates how to launch **testpmd** with Chelsio T5 devices managed by librte_pmd_cxgbe in Linux operating system.

1. Change to DPDK source directory where the target has been compiled in section *Driver Compilation*:

   ```
   cd <DPDK-source-directory>
   ```

2. Load the kernel module:

```
modprobe cxgb4
```

3. Get the PCI bus addresses of the interfaces bound to cxgb4 driver:

```
dmesg | tail -2
```

Example output:

```
cxgb4 0000:02:00.4 p1p1: renamed from eth0
cxgb4 0000:02:00.4 p1p2: renamed from eth1
```

Note: Both the interfaces of a Chelsio T5 2-port adapter are bound to the same PCI bus address.

4. Unload the kernel module:

```
modprobe -ar cxgb4 csiostor
```

5. Request huge pages:

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

6. Mount huge pages:

```
mkdir /mnt/huge
mount -t hugetlbfs nodev /mnt/huge
```

7. Load igb_uio or vfio-pci driver:

```
insmod ./x86_64-native-linuxapp-gcc/kmod/igb_uio.ko
```

or

```
modprobe vfio-pci
```

8. Bind the Chelsio T5 adapters to igb_uio or vfio-pci loaded in the previous step:

```
./tools/dpdk_nic_bind.py --bind igb_uio 0000:02:00.4
```

or

Setup VFIO permissions for regular users and then bind to vfio-pci:

```
sudo chmod a+x /dev/vfio

sudo chmod 0666 /dev/vfio/*

./tools/dpdk_nic_bind.py --bind vfio-pci 0000:02:00.4
```

Note: Currently, CXGBE PMD only supports the binding of PF4 for Chelsio T5 NICs.

9. Start testpmd with basic parameters:

```
./x86_64-native-linuxapp-gcc/app/testpmd -c 0xf -n 4 -w 0000:02:00.4 -- -i
```

Example output:

```
[...]
EAL: PCI device 0000:02:00.4 on NUMA socket -1
EAL:   probe driver: 1425:5401 rte_cxgbe_pmd
EAL:   PCI memory mapped at 0x7fd7c0200000
EAL:   PCI memory mapped at 0x7fd77cdfd000
EAL:   PCI memory mapped at 0x7fd7c10b7000
PMD: rte_cxgbe_pmd: fw: 1.13.32.0, TP: 0.1.4.8
PMD: rte_cxgbe_pmd: Coming up as MASTER: Initializing adapter
Interactive-mode selected
Configuring Port 0 (socket 0)
```

```
Port 0: 00:07:43:2D:EA:C0
Configuring Port 1 (socket 0)
Port 1: 00:07:43:2D:EA:C8
Checking link statuses...
PMD: rte_cxgbe_pmd: Port0: passive DA port module inserted
PMD: rte_cxgbe_pmd: Port1: passive DA port module inserted
Port 0 Link Up - speed 10000 Mbps - full-duplex
Port 1 Link Up - speed 10000 Mbps - full-duplex
Done
testpmd>
```

Note: Flow control pause TX/RX is disabled by default and can be enabled via testpmd. Refer section *Enable/Disable Flow Control* for more details.

2.7 FreeBSD

2.7.1 FreeBSD Installation

Steps to manually install the latest firmware from the downloaded Chelsio Unified Wire package for FreeBSD operating system are as follows:

1. Load the kernel module:

   ```
   kldload if_cxgbe
   ```

2. Use dmesg to get the t5nex instance assigned to the Chelsio card:

   ```
   dmesg | grep "t5nex"
   ```

   Example output:

   ```
   t5nex0: <Chelsio T520-CR> irq 16 at device 0.4 on pci2
   cxl0: <port 0> on t5nex0
   cxl1: <port 1> on t5nex0
   t5nex0: PCIe x8, 2 ports, 14 MSI-X interrupts, 31 eq, 13 iq
   ```

   In the example above, a Chelsio T520-CR card is bound to a t5nex0 instance.

3. Install cxgbetool from FreeBSD source repository:

   ```
   cd <path_to_FreeBSD_source>/tools/tools/cxgbetool/
   make && make install
   ```

4. Use cxgbetool to load the firmware image onto the card:

   ```
   cxgbetool t5nex0 loadfw <path_to_uwire>/src/network/firmware/t5fw-*.bin
   ```

5. Unload and reload the kernel module:

   ```
   kldunload if_cxgbe
   kldload if_cxgbe
   ```

6. Verify with sysctl:

   ```
   sysctl -a | grep "t5nex" | grep "firmware"
   ```

   Example output:

   ```
   dev.t5nex.0.firmware_version: 1.13.32.0
   ```

2.7.2 Running testpmd

This section demonstrates how to launch **testpmd** with Chelsio T5 devices managed by librte_pmd_cxgbe in FreeBSD operating system.

1. Change to DPDK source directory where the target has been compiled in section *Driver Compilation*:

   ```
   cd <DPDK-source-directory>
   ```

2. Copy the contigmem kernel module to /boot/kernel directory:

   ```
   cp x86_64-native-bsdapp-clang/kmod/contigmem.ko /boot/kernel/
   ```

3. Add the following lines to /boot/loader.conf:

   ```
   # reserve 2 x 1G blocks of contiguous memory using contigmem driver
   hw.contigmem.num_buffers=2
   hw.contigmem.buffer_size=1073741824
   # load contigmem module during boot process
   contigmem_load="YES"
   ```

   The above lines load the contigmem kernel module during boot process and allocate 2 x 1G blocks of contiguous memory to be used for DPDK later on. This is to avoid issues with potential memory fragmentation during later system up time, which may result in failure of allocating the contiguous memory required for the contigmem kernel module.

4. Restart the system and ensure the contigmem module is loaded successfully:

   ```
   reboot
   kldstat | grep "contigmem"
   ```

   Example output:

   ```
   2    1 0xffffffff817f1000 3118     contigmem.ko
   ```

5. Repeat step 1 to ensure that you are in the DPDK source directory.

6. Load the cxgbe kernel module:

   ```
   kldload if_cxgbe
   ```

7. Get the PCI bus addresses of the interfaces bound to t5nex driver:

   ```
   pciconf -l | grep "t5nex"
   ```

   Example output:

   ```
   t5nex0@pci0:2:0:4: class=0x020000 card=0x00001425 chip=0x54011425 rev=0x00
   ```

   In the above example, the t5nex0 is bound to 2:0:4 bus address.

   Note: Both the interfaces of a Chelsio T5 2-port adapter are bound to the same PCI bus address.

   ---

8. Unload the kernel module:

   ```
   kldunload if_cxgbe
   ```

9. Set the PCI bus addresses to hw.nic_uio.bdfs kernel environment parameter:

   ```
   kenv hw.nic_uio.bdfs="2:0:4"
   ```

   This automatically binds 2:0:4 to nic_uio kernel driver when it is loaded in the next step.

Note: Currently, CXGBE PMD only supports the binding of PF4 for Chelsio T5 NICs.

10. Load nic_uio kernel driver:

```
kldload ./x86_64-native-bsdapp-clang/kmod/nic_uio.ko
```

11. Start testpmd with basic parameters:

```
./x86_64-native-bsdapp-clang/app/testpmd -c 0xf -n 4 -w 0000:02:00.4 -- -i
```

Example output:

```
[...]
EAL: PCI device 0000:02:00.4 on NUMA socket 0
EAL:   probe driver: 1425:5401 rte_cxgbe_pmd
EAL:   PCI memory mapped at 0x8007ec000
EAL:   PCI memory mapped at 0x842800000
EAL:   PCI memory mapped at 0x80086c000
PMD: rte_cxgbe_pmd: fw: 1.13.32.0, TP: 0.1.4.8
PMD: rte_cxgbe_pmd: Coming up as MASTER: Initializing adapter
Interactive-mode selected
Configuring Port 0 (socket 0)
Port 0: 00:07:43:2D:EA:C0
Configuring Port 1 (socket 0)
Port 1: 00:07:43:2D:EA:C8
Checking link statuses...
PMD: rte_cxgbe_pmd: Port0: passive DA port module inserted
PMD: rte_cxgbe_pmd: Port1: passive DA port module inserted
Port 0 Link Up - speed 10000 Mbps - full-duplex
Port 1 Link Up - speed 10000 Mbps - full-duplex
Done
testpmd>
```

Note: Flow control pause TX/RX is disabled by default and can be enabled via testpmd. Refer section *Enable/Disable Flow Control* for more details.

2.8 Sample Application Notes

2.8.1 Enable/Disable Flow Control

Flow control pause TX/RX is disabled by default and can be enabled via testpmd as follows:

```
testpmd> set flow_ctrl rx on tx on 0 0 0 0 mac_ctrl_frame_fwd off autoneg on 0
testpmd> set flow_ctrl rx on tx on 0 0 0 0 mac_ctrl_frame_fwd off autoneg on 1
```

To disable again, run:

```
testpmd> set flow_ctrl rx off tx off 0 0 0 0 mac_ctrl_frame_fwd off autoneg off 0
testpmd> set flow_ctrl rx off tx off 0 0 0 0 mac_ctrl_frame_fwd off autoneg off 1
```

2.8.2 Jumbo Mode

There are two ways to enable sending and receiving of jumbo frames via testpmd. One method involves using the **mtu** command, which changes the mtu of an individual port without having to stop the selected port. Another method involves stopping all the ports first and then running **max-pkt-len** command to configure the mtu of all the ports with a single command.

- To configure each port individually, run the mtu command as follows:

```
testpmd> port config mtu 0 9000
testpmd> port config mtu 1 9000
```

- To configure all the ports at once, stop all the ports first and run the max-pkt-len command as follows:

```
testpmd> port stop all
testpmd> port config all max-pkt-len 9000
```

CHAPTER

THREE

DRIVER FOR VM EMULATED DEVICES

The DPDK EM poll mode driver supports the following emulated devices:

- qemu-kvm emulated Intel® 82540EM Gigabit Ethernet Controller (qemu e1000 device)
- VMware* emulated Intel® 82545EM Gigabit Ethernet Controller
- VMware emulated Intel® 8274L Gigabit Ethernet Controller.

3.1 Validated Hypervisors

The validated hypervisors are:

- KVM (Kernel Virtual Machine) with Qemu, version 0.14.0
- KVM (Kernel Virtual Machine) with Qemu, version 0.15.1
- VMware ESXi 5.0, Update 1

3.2 Recommended Guest Operating System in Virtual Machine

The recommended guest operating system in a virtualized environment is:

- Fedora* 18 (64-bit)

For supported kernel versions, refer to the *DPDK Release Notes*.

3.3 Setting Up a KVM Virtual Machine

The following describes a target environment:

- Host Operating System: Fedora 14
- Hypervisor: KVM (Kernel Virtual Machine) with Qemu version, 0.14.0
- Guest Operating System: Fedora 14
- Linux Kernel Version: Refer to the DPDK Getting Started Guide
- Target Applications: testpmd

The setup procedure is as follows:

1. Download qemu-kvm-0.14.0 from http://sourceforge.net/projects/kvm/files/qemu-kvm/ and install it in the Host OS using the following steps:

   When using a recent kernel (2.6.25+) with kvm modules included:

   ```
   tar xzf qemu-kvm-release.tar.gz cd qemu-kvm-release
   ./configure --prefix=/usr/local/kvm
   make
   sudo make install
   sudo /sbin/modprobe kvm-intel
   ```

   When using an older kernel or a kernel from a distribution without the kvm modules, you must download (from the same link), compile and install the modules yourself:

   ```
   tar xjf kvm-kmod-release.tar.bz2
   cd kvm-kmod-release
   ./configure
   make
   sudo make install
   sudo /sbin/modprobe kvm-intel
   ```

   Note that qemu-kvm installs in the /usr/local/bin directory.

   For more details about KVM configuration and usage, please refer to: http://www.linux-kvm.org/page/HOWTO1.

2. Create a Virtual Machine and install Fedora 14 on the Virtual Machine. This is referred to as the Guest Operating System (Guest OS).

3. Start the Virtual Machine with at least one emulated e1000 device.

Note: The Qemu provides several choices for the emulated network device backend. Most commonly used is a TAP networking backend that uses a TAP networking device in the host. For more information about Qemu supported networking backends and different options for configuring networking at Qemu, please refer to:

— http://www.linux-kvm.org/page/Networking

— http://wiki.qemu.org/Documentation/Networking

— http://qemu.weilnetz.de/qemu-doc.html

For example, to start a VM with two emulated e1000 devices, issue the following command:

```
/usr/local/kvm/bin/qemu-system-x86_64 -cpu host -smp 4 -hda qemu1.raw -m 1024
-net nic,model=e1000,vlan=1,macaddr=DE:AD:1E:00:00:01
-net tap,vlan=1,ifname=tapvm01,script=no,downscript=no
-net nic,model=e1000,vlan=2,macaddr=DE:AD:1E:00:00:02
-net tap,vlan=2,ifname=tapvm02,script=no,downscript=no
```

where:

— -m = memory to assign

— -smp = number of smp cores

— -hda = virtual disk image

This command starts a new virtual machine with two emulated 82540EM devices, backed up with two TAP networking host interfaces, tapvm01 and tapvm02.

```
# ip tuntap show
tapvm01: tap
tapvm02: tap
```

4. Configure your TAP networking interfaces using ip/ifconfig tools.
5. Log in to the guest OS and check that the expected emulated devices exist:

```
# lspci -d 8086:100e
00:04.0 Ethernet controller: Intel Corporation 82540EM Gigabit Ethernet Controller (rev 03
00:05.0 Ethernet controller: Intel Corporation 82540EM Gigabit Ethernet Controller (rev 03
```

6. Install the DPDK and run testpmd.

3.4 Known Limitations of Emulated Devices

The following are known limitations:

1. The Qemu e1000 RX path does not support multiple descriptors/buffers per packet. Therefore, rte_mbuf should be big enough to hold the whole packet. For example, to allow testpmd to receive jumbo frames, use the following:

   testpmd [options] – –mbuf-size=<your-max-packet-size>

2. Qemu e1000 does not validate the checksum of incoming packets.
3. Qemu e1000 only supports one interrupt source, so link and Rx interrupt should be exclusive.
4. Qemu e1000 does not support interrupt auto-clear, application should disable interrupt immediately when woken up.

CHAPTER

FOUR

ENIC POLL MODE DRIVER

ENIC PMD is the DPDK poll-mode driver for the Cisco System Inc. VIC Ethernet NICs. These adapters are also referred to as vNICs below. If you are running or would like to run DPDK software applications on Cisco UCS servers using Cisco VIC adapters the following documentation is relevant.

4.1 Version Information

The version of the ENIC PMD driver is 1.0.0.6 and will be printed by ENIC PMD during the initialization.

4.2 How to obtain ENIC PMD integrated DPDK

ENIC PMD support is integrated into the DPDK suite. dpdk-<version>.tar.gz should be downloaded from http://dpdk.org

4.3 Configuration information

- **DPDK Configuration Parameters**

  The following configuration options are available for the ENIC PMD:

  - **CONFIG_RTE_LIBRTE_ENIC_PMD** (default y): Enables or disables inclusion of the ENIC PMD driver in the DPDK compilation.
  - **CONFIG_RTE_LIBRTE_ENIC_DEBUG** (default n): Enables or disables debug logging within the ENIC PMD driver.

- **vNIC Configuration Parameters**

  - **Number of Queues**

    The maximum number of receive and transmit queues are configurable on a per vNIC basis through the Cisco UCS Manager (CIMC or UCSM). These values should be configured to be greater than or equal to the nb_rx_q and nb_tx_q parameters expected to used in the call to the rte_eth_dev_configure() function.

  - **Size of Queues**

Likewise, the number of receive and transmit descriptors are configurable on a per vNIC bases via the UCS Manager and should be greater than or equal to the nb_rx_desc and nb_tx_desc parameters expected to be used in the calls to rte_eth_rx_queue_setup() and rte_eth_tx_queue_setup() respectively.

– **Interrupts**

Only one interrupt per vNIC interface should be configured in the UCS manager regardless of the number receive/transmit queues. The ENIC PMD uses this interrupt to get information about errors in the fast path.

4.4 Limitations

- **VLAN 0 Priority Tagging**

  If a vNIC is configured in TRUNK mode by the UCS manager, the adapter will priority tag egress packets according to 802.1Q if they were not already VLAN tagged by software. If the adapter is connected to a properly configured switch, there will be no unexpected behavior.

  In test setups where an Ethernet port of a Cisco adapter in TRUNK mode is connected point-to-point to another adapter port or connected though a router instead of a switch, all ingress packets will be VLAN tagged. Programs such as l3fwd which do not account for VLAN tags in packets will misbehave. The solution is to enable VLAN stripping on ingress. The follow code fragment is example of how to accomplish this:

```
vlan_offload = rte_eth_dev_get_vlan_offload(port);
vlan_offload |= ETH_VLAN_STRIP_OFFLOAD;
rte_eth_dev_set_vlan_offload(port, vlan_offload);
```

4.5 How to build the suite?

The build instructions for the DPDK suite should be followed. By default the ENIC PMD library will be built into the DPDK library.

For configuring and using UIO and VFIO frameworks, please refer the documentation that comes with DPDK suite.

4.6 Supported Cisco VIC adapters

ENIC PMD supports all recent generations of Cisco VIC adapters including:

- VIC 1280
- VIC 1240
- VIC 1225
- VIC 1285
- VIC 1225T
- VIC 1227

- VIC 1227T
- VIC 1380
- VIC 1340
- VIC 1385
- VIC 1387
- **Flow director features are not supported on generation 1 Cisco VIC adapters** (M81KR and P81E)

4.7 Supported Operating Systems

Any Linux distribution fulfilling the conditions described in Dependencies section of DPDK documentation.

4.8 Supported features

- Unicast, multicast and broadcast transmission and reception
- Receive queue polling
- Port Hardware Statistics
- Hardware VLAN acceleration
- IP checksum offload
- Receive side VLAN stripping
- Multiple receive and transmit queues
- Flow Director ADD, UPDATE, DELETE, STATS operation support for IPV4 5-TUPLE flows
- Promiscuous mode
- Setting RX VLAN (supported via UCSM/CIMC only)
- VLAN filtering (supported via UCSM/CIMC only)
- Execution of application by unprivileged system users
- IPV4, IPV6 and TCP RSS hashing

4.9 Known bugs and Unsupported features in this release

- Signature or flex byte based flow direction
- Drop feature of flow direction
- VLAN based flow direction
- non-IPV4 flow direction
- Setting of extended VLAN

- UDP RSS hashing

4.10 Prerequisites

- Prepare the system as recommended by DPDK suite. This includes environment variables, hugepages configuration, tool-chains and configuration
- Insert vfio-pci kernel module using the command 'modprobe vfio-pci' if the user wants to use VFIO framework
- Insert uio kernel module using the command 'modprobe uio' if the user wants to use UIO framework
- DPDK suite should be configured based on the user's decision to use VFIO or UIO framework
- If the vNIC device(s) to be used is bound to the kernel mode Ethernet driver (enic), use 'ifconfig' to bring the interface down. The dpdk_nic_bind.py tool can then be used to unbind the device's bus id from the enic kernel mode driver.
- Bind the intended vNIC to vfio-pci in case the user wants ENIC PMD to use VFIO framework using dpdk_nic_bind.py.
- Bind the intended vNIC to igb_uio in case the user wants ENIC PMD to use UIO framework using dpdk_nic_bind.py.

At this point the system should be ready to run DPDK applications. Once the application runs to completion, the vNIC can be detached from vfio-pci or igb_uio if necessary.

Root privilege is required to bind and unbind vNICs to/from VFIO/UIO. VFIO framework helps an unprivileged user to run the applications. For an unprivileged user to run the applications on DPDK and ENIC PMD, it may be necessary to increase the maximum locked memory of the user. The following command could be used to do this.

```
sudo sh -c "ulimit -l <value in Kilo Bytes>"
```

The value depends on the memory configuration of the application, DPDK and PMD. Typically, the limit has to be raised to higher than 2GB. e.g., 2621440

The compilation of any unused drivers can be disabled using the configuration file in config/ directory (e.g., config/common_linuxapp). This would help in bringing down the time taken for building the libraries and the initialization time of the application.

4.11 Additional Reference

- http://www.cisco.com/c/en/us/products/servers-unified-computing

4.12 Contact Information

Any questions or bugs should be reported to DPDK community and to the ENIC PMD maintainers:

- John Daley <johndale@cisco.com>

- Sujith Sankar <ssujith@cisco.com>

CHAPTER

FIVE

FM10K POLL MODE DRIVER

The FM10K poll mode driver library provides support for the Intel FM10000 (FM10K) family of 40GbE/100GbE adapters.

5.1 Limitations

5.1.1 Switch manager

The Intel FM10000 family of NICs integrate a hardware switch and multiple host interfaces. The FM10000 PMD driver only manages host interfaces. For the switch component another switch driver has to be loaded prior to to the FM10000 PMD driver. The switch driver can be acquired for Intel support or from the Match Interface project. Only Testpoint is validated with DPDK, the latest version that has been validated with DPDK2.2 is 4.1.6.

5.1.2 CRC striping

The FM10000 family of NICs strip the CRC for every packets coming into the host interface. So, CRC will be stripped even when the `rxmode.hw_strip_crc` member is set to 0 in `struct rte_eth_conf`.

5.1.3 Maximum packet length

The FM10000 family of NICS support a maximum of a 15K jumbo frame. The value is fixed and cannot be changed. So, even when the `rxmode.max_rx_pkt_len` member of `struct rte_eth_conf` is set to a value lower than 15364, frames up to 15364 bytes can still reach the host interface.

CHAPTER

SIX

IXGBE DRIVER

6.1 Vector PMD for IXGBE

Vector PMD uses Intel® SIMD instructions to optimize packet I/O. It improves load/store bandwidth efficiency of L1 data cache by using a wider SSE/AVX register 1 (1). The wider register gives space to hold multiple packet buffers so as to save instruction number when processing bulk of packets.

There is no change to PMD API. The RX/TX handler are the only two entries for vPMD packet I/O. They are transparently registered at runtime RX/TX execution if all condition checks pass.

1. To date, only an SSE version of IX GBE vPMD is available. To ensure that vPMD is in the binary code, ensure that the option CONFIG_RTE_IXGBE_INC_VECTOR=y is in the configure file.

Some constraints apply as pre-conditions for specific optimizations on bulk packet transfers. The following sections explain RX and TX constraints in the vPMD.

6.1.1 RX Constraints

Prerequisites and Pre-conditions

The following prerequisites apply:

- To enable vPMD to work for RX, bulk allocation for Rx must be allowed.

Ensure that the following pre-conditions are satisfied:

- rxq->rx_free_thresh >= RTE_PMD_IXGBE_RX_MAX_BURST
- rxq->rx_free_thresh < rxq->nb_rx_desc
- (rxq->nb_rx_desc % rxq->rx_free_thresh) == 0
- rxq->nb_rx_desc < (IXGBE_MAX_RING_DESC - RTE_PMD_IXGBE_RX_MAX_BURST)

These conditions are checked in the code.

Scattered packets are not supported in this mode. If an incoming packet is greater than the maximum acceptable length of one “mbuf” data size (by default, the size is 2 KB), vPMD for RX would be disabled.

By default, IXGBE_MAX_RING_DESC is set to 4096 and RTE_PMD_IXGBE_RX_MAX_BURST is set to 32.

Feature not Supported by RX Vector PMD

Some features are not supported when trying to increase the throughput in vPMD. They are:

- IEEE1588
- FDIR
- Header split
- RX checksum off load

Other features are supported using optional MACRO configuration. They include:

- HW VLAN strip
- HW extend dual VLAN
- Enabled by RX_OLFLAGS (RTE_IXGBE_RX_OLFLAGS_ENABLE=y)

To guarantee the constraint, configuration flags in dev_conf.rxmode will be checked:

- hw_vlan_strip
- hw_vlan_extend
- hw_ip_checksum
- header_split
- dev_conf

fdir_conf->mode will also be checked.

RX Burst Size

As vPMD is focused on high throughput, it assumes that the RX burst size is equal to or greater than 32 per burst. It returns zero if using nb_pkt < 32 as the expected packet number in the receive handler.

6.1.2 TX Constraint

Prerequisite

The only prerequisite is related to tx_rs_thresh. The tx_rs_thresh value must be greater than or equal to RTE_PMD_IXGBE_TX_MAX_BURST, but less or equal to RTE_IXGBE_TX_MAX_FREE_BUF_SZ. Consequently, by default the tx_rs_thresh value is in the range 32 to 64.

Feature not Supported by RX Vector PMD

TX vPMD only works when txq_flags is set to IXGBE_SIMPLE_FLAGS.

This means that it does not support TX multi-segment, VLAN offload and TX csum offload. The following MACROs are used for these three features:

- ETH_TXQ_FLAGS_NOMULTSEGS

- ETH_TXQ_FLAGS_NOVLANOFFL
- ETH_TXQ_FLAGS_NOXSUMSCTP
- ETH_TXQ_FLAGS_NOXSUMUDP
- ETH_TXQ_FLAGS_NOXSUMTCP

6.1.3 Sample Application Notes

testpmd

By default, using CONFIG_RTE_IXGBE_RX_OLFLAGS_ENABLE=y:

```
./x86_64-native-linuxapp-gcc/app/testpmd -c 300 -n 4 -- -i --burst=32 --rxfreet=32 --mbcache=25
```

When CONFIG_RTE_IXGBE_RX_OLFLAGS_ENABLE=n, better performance can be achieved:

```
./x86_64-native-linuxapp-gcc/app/testpmd -c 300 -n 4 -- -i --burst=32 --rxfreet=32 --mbcache=25
```

l3fwd

When running l3fwd with vPMD, there is one thing to note. In the configuration, ensure that port_conf.rxmode.hw_ip_checksum=0. Otherwise, by default, RX vPMD is disabled.

load_balancer

As in the case of l3fwd, set configure port_conf.rxmode.hw_ip_checksum=0 to enable vPMD. In addition, for improved performance, use -bsz “(32,32),(64,64),(32,32)” in load_balancer to avoid using the default burst size of 144.

CHAPTER

SEVEN

I40E/IXGBE/IGB VIRTUAL FUNCTION DRIVER

Supported Intel® Ethernet Controllers (see the *DPDK Release Notes* for details) support the following modes of operation in a virtualized environment:

- **SR-IOV mode**: Involves direct assignment of part of the port resources to different guest operating systems using the PCI-SIG Single Root I/O Virtualization (SR IOV) standard, also known as "native mode" or "pass-through" mode. In this chapter, this mode is referred to as IOV mode.
- **VMDq mode**: Involves central management of the networking resources by an IO Virtual Machine (IOVM) or a Virtual Machine Monitor (VMM), also known as software switch acceleration mode. In this chapter, this mode is referred to as the Next Generation VMDq mode.

7.1 SR-IOV Mode Utilization in a DPDK Environment

The DPDK uses the SR-IOV feature for hardware-based I/O sharing in IOV mode. Therefore, it is possible to partition SR-IOV capability on Ethernet controller NIC resources logically and expose them to a virtual machine as a separate PCI function called a "Virtual Function". Refer to Fig. 7.1.

Therefore, a NIC is logically distributed among multiple virtual machines (as shown in Fig. 7.1), while still having global data in common to share with the Physical Function and other Virtual Functions. The DPDK fm10kvf, i40evf, igbvf or ixgbevf as a Poll Mode Driver (PMD) serves for the Intel® 82576 Gigabit Ethernet Controller, Intel® Ethernet Controller I350 family, Intel® 82599 10 Gigabit Ethernet Controller NIC, Intel® Fortville 10/40 Gigabit Ethernet Controller NIC's virtual PCI function, or PCIe host-interface of the Intel Ethernet Switch FM10000 Series. Meanwhile the DPDK Poll Mode Driver (PMD) also supports "Physical Function" of such NIC's on the host.

The DPDK PF/VF Poll Mode Driver (PMD) supports the Layer 2 switch on Intel® 82576 Gigabit Ethernet Controller, Intel® Ethernet Controller I350 family, Intel® 82599 10 Gigabit Ethernet Controller, and Intel® Fortville 10/40 Gigabit Ethernet Controller NICs so that guest can choose it for inter virtual machine traffic in SR-IOV mode.

For more detail on SR-IOV, please refer to the following documents:

- SR-IOV provides hardware based I/O sharing
- PCI-SIG-Single Root I/O Virtualization Support on IA
- Scalable I/O Virtualized Servers

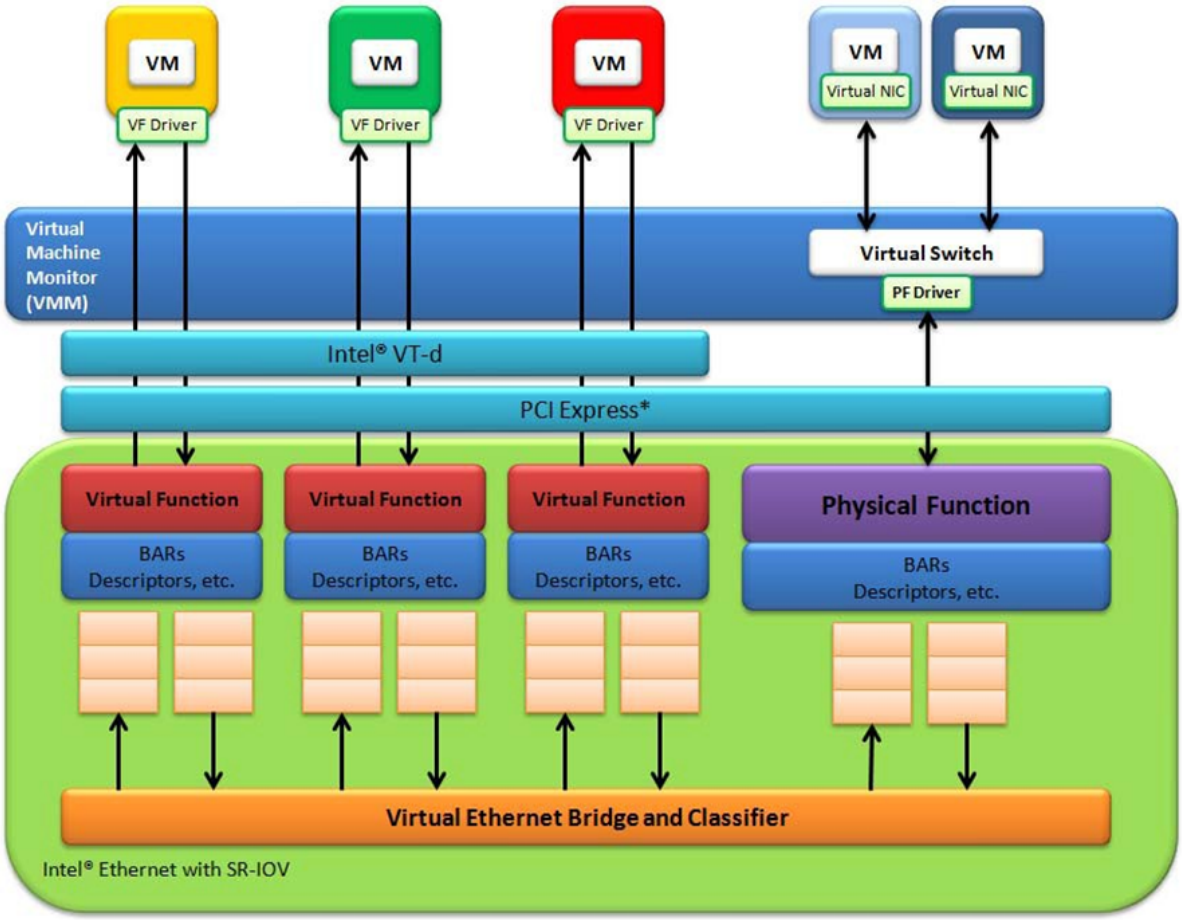

Fig. 7.1: Virtualization for a Single Port NIC in SR-IOV Mode

7.1.1 Physical and Virtual Function Infrastructure

The following describes the Physical Function and Virtual Functions infrastructure for the supported Ethernet Controller NICs.

Virtual Functions operate under the respective Physical Function on the same NIC Port and therefore have no access to the global NIC resources that are shared between other functions for the same NIC port.

A Virtual Function has basic access to the queue resources and control structures of the queues assigned to it. For global resource access, a Virtual Function has to send a request to the Physical Function for that port, and the Physical Function operates on the global resources on behalf of the Virtual Function. For this out-of-band communication, an SR-IOV enabled NIC provides a memory buffer for each Virtual Function, which is called a "Mailbox".

The PCIE host-interface of Intel Ethernet Switch FM10000 Series VF infrastructure

In a virtualized environment, the programmer can enable a maximum of *64 Virtual Functions (VF)* globally per PCIE host-interface of the Intel Ethernet Switch FM10000 Series device. Each VF can have a maximum of 16 queue pairs. The Physical Function in host could be only configured by the Linux* fm10k driver (in the case of the Linux Kernel-based Virtual Machine [KVM]), DPDK PMD PF driver doesn't support it yet.

For example,

- Using Linux* fm10k driver:

```
rmmod fm10k (To remove the fm10k module)
insmod fm0k.ko max_vfs=2,2 (To enable two Virtual Functions per port)
```

Virtual Function enumeration is performed in the following sequence by the Linux* pci driver for a dual-port NIC. When you enable the four Virtual Functions with the above command, the four enabled functions have a Function# represented by (Bus#, Device#, Function#) in sequence starting from 0 to 3. However:

- Virtual Functions 0 and 2 belong to Physical Function 0
- Virtual Functions 1 and 3 belong to Physical Function 1

Note: The above is an important consideration to take into account when targeting specific packets to a selected port.

Intel® Fortville 10/40 Gigabit Ethernet Controller VF Infrastructure

In a virtualized environment, the programmer can enable a maximum of *128 Virtual Functions (VF)* globally per Intel® Fortville 10/40 Gigabit Ethernet Controller NIC device. Each VF can have a maximum of 16 queue pairs. The Physical Function in host could be either configured by the Linux* i40e driver (in the case of the Linux Kernel-based Virtual Machine [KVM]) or by DPDK PMD PF driver. When using both DPDK PMD PF/VF drivers, the whole NIC will be taken over by DPDK based application.

For example,

- Using Linux* i40e driver:

```
rmmod i40e (To remove the i40e module)
insmod i40e.ko max_vfs=2,2 (To enable two Virtual Functions per port)
```

- Using the DPDK PMD PF i40e driver:

  Kernel Params: iommu=pt, intel_iommu=on

  ```
  modprobe uio
  insmod igb_uio
  ./dpdk_nic_bind.py -b igb_uio bb:ss.f
  echo 2 > /sys/bus/pci/devices/0000\:bb\:ss.f/max_vfs (To enable two VFs on a specific PCI
  ```

  Launch the DPDK testpmd/example or your own host daemon application using the DPDK PMD library.

- Using the DPDK PMD PF ixgbe driver to enable VF RSS:

  Same steps as above to install the modules of uio, igb_uio, specify max_vfs for PCI device, and launch the DPDK testpmd/example or your own host daemon application using the DPDK PMD library.

  The available queue number(at most 4) per VF depends on the total number of pool, which is determined by the max number of VF at PF initialization stage and the number of queue specified in config:

  – If the max number of VF is set in the range of 1 to 32:

    If the number of rxq is specified as 4(e.g. '–rxq 4' in testpmd), then there are totally 32 pools(ETH_32_POOLS), and each VF could have 4 or less(e.g. 2) queues;

    If the number of rxq is specified as 2(e.g. '–rxq 2' in testpmd), then there are totally 32 pools(ETH_32_POOLS), and each VF could have 2 queues;

  – If the max number of VF is in the range of 33 to 64:

    If the number of rxq is 4 ('–rxq 4' in testpmd), then error message is expected as rxq is not correct at this case;

    If the number of rxq is 2 ('–rxq 2' in testpmd), then there is totally 64 pools(ETH_64_POOLS), and each VF have 2 queues;

  On host, to enable VF RSS functionality, rx mq mode should be set as ETH_MQ_RX_VMDQ_RSS or ETH_MQ_RX_RSS mode, and SRIOV mode should be activated(max_vfs >= 1). It also needs config VF RSS information like hash function, RSS key, RSS key length.

  ```
  testpmd -c 0xffff -n 4 -- --coremask=<core-mask> --rxq=4 --txq=4 -i
  ```

  The limitation for VF RSS on Intel® 82599 10 Gigabit Ethernet Controller is: The hash and key are shared among PF and all VF, the RETA table with 128 entries is also shared among PF and all VF; So it could not to provide a method to query the hash and reta content per VF on guest, while, if possible, please query them on host(PF) for the shared RETA information.

Virtual Function enumeration is performed in the following sequence by the Linux* pci driver for a dual-port NIC. When you enable the four Virtual Functions with the above command, the four enabled functions have a Function# represented by (Bus#, Device#, Function#) in sequence starting from 0 to 3. However:

- Virtual Functions 0 and 2 belong to Physical Function 0
- Virtual Functions 1 and 3 belong to Physical Function 1

Note: The above is an important consideration to take into account when targeting specific packets to a selected port.

Intel® 82599 10 Gigabit Ethernet Controller VF Infrastructure

The programmer can enable a maximum of *63 Virtual Functions* and there must be *one Physical Function* per Intel® 82599 10 Gigabit Ethernet Controller NIC port. The reason for this is that the device allows for a maximum of 128 queues per port and a virtual/physical function has to have at least one queue pair (RX/TX). The current implementation of the DPDK ixgbevf driver supports a single queue pair (RX/TX) per Virtual Function. The Physical Function in host could be either configured by the Linux* ixgbe driver (in the case of the Linux Kernel-based Virtual Machine [KVM]) or by DPDK PMD PF driver. When using both DPDK PMD PF/VF drivers, the whole NIC will be taken over by DPDK based application.

For example,

- Using Linux* ixgbe driver:

```
rmmod ixgbe (To remove the ixgbe module)
insmod ixgbe max_vfs=2,2 (To enable two Virtual Functions per port)
```

- Using the DPDK PMD PF ixgbe driver:

  Kernel Params: iommu=pt, intel_iommu=on

```
modprobe uio
insmod igb_uio
./dpdk_nic_bind.py -b igb_uio bb:ss.f
echo 2 > /sys/bus/pci/devices/0000\:bb\:ss.f/max_vfs (To enable two VFs on a specific PCI
```

  Launch the DPDK testpmd/example or your own host daemon application using the DPDK PMD library.

Virtual Function enumeration is performed in the following sequence by the Linux* pci driver for a dual-port NIC. When you enable the four Virtual Functions with the above command, the four enabled functions have a Function# represented by (Bus#, Device#, Function#) in sequence starting from 0 to 3. However:

- Virtual Functions 0 and 2 belong to Physical Function 0
- Virtual Functions 1 and 3 belong to Physical Function 1

Note: The above is an important consideration to take into account when targeting specific packets to a selected port.

Intel® 82576 Gigabit Ethernet Controller and Intel® Ethernet Controller I350 Family VF Infrastructure

In a virtualized environment, an Intel® 82576 Gigabit Ethernet Controller serves up to eight virtual machines (VMs). The controller has 16 TX and 16 RX queues. They are generally referred to (or thought of) as queue pairs (one TX and one RX queue). This gives the controller 16 queue pairs.

A pool is a group of queue pairs for assignment to the same VF, used for transmit and receive operations. The controller has eight pools, with each pool containing two queue pairs, that is, two TX and two RX queues assigned to each VF.

In a virtualized environment, an Intel® Ethernet Controller I350 family device serves up to eight virtual machines (VMs) per port. The eight queues can be accessed by eight different VMs if configured correctly (the i350 has 4x1GbE ports each with 8T X and 8 RX queues), that means, one Transmit and one Receive queue assigned to each VF.

For example,

- Using Linux* igb driver:

```
rmmod igb (To remove the igb module)
insmod igb max_vfs=2,2 (To enable two Virtual Functions per port)
```

- Using Intel® DPDK PMD PF igb driver:

  Kernel Params: iommu=pt, intel_iommu=on modprobe uio

```
insmod igb_uio
./dpdk_nic_bind.py -b igb_uio bb:ss.f
echo 2 > /sys/bus/pci/devices/0000\:bb\:ss.f/max_vfs (To enable two VFs on a specific pci
```

  Launch DPDK testpmd/example or your own host daemon application using the DPDK PMD library.

Virtual Function enumeration is performed in the following sequence by the Linux* pci driver for a four-port NIC. When you enable the four Virtual Functions with the above command, the four enabled functions have a Function# represented by (Bus#, Device#, Function#) in sequence, starting from 0 to 7. However:

- Virtual Functions 0 and 4 belong to Physical Function 0
- Virtual Functions 1 and 5 belong to Physical Function 1
- Virtual Functions 2 and 6 belong to Physical Function 2
- Virtual Functions 3 and 7 belong to Physical Function 3

Note: The above is an important consideration to take into account when targeting specific packets to a selected port.

7.1.2 Validated Hypervisors

The validated hypervisor is:

- KVM (Kernel Virtual Machine) with Qemu, version 0.14.0

However, the hypervisor is bypassed to configure the Virtual Function devices using the Mailbox interface, the solution is hypervisor-agnostic. Xen* and VMware* (when SR- IOV is supported) will also be able to support the DPDK with Virtual Function driver support.

7.1.3 Expected Guest Operating System in Virtual Machine

The expected guest operating systems in a virtualized environment are:

- Fedora* 14 (64-bit)

- Ubuntu* 10.04 (64-bit)

For supported kernel versions, refer to the *DPDK Release Notes*.

7.2 Setting Up a KVM Virtual Machine Monitor

The following describes a target environment:

- Host Operating System: Fedora 14
- Hypervisor: KVM (Kernel Virtual Machine) with Qemu version 0.14.0
- Guest Operating System: Fedora 14
- Linux Kernel Version: Refer to the *DPDK Getting Started Guide*
- Target Applications: l2fwd, l3fwd-vf

The setup procedure is as follows:

1. Before booting the Host OS, open **BIOS setup** and enable **Intel® VT features**.
2. While booting the Host OS kernel, pass the intel_iommu=on kernel command line argument using GRUB. When using DPDK PF driver on host, pass the iommu=pt kernel command line argument in GRUB.
3. Download qemu-kvm-0.14.0 from http://sourceforge.net/projects/kvm/files/qemu-kvm/ and install it in the Host OS using the following steps:

   When using a recent kernel (2.6.25+) with kvm modules included:

   ```
   tar xzf qemu-kvm-release.tar.gz
   cd qemu-kvm-release
   ./configure --prefix=/usr/local/kvm
   make
   sudo make install
   sudo /sbin/modprobe kvm-intel
   ```

   When using an older kernel, or a kernel from a distribution without the kvm modules, you must download (from the same link), compile and install the modules yourself:

   ```
   tar xjf kvm-kmod-release.tar.bz2
   cd kvm-kmod-release
   ./configure
   make
   sudo make install
   sudo /sbin/modprobe kvm-intel
   ```

   qemu-kvm installs in the /usr/local/bin directory.

   For more details about KVM configuration and usage, please refer to:

   http://www.linux-kvm.org/page/HOWTO1.

4. Create a Virtual Machine and install Fedora 14 on the Virtual Machine. This is referred to as the Guest Operating System (Guest OS).
5. Download and install the latest ixgbe driver from:

   http://downloadcenter.intel.com/Detail_Desc.aspx?agr=Y&DwnldID=14687

6. In the Host OS

   When using Linux kernel ixgbe driver, unload the Linux ixgbe driver and reload it with the max_vfs=2,2 argument:

   ```
   rmmod ixgbe
   modprobe ixgbe max_vfs=2,2
   ```

   When using DPDK PMD PF driver, insert DPDK kernel module igb_uio and set the number of VF by sysfs max_vfs:

   ```
   modprobe uio
   insmod igb_uio
   ./dpdk_nic_bind.py -b igb_uio 02:00.0 02:00.1 0e:00.0 0e:00.1
   echo 2 > /sys/bus/pci/devices/0000\:02\:00.0/max_vfs
   echo 2 > /sys/bus/pci/devices/0000\:02\:00.1/max_vfs
   echo 2 > /sys/bus/pci/devices/0000\:0e\:00.0/max_vfs
   echo 2 > /sys/bus/pci/devices/0000\:0e\:00.1/max_vfs
   ```

   Note: You need to explicitly specify number of vfs for each port, for example, in the command above, it creates two vfs for the first two ixgbe ports.

   Let say we have a machine with four physical ixgbe ports:

   0000:02:00.0

   0000:02:00.1

   0000:0e:00.0

   0000:0e:00.1

   The command above creates two vfs for device 0000:02:00.0:

   ```
   ls -alrt /sys/bus/pci/devices/0000\:02\:00.0/virt*
   lrwxrwxrwx. 1 root root 0 Apr 13 05:40 /sys/bus/pci/devices/0000:02:00.0/virtfn1 -> ../000
   lrwxrwxrwx. 1 root root 0 Apr 13 05:40 /sys/bus/pci/devices/0000:02:00.0/virtfn0 -> ../000
   ```

   It also creates two vfs for device 0000:02:00.1:

   ```
   ls -alrt /sys/bus/pci/devices/0000\:02\:00.1/virt*
   lrwxrwxrwx. 1 root root 0 Apr 13 05:51 /sys/bus/pci/devices/0000:02:00.1/virtfn1 -> ../000
   lrwxrwxrwx. 1 root root 0 Apr 13 05:51 /sys/bus/pci/devices/0000:02:00.1/virtfn0 -> ../000
   ```

7. List the PCI devices connected and notice that the Host OS shows two Physical Functions (traditional ports) and four Virtual Functions (two for each port). This is the result of the previous step.

8. Insert the pci_stub module to hold the PCI devices that are freed from the default driver using the following command (see http://www.linux-kvm.org/page/How_to_assign_devices_with_VT-d_in_KVM Section 4 for more information):

   ```
   sudo /sbin/modprobe pci-stub
   ```

   Unbind the default driver from the PCI devices representing the Virtual Functions. A script to perform this action is as follows:

   ```
   echo "8086 10ed" > /sys/bus/pci/drivers/pci-stub/new_id
   echo 0000:08:10.0 > /sys/bus/pci/devices/0000:08:10.0/driver/unbind
   echo 0000:08:10.0 > /sys/bus/pci/drivers/pci-stub/bind
   ```

   where, 0000:08:10.0 belongs to the Virtual Function visible in the Host OS.

9. Now, start the Virtual Machine by running the following command:

```
/usr/local/kvm/bin/qemu-system-x86_64 -m 4096 -smp 4 -boot c -hda lucid.qcow2 -device pci-
```

where:

— -m = memory to assign

—-smp = number of smp cores

— -boot = boot option

—-hda = virtual disk image

— -device = device to attach

Note: — The pci-assign,host=08:10.0 alue indicates that you want to attach a PCI device to a Virtual Machine and the respective (Bus:Device.Function) numbers should be passed for the Virtual Function to be attached.

— qemu-kvm-0.14.0 allows a maximum of four PCI devices assigned to a VM, but this is qemu-kvm version dependent since qemu-kvm-0.14.1 allows a maximum of five PCI devices.

— qemu-system-x86_64 also has a -cpu command line option that is used to select the cpu_model to emulate in a Virtual Machine. Therefore, it can be used as:

```
/usr/local/kvm/bin/qemu-system-x86_64 -cpu ?

(to list all available cpu_models)

/usr/local/kvm/bin/qemu-system-x86_64 -m 4096 -cpu host -smp 4 -boot c -hda lucid.qcow2 -c

(to use the same cpu_model equivalent to the host cpu)
```

For more information, please refer to: http://wiki.qemu.org/Features/CPUModels.

---

10. Install and run DPDK host app to take over the Physical Function. Eg.

```
make install T=x86_64-native-linuxapp-gcc
./x86_64-native-linuxapp-gcc/app/testpmd -c f -n 4 -- -i
```

11. Finally, access the Guest OS using vncviewer with the localhost:5900 port and check the lspci command output in the Guest OS. The virtual functions will be listed as available for use.

12. Configure and install the DPDK with an x86_64-native-linuxapp-gcc configuration on the Guest OS as normal, that is, there is no change to the normal installation procedure.

```
make config T=x86_64-native-linuxapp-gcc O=x86_64-native-linuxapp-gcc
cd x86_64-native-linuxapp-gcc
make
```

Note: If you are unable to compile the DPDK and you are getting “error: CPU you selected does not support x86-64 instruction set”, power off the Guest OS and start the virtual machine with the correct -cpu option in the qemu- system-x86_64 command as shown in step 9. You must select the best x86_64 cpu_model to emulate or you can select host option if available.

Note: Run the DPDK l2fwd sample application in the Guest OS with Hugepages enabled. For the expected benchmark performance, you must pin the cores from the Guest OS to the Host

OS (taskset can be used to do this) and you must also look at the PCI Bus layout on the board to ensure you are not running the traffic over the QPI Interface.

Note:

- The Virtual Machine Manager (the Fedora package name is virt-manager) is a utility for virtual machine management that can also be used to create, start, stop and delete virtual machines. If this option is used, step 2 and 6 in the instructions provided will be different.
- virsh, a command line utility for virtual machine management, can also be used to bind and unbind devices to a virtual machine in Ubuntu. If this option is used, step 6 in the instructions provided will be different.
- The Virtual Machine Monitor (see Fig. 7.2) is equivalent to a Host OS with KVM installed as described in the instructions.

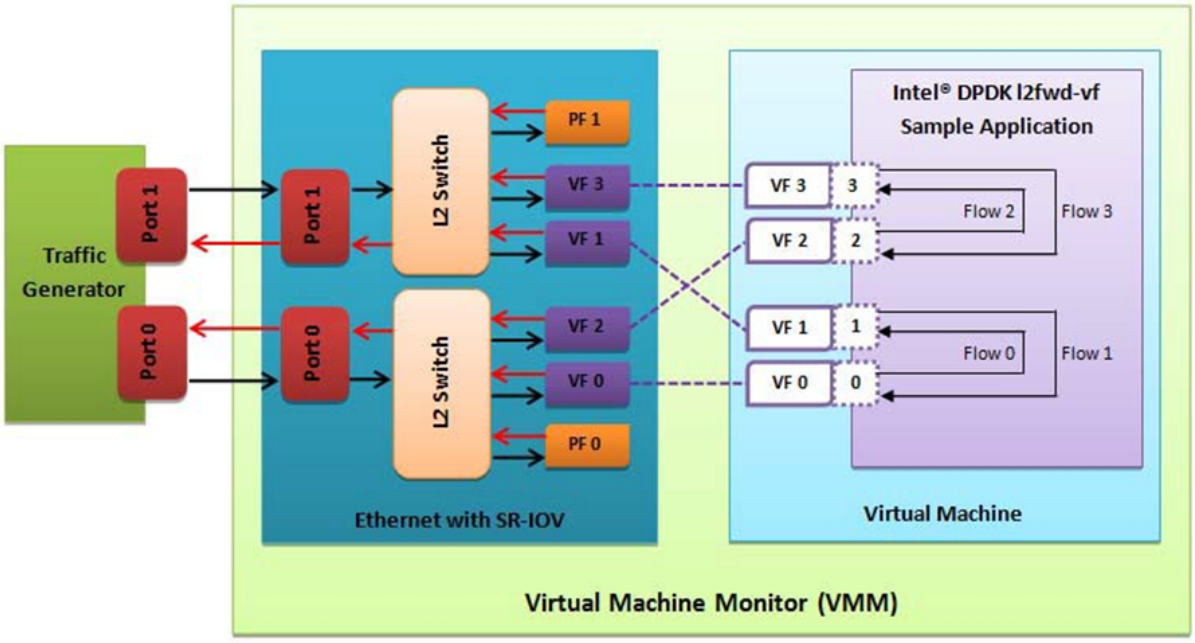

Fig. 7.2: Performance Benchmark Setup

7.3 DPDK SR-IOV PMD PF/VF Driver Usage Model

7.3.1 Fast Host-based Packet Processing

Software Defined Network (SDN) trends are demanding fast host-based packet handling. In a virtualization environment, the DPDK VF PMD driver performs the same throughput result as a non-VT native environment.

With such host instance fast packet processing, lots of services such as filtering, QoS, DPI can be offloaded on the host fast path.

Fig. 7.3 shows the scenario where some VMs directly communicate externally via a VFs, while others connect to a virtual switch and share the same uplink bandwidth.

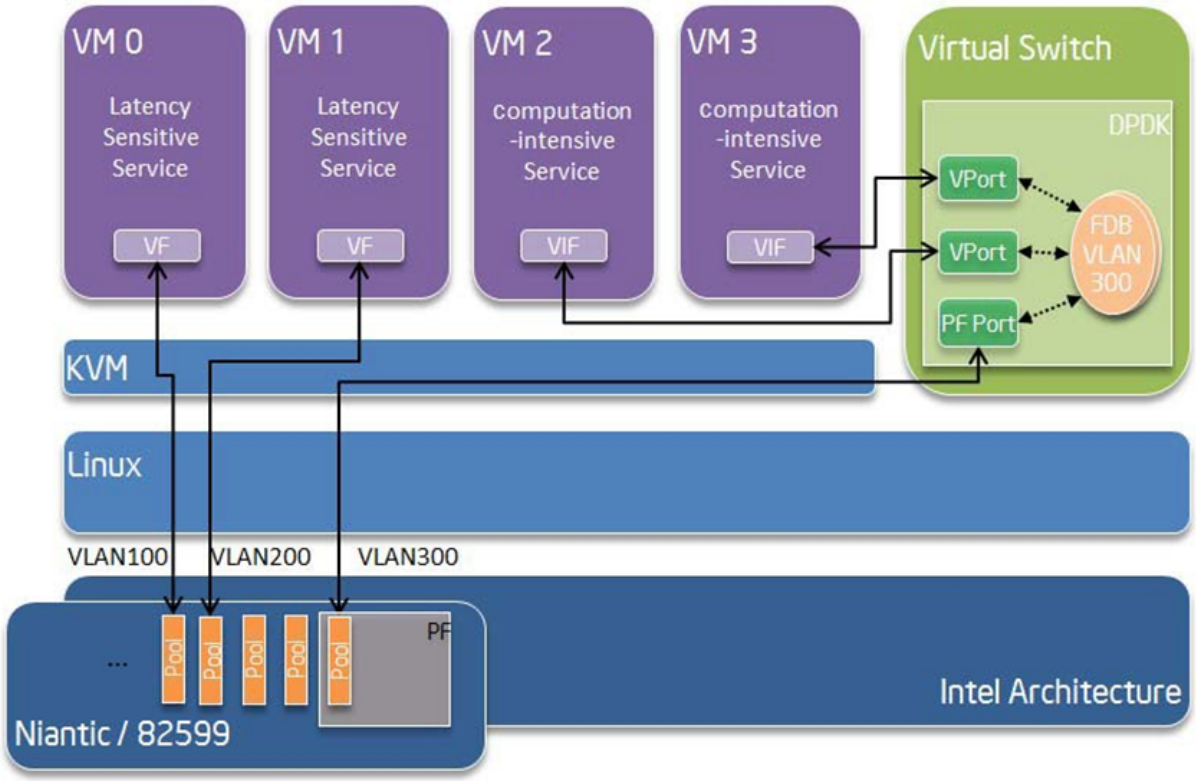

Fig. 7.3: Fast Host-based Packet Processing

7.4 SR-IOV (PF/VF) Approach for Inter-VM Communication

Inter-VM data communication is one of the traffic bottle necks in virtualization platforms. SR-IOV device assignment helps a VM to attach the real device, taking advantage of the bridge in the NIC. So VF-to-VF traffic within the same physical port (VM0<->VM1) have hardware acceleration. However, when VF crosses physical ports (VM0<->VM2), there is no such hardware bridge. In this case, the DPDK PMD PF driver provides host forwarding between such VMs.

Fig. 7.4 shows an example. In this case an update of the MAC address lookup tables in both the NIC and host DPDK application is required.

In the NIC, writing the destination of a MAC address belongs to another cross device VM to the PF specific pool. So when a packet comes in, its destination MAC address will match and forward to the host DPDK PMD application.

In the host DPDK application, the behavior is similar to L2 forwarding, that is, the packet is forwarded to the correct PF pool. The SR-IOV NIC switch forwards the packet to a specific VM according to the MAC destination address which belongs to the destination VF on the VM.

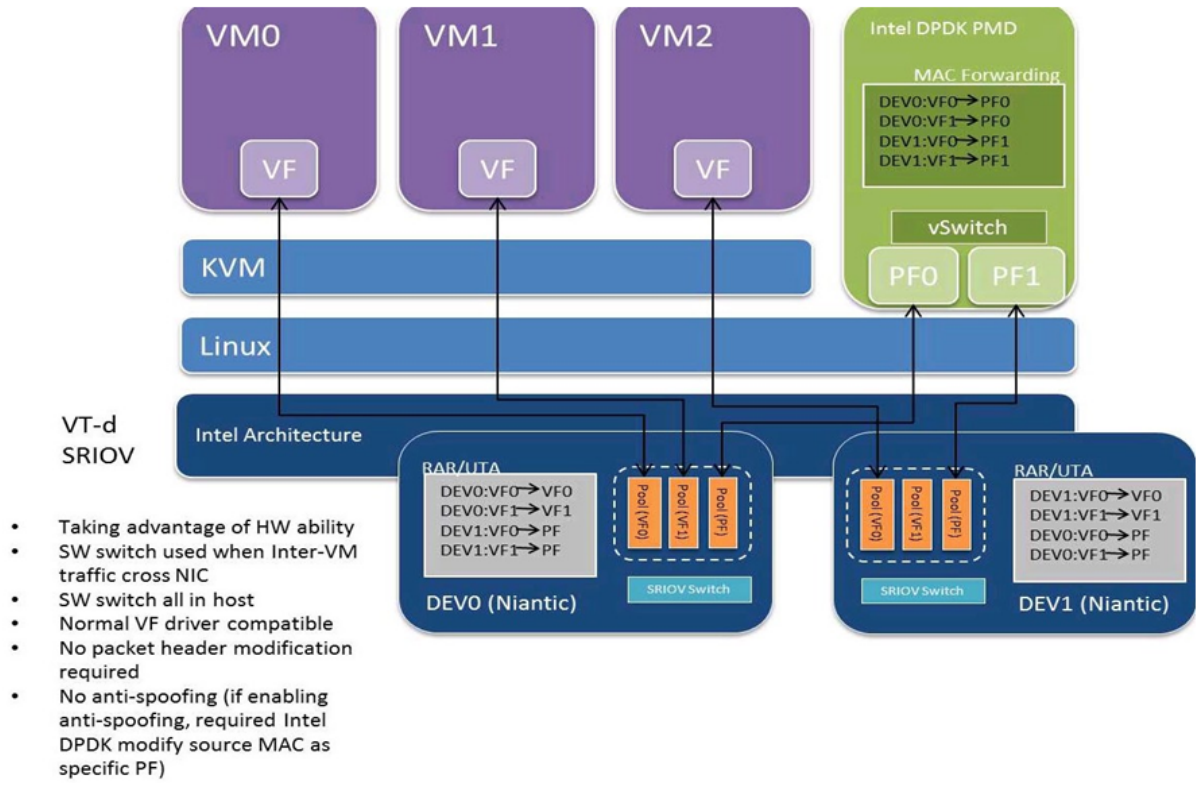

Fig. 7.4: Inter-VM Communication

CHAPTER

EIGHT

MLX4 POLL MODE DRIVER LIBRARY

The MLX4 poll mode driver library (**librte_pmd_mlx4**) implements support for **Mellanox ConnectX-3** and **Mellanox ConnectX-3 Pro** 10/40 Gbps adapters as well as their virtual functions (VF) in SR-IOV context.

Information and documentation about this family of adapters can be found on the Mellanox website. Help is also provided by the Mellanox community.

There is also a section dedicated to this poll mode driver.

Note: Due to external dependencies, this driver is disabled by default. It must be enabled manually by setting `CONFIG_RTE_LIBRTE_MLX4_PMD=y` and recompiling DPDK.

Warning: `CONFIG_RTE_BUILD_COMBINE_LIBS` with `CONFIG_RTE_BUILD_SHARED_LIB` is not supported and thus the compilation will fail with this configuration.

8.1 Implementation details

Most Mellanox ConnectX-3 devices provide two ports but expose a single PCI bus address, thus unlike most drivers, librte_pmd_mlx4 registers itself as a PCI driver that allocates one Ethernet device per detected port.

For this reason, one cannot white/blacklist a single port without also white/blacklisting the others on the same device.

Besides its dependency on libibverbs (that implies libmlx4 and associated kernel support), librte_pmd_mlx4 relies heavily on system calls for control operations such as querying/updating the MTU and flow control parameters.

For security reasons and robustness, this driver only deals with virtual memory addresses. The way resources allocations are handled by the kernel combined with hardware specifications that allow it to handle virtual memory addresses directly ensure that DPDK applications cannot access random physical memory (or memory that does not belong to the current process).

This capability allows the PMD to coexist with kernel network interfaces which remain functional, although they stop receiving unicast packets as long as they share the same MAC address.

Compiling librte_pmd_mlx4 causes DPDK to be linked against libibverbs.

8.2 Features

- RSS, also known as RCA, is supported. In this mode the number of configured RX queues must be a power of two.
- VLAN filtering is supported.
- Link state information is provided.
- Promiscuous mode is supported.
- All multicast mode is supported.
- Multiple MAC addresses (unicast, multicast) can be configured.
- Scattered packets are supported for TX and RX.
- Inner L3/L4 (IP, TCP and UDP) TX/RX checksum offloading and validation.
- Outer L3 (IP) TX/RX checksum offloading and validation for VXLAN frames.
- Secondary process TX is supported.

8.3 Limitations

- RSS hash key cannot be modified.
- RSS RETA cannot be configured
- RSS always includes L3 (IPv4/IPv6) and L4 (UDP/TCP). They cannot be dissociated.
- Hardware counters are not implemented (they are software counters).
- Secondary process RX is not supported.

8.4 Configuration

8.4.1 Compilation options

These options can be modified in the `.config` file.

- `CONFIG_RTE_LIBRTE_MLX4_PMD` (default **n**)

  Toggle compilation of librte_pmd_mlx4 itself.

- `CONFIG_RTE_LIBRTE_MLX4_DEBUG` (default **n**)

  Toggle debugging code and stricter compilation flags. Enabling this option adds additional run-time checks and debugging messages at the cost of lower performance.

- `CONFIG_RTE_LIBRTE_MLX4_SGE_WR_N` (default **4**)

  Number of scatter/gather elements (SGEs) per work request (WR). Lowering this number improves performance but also limits the ability to receive scattered packets (packets that do not fit a single mbuf). The default value is a safe tradeoff.

- `CONFIG_RTE_LIBRTE_MLX4_MAX_INLINE` (default **0**)

  Amount of data to be inlined during TX operations. Improves latency but lowers throughput.

- `CONFIG_RTE_LIBRTE_MLX4_TX_MP_CACHE` (default **8**)

  Maximum number of cached memory pools (MPs) per TX queue. Each MP from which buffers are to be transmitted must be associated to memory regions (MRs). This is a slow operation that must be cached.

  This value is always 1 for RX queues since they use a single MP.

- `CONFIG_RTE_LIBRTE_MLX4_SOFT_COUNTERS` (default **1**)

  Toggle software counters. No counters are available if this option is disabled since hardware counters are not supported.

8.4.2 Environment variables

- `MLX4_INLINE_RECV_SIZE`

  A nonzero value enables inline receive for packets up to that size. May significantly improve performance in some cases but lower it in others. Requires careful testing.

8.4.3 Run-time configuration

- The only constraint when RSS mode is requested is to make sure the number of RX queues is a power of two. This is a hardware requirement.
- librte_pmd_mlx4 brings kernel network interfaces up during initialization because it is affected by their state. Forcing them down prevents packets reception.
- **ethtool** operations on related kernel interfaces also affect the PMD.

8.4.4 Kernel module parameters

The **mlx4_core** kernel module has several parameters that affect the behavior and/or the performance of librte_pmd_mlx4. Some of them are described below.

- **num_vfs** (integer or triplet, optionally prefixed by device address strings)

  Create the given number of VFs on the specified devices.

- **log_num_mgm_entry_size** (integer)

  Device-managed flow steering (DMFS) is required by DPDK applications. It is enabled by using a negative value, the last four bits of which have a special meaning.

  - **-1**: force device-managed flow steering (DMFS).
  - **-7**: configure optimized steering mode to improve performance with the following limitation: VLAN filtering is not supported with this mode. This is the recommended mode in case VLAN filter is not needed.

8.5 Prerequisites

This driver relies on external libraries and kernel drivers for resources allocations and initialization. The following dependencies are not part of DPDK and must be installed separately:

- **libibverbs**

  User space verbs framework used by librte_pmd_mlx4. This library provides a generic interface between the kernel and low-level user space drivers such as libmlx4.

  It allows slow and privileged operations (context initialization, hardware resources allocations) to be managed by the kernel and fast operations to never leave user space.

- **libmlx4**

  Low-level user space driver library for Mellanox ConnectX-3 devices, it is automatically loaded by libibverbs.

  This library basically implements send/receive calls to the hardware queues.

- **Kernel modules** (mlnx-ofed-kernel)

  They provide the kernel-side verbs API and low level device drivers that manage actual hardware initialization and resources sharing with user space processes.

  Unlike most other PMDs, these modules must remain loaded and bound to their devices:

  - mlx4_core: hardware driver managing Mellanox ConnectX-3 devices.
  - mlx4_en: Ethernet device driver that provides kernel network interfaces.
  - mlx4_ib: InifiniBand device driver.
  - ib_uverbs: user space driver for verbs (entry point for libibverbs).

- **Firmware update**

  Mellanox OFED releases include firmware updates for ConnectX-3 adapters.

  Because each release provides new features, these updates must be applied to match the kernel modules and libraries they come with.

Note: Both libraries are BSD and GPL licensed. Linux kernel modules are GPL licensed.

Currently supported by DPDK:

- Mellanox OFED **3.1**.
- Firmware version **2.35.5100** and higher.
- Supported architectures: **x86_64** and **POWER8**.

8.5.1 Getting Mellanox OFED

While these libraries and kernel modules are available on OpenFabrics Alliance's website and provided by package managers on most distributions, this PMD requires Ethernet extensions that may not be supported at the moment (this is a work in progress).

Mellanox OFED includes the necessary support and should be used in the meantime. For DPDK, only libibverbs, libmlx4, mlnx-ofed-kernel packages and firmware updates are required from that distribution.

Note: Several versions of Mellanox OFED are available. Installing the version this DPDK release was developed and tested against is strongly recommended. Please check the *prerequisites*.

8.6 Usage example

This section demonstrates how to launch **testpmd** with Mellanox ConnectX-3 devices managed by librte_pmd_mlx4.

1. Load the kernel modules:

   ```
   modprobe -a ib_uverbs mlx4_en mlx4_core mlx4_ib
   ```

   Alternatively if MLNX_OFED is fully installed, the following script can be run:

   ```
   /etc/init.d/openibd restart
   ```

   Note: User space I/O kernel modules (uio and igb_uio) are not used and do not have to be loaded.

   ---

2. Make sure Ethernet interfaces are in working order and linked to kernel verbs. Related sysfs entries should be present:

   ```
   ls -d /sys/class/net/*/device/infiniband_verbs/uverbs* | cut -d / -f 5
   ```

   Example output:

   ```
   eth2
   eth3
   eth4
   eth5
   ```

3. Optionally, retrieve their PCI bus addresses for whitelisting:

   ```
   {
       for intf in eth2 eth3 eth4 eth5;
       do
           (cd "/sys/class/net/${intf}/device/" && pwd -P);
       done;
   } |
   sed -n 's,.*/\(.*\),-w \1,p'
   ```

   Example output:

   ```
   -w 0000:83:00.0
   -w 0000:83:00.0
   -w 0000:84:00.0
   -w 0000:84:00.0
   ```

   Note: There are only two distinct PCI bus addresses because the Mellanox ConnectX-3 adapters installed on this system are dual port.

   ---

4. Request huge pages:

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

5. Start testpmd with basic parameters:

```
testpmd -c 0xff00 -n 4 -w 0000:83:00.0 -w 0000:84:00.0 -- --rxq=2 --txq=2 -i
```

Example output:

```
[...]
EAL: PCI device 0000:83:00.0 on NUMA socket 1
EAL:   probe driver: 15b3:1007 librte_pmd_mlx4
PMD: librte_pmd_mlx4: PCI information matches, using device "mlx4_0" (VF: false)
PMD: librte_pmd_mlx4: 2 port(s) detected
PMD: librte_pmd_mlx4: port 1 MAC address is 00:02:c9:b5:b7:50
PMD: librte_pmd_mlx4: port 2 MAC address is 00:02:c9:b5:b7:51
EAL: PCI device 0000:84:00.0 on NUMA socket 1
EAL:   probe driver: 15b3:1007 librte_pmd_mlx4
PMD: librte_pmd_mlx4: PCI information matches, using device "mlx4_1" (VF: false)
PMD: librte_pmd_mlx4: 2 port(s) detected
PMD: librte_pmd_mlx4: port 1 MAC address is 00:02:c9:b5:ba:b0
PMD: librte_pmd_mlx4: port 2 MAC address is 00:02:c9:b5:ba:b1
Interactive-mode selected
Configuring Port 0 (socket 0)
PMD: librte_pmd_mlx4: 0x867d60: TX queues number update: 0 -> 2
PMD: librte_pmd_mlx4: 0x867d60: RX queues number update: 0 -> 2
Port 0: 00:02:C9:B5:B7:50
Configuring Port 1 (socket 0)
PMD: librte_pmd_mlx4: 0x867da0: TX queues number update: 0 -> 2
PMD: librte_pmd_mlx4: 0x867da0: RX queues number update: 0 -> 2
Port 1: 00:02:C9:B5:B7:51
Configuring Port 2 (socket 0)
PMD: librte_pmd_mlx4: 0x867de0: TX queues number update: 0 -> 2
PMD: librte_pmd_mlx4: 0x867de0: RX queues number update: 0 -> 2
Port 2: 00:02:C9:B5:BA:B0
Configuring Port 3 (socket 0)
PMD: librte_pmd_mlx4: 0x867e20: TX queues number update: 0 -> 2
PMD: librte_pmd_mlx4: 0x867e20: RX queues number update: 0 -> 2
Port 3: 00:02:C9:B5:BA:B1
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Port 1 Link Up - speed 40000 Mbps - full-duplex
Port 2 Link Up - speed 10000 Mbps - full-duplex
Port 3 Link Up - speed 40000 Mbps - full-duplex
Done
testpmd>
```

CHAPTER

NINE

MLX5 POLL MODE DRIVER

The MLX5 poll mode driver library (**librte_pmd_mlx5**) provides support for **Mellanox ConnectX-4** and **Mellanox ConnectX-4 Lx** families of 10/25/40/50/100 Gb/s adapters as well as their virtual functions (VF) in SR-IOV context.

Information and documentation about these adapters can be found on the Mellanox website. Help is also provided by the Mellanox community.

There is also a section dedicated to this poll mode driver.

Note: Due to external dependencies, this driver is disabled by default. It must be enabled manually by setting `CONFIG_RTE_LIBRTE_MLX5_PMD=y` and recompiling DPDK.

Warning: `CONFIG_RTE_BUILD_COMBINE_LIBS` with `CONFIG_RTE_BUILD_SHARED_LIB` is not supported and thus the compilation will fail with this configuration.

9.1 Implementation details

Besides its dependency on libibverbs (that implies libmlx5 and associated kernel support), librte_pmd_mlx5 relies heavily on system calls for control operations such as querying/updating the MTU and flow control parameters.

For security reasons and robustness, this driver only deals with virtual memory addresses. The way resources allocations are handled by the kernel combined with hardware specifications that allow it to handle virtual memory addresses directly ensure that DPDK applications cannot access random physical memory (or memory that does not belong to the current process).

This capability allows the PMD to coexist with kernel network interfaces which remain functional, although they stop receiving unicast packets as long as they share the same MAC address.

Enabling librte_pmd_mlx5 causes DPDK applications to be linked against libibverbs.

9.2 Features

- Multiple TX and RX queues.
- Support for scattered TX and RX frames.

- IPv4, IPv6, TCPv4, TCPv6, UDPv4 and UDPv6 RSS on any number of queues.
- Several RSS hash keys, one for each flow type.
- Configurable RETA table.
- Support for multiple MAC addresses.
- VLAN filtering.
- Promiscuous mode.
- Multicast promiscuous mode.
- Hardware checksum offloads.

9.3 Limitations

- KVM and VMware ESX SR-IOV modes are not supported yet.
- Inner RSS for VXLAN frames is not supported yet.
- Port statistics through software counters only.
- Hardware checksum offloads for VXLAN inner header are not supported yet.
- Secondary processes are not supported yet.

9.4 Configuration

9.4.1 Compilation options

These options can be modified in the `.config` file.

- `CONFIG_RTE_LIBRTE_MLX5_PMD` (default **n**)

  Toggle compilation of librte_pmd_mlx5 itself.

- `CONFIG_RTE_LIBRTE_MLX5_DEBUG` (default **n**)

  Toggle debugging code and stricter compilation flags. Enabling this option adds additional run-time checks and debugging messages at the cost of lower performance.

- `CONFIG_RTE_LIBRTE_MLX5_SGE_WR_N` (default **4**)

  Number of scatter/gather elements (SGEs) per work request (WR). Lowering this number improves performance but also limits the ability to receive scattered packets (packets that do not fit a single mbuf). The default value is a safe tradeoff.

- `CONFIG_RTE_LIBRTE_MLX5_MAX_INLINE` (default **0**)

  Amount of data to be inlined during TX operations. Improves latency. Can improve PPS performance when PCI backpressure is detected and may be useful for scenarios involving heavy traffic on many queues.

  Since the additional software logic necessary to handle this mode can lower performance when there is no backpressure, it is not enabled by default.

- `CONFIG_RTE_LIBRTE_MLX5_TX_MP_CACHE` (default **8**)

  Maximum number of cached memory pools (MPs) per TX queue. Each MP from which buffers are to be transmitted must be associated to memory regions (MRs). This is a slow operation that must be cached.

  This value is always 1 for RX queues since they use a single MP.

9.4.2 Environment variables

- `MLX5_ENABLE_CQE_COMPRESSION`

  A nonzero value lets ConnectX-4 return smaller completion entries to improve performance when PCI backpressure is detected. It is most useful for scenarios involving heavy traffic on many queues.

  Since the additional software logic necessary to handle this mode can lower performance when there is no backpressure, it is not enabled by default.

9.4.3 Run-time configuration

- librte_pmd_mlx5 brings kernel network interfaces up during initialization because it is affected by their state. Forcing them down prevents packets reception.
- **ethtool** operations on related kernel interfaces also affect the PMD.

9.5 Prerequisites

This driver relies on external libraries and kernel drivers for resources allocations and initialization. The following dependencies are not part of DPDK and must be installed separately:

- **libibverbs**

  User space Verbs framework used by librte_pmd_mlx5. This library provides a generic interface between the kernel and low-level user space drivers such as libmlx5.

  It allows slow and privileged operations (context initialization, hardware resources allocations) to be managed by the kernel and fast operations to never leave user space.

- **libmlx5**

  Low-level user space driver library for Mellanox ConnectX-4 devices, it is automatically loaded by libibverbs.

  This library basically implements send/receive calls to the hardware queues.

- **Kernel modules** (mlnx-ofed-kernel)

  They provide the kernel-side Verbs API and low level device drivers that manage actual hardware initialization and resources sharing with user space processes.

  Unlike most other PMDs, these modules must remain loaded and bound to their devices:

  - mlx5_core: hardware driver managing Mellanox ConnectX-4 devices and related Ethernet kernel network devices.
  - mlx5_ib: InifiniBand device driver.

  - ib_uverbs: user space driver for Verbs (entry point for libibverbs).

- **Firmware update**

  Mellanox OFED releases include firmware updates for ConnectX-4 adapters.

  Because each release provides new features, these updates must be applied to match the kernel modules and libraries they come with.

Note: Both libraries are BSD and GPL licensed. Linux kernel modules are GPL licensed.

Currently supported by DPDK:

- Mellanox OFED **3.1-1.0.3** or **3.1-1.5.7.1** depending on usage.

  The following features are supported with version **3.1-1.5.7.1** and above only:

  - IPv6, UPDv6, TCPv6 RSS.
  - RX checksum offloads.
  - IBM POWER8.

- Minimum firmware version:

  With MLNX_OFED **3.1-1.0.3**:

  - ConnectX-4: **12.12.1240**
  - ConnectX-4 Lx: **14.12.1100**

  With MLNX_OFED **3.1-1.5.7.1**:

  - ConnectX-4: **12.13.0144**
  - ConnectX-4 Lx: **14.13.0144**

9.5.1 Getting Mellanox OFED

While these libraries and kernel modules are available on OpenFabrics Alliance's website and provided by package managers on most distributions, this PMD requires Ethernet extensions that may not be supported at the moment (this is a work in progress).

Mellanox OFED includes the necessary support and should be used in the meantime. For DPDK, only libibverbs, libmlx5, mlnx-ofed-kernel packages and firmware updates are required from that distribution.

Note: Several versions of Mellanox OFED are available. Installing the version this DPDK release was developed and tested against is strongly recommended. Please check the *prerequisites*.

9.6 Notes for testpmd

Compared to librte_pmd_mlx4 that implements a single RSS configuration per port, librte_pmd_mlx5 supports per-protocol RSS configuration.

Since `testpmd` defaults to IP RSS mode and there is currently no command-line parameter to enable additional protocols (UDP and TCP as well as IP), the following commands must be entered from its CLI to get the same behavior as librte_pmd_mlx4:

```
> port stop all
> port config all rss all
> port start all
```

9.7 Usage example

This section demonstrates how to launch **testpmd** with Mellanox ConnectX-4 devices managed by librte_pmd_mlx5.

1. Load the kernel modules:

   ```
   modprobe -a ib_uverbs mlx5_core mlx5_ib
   ```

   Alternatively if MLNX_OFED is fully installed, the following script can be run:

   ```
   /etc/init.d/openibd restart
   ```

   Note: User space I/O kernel modules (uio and igb_uio) are not used and do not have to be loaded.

   ---

2. Make sure Ethernet interfaces are in working order and linked to kernel verbs. Related sysfs entries should be present:

   ```
   ls -d /sys/class/net/*/device/infiniband_verbs/uverbs* | cut -d / -f 5
   ```

   Example output:

   ```
   eth30
   eth31
   eth32
   eth33
   ```

3. Optionally, retrieve their PCI bus addresses for whitelisting:

   ```
   {
       for intf in eth2 eth3 eth4 eth5;
       do
           (cd "/sys/class/net/${intf}/device/" && pwd -P);
       done;
   } |
   sed -n 's,.*/\(.*\),-w \1,p'
   ```

   Example output:

   ```
   -w 0000:05:00.1
   -w 0000:06:00.0
   -w 0000:06:00.1
   -w 0000:05:00.0
   ```

4. Request huge pages:

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

5. Start testpmd with basic parameters:

   ```
   testpmd -c 0xff00 -n 4 -w 05:00.0 -w 05:00.1 -w 06:00.0 -w 06:00.1 -- --rxq=2 --txq=2 -i
   ```

   Example output:

```
[...]
EAL: PCI device 0000:05:00.0 on NUMA socket 0
EAL:   probe driver: 15b3:1013 librte_pmd_mlx5
PMD: librte_pmd_mlx5: PCI information matches, using device "mlx5_0" (VF: false)
PMD: librte_pmd_mlx5: 1 port(s) detected
PMD: librte_pmd_mlx5: port 1 MAC address is e4:1d:2d:e7:0c:fe
EAL: PCI device 0000:05:00.1 on NUMA socket 0
EAL:   probe driver: 15b3:1013 librte_pmd_mlx5
PMD: librte_pmd_mlx5: PCI information matches, using device "mlx5_1" (VF: false)
PMD: librte_pmd_mlx5: 1 port(s) detected
PMD: librte_pmd_mlx5: port 1 MAC address is e4:1d:2d:e7:0c:ff
EAL: PCI device 0000:06:00.0 on NUMA socket 0
EAL:   probe driver: 15b3:1013 librte_pmd_mlx5
PMD: librte_pmd_mlx5: PCI information matches, using device "mlx5_2" (VF: false)
PMD: librte_pmd_mlx5: 1 port(s) detected
PMD: librte_pmd_mlx5: port 1 MAC address is e4:1d:2d:e7:0c:fa
EAL: PCI device 0000:06:00.1 on NUMA socket 0
EAL:   probe driver: 15b3:1013 librte_pmd_mlx5
PMD: librte_pmd_mlx5: PCI information matches, using device "mlx5_3" (VF: false)
PMD: librte_pmd_mlx5: 1 port(s) detected
PMD: librte_pmd_mlx5: port 1 MAC address is e4:1d:2d:e7:0c:fb
Interactive-mode selected
Configuring Port 0 (socket 0)
PMD: librte_pmd_mlx5: 0x8cba80: TX queues number update: 0 -> 2
PMD: librte_pmd_mlx5: 0x8cba80: RX queues number update: 0 -> 2
Port 0: E4:1D:2D:E7:0C:FE
Configuring Port 1 (socket 0)
PMD: librte_pmd_mlx5: 0x8ccac8: TX queues number update: 0 -> 2
PMD: librte_pmd_mlx5: 0x8ccac8: RX queues number update: 0 -> 2
Port 1: E4:1D:2D:E7:0C:FF
Configuring Port 2 (socket 0)
PMD: librte_pmd_mlx5: 0x8cdb10: TX queues number update: 0 -> 2
PMD: librte_pmd_mlx5: 0x8cdb10: RX queues number update: 0 -> 2
Port 2: E4:1D:2D:E7:0C:FA
Configuring Port 3 (socket 0)
PMD: librte_pmd_mlx5: 0x8ceb58: TX queues number update: 0 -> 2
PMD: librte_pmd_mlx5: 0x8ceb58: RX queues number update: 0 -> 2
Port 3: E4:1D:2D:E7:0C:FB
Checking link statuses...
Port 0 Link Up - speed 40000 Mbps - full-duplex
Port 1 Link Up - speed 40000 Mbps - full-duplex
Port 2 Link Up - speed 10000 Mbps - full-duplex
Port 3 Link Up - speed 10000 Mbps - full-duplex
Done
testpmd>
```

CHAPTER

TEN

NFP POLL MODE DRIVER LIBRARY

Netronome's sixth generation of flow processors pack 216 programmable cores and over 100 hardware accelerators that uniquely combine packet, flow, security and content processing in a single device that scales up to 400 Gbps.

This document explains how to use DPDK with the Netronome Poll Mode Driver (PMD) supporting Netronome's Network Flow Processor 6xxx (NFP-6xxx).

Currently the driver supports virtual functions (VFs) only.

10.1 Dependencies

Before using the Netronome's DPDK PMD some NFP-6xxx configuration, which is not related to DPDK, is required. The system requires installation of **Netronome's BSP (Board Support Package)** which includes Linux drivers, programs and libraries.

If you have a NFP-6xxx device you should already have the code and documentation for doing this configuration. Contact **support@netronome.com** to obtain the latest available firmware.

The NFP Linux kernel drivers (including the required PF driver for the NFP) are available on Github at **https://github.com/Netronome/nfp-drv-kmods** along with build instructions.

DPDK runs in userspace and PMDs uses the Linux kernel UIO interface to allow access to physical devices from userspace. The NFP PMD requires a separate UIO driver, **nfp_uio**, to perform correct initialization. This driver is part of Netronome´s BSP and it is equivalent to Intel's igb_uio driver.

10.2 Building the software

Netronome's PMD code is provided in the **drivers/net/nfp** directory. Because Netronome´s BSP dependencies the driver is disabled by default in DPDK build using **common_linuxapp configuration** file. Enabling the driver or if you use another configuration file and want to have NFP support, this variable is needed:

- **CONFIG_RTE_LIBRTE_NFP_PMD=y**

Once DPDK is built all the DPDK apps and examples include support for the NFP PMD.

10.3 System configuration

Using the NFP PMD is not different to using other PMDs. Usual steps are:

1. **Configure hugepages:** All major Linux distributions have the hugepages functionality enabled by default. By default this allows the system uses for working with transparent hugepages. But in this case some hugepages need to be created/reserved for use with the DPDK through the hugetlbfs file system. First the virtual file system need to be mounted:

   ```
   mount -t hugetlbfs none /mnt/hugetlbfs
   ```

   The command uses the common mount point for this file system and it needs to be created if necessary.

   Configuring hugepages is performed via sysfs:

   ```
   /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
   ```

   This sysfs file is used to specify the number of hugepages to reserve. For example:

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

   This will reserve 2GB of memory using 1024 2MB hugepages. The file may be read to see if the operation was performed correctly:

   ```
   cat /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
   ```

   The number of unused hugepages may also be inspected.

   Before executing the DPDK app it should match the value of nr_hugepages.

   ```
   cat /sys/kernel/mm/hugepages/hugepages-2048kB/free_hugepages
   ```

   The hugepages reservation should be performed at system initialization and it is usual to use a kernel parameter for configuration. If the reservation is attempted on a busy system it will likely fail. Reserving memory for hugepages may be done adding the following to the grub kernel command line:

   ```
   default_hugepagesz=1M hugepagesz=2M hugepages=1024
   ```

   This will reserve 2GBytes of memory using 2Mbytes huge pages.

   Finally, for a NUMA system the allocation needs to be made on the correct NUMA node. In a DPDK app there is a master core which will (usually) perform memory allocation. It is important that some of the hugepages are reserved on the NUMA memory node where the network device is attached. This is because of a restriction in DPDK by which TX and RX descriptors rings must be created on the master code.

   Per-node allocation of hugepages may be inspected and controlled using sysfs. For example:

   ```
   cat /sys/devices/system/node/node0/hugepages/hugepages-2048kB/nr_hugepages
   ```

   For a NUMA system there will be a specific hugepage directory per node allowing control of hugepage reservation. A common problem may occur when hugepages reservation is performed after the system has been working for some time. Configuration using the global sysfs hugepage interface will succeed but the per-node allocations may be unsatisfactory.

   The number of hugepages that need to be reserved depends on how the app uses TX and RX descriptors, and packets mbufs.

2. **Enable SR-IOV on the NFP-6xxx device:** The current NFP PMD works with Virtual Functions (VFs) on a NFP device. Make sure that one of the Physical Function (PF) drivers from the above Github repository is installed and loaded.

   Virtual Functions need to be enabled before they can be used with the PMD. Before enabling the VFs it is useful to obtain information about the current NFP PCI device detected by the system:

   ```
   lspci -d19ee:
   ```

   Now, for example, configure two virtual functions on a NFP-6xxx device whose PCI system identity is "0000:03:00.0":

   ```
   echo 2 > /sys/bus/pci/devices/0000:03:00.0/sriov_numvfs
   ```

   The result of this command may be shown using lspci again:

   ```
   lspci -d19ee: -k
   ```

   Two new PCI devices should appear in the output of the above command. The -k option shows the device driver, if any, that devices are bound to. Depending on the modules loaded at this point the new PCI devices may be bound to nfp_netvf driver.

3. **To install the uio kernel module (manually):** All major Linux distributions have support for this kernel module so it is straightforward to install it:

   ```
   modprobe uio
   ```

   The module should now be listed by the lsmod command.

4. **To install the nfp_uio kernel module (manually):** This module supports NFP-6xxx devices through the UIO interface.

   This module is part of Netronome´s BSP and it should be available when the BSP is installed.

   ```
   modprobe nfp_uio.ko
   ```

   The module should now be listed by the lsmod command.

   Depending on which NFP modules are loaded, nfp_uio may be automatically bound to the NFP PCI devices by the system. Otherwise the binding needs to be done explicitly. This is the case when nfp_netvf, the Linux kernel driver for NFP VFs, was loaded when VFs were created. As described later in this document this configuration may also be performed using scripts provided by the Netronome´s BSP.

   First the device needs to be unbound, for example from the nfp_netvf driver:

   ```
   echo 0000:03:08.0 > /sys/bus/pci/devices/0000:03:08.0/driver/unbind

   lspci -d19ee: -k
   ```

   The output of lspci should now show that 0000:03:08.0 is not bound to any driver.

   The next step is to add the NFP PCI ID to the NFP UIO driver:

   ```
   echo 19ee 6003 > /sys/bus/pci/drivers/nfp_uio/new_id
   ```

   And then to bind the device to the nfp_uio driver:

   ```
   echo 0000:03:08.0 > /sys/bus/pci/drivers/nfp_uio/bind

   lspci -d19ee: -k
   ```

   lspci should show that device bound to nfp_uio driver.

5. **Using tools from Netronome´s BSP to install and bind modules:** DPDK provides scripts which are useful for installing the UIO modules and for binding the right device to those modules avoiding doing so manually. However, these scripts have not support for Netronome´s UIO driver. Along with drivers, the BSP installs those DPDK scripts slightly modified with support for Netronome´s UIO driver.

   Those specific scripts can be found in Netronome´s BSP installation directory. Refer to BSP documentation for more information.

   - **setup.sh**
   - **dpdk_nic_bind.py**

   Configuration may be performed by running setup.sh which invokes dpdk_nic_bind.py as needed. Executing setup.sh will display a menu of configuration options.

CHAPTER

ELEVEN

SZEDATA2 POLL MODE DRIVER LIBRARY

The SZEDATA2 poll mode driver library implements support for cards from COMBO family (**COMBO-80G**, **COMBO-100G**). The SZEDATA2 PMD is virtual PMD which uses interface provided by libsze2 library to communicate with COMBO cards over sze2 layer.

More information about family of COMBO cards and used technology (NetCOPE platform) can be found on the Liberouter website.

Note: This driver has external dependencies. Therefore it is disabled in default configuration files. It can be enabled by setting `CONFIG_RTE_LIBRTE_PMD_SZEDATA2=y` and recompiling.

Note: Currently the driver is supported only on x86_64 architectures. Only x86_64 versions of the external libraries are provided.

11.1 Prerequisites

This PMD requires kernel modules which are responsible for initialization and allocation of resources needed for sze2 layer function. Communication between PMD and kernel modules is mediated by libsze2 library. These kernel modules and library are not part of DPDK and must be installed separately:

- **libsze2 library**

  The library provides API for initialization of sze2 transfers, receiving and transmitting data segments.

- **Kernel modules**

  - combov3
  - szedata2_cv3

  Kernel modules manage initialization of hardware, allocation and sharing of resources for user space applications:

Information about getting the dependencies can be found here.

11.2 Using the SZEDATA2 PMD

SZEDATA2 PMD can be created by passing `--vdev=` option to EAL in the following format:

```
--vdev 'DEVICE,dev_path=PATH,rx_ifaces=RX_MASK,tx_ifaces=TX_MASK'
```

DEVICE and options dev_path, rx_ifaces, tx_ifaces are mandatory and must be separated by commas.

- DEVICE: contains prefix eth_szedata2 followed by numbers or letters, must be unique for each virtual device
- dev_path: Defines path to szedata2 device. Value is valid path to szedata2 device. Example:

  ```
  dev_path=/dev/szedataII0
  ```

- rx_ifaces: Defines which receive channels will be used. For each channel is created one queue. Value is mask for selecting which receive channels are required. Example:

  ```
  rx_ifaces=0x3
  ```

- tx_ifaces: Defines which transmit channels will be used. For each channel is created one queue. Value is mask for selecting which transmit channels are required. Example:

  ```
  tx_ifaces=0x3
  ```

11.3 Example of usage

Read packets from 0. and 1. receive channel and write them to 0. and 1. transmit channel:

```
$RTE_TARGET/app/testpmd -c 0xf -n 2 \
--vdev 'eth_szedata20,dev_path=/dev/szedataII0,rx_ifaces=0x3,tx_ifaces=0x3' \
-- --port-topology=chained --rxq=2 --txq=2 --nb-cores=2
```

CHAPTER

TWELVE

POLL MODE DRIVER FOR EMULATED VIRTIO NIC

Virtio is a para-virtualization framework initiated by IBM, and supported by KVM hypervisor. In the Data Plane Development Kit (DPDK), we provide a virtio Poll Mode Driver (PMD) as a software solution, comparing to SRIOV hardware solution, for fast guest VM to guest VM communication and guest VM to host communication.

Vhost is a kernel acceleration module for virtio qemu backend. The DPDK extends kni to support vhost raw socket interface, which enables vhost to directly read/ write packets from/to a physical port. With this enhancement, virtio could achieve quite promising performance.

In future release, we will also make enhancement to vhost backend, releasing peak performance of virtio PMD driver.

For basic qemu-KVM installation and other Intel EM poll mode driver in guest VM, please refer to Chapter “Driver for VM Emulated Devices”.

In this chapter, we will demonstrate usage of virtio PMD driver with two backends, standard qemu vhost back end and vhost kni back end.

12.1 Virtio Implementation in DPDK

For details about the virtio spec, refer to Virtio PCI Card Specification written by Rusty Russell.

As a PMD, virtio provides packet reception and transmission callbacks virtio_recv_pkts and virtio_xmit_pkts.

In virtio_recv_pkts, index in range [vq->vq_used_cons_idx , vq->vq_ring.used->idx) in vring is available for virtio to burst out.

In virtio_xmit_pkts, same index range in vring is available for virtio to clean. Virtio will enqueue to be transmitted packets into vring, advance the vq->vq_ring.avail->idx, and then notify the host back end if necessary.

12.2 Features and Limitations of virtio PMD

In this release, the virtio PMD driver provides the basic functionality of packet reception and transmission.

- It supports merge-able buffers per packet when receiving packets and scattered buffer per packet when transmitting packets. The packet size supported is from 64 to 1518.
- It supports multicast packets and promiscuous mode.

- The descriptor number for the RX/TX queue is hard-coded to be 256 by qemu. If given a different descriptor number by the upper application, the virtio PMD generates a warning and fall back to the hard-coded value.
- Features of mac/vlan filter are supported, negotiation with vhost/backend are needed to support them. When backend can't support vlan filter, virtio app on guest should disable vlan filter to make sure the virtio port is configured correctly. E.g. specify '–disable-hw-vlan' in testpmd command line.
- RTE_PKTMBUF_HEADROOM should be defined larger than sizeof(struct virtio_net_hdr), which is 10 bytes.
- Virtio does not support runtime configuration.
- Virtio supports Link State interrupt.
- Virtio supports software vlan stripping and inserting.
- Virtio supports using port IO to get PCI resource when uio/igb_uio module is not available.

12.3 Prerequisites

The following prerequisites apply:

- In the BIOS, turn VT-x and VT-d on
- Linux kernel with KVM module; vhost module loaded and ioeventfd supported. Qemu standard backend without vhost support isn't tested, and probably isn't supported.

12.4 Virtio with kni vhost Back End

This section demonstrates kni vhost back end example setup for Phy-VM Communication.

Host2VM communication example

1. Load the kni kernel module:

   ```
   insmod rte_kni.ko
   ```

   Other basic DPDK preparations like hugepage enabling, uio port binding are not listed here. Please refer to the *DPDK Getting Started Guide* for detailed instructions.

2. Launch the kni user application:

   ```
   examples/kni/build/app/kni -c 0xf -n 4 -- -p 0x1 -P --config="(0,1,3)"
   ```

   This command generates one network device vEth0 for physical port. If specify more physical ports, the generated network device will be vEth1, vEth2, and so on.

   For each physical port, kni creates two user threads. One thread loops to fetch packets from the physical NIC port into the kni receive queue. The other user thread loops to send packets in the kni transmit queue.

   For each physical port, kni also creates a kernel thread that retrieves packets from the kni receive queue, place them onto kni's raw socket's queue and wake up the vhost kernel thread to exchange packets with the virtio virt queue.

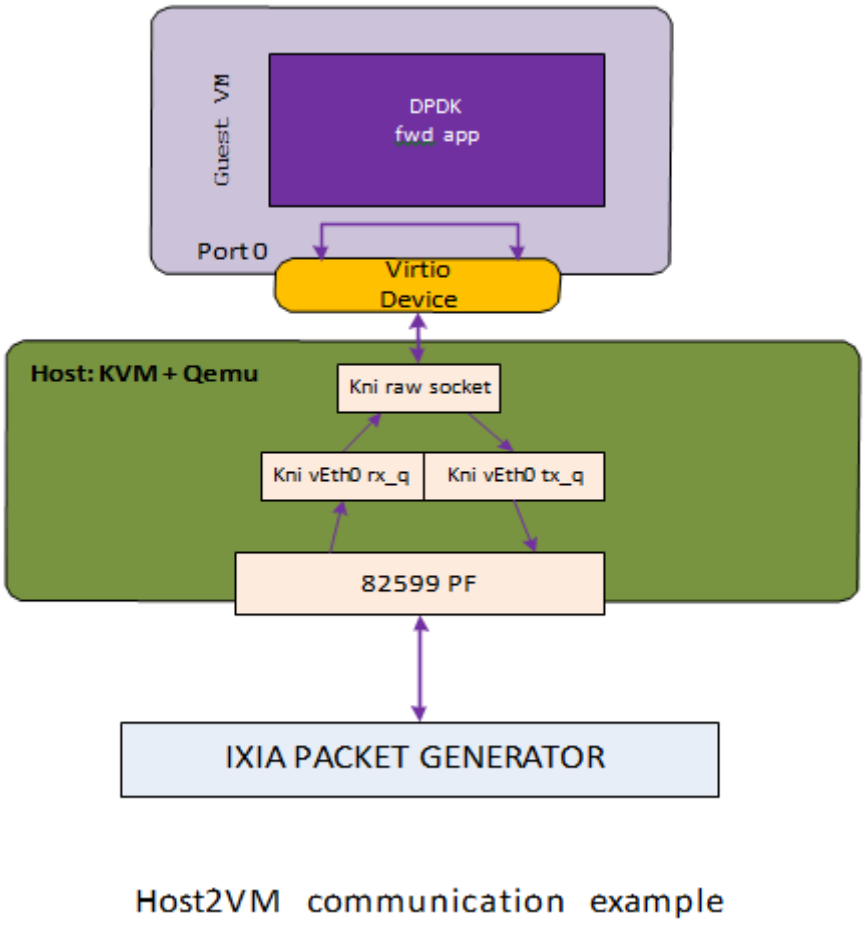

Fig. 12.1: Host2VM Communication Example Using kni vhost Back End

For more details about kni, please refer to Chapter 24 “Kernel NIC Interface”.

3. Enable the kni raw socket functionality for the specified physical NIC port, get the generated file descriptor and set it in the qemu command line parameter. Always remember to set ioeventfd_on and vhost_on.

   Example:

   ```
   echo 1 > /sys/class/net/vEth0/sock_en
   fd=`cat /sys/class/net/vEth0/sock_fd`
   exec qemu-system-x86_64 -enable-kvm -cpu host \
   -m 2048 -smp 4 -name dpdk-test1-vm1 \
   -drive file=/data/DPDKVMS/dpdk-vm.img \
   -netdev tap, fd=$fd,id=mynet_kni, script=no,vhost=on \
   -device virtio-net-pci,netdev=mynet_kni,bus=pci.0,addr=0x3,ioeventfd=on \
   -vnc:1 -daemonize
   ```

   In the above example, virtio port 0 in the guest VM will be associated with vEth0, which in turns corresponds to a physical port, which means received packets come from vEth0, and transmitted packets is sent to vEth0.

4. In the guest, bind the virtio device to the uio_pci_generic kernel module and start the forwarding application. When the virtio port in guest bursts rx, it is getting packets from the raw socket’s receive queue. When the virtio port bursts tx, it is sending packet to the tx_q.

   ```
   modprobe uio
   echo 512 > /sys/devices/system/node/node0/hugepages/hugepages-2048kB/nr_hugepages
   modprobe uio_pci_generic
   python tools/dpdk_nic_bind.py -b uio_pci_generic 00:03.0
   ```

   We use testpmd as the forwarding application in this example.

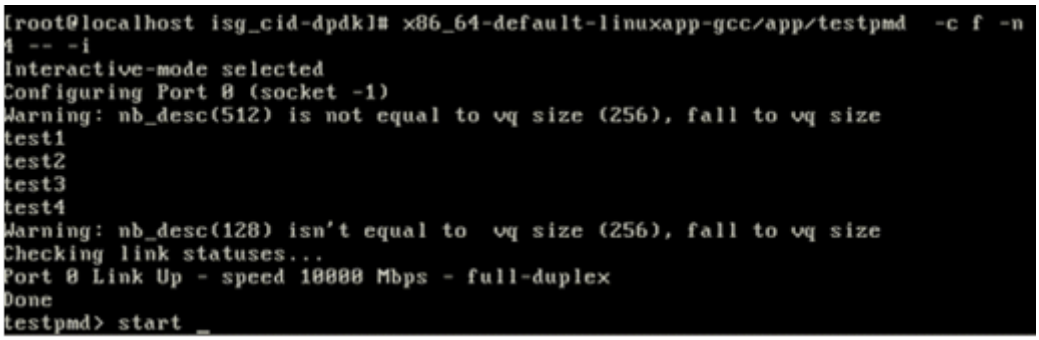

Fig. 12.2: Running testpmd

5. Use IXIA packet generator to inject a packet stream into the KNI physical port.

   The packet reception and transmission flow path is:

   IXIA packet generator->82599 PF->KNI rx queue->KNI raw socket queue->Guest VM virtio port 0 rx burst->Guest VM virtio port 0 tx burst-> KNI tx queue->82599 PF-> IXIA packet generator

12.5 Virtio with qemu virtio Back End

```
qemu-system-x86_64 -enable-kvm -cpu host -m 2048 -smp 2 -mem-path /dev/
hugepages -mem-prealloc
```

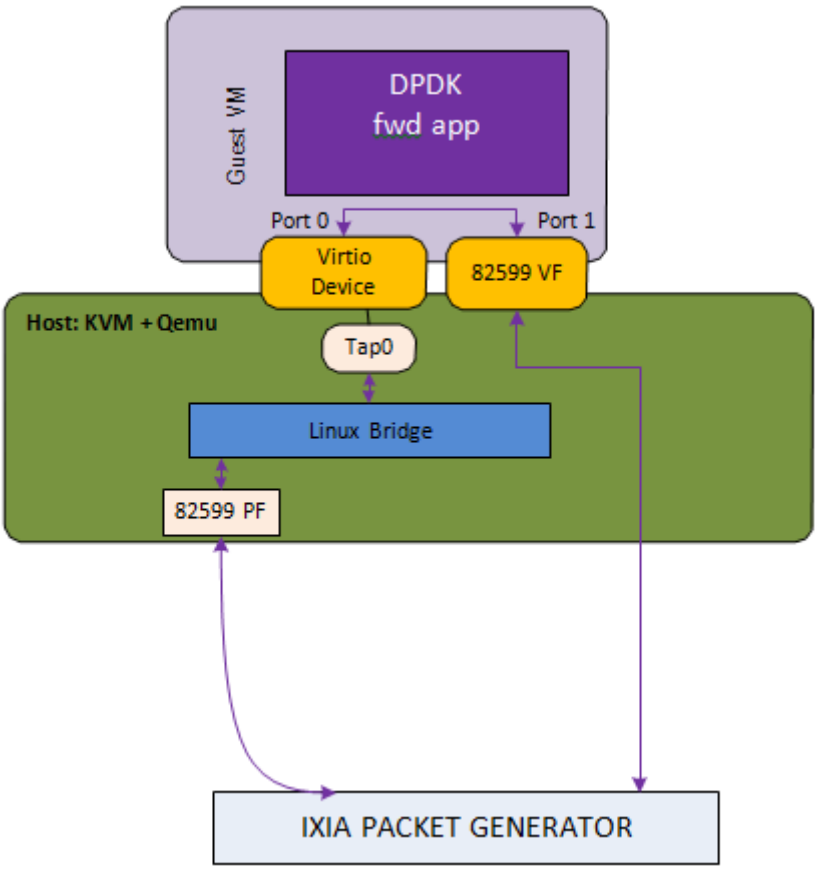

Fig. 12.3: Host2VM Communication Example Using qemu vhost Back End

```
-drive file=/data/DPDKVMS/dpdk-vm1
-netdev tap,id=vm1_p1,ifname=tap0,script=no,vhost=on
-device virtio-net-pci,netdev=vm1_p1,bus=pci.0,addr=0x3,ioeventfd=on
-device pci-assign,host=04:10.1 \
```

In this example, the packet reception flow path is:

> IXIA packet generator->82599 PF->Linux Bridge->TAP0's socket queue-> Guest VM virtio port 0 rx burst-> Guest VM 82599 VF port1 tx burst-> IXIA packet generator

The packet transmission flow is:

> IXIA packet generator-> Guest VM 82599 VF port1 rx burst-> Guest VM virtio port 0 tx burst-> tap -> Linux Bridge->82599 PF-> IXIA packet generator

CHAPTER

THIRTEEN

POLL MODE DRIVER FOR PARAVIRTUAL VMXNET3 NIC

The VMXNET3 adapter is the next generation of a paravirtualized NIC, introduced by VMware* ESXi. It is designed for performance and is not related to VMXNET or VMXENET2. It offers all the features available in VMXNET2, and adds several new features such as, multi-queue support (also known as Receive Side Scaling, RSS), IPv6 offloads, and MSI/MSI-X interrupt delivery. Because operating system vendors do not provide built-in drivers for this card, VMware Tools must be installed to have a driver for the VMXNET3 network adapter available. One can use the same device in a DPDK application with VMXNET3 PMD introduced in DPDK API.

Currently, the driver provides basic support for using the device in a DPDK application running on a guest OS. Optimization is needed on the backend, that is, the VMware* ESXi vmkernel switch, to achieve optimal performance end-to-end.

In this chapter, two setups with the use of the VMXNET3 PMD are demonstrated:

1. Vmxnet3 with a native NIC connected to a vSwitch
2. Vmxnet3 chaining VMs connected to a vSwitch

13.1 VMXNET3 Implementation in the DPDK

For details on the VMXNET3 device, refer to the VMXNET3 driver's vmxnet3 directory and support manual from VMware*.

For performance details, refer to the following link from VMware:

http://www.vmware.com/pdf/vsp_4_vmxnet3_perf.pdf

As a PMD, the VMXNET3 driver provides the packet reception and transmission callbacks, vmxnet3_recv_pkts and vmxnet3_xmit_pkts. It does not support scattered packet reception as part of vmxnet3_recv_pkts and vmxnet3_xmit_pkts. Also, it does not support scattered packet reception as part of the device operations supported.

The VMXNET3 PMD handles all the packet buffer memory allocation and resides in guest address space and it is solely responsible to free that memory when not needed. The packet buffers and features to be supported are made available to hypervisor via VMXNET3 PCI configuration space BARs. During RX/TX, the packet buffers are exchanged by their GPAs, and the hypervisor loads the buffers with packets in the RX case and sends packets to vSwitch in the TX case.

The VMXNET3 PMD is compiled with vmxnet3 device headers. The interface is similar to that of the other PMDs available in the DPDK API. The driver pre-allocates the packet buffers and loads the command ring descriptors in advance. The hypervisor fills those packet buffers on

packet arrival and write completion ring descriptors, which are eventually pulled by the PMD. After reception, the DPDK application frees the descriptors and loads new packet buffers for the coming packets. The interrupts are disabled and there is no notification required. This keeps performance up on the RX side, even though the device provides a notification feature.

In the transmit routine, the DPDK application fills packet buffer pointers in the descriptors of the command ring and notifies the hypervisor. In response the hypervisor takes packets and passes them to the vSwitch. It writes into the completion descriptors ring. The rings are read by the PMD in the next transmit routine call and the buffers and descriptors are freed from memory.

13.2 Features and Limitations of VMXNET3 PMD

In release 1.6.0, the VMXNET3 PMD provides the basic functionality of packet reception and transmission. There are several options available for filtering packets at VMXNET3 device level including:

1. MAC Address based filtering:
    - Unicast, Broadcast, All Multicast modes - SUPPORTED BY DEFAULT
    - Multicast with Multicast Filter table - NOT SUPPORTED
    - Promiscuous mode - SUPPORTED
    - RSS based load balancing between queues - SUPPORTED
2. VLAN filtering:
    - VLAN tag based filtering without load balancing - SUPPORTED

Note:

- Release 1.6.0 does not support separate headers and body receive cmd_ring and hence, multiple segment buffers are not supported. Only cmd_ring_0 is used for packet buffers, one for each descriptor.
- Receive and transmit of scattered packets is not supported.
- Multicast with Multicast Filter table is not supported.

13.3 Prerequisites

The following prerequisites apply:

- Before starting a VM, a VMXNET3 interface to a VM through VMware vSphere Client must be assigned. This is shown in the figure below.

Note: Depending on the Virtual Machine type, the VMware vSphere Client shows Ethernet adaptors while adding an Ethernet device. Ensure that the VM type used offers a VMXNET3 device. Refer to the VMware documentation for a listed of VMs.

Note: Follow the *DPDK Getting Started Guide* to setup the basic DPDK environment.

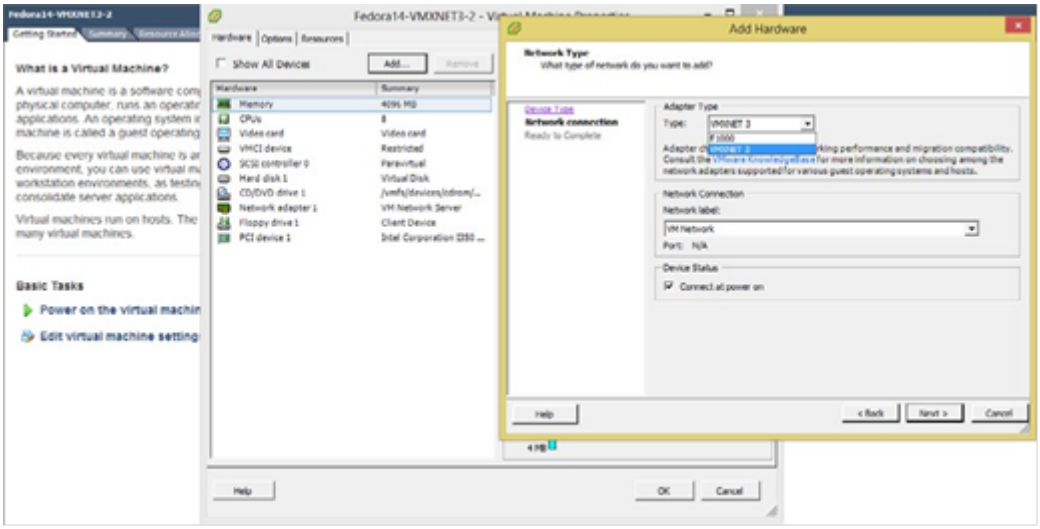

Fig. 13.1: Assigning a VMXNET3 interface to a VM using VMware vSphere Client

Note: Follow the *DPDK Sample Application's User Guide*, L2 Forwarding/L3 Forwarding and TestPMD for instructions on how to run a DPDK application using an assigned VMXNET3 device.

13.4 VMXNET3 with a Native NIC Connected to a vSwitch

This section describes an example setup for Phy-vSwitch-VM-Phy communication.

Note: Other instructions on preparing to use DPDK such as, hugepage enabling, uio port binding are not listed here. Please refer to *DPDK Getting Started Guide and DPDK Sample Application's User Guide* for detailed instructions.

The packet reception and transmission flow path is:

```
Packet generator -> 82576
                 -> VMware ESXi vSwitch
                 -> VMXNET3 device
                 -> Guest VM VMXNET3 port 0 rx burst
                 -> Guest VM 82599 VF port 0 tx burst
                 -> 82599 VF
                 -> Packet generator
```

13.5 VMXNET3 Chaining VMs Connected to a vSwitch

The following figure shows an example VM-to-VM communication over a Phy-VM-vSwitch-VM-Phy communication channel.

Note: When using the L2 Forwarding or L3 Forwarding applications, a destination MAC address needs to be written in packets to hit the other VM's VMXNET3 interface.

In this example, the packet flow path is:

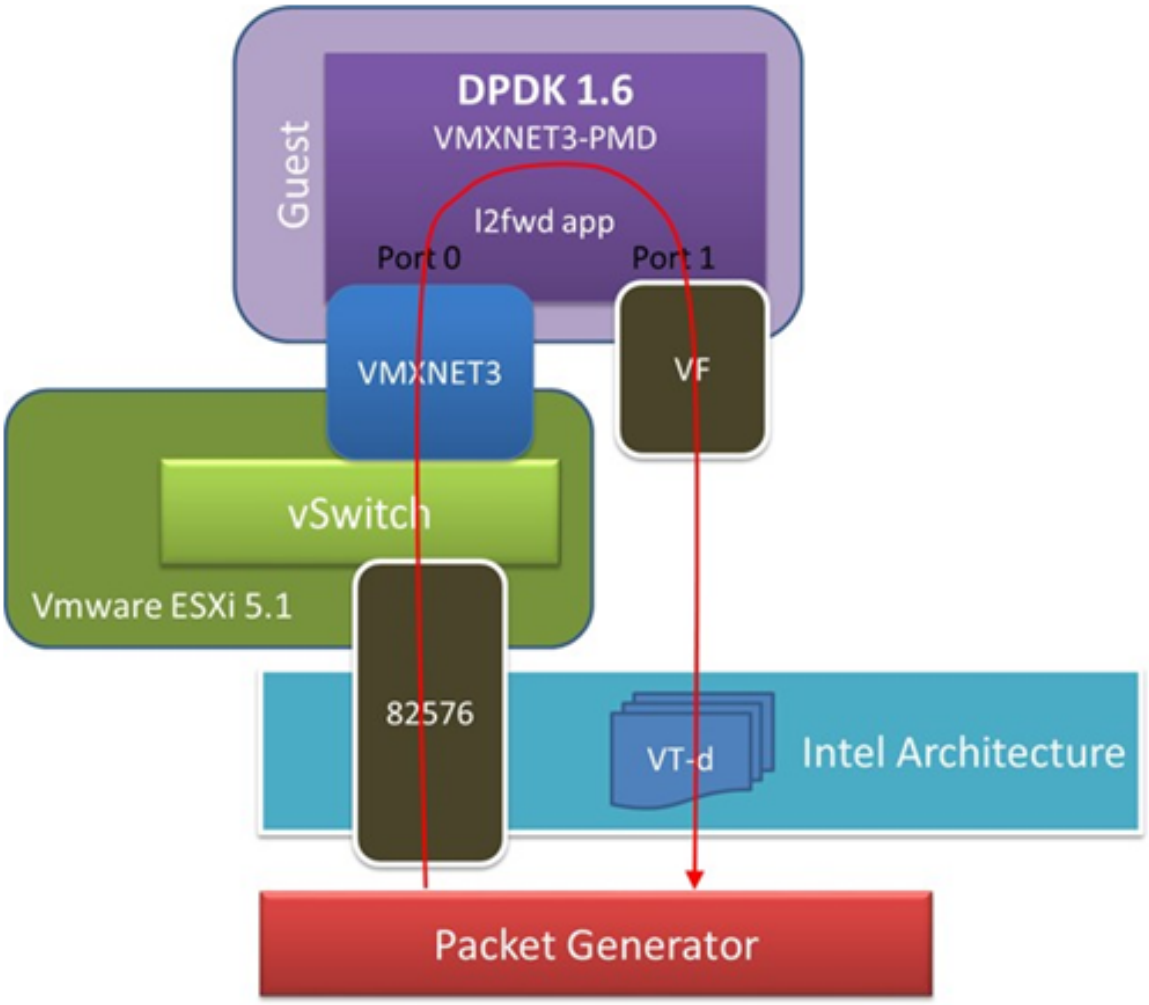

Fig. 13.2: VMXNET3 with a Native NIC Connected to a vSwitch

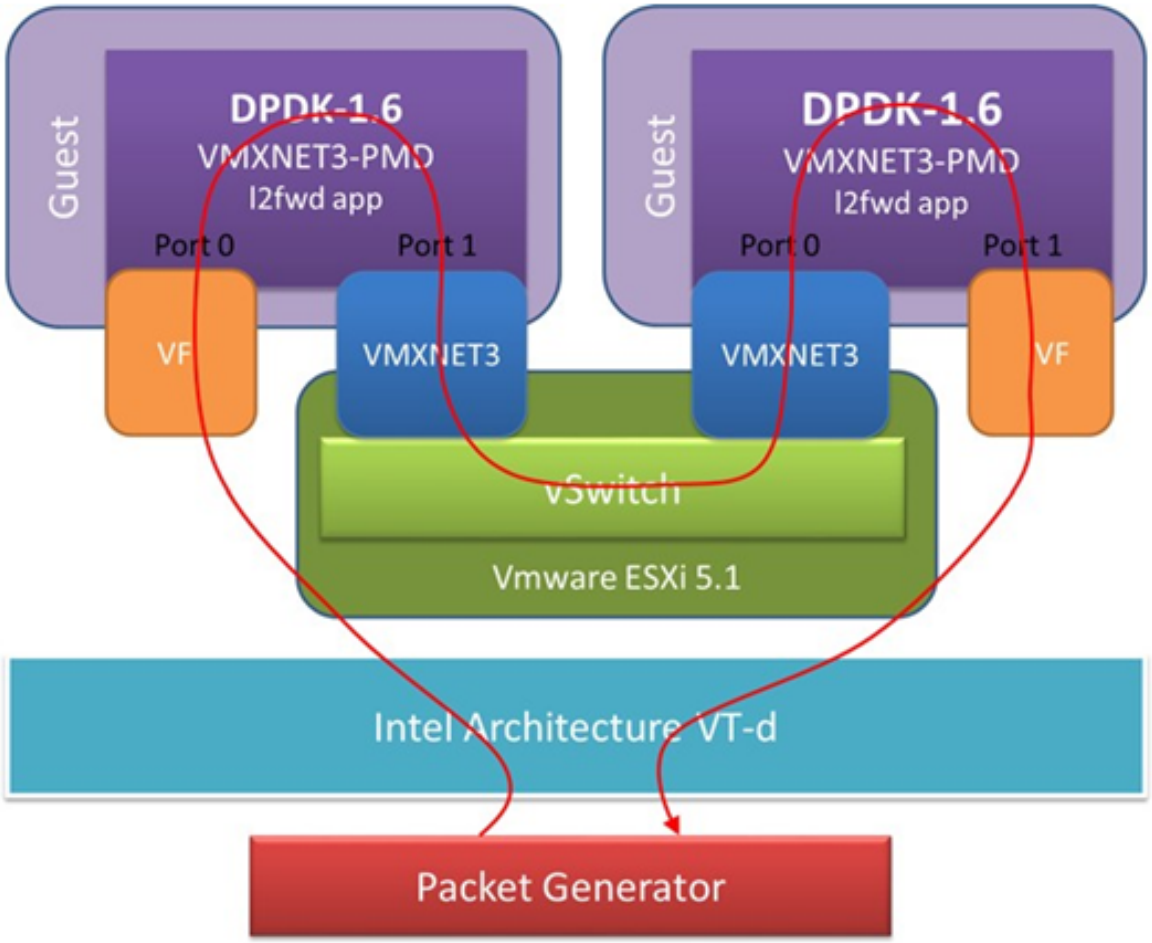

Fig. 13.3: VMXNET3 Chaining VMs Connected to a vSwitch

```
Packet generator -> 82599 VF
                 -> Guest VM 82599 port 0 rx burst
                 -> Guest VM VMXNET3 port 1 tx burst
                 -> VMXNET3 device
                 -> VMware ESXi vSwitch
                 -> VMXNET3 device
                 -> Guest VM VMXNET3 port 0 rx burst
                 -> Guest VM 82599 VF port 1 tx burst
                 -> 82599 VF
                 -> Packet generator
```

CHAPTER

FOURTEEN

LIBPCAP AND RING BASED POLL MODE DRIVERS

In addition to Poll Mode Drivers (PMDs) for physical and virtual hardware, the DPDK also includes two pure-software PMDs. These two drivers are:

- A libpcap -based PMD (librte_pmd_pcap) that reads and writes packets using libpcap, - both from files on disk, as well as from physical NIC devices using standard Linux kernel drivers.
- A ring-based PMD (librte_pmd_ring) that allows a set of software FIFOs (that is, rte_ring) to be accessed using the PMD APIs, as though they were physical NICs.

Note: The libpcap -based PMD is disabled by default in the build configuration files, owing to an external dependency on the libpcap development files which must be installed on the board. Once the libpcap development files are installed, the library can be enabled by setting CONFIG_RTE_LIBRTE_PMD_PCAP=y and recompiling the Intel® DPDK.

14.1 Using the Drivers from the EAL Command Line

For ease of use, the DPDK EAL also has been extended to allow pseudo-Ethernet devices, using one or more of these drivers, to be created at application startup time during EAL initialization.

To do so, the –vdev= parameter must be passed to the EAL. This takes take options to allow ring and pcap-based Ethernet to be allocated and used transparently by the application. This can be used, for example, for testing on a virtual machine where there are no Ethernet ports.

14.1.1 Libpcap-based PMD

Pcap-based devices can be created using the virtual device –vdev option. The device name must start with the eth_pcap prefix followed by numbers or letters. The name is unique for each device. Each device can have multiple stream options and multiple devices can be used. Multiple device definitions can be arranged using multiple –vdev. Device name and stream options must be separated by commas as shown below:

```
$RTE_TARGET/app/testpmd -c f -n 4 --vdev  'eth_pcap0,stream_opt0=..,stream_opt1=..' --vdev='eth
```

Device Streams

Multiple ways of stream definitions can be assessed and combined as long as the following two rules are respected:

- A device is provided with two different streams - reception and transmission.
- A device is provided with one network interface name used for reading and writing packets.

The different stream types are:

- rx_pcap: Defines a reception stream based on a pcap file. The driver reads each packet within the given pcap file as if it was receiving it from the wire. The value is a path to a valid pcap file.

  rx_pcap=/path/to/file.pcap

- tx_pcap: Defines a transmission stream based on a pcap file. The driver writes each received packet to the given pcap file. The value is a path to a pcap file. The file is overwritten if it already exists and it is created if it does not.

  tx_pcap=/path/to/file.pcap

- rx_iface: Defines a reception stream based on a network interface name. The driver reads packets coming from the given interface using the Linux kernel driver for that interface. The value is an interface name.

  rx_iface=eth0

- tx_iface: Defines a transmission stream based on a network interface name. The driver sends packets to the given interface using the Linux kernel driver for that interface. The value is an interface name.

  tx_iface=eth0

- iface: Defines a device mapping a network interface. The driver both reads and writes packets from and to the given interface. The value is an interface name.

  iface=eth0

Examples of Usage

Read packets from one pcap file and write them to another:

```
$RTE_TARGET/app/testpmd -c '0xf' -n 4 --vdev 'eth_pcap0,rx_pcap=/path/to/ file_rx.pcap,tx_pcap=
```

Read packets from a network interface and write them to a pcap file:

```
$RTE_TARGET/app/testpmd -c '0xf' -n 4 --vdev 'eth_pcap0,rx_iface=eth0,tx_pcap=/path/to/file_tx.
```

Read packets from a pcap file and write them to a network interface:

```
$RTE_TARGET/app/testpmd -c '0xf' -n 4 --vdev 'eth_pcap0,rx_pcap=/path/to/ file_rx.pcap,tx_iface
```

Forward packets through two network interfaces:

```
$RTE_TARGET/app/testpmd -c '0xf' -n 4 --vdev 'eth_pcap0,iface=eth0' --vdev='eth_pcap1;iface=eth
```

Using libpcap-based PMD with the testpmd Application

One of the first things that testpmd does before starting to forward packets is to flush the RX streams by reading the first 512 packets on every RX stream and discarding them. When using a libpcap-based PMD this behavior can be turned off using the following command line option:

```
--no-flush-rx
```

It is also available in the runtime command line:

```
set flush_rx on/off
```

It is useful for the case where the rx_pcap is being used and no packets are meant to be discarded. Otherwise, the first 512 packets from the input pcap file will be discarded by the RX flushing operation.

```
$RTE_TARGET/app/testpmd -c '0xf' -n 4 --vdev 'eth_pcap0,rx_pcap=/path/to/ file_rx.pcap,tx_pcap=
```

14.1.2 Rings-based PMD

To run a DPDK application on a machine without any Ethernet devices, a pair of ring-based rte_ethdevs can be used as below. The device names passed to the –vdev option must start with eth_ring and take no additional parameters. Multiple devices may be specified, separated by commas.

```
./testpmd -c E -n 4 --vdev=eth_ring0 --vdev=eth_ring1 -- -i
EAL: Detected lcore 1 as core 1 on socket 0
...

Interactive-mode selected
Configuring Port 0 (socket 0)
Configuring Port 1 (socket 0)
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Port 1 Link Up - speed 10000 Mbps - full-duplex
Done

testpmd> start tx_first
io packet forwarding - CRC stripping disabled - packets/burst=16
nb forwarding cores=1 - nb forwarding ports=2
RX queues=1 - RX desc=128 - RX free threshold=0
RX threshold registers: pthresh=8 hthresh=8 wthresh=4
TX queues=1 - TX desc=512 - TX free threshold=0
TX threshold registers: pthresh=36 hthresh=0 wthresh=0
TX RS bit threshold=0 - TXQ flags=0x0

testpmd> stop
Telling cores to stop...
Waiting for lcores to finish...
---------------------- Forward statistics for port 0  ----------------------
RX-packets: 231192368      RX-dropped: 0             RX-total: 231192368
TX-packets: 231192384      TX-dropped: 0             TX-total: 231192384
----------------------------------------------------------------------------

---------------------- Forward statistics for port 1  ----------------------
RX-packets: 231192368      RX-dropped: 0             RX-total: 231192368
TX-packets: 231192384      TX-dropped: 0             TX-total: 231192384
----------------------------------------------------------------------------

++++++++++++++++ Accumulated forward statistics for allports++++++++++
RX-packets: 462384736  RX-dropped: 0 RX-total: 462384736
```

```
TX-packets: 462384768  TX-dropped: 0 TX-total: 462384768
++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Done.
```

14.1.3 Using the Poll Mode Driver from an Application

Both drivers can provide similar APIs to allow the user to create a PMD, that is, rte_ethdev structure, instances at run-time in the end-application, for example, using rte_eth_from_rings() or rte_eth_from_pcaps() APIs. For the rings-based PMD, this functionality could be used, for example, to allow data exchange between cores using rings to be done in exactly the same way as sending or receiving packets from an Ethernet device. For the libpcap-based PMD, it allows an application to open one or more pcap files and use these as a source of packet input to the application.

Usage Examples

To create two pseudo-Ethernet ports where all traffic sent to a port is looped back for reception on the same port (error handling omitted for clarity):

```
#define RING_SIZE 256
#define NUM_RINGS 2
#define SOCKET0 0

struct rte_ring *ring[NUM_RINGS];
int port0, port1;

ring[0] = rte_ring_create("R0", RING_SIZE, SOCKET0, RING_F_SP_ENQ|RING_F_SC_DEQ);
ring[1] = rte_ring_create("R1", RING_SIZE, SOCKET0, RING_F_SP_ENQ|RING_F_SC_DEQ);

/* create two ethdev's */

port0 = rte_eth_from_rings("eth_ring0", ring, NUM_RINGS, ring, NUM_RINGS, SOCKET0);
port1 = rte_eth_from_rings("eth_ring1", ring, NUM_RINGS, ring, NUM_RINGS, SOCKET0);
```

To create two pseudo-Ethernet ports where the traffic is switched between them, that is, traffic sent to port 0 is read back from port 1 and vice-versa, the final two lines could be changed as below:

```
port0 = rte_eth_from_rings("eth_ring0", &ring[0], 1, &ring[1], 1, SOCKET0);
port1 = rte_eth_from_rings("eth_ring1", &ring[1], 1, &ring[0], 1, SOCKET0);
```

This type of configuration could be useful in a pipeline model, for example, where one may want to have inter-core communication using pseudo Ethernet devices rather than raw rings, for reasons of API consistency.

Enqueuing and dequeuing items from an rte_ring using the rings-based PMD may be slower than using the native rings API. This is because DPDK Ethernet drivers make use of function pointers to call the appropriate enqueue or dequeue functions, while the rte_ring specific functions are direct function calls in the code and are often inlined by the compiler.

> Once an ethdev has been created, for either a ring or a pcap-based PMD, it should be configured and started in the same way as a regular Ethernet device, that is, by calling rte_eth_dev_configure() to set the number of receive and transmit queues, then calling rte_eth_rx_queue_setup() / tx_queue_setup() for each of those queues and finally calling rte_eth_dev_start() to allow transmission and reception of packets to begin.

Figures