# Introducing the Data Plane Development Kit (DPDK) on Lenovo Servers

Introduces the DPDK application development libraries

Explains how DPDK improves network performance

Shows how to implement a simple l2fwd DPDK application

Shows how to implement a DPDK-accelerated Open vSwitch with a virtual machine

Yao Aili

# Abstract

The Data Plane Development Kit (DPDK) is a set of fundamental development libraries for fast data plane packet processing. It includes core components, poll mode drivers, an accelerator, classification components, quality of service (QoS), extensions, and packet frameworks. Together, all of these components make DPDK an ideal development kit for data planes on server platforms such as Lenovo® ThinkSystem™. It is a key feature for high-performance data transmission scenarios like Network Functions Virtualization (NFV) and data center virtualization.

In this paper, we introduce DPDK and its technology and we explain the steps to deploy a DPDK debug environment from the source code on the Lenovo ThinkSystem ST550 server. Finally, we use an Open vSwitch application example to show how the DPDK works.

This paper is intended for software developers, systems architects, and Linux systems engineers. It is expected that readers will have a good knowledge of Linux and networking.

At Lenovo Press, we bring together experts to produce technical publications around topics of importance to you, providing information and best practices for using Lenovo products and solutions to solve IT challenges.

See a list of our most recent publications at the Lenovo Press web site:

http://lenovopress.com

**Do you have the latest version?** We update our papers from time to time, so check whether you have the latest version of this document by clicking the **Check for Updates** button on the front page of the PDF. Pressing this button will take you to a web page that will tell you if you are reading the latest version of the document and give you a link to the latest if needed. While you're there, you can also sign up to get notified via email whenever we make an update.

# Contents

## Introduction

The open-source Data Plane Development Kit (DPDK) is a set of data plane libraries and network interface controller drivers for fast packet processing. The DPDK provides a programming framework and enables faster development of high speed data packet networking applications.

There are three components for networking devices, the data plane, the control plane and the management plane. These three planes enable the networking device to move a packet from interface A to another interface B.

- Management plane: This plane is used for human interaction and provides a command shell to configuration the networking device.
- Control plane: This plane will determine where the packet should go. Usually, it includes multiple protocols, such as RIP, OSPF and EIGRP.
- Data plane: This plane is used for fast packet forwarding. Usually, it includes a small cache for packet forwarding instructions. If there is a cache hit, the packet will be forwarded, or it will instruct the control plane to decide what to do.

### Background

Modern servers receive an ever-growing number of data packets from the Internet. Concurrency and response speed demands are increasing as the payload of server net path increases. Faster network adapters have been developed to support these requirements, and connections such as 40Gb, 50Gb, and even 100Gb are readily available.

However, the traditional Linux packet processing method is still comparatively slow. To receive packets, the kernel must handle many interrupts, process the packet in the kernel net stack and, if needed, inform the related user space applications to fetch the packet. Finally, the application copies the packet and processes it.

For high-speed interfaces and high workloads, the kernel has to handle millions of interrupts due to the high packet volume. This can overload the system and reduce the system response time. In addition, the Linux network stack must satisfy various application and user needs, which leads to low network efficiency and an inability to achieve fast packet processing. If an application wants to process the packet, it has to copy it from the kernel space, which takes many CPU cycles and results inhigh latenciess due to the context switch between the kernel space and user space.

At the same time, x86 CPUs have become more powerful by integrating more cores into a single socket. If the system wants to take advantage of the power of multicore CPUs, the kernel and system must be optimized for such functions. The DPDK was invented by Intel for the purpose of processing network packets, and has now become an open-source project.

ThinkSystem servers are fully compliant with the DPDK and its related technology, and it's an ideal platform for DPDK-related applications

### Maximizing network performance

The DPDK utilizes a set of techniques to maximize network performance:

- Poll mode driver (PMD)

  Compared with the IRQ mode driver, the poll mode driver (PMD) uses a dedicated process (or thread) to poll the queue of the NIC. When packets are received, the PMD will receive them continuously. If there are no packets, the PMD will just poll in an endless loop. Usually, we will dedicate more than one core to the poll mode thread and isolate the dedicated CPU from scheduling. This will avoid the context switch and reduce the cache miss rate.

- User mode driver

  Compared with the kernel mode driver, the user mode driver eliminates the unnecessary memory copy between the kernel and user space, and avoids context switches (because it doesn't have to call the system). Without the constraints of the kernel data structure, the user mode driver can utilize optimized mbuf definitions to achieve high performance. The application can use this driver with more freedom and implement specific optimizations.

- Affinity and CPU binding

  The app and driver in user mode will still be scheduled by the kernel. This will lead to a high cache miss rate and low performance. For some key workloads and situations, this is unacceptable. The DPDK uses the Linux kernel's thread affinity feature to bind the thread to a specific core to avoid this situation. We recommend using isocpu kernel parameters to remove specific cores from the scheduling policy.

  Thus, the app and poll mode driver will be bound to a specific CPU, which will never be scheduled for other tasks.

- Lower memory access costs

  The DPDK provides methods that help reduce the impacts on performance related to memory access:

  - For the NUMA system, DPDK uses a dedicated malloc function to provide the local memory to a special node.
  - For frequently used objects, such as mbuf structs, DPDK adds special pads to align with different channels and ranks in the memory, so that access is distributed evenly among all channels.
  - Uses the huge page technique to reduce the TLB miss rate. This will dramatically reduce the memory access time and ultimately enhance performance.
  - Uses the DDIO technique to reduce the cache write back time and memory access time.

- Lockless technique

  For multicore systems, especially for data path packet processing, competition is an actual problem which will lead to high CPU overhead and low efficiency. DPDK tries to avoid this competition situation using mallocing structs and objects dedicated to the local core.

  In addition, to use lockless code to reduce lock costs, DPDK will use the Intel atomic cmpxchg instruction to replace spinlock and use a multi/single producer and multi/single consumer ring buffer to achieve high performance packet queues and synchronization between different cores.

- Other software optimizations

  In the DPDK source code, you can see many small optimizations that result in significant performance improvements. These include:

  - Using SIMD instructions to accelerate memcpy.
  - Using the cache prefetch technique to reduce the cache miss rate.
  - Using line aligned data structures.

## DPDK framework

The DPDK includes core components, PMD drivers, accelerators, classification components, QOS, extensions, and packet frameworks. Figure 1 shows the components:

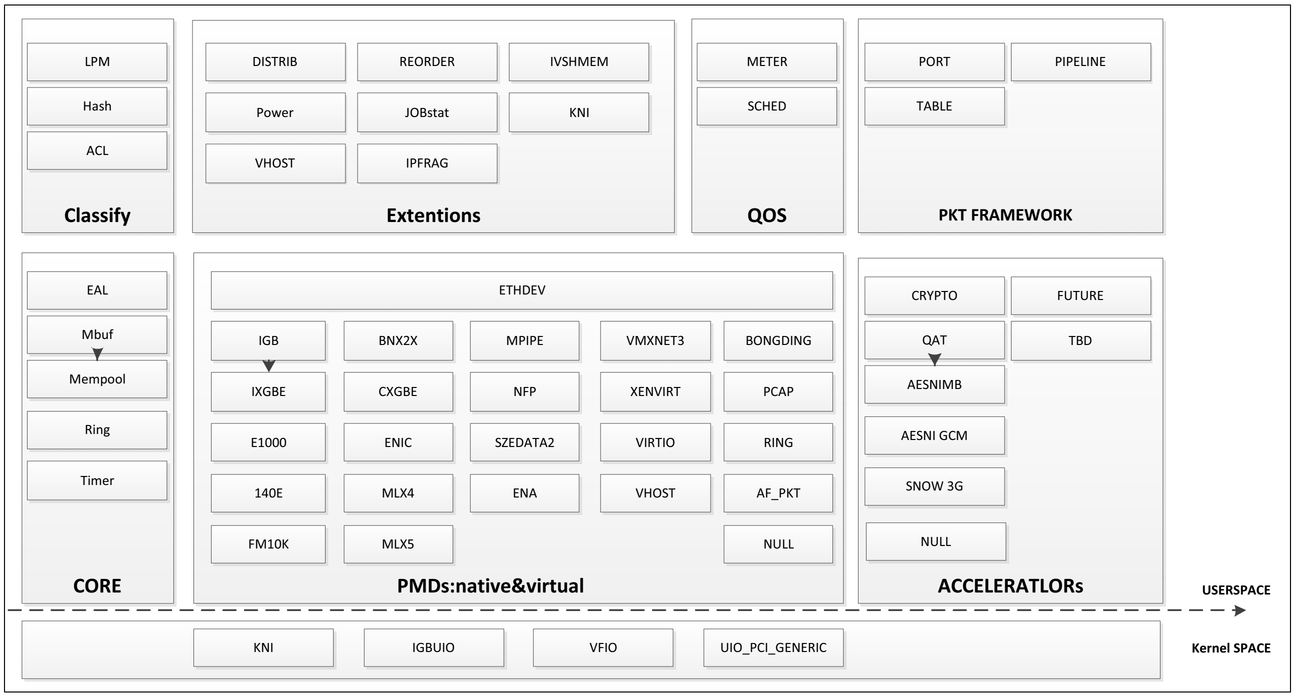

*Figure 1 DPDK framework*

The components in the above figure are as follows:

- Core: Basic functions, including EAL (environment abstraction layer) and four core components: Ring Manager (librte_ring), Memory Pool Manager (librte_mempool), Network Packet Buffer Management (librte_mbuf), and Timer Manager (librte_timer).
- PMD: Poll mode drivers for all supported NICs.
- Accelerators: These refer to hardware accelerators, such as QAT and CRYPTO.
- Classification components: These components facilitate searches for incoming packets in the flow table.
- Extensions: Various extensions for different use scenarios.
- QoS: Quality of service.
- Pkt framework: This is for packet processing and forwarding.

## Usage scenarios

In this section, we describe two scenarios that show the use of DPDK with network data processing.

## DPDK for data center virtualization

DPDK is a server software development kit, so its typical usage scenario is for data center virtualization. In Figure 2 on page 6, one server has two physical NICs (10 GbE or 40 GbE) with high-speed workload capabilities. These NICs are driven by user space PMD and form a user space DPDK interface. The virtual switches attached to these physical DPDK ports and many other virtio ports (of a virtual machine) are emulated by the vhost based on DPDK.

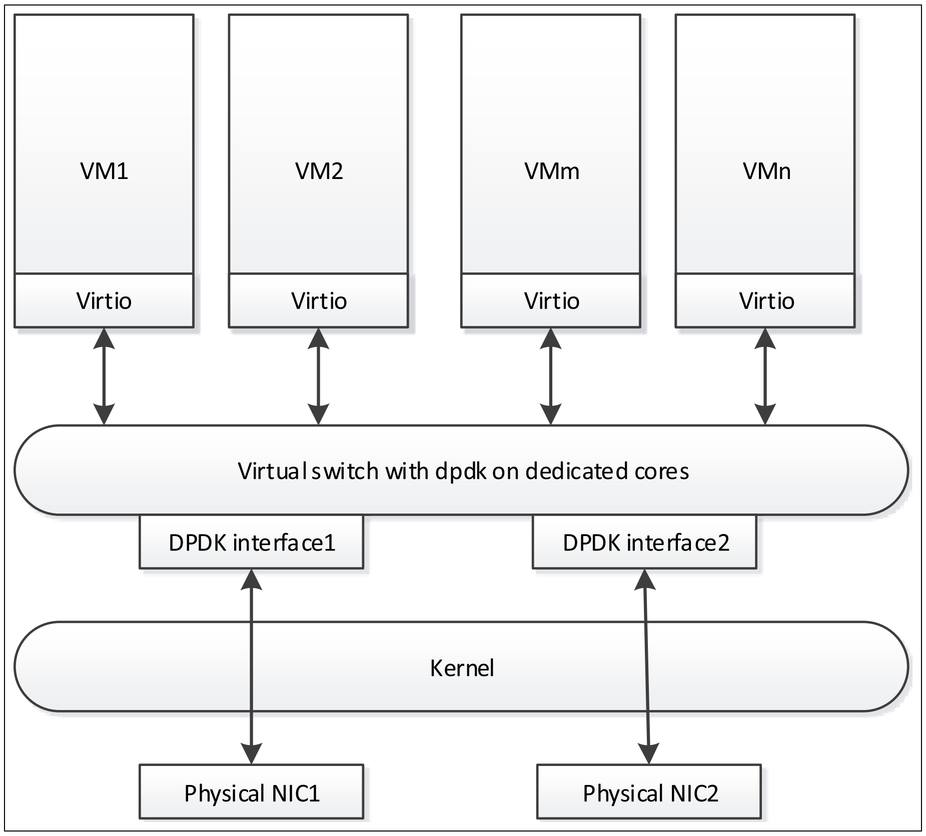

*Figure 2 DPDK Virtualization*

The workload comes in from the physical port and will be distributed by the virtual switch to different VMs based on the configured flow rules.

With DPDK acceleration, the critical app in each VM will meet the performance and latency targets (with up to a ten-fold performance enhancement). Meanwhile, using the DPDK accelerated virtualization technique will reduce the hardware costs associated with physical switches.

## NFV usage scenario

DPDK is a low-level component and is usually treated as the entry point for advanced data packet processing. A lot of services (such as Virtual Network Function, VNF) will get packets from DPDK, and send packets to DPDK. The DPDK provides the packet to the virtual switch at high speed. Then, the VNFs in different VMs will utilize the DPDK user poll mode driver to receive and transmit packets between different VNFs and virtual switches.

Figure 3 on page 7 shows how DPDK works in NFV.

- DPDK works in the NFVI layer to accelerate the physical NIC.

- DPDK work in the VNF layer to accelerate the packet forwarding between VNFs.

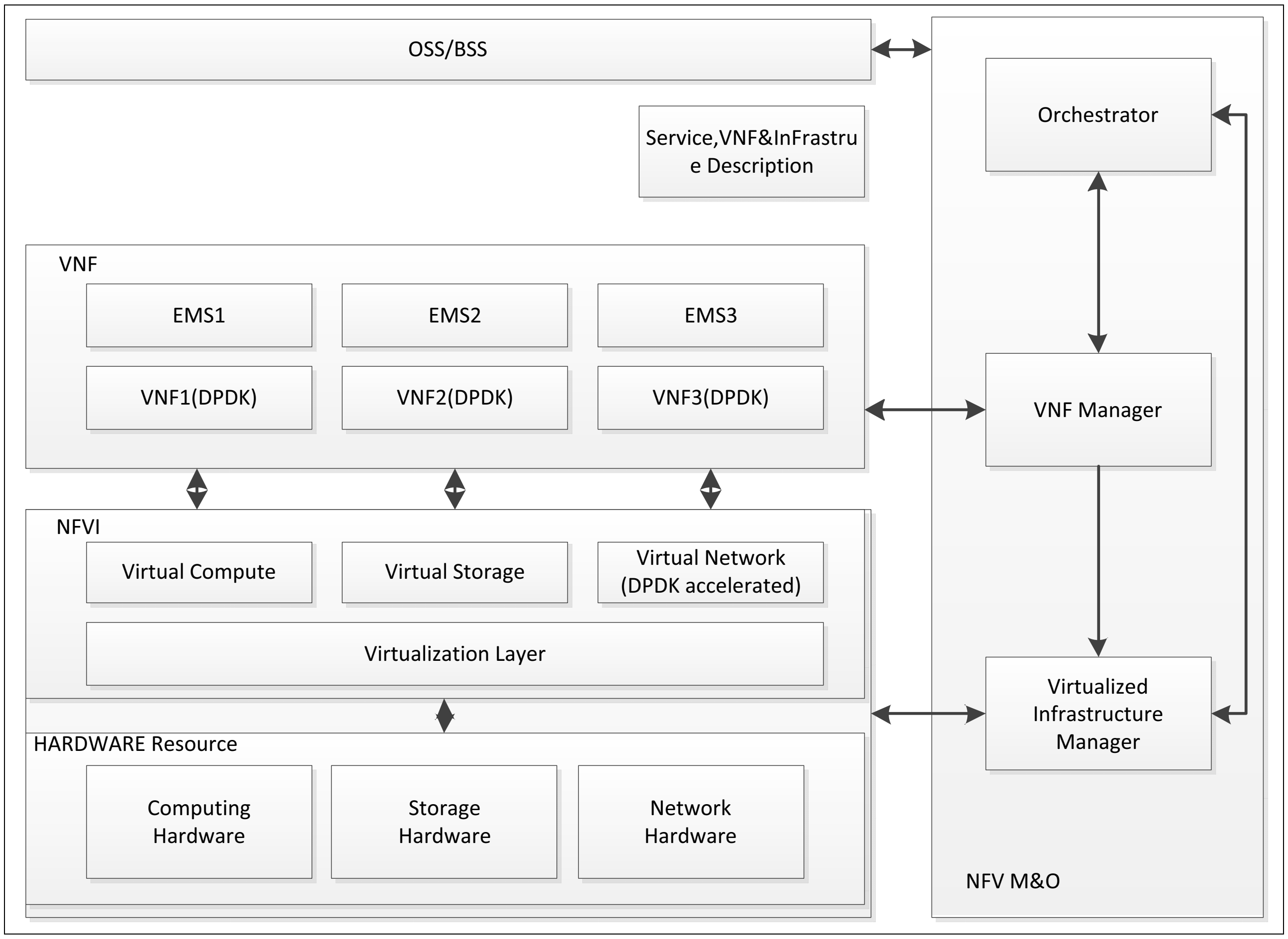

*Figure 3 NFV Architectural Framework*

## Example applications

In the remainder of this paper, we present two DPDK applications as examples that show how to deploy the DPDK development environment: l2fwd, a simple DPDK application, and a DPDK-accelerated Open vSwitch.

In this paper, we are running these samples on a Lenovo ThinkSystem ST550 server with Intel Xeon Scalable Family processors.

As a development environment for software developers, all we need is the source code.

## L2fwd example

The L2 Forwarding (l2fwd) sample application operates in real and virtualized environments and performs L2 forwarding for each packet that is received on an RX_PORT. See Figure 4 on page 8.

The destination port is the port adjacent to the enabled portmask. That is, if the first four ports are enabled (portmask 0xf), ports 1 and 2 forward to each other, and ports 3 and 4 forward to each other. Also, if MAC address updating is enabled, the MAC addresses are affected as follows:

- The source MAC address is replaced by the TX_PORT MAC addressi
- The destination MAC address is replaced by 02:00:00:00:00:TX_PORT_ID

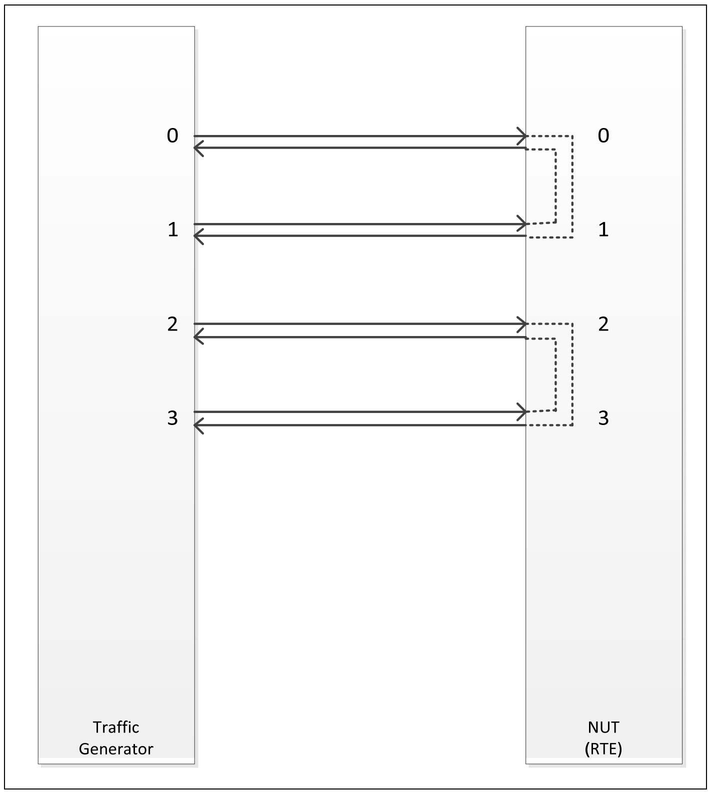

*Figure 4   Data flow of the L2 Forwarding (l2fwd) sample application*

The overall steps are as follows:

1. Download and compile the L2fwd application source
2. Configure huge pages
3. Bind the NIC to the UIO driver
4. Run the L2fwd application

Our lab environment is as follows:

- Lab host server: Lenovo ThinkSystem ST550
- Host OS: SLES 12.2.

## Download and compile the L2fwd application source

The detailed steps to implement the l2fwd application are as follows:

1. Download the DPDK source code to compile and install it. Get the DPDK source code from http://dpdk.org/download. In our environment, we selected the latest stable version, 17.02.01.
2. Extract the source code to ST550 server to directory /home/dpdk.
3. The packages required for compiling are as follows:
   - Gcc (version 4.9 or later is recommended for all platforms)
   - glibc-devel.x86_64
   - kernel-devel.x86_64
   - Python
   - libpcap1 (optional)
4. Compile the DPDK environment. Switch to the DPDK source directory and run the following command:

   ```
   # cd /home/dpdk/dpdk-stable-16.11.1
   # make install T=x86_64-native-linuxapp-gcc DESTDIR=/home/dpdk/Destdir
   ```

   The parameters are as follows:

   - T indicates the target type. The syntax is: `ARCH-MACHINE-EXECENV-TOOLCHAIN`
     - ARCH can be: i686, x86_64, ppc_64
     - MACHINE can be: native, power8
     - EXECENV can be: linuxapp, bsdapp
     - TOOLCHAIN can be: gcc, icc

     The targets to be installed depend on the 32-bit and/or 64-bit packages and compilers installed on the host. Available targets can be found in the DPDK/config directory.
   - DESTDIR

     This will install the binary SDK (bin + modules + libs + headers + mk) in the specified directory. This directory can be used as RTE_SDK by external applications.
5. After DPDK environment compilation and installation, the system will show the following message to indicate success:

   ```
   Installation in /home/dpdk/Destdir/ complete
   ```

   The DPDK related lib has now been built and provided in the directory, and the binary lib has been built in DESTDIR directory:

   /home/dpdk/dpdk-stable-16.11.1/x86_64-native-linuxapp-gcc
6. Compile the application

   To use DPDK, the next step is to develop and run an application with this library. The DPDK package provides various examples that we can use as a basis to develop our app. We will use l2fwd in the DPDK source code as an example of how to compile with DPDK:

   ```
   export RTE_SDK=/home/dpdk/Destdir/share/dpdk/
   export RTE_TARGET=x86_64-native-linuxapp-gcc
   cd /home/dpdk/dpdk-stable-16.11.1/examples/l2fwd
   make
   ```

8.The system will show the following message to indicate success:

```
CC main.o
LD l2fwd
INSTALL-APP l2fwd
INSTALL-MAP l2fwd.map
```

The app is now compiled to directory /home/dpdk/dpdk-stable-16.11.1/examples/l2fwd/build/app.

7. Before running the application, we configure huge pages for the system and bind the NIC to the PMD driver.

   Huge page support is required for the large memory pool allocation used for packet buffers. By using hugepage allocations, performance is increased, as fewer pages are needed. Therefore, there will be fewer Translation Lookaside Buffers (TLBs, high speed translation caches), which reduces the time it takes to translate a virtual page address to a physical page address.

## Configure huge pages

Allocate 2048 2MB huge pages as follows:

1. For persistent allocation of huge pages, write to hugepages.conf file in /etc/sysctl.d:

```
$ echo 'vm.nr_hugepages=2048' > /etc/sysctl.d/hugepages.conf
```

2. For runtime allocation of huge pages, use the sysctl utility:

```
$ sysctl -w vm.nr_hugepages=2048
```

3. Mount the hugepages, if not already mounted by default:

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

4. The mount point can be made permanent across reboots, by adding the following line to the /etc/fstab file:

```
nodev /mnt/huge hugetlbfs defaults 0 0
```

5. For 1GB pages, the page size must be specified as a mount option:

```
nodev /mnt/huge_1GB hugetlbfs pagesize=1GB 0 0
```

## Bind the NIC to the UIO driver

Bind the NIC to the user space I/O (UIO) driver. In our lab, we are using a four-port Intel I350 Gigabit Ethernet Adapter.

About UIO: DPDK uses PMD to drive the NIC. This UIO driver is a user-space driver utilizing the UIO framework of Linux. UIO does not eliminate the need for kernel-space code: a module is required to set up the device and an interrupt handler needs to be registered. Much can be done in user space, but there needs to be an in-kernel interrupt handler that knows how to tell the device to stop demanding attention.

In our case, we need a small kernel mode driver named uio_pci_generic to unbind the NIC from original driver and release it to the PMD of DPDK.

1. Insert the kernel UIO driver to replace the origin kernel driver (igb.ko in this case)

```
modprobe uio_pci_generic
```

Alternatively, you can use the driver provided by the DPDK source code in directory /home/dpdk/dpdk-stable-16.11.1/x86_64-native-linuxapp-gcc/kmod

2. Use the dpdk-devbind.py script in the DPDK source for unbinding the kernel's original driver. First, we use this script to show the port status:

   ```
   /home/dpdk/dpdk-stable-16.11.1/tools/ dpdk-devbind.py -s
   ```

3. Bind the related pci dev to the uio_pci_generic driver:

   ```
   ./dpdk-devbind.py --bind uio_pci_generic 0000:5b:00.0 0000:5b:00.1 0000:5b:00.2
   0000:5b:00.3
   ```

4. Check the port status again:

   ```
   dpdk-devbind.py -s
   ```

The port status output will look similar to Example 1:

*Example 1 Output from the* **`dpdk-devbind.py -s`** *command*

```
Network devices using DPDK-compatible driver
============================================
0000:5b:00.0 'I350 Gigabit Network Connection' drv=uio_pci_generic unused=igb,igb_uio
0000:5b:00.1 'I350 Gigabit Network Connection' drv=uio_pci_generic unused=igb,igb_uio
0000:5b:00.2 'I350 Gigabit Network Connection' drv=uio_pci_generic unused=igb,igb_uio
0000:5b:00.3 'I350 Gigabit Network Connection' drv=uio_pci_generic unused=igb,igb_uio
Network devices using kernel driver
===================================
0000:b0:00.0 'Ethernet Connection X722 for 1GbE' if=eth2 drv=i40e
unused=igb_uio,uio_pci_generic *Active*
0000:b0:00.1 'Ethernet Connection X722 for 1GbE' if=eth5 drv=i40e
unused=igb_uio,uio_pci_generic *Active*
Other network devices
=====================
<none>
Crypto devices using DPDK-compatible driver
===========================================
<none>
Crypto devices using kernel driver
==================================
<none>
Other crypto devices
====================
<none>
```

## Run the l2fwd application

The l2fwd application requires a number of command line options:

```
./build/l2fwd [EAL options] -- -p PORTMASK [-q NQ] --[no-]mac-updating
```

Where

- `[EAL options]:`

  The following is the list of options that can be given to the EAL:

  ```
  ./ build/l2fwd [-c COREMASK | -l CORELIST] [-n NUM] \
        [-b <domain:bus:devid.func>] [--socket-mem=MB,...] [-d LIB.so|DIR] \
        [-m MB] [-r NUM] [-v] [--file-prefix] \
        [--proc-type <primary|secondary|auto>] [-- xen-dom0]
  ```

`-c COREMASK` or `-l CORELIST`: A hexadecimal bit mask of the cores to run on. Note that core numbering can change between platforms and should be determined beforehand. The corelist is a set of core numbers instead of a bitmap core mask.

`-n NUM`: Number of memory channels per processor socket.

`--socket-mem`: Memory to allocate from hugepages on specific sockets.

`-r NUM`: Number of memory ranks.

`--huge-dir`: The directory where hugetlbfs is mounted.

`--file-prefix`: The prefix text used for hugepage filenames.

The `-c` or `-l` and option is mandatory; the others are optional.

In our case:

- `-n 6`, because we are using Intel Xeon Gold 5117 processor with 2.0 GHz core and 6 channels per socket
- `-r 4`, because we ran **`dmidecode -t 17`** to check the ranks of memory to determine the number to use is 4
- `--huge-dir /mnt/huge`

- `p PORTMASK`: A hexadecimal bitmask of the ports to configure
- `-q NQ`: The number of queues (=ports) per lcore (default is 1)
- `–[no-]mac-updating`: Enable or disable MAC addresses updating (enabled by default).

To run the application in linuxapp environment with 4 lcores, 16 ports and 8 RX queues per lcore and MAC address updating enabled, issue the command:

```
./build/l2fwd -l 0-3 -n 6 -r 4 -- -q 8 -p ffff
```

### Result

In our tests we used a Gigabit Ethernet adapter, and therefore easily achieved the needed line-speed packet per port. Typically, even with 40 GbE, you can get line-speed performance for simple forwarding.

## DPDK-accelerated Open vSwitch

The l2fwd example presented in the previous section is not the typical way that DPDK is used. It is simply a test example that provides no useful function to customers.

The main purpose of DPDK is Network Functions Virtualization (NFV), so in this section we use Open vSwitch as an example.

### About Open vSwitch

As defined in Wikipedia, Open vSwitch is a software-implemented virtual multilayer network switch, designed to enable effective network automation through programmatic extensions. It supports standard management interfaces and protocols, such as NetFlow, sFlow, SPAN, RSPAN, CLI, LACP and 802.1ag. In addition, Open vSwitch is designed to support transparent distribution across multiple physical servers by enabling the creation of cross-server switches in a way that abstracts out the underlying server architecture.

Open vSwitch, without DPDK, includes two parts, the user space component and the ykernel component.

The user space component is mainly used for data switching and OpenFlow. The kernel component is data path for fast packet forwarding. Figure 5 on page 13 shows the Open vSwitch framework.

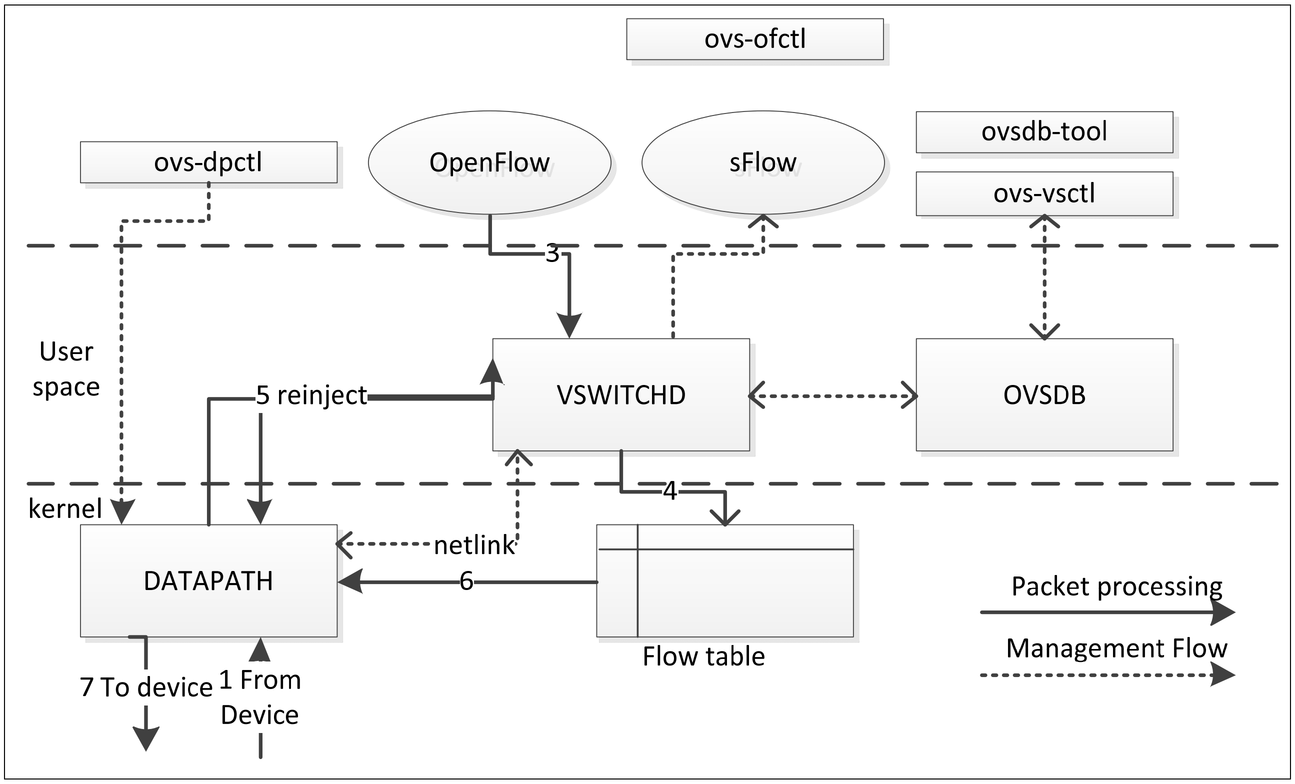

*Figure 5 Open vSwitch framework*

Packet flow is as follows:

1. The netdevice receives one packet.
2. The data path (kernel module Open vSwitch.ko) will process the packet first, if there is a flow table hit, and then forward it.
3. If the flow table in the kernel misses, the system informs vswitchd through netlink upcall.
4. vswitchd searches the flow table, and if there is a hit, vswitchd will update the flow table in the kernel.
5. If there is no hit, the vswitchd will turn it to Openflow or sFlow to make a decision for the packet, and then update the flow table in vswitchd and the kernel module.

## DPDK-accelerated Open vSwitch

When DPDK accelerates Open vSwitch, there are three main changes:

- Bypass the kernel datapath and release it to user space PMD drivers.
- Implement the netdev-dpdk code to let the vswitchd recognize the user space interface. In addition, the dpdk vhost-user interface is implemented in user space for virtualization.
- dpif-netdev: a user-space fast forwarding path.

## Configuring and compiling Open vSwitch

In the previous example, we set up the DPDK environment. Now, we to set up Open vSwitch with DPDK.

1. Go to http://openvswitch.org/download/ to download the Open vSwitch source code, or you can use git:

   ```
   git clone https://github.com/openvswitch/ovs.git
   ```

2. Extract the source code to directory /home/dpdk/openvswitch-2.7.0
3. Run boot.sh in the top source directory to build the “configure” script

   ```
   $ ./boot.sh
   ```

4. Configure the Open vSwitch project with the following command:

   ```
   ./configure
   --with-dpdk=/home/dpdk/dpdk-stable-16.11.1/x86_64-native-linuxapp-gcc
   CFLAGS="-g -O2 -msse4.2"
   ```

   DPDK support is disabled by default. We must enable it using the `--with-dpdk` option

5. Compile and install:

   ```
   make
   make install
   ```

## Run the Open vSwitch

Before we run the Open vSwitch, we need to insert the Open vSwitch module. Then we will start the Open vSwitch as follows:

### Start database daemon

1. Start the configuration database, ovsdb-server.

   Each machine on which Open vSwitch is installed should run its own copy of ovsdb-server. Before ovsdb-server itself can be started, we need to configure a database that it can use:

   ```
   $ export PATH=$PATH:/usr/local/share/openvswitch/scripts
   $ export DB_SOCK=/usr/local/var/run/openvswitch/db.sock
   $ mkdir -p /usr/local/etc/openvswitch
   $ ovsdb-tool create /usr/local/etc/openvswitch/conf.db \
       vswitchd/vswitch.ovsschema
   ```

2. Configure ovsdb-server to use the database created above, to listen on a Unix domain socket, and to connect to any managers specified in the database itself:

   ```
   $ mkdir -p /usr/local/var/run/openvswitch
   $ ovsdb-server --remote=punix:/usr/local/var/run/openvswitch/db.sock \
   --remote=db:Open_vSwitch,Open_vSwitch,manager_options \
   --pidfile --detach --log-file
   $ ovs-vsctl --no-wait init
   ```

### Configure DPDK-related parameters

Before we start the main Open vSwitch daemon, we should do some DPDK-related configuration work. DPDK configuration arguments can be passed to ovs-vswitchd via the other_config column of the Open_vSwitch table. At a minimum, the dpdk-init option must be set to true.

For example:

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

There are many other configuration options, the most important of which are listed below. Defaults will be provided for all values not explicitly set.

- `dpdk-init`

  Specifies whether OVS should initialize and support DPDK ports. This is a Boolean value, and defaults to false.

- `dpdk-lcore-mask`

  Specifies the CPU cores on which dpdk lcore threads should be spawned and expects a hex string value (e.g. ‘0x123’).

- `dpdk-socket-mem`

  Comma-separated list of memory to pre-allocate from hugepages on specific sockets.

- `dpdk-hugepage-dir`

  Directory where hugetlbfs is mounted

- `vhost-sock-dir`

  This option is used to set the path to the vhost-user Unix socket files.

If allocating more than one GB hugepage, you can configure the amount of memory used from any given NUMA node. For example, to use 1GB from NUMA node 0 and 0GB for all other NUMA nodes, run:

```
$ ovs-vsctl --no-wait set Open_vSwitch . other_config:dpdk-socket-mem="1024,0"
```

or:

```
$ ovs-vsctl --no-wait set Open_vSwitch . other_config:dpdk-socket-mem="1024"
```

Similarly, if you wish to better scale the workloads across cores, then multiple PMD threads can be created and pinned to CPU cores by explicitly specifying pmd-cpu-masks. Cores are numbered from 0, so to spawn two PMD threads and pin them to cores 1 and 2, run:

```
$ ovs-vsctl set Open_vSwitch . other_config:pmd-cpu-mask=0x6
```

## Start the main Open vSwitch daemon

Start the daemon using the following command:

```
$ ovs-vswitchd --pidfile --detach --log-file;
```

It will show the following log which indicates the Open vSwitch has started to run:

*Example 2 Output from starting the Open vSwitch daemon*

```
linux-sm0h:~ # ovs-ctl --no-ovsdb-server --db-sock="$DB_SOCK" start
EAL: Detected 28 lcore(s)
EAL: No free hugepages reported in hugepages-1048576kB
EAL: Probing VFIO support...
EAL: PCI device 0000:5b:00.0 on NUMA socket 0
EAL:   probe driver: 8086:1521 net_e1000_igb
EAL: PCI device 0000:5b:00.1 on NUMA socket 0
EAL:   probe driver: 8086:1521 net_e1000_igb
EAL: PCI device 0000:5b:00.2 on NUMA socket 0
EAL:   probe driver: 8086:1521 net_e1000_igb
EAL: PCI device 0000:5b:00.3 on NUMA socket 0
EAL:   probe driver: 8086:1521 net_e1000_igb
```

```
EAL: PCI device 0000:b0:00.0 on NUMA socket 0
EAL:   probe driver: 8086:37d1 net_i40e
EAL: PCI device 0000:b0:00.1 on NUMA socket 0
EAL:   probe driver: 8086:37d1 net_i40e
Zone 0: name:<rte_eth_dev_data>, phys:0x6f1cec40, len:0x30100, virt:0x7fa70afcec40,
socket_id:0, flags:0
VHOST_CONFIG: vhost-user server: socket created, fd: 57
VHOST_CONFIG: bind to /usr/local/var/run/openvswitch/subdir/vhost-user-3
VHOST_CONFIG: vhost-user client: socket created, fd: 64
VHOST_CONFIG: failed to connect to /home/dpdk/socket: Connection refused
VHOST_CONFIG: /home/dpdk/socket: reconnecting...
VHOST_CONFIG: vhost-user server: socket created, fd: 65
VHOST_CONFIG: bind to /usr/local/var/run/openvswitch/subdir/vhost-user-4
Starting ovs-vswitchd done
Enabling remote OVSDB managers
```

## Configure DPDK physical ports during runtime

We first need to create one bridge, and add a DPDK port to it. Specifically, the dpdk-devargs will assign a specified DPDK interface (using PCI address) to Open vSwitch.

In the following example, we add two physical ports (0000:5b:00.2, 0000:5b:00.3) to bridge br0. These two ports are the Internet ports that will allow the VM in the host to transmit packets externally.

```
$ ovs-vsctl add-br br0 -- set bridge br0 datapath_type=netdev
$ ovs-vsctl add-port br0 myportnameone -- set Interface myportnameone \
    type=dpdk options:dpdk-devargs=0000:5b:00.2
$ ovs-vsctl add-port br0 myportnametwo -- set Interface myportnametwo \
type=dpdk options:dpdk-devargs= 0000:5b:00.3
```

## Configure virtIO port and VM

Now we need to add vhost-user ports to bridge br0, which will include the VM in the bridge domain.

Open vSwitch provides two types of vHost User ports:

- vhost-user (dpdkvhostuser)
- vhost-user-client (dpdkvhostuserclient)

vHost User uses a client-server model. The server creates/manages/destroys the vHost User sockets, and the client connects to the server. Depending on which port type you use, dpdkvhostuser or dpdkvhostuserclient, a different configuration of the client-server model is used.

For vhost-user ports, Open vSwitch acts as the server and QEMU the client. This means if OVS crashes, all VMs must be restarted. On the other hand, for vhost-user-client ports, OVS acts as the client and QEMU the server. This means OVS can crash and be restarted without issue, and it is also possible to restart the instance itself. For this reason, vhost-user-client ports are the preferred type in all known scenarios. The only limitation is that vhost-user client mode ports require QEMU version 2.7. Ports of the vhost-user type are currently deprecated and will be removed in a future release.

To use vhost-user ports for guest virtual machines, we must first add said ports to the switch. DPDK vhost-user ports can have arbitrary names, except for forward and backward slashes, which are prohibited. For vhost-user, the port type is dpdkvhostuser:

```
$ ovs-vsctl add-port br0 vhost-user-1 -- set Interface vhost-user-1 type=dpdkvhostuser
$ ovs-vsctl add-port br0 vhost-user-2 -- set Interface vhost-user-2 type=dpdkvhostuser
```

Now, we should install several VMs using KVM. The installation process is outside the scope of this document.

In our example, we will install two RHEL 7.2 images which already include virtio front-end drivers. The VM images are in the following locations:

- /var/lib/libvirt/images/generic-3.qcow2
- /var/lib/libvirt/images/generic-3-clone.qcow2

Now we can input several virtio related parameters to start the VM, as follows:

```
qemu-kvm -cpu host -smp 2 -hda /var/lib/libvirt/images/generic-3.qcow2 -m 1024M
--enable-kvm -object
memory-backend-file,id=mem,size=1024M,mem-path=/dev/hugepages,share=on -numa
node,memdev=mem -mem-prealloc -chardev
socket,id=char1,path=/usr/local/var/run/openvswitch/subdir/vhost1 -netdev
type=vhost-user,id=mynet1,chardev=char1,vhostforce,queues=2 -device
virtio-net-pci,mac=11:22:33:44:55:66,netdev=mynet1,mq=on,vectors=6 -vnc :1 &

qemu-kvm -cpu host -smp 2 -hda /var/lib/libvirt/images/generic-3-clone.qcow2 -m
1024M --enable-kvm -object
memory-backend-file,id=mem,size=1024M,mem-path=/dev/hugepages,share=on -numa
node,memdev=mem -mem-prealloc -chardev
socket,id=char1,path=/usr/local/var/run/openvswitch/subdir/vhost2 -netdev
type=vhost-user,id=mynet1,chardev=char1,vhostforce,queues=2 -device
virtio-net-pci,mac=11:22:33:44:55:67,netdev=mynet1,mq=on,vectors=6 -vnc :2 &
```

The `-chardev` option will pass the previous socket (created by vhost-user port) to the virtual machine.

The `-netdev` option will assign the NIC type and related parameters to the virtual machine.

In this example, we will start two virtual machines bound to the two previously created virtio parts. Their MAC addresses are 11:22:33:44:55:66 and 11:22:33:44:55:67.

When these two VMs are started, we should log in and config the NIC we passed to the VM with an IP address. In this example, we will set one address to 5.5.5.5/24 and the other to 5.5.5.6/25

## Start bridge br0

Now, we start the bridge br0 on the host using the following command:

```
ifconfig br0 5.5.5.1/24 up
```

### Result

We are now able to generate packets on one physical port with different MACs. The packets will be received by the DPDK PMD driver, and then submitted to the Open vSwitch daemon. The Open vSwitch will distribute the packets to different virtio interfaces according to their MACs and ultimately reach the corresponding VM.

If we connect one DPDK physical port to a remote server, we can ping the VMs on the server.

## Online resources

- DPDK home page

  http://dpdk.org/
- Open vSwitch Documentation

  http://docs.openvswitch.org
- Intel article, *Open vSwitch with DPDK Overview*

  https://software.intel.com/en-us/articles/open-vswitch-with-dpdk-overview

## Change history

January 2018:

- Corrrections for grammar and readability

## Author

**Yao Aili** is a Linux Engineer at the Lenovo Data Center Group in Beijing, China. He has worked with Linux for more than ten years, mainly focusing on networking and kernel optimization.

Thanks to the following people for their contributions to this project:

- David Watts, Lenovo Press
- Mark T. Chapman, Lenovo editor

## Notices

Lenovo may not offer the products, services, or features discussed in this document in all countries. Consult your local Lenovo representative for information on the products and services currently available in your area. Any reference to a Lenovo product, program, or service is not intended to state or imply that only that Lenovo product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any Lenovo intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any other product, program, or service.

Lenovo may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

Lenovo (United States), Inc.
1009 Think Place - Building One
Morrisville, NC 27560
U.S.A.
Attention: Lenovo Director of Licensing

LENOVO PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. Lenovo may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

The products described in this document are not intended for use in implantation or other life support applications where malfunction may result in injury or death to persons. The information contained in this document does not affect or change Lenovo product specifications or warranties. Nothing in this document shall operate as an express or implied license or indemnity under the intellectual property rights of Lenovo or third parties. All information contained in this document was obtained in specific environments and is presented as an illustration. The result obtained in other operating environments may vary.

Lenovo may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Any references in this publication to non-Lenovo Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this Lenovo product, and use of those Web sites is at your own risk.

Any performance data contained herein was determined in a controlled environment. Therefore, the result obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

**© Copyright Lenovo 2018. All rights reserved.**
Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by Global Services Administration (GSA) ADP Schedule Contract

This document was created or updated on January 28, 2018.

Send us your comments via the **Rate & Provide Feedback** form found at
http://lenovopress.com/lp0749

## Trademarks

Lenovo, the Lenovo logo, and For Those Who Do are trademarks or registered trademarks of Lenovo in the United States, other countries, or both. These and other Lenovo trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by Lenovo at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of Lenovo trademarks is available on the Web at http://www.lenovo.com/legal/copytrade.html.

The following terms are trademarks of Lenovo in the United States, other countries, or both:

| | | |
|---|---|---|
| Lenovo® | Lenovo(logo)® | ThinkSystem™ |

The following terms are trademarks of other companies:

Intel, Xeon, and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.