dpdk

Release 0.11

Nov 23, 2017

Contents

1 Linux平台上DPDK入门指南 1

2 Getting Started Guide for FreeBSD 25

3 Sample Applications User Guides 37

4 编程指南 249

5 HowTo Guides 435

6 DPDK Tools User Guides 471

7 Testpmd Application User Guide 483

8 Network Interface Controller Drivers 537

9 Crypto Device Drivers 641

10 Event Device Drivers 659

11 Xen Guide 665

12 Contributor's Guidelines 673

13 Release Notes 717

14 FAQ 793

CHAPTER 1

Linux平台上DPDK入门指南

1.1 简介

本文档包含DPDK软件安装和配置的相关说明。旨在帮助用户快速启动和运行软件。文档主要描述了在Linux环境下编译和 运行DPDK应用程序，但是文档并不深入DPDK的具体实现细节。

1.1.1 文档地图

以下是一份建议顺序阅读的DPDK参考文档列表：

- **发布说明**：提供特性发行版本的信息，包括支持的功能，限制，修复的问题，已知的问题等等。此外，还以FAQ方式提供了常见问题及解答。
- **入门指南**（本文档）：介绍如何安装和配置DPDK，旨在帮助用户快速上手。
- **编程指南**：描述如下内容：
  - 软件架构及如何使用（实例介绍），特别是在Linux环境中的用法
  - DPDK的主要内容，系统构建（包括可以在DPDK根目录Makefile中来构建工具包和应用程序的命令）及应用移植细则。
  - 软件中使用的，以及新开发中需要考虑的一些优化。

  还提供了文档使用的术语表。
- **API参考**：提供有关DPDK功能、数据结构和其他编程结构的详细信息。
- **示例程序用户指南**：描述了一组例程。 每个章节描述了一个用例，展示了具体的功能，并提供了有关如何编译、运行和使用的说明。

1.2 系统要求

本章描述了编译DPDK所需的软件包。

Note: 假如在Intel公司的89xx通信芯片组平台上使用DPDK，请参阅文档 *Intel® Communications Chipset 89xx Series Software for Linux Getting Started Guide*。

1.2.1 X86 上预先设置 BIOS

对大多数平台，使用基本DPDK功能无需对BIOS进行特殊设置。然而，对于HPET定时器和电源管理功能，以及为了获得40G网卡上小包处理的高性能，则可能需要更改BIOS设置。可以参阅章节 *Enabling Additional Functionality* 以获取更为详细的信息。

1.2.2 编译DPDK

工具集：

Note: 以下说明在Fedora 18上通过了测试。其他系统所需要的安装命令和软件包可能有所不同。有关其他Linux发行版本和测试版本的详细信息，请参阅DPDK发布说明。

- GNU `make`.
- coreutils: `cmp`, `sed`, `grep`, `arch` 等.
- gcc: 4.9以上的版本适用于所有的平台。 在某些发布版本中，启用了一些特定的编译器标志和链接标志（例如‘‘-fstack-protector‘‘）。请参阅文档的发布版本和 `gcc -dumpspecs`.
- libc 头文件，通常打包成 `gcc-multilib` (`glibc-devel.i686 / libc6-dev-i386`; `glibc-devel.x86_64 / libc6-dev` 用于Intel 64位架构编译; `glibc-devel.ppc64` 用于IBM 64位架构编译;)
- 构建Linux内核模块所需要的头文件和源文件。(kernel - devel.x86_64; kernel - devel.ppc64)
- 在64位系统上编译32位软件包额外需要的软件为：
  - glibc.i686, libgcc.i686, libstdc++.i686 及 glibc-devel.i686， 适用于Intel的i686/x86_64;
  - glibc.ppc64, libgcc.ppc64, libstdc++.ppc64 及 glibc-devel.ppc64 适用于 IBM ppc_64;

  Note: x86_x32 ABI目前仅在Ubuntu 13.10及以上版本或者Debian最近的发行版本上支持。编译器必须是gcc 4.9+版本。

- Python, 2.7+ or 3.2+版本, 用于运行DPDK软件包中的各种帮助脚本。

可选工具：

- Intel® C++ Compiler (icc). 安装icc可能需要额外的库，请参阅编译器安装目录下的icc安装指南。
- IBM® Advance ToolChain for Powerlinux. 这是一组开源开发工具和运行库。允许用户在Linux上使用IBM最新POWER硬件的优势。具体安装请参阅IBM的官方安装文档。
- libpcap 头文件和库 (libpcap-devel)，用于编译和使用基于libcap的轮询模式驱动程序。默认情况下，该驱动程序被禁用，可以通过在构建时修改配置文件 `CONFIG_RTE_LIBRTE_PMD_PCAP=y` 来开启。
- 需要使用libarchive 头文件和库来进行某些使用tar获取资源的单元测试。

1.2.3 运行DPDK应用程序

要运行DPDK应用程序，需要在目标机器上进行某些定制。

系统软件

需求：

- Kernel version >= 2.6.34

  当前内核版本可以通过命令查看:

```
uname -r
```

- glibc >= 2.7 (方便使用cpuset相关特性)

  版本信息通命令 `ldd --version` 查看。

- Kernel configuration

  在 Fedora OS 及其他常见的发行版本中，如 Ubuntu 或 Red Hat Enterprise Linux，供应商提供的配置可以运行大多数
  对于其他内核构件，应为DPDK开启的选项包括：

  - UIO 支持
  - HUGETLBFS 支持
  - PROC_PAGE_MONITOR 支持
  - 如果需要HPET支持，还应开启 HPET and HPET_MMAP 配置选项。有关信息参考 *High Precision Event Timer (HPET) Functionality* 章节获取更多信息。

在 Linux 环境中使用 Hugepages

用于数据包缓冲区的大型内存池分配需要 Hugepages 支持（如上节所述，必须在运行的内核中开启 HUGETLBFS 选项）。通过使用大页分配，程序需要更少的页面，性能增加， 因为较少的TLB减少了将虚拟页面地址翻译成物理页面地址所需的时间。如果没有大页，标准大小4k的页面会导致频繁的TLB miss，性能下降。

预留 Hugepages 给 DPDK 使用

大页分配应该在系统引导时或者启动后尽快完成，以避免物理内存碎片化。要在引导时预留大页，需要给Linux内核命令行传递一个参数。

对于2MB大小的页面，只需要将hugepages选项传递给内核。如，预留1024个2MB大小的page，使用:

```
hugepages=1024
```

对于其他大小的hugepage，例如1G的页，大小必须同时指定。例如，要预留4个1G大小的页面给程序，需要传递以下选项给内核:

```
default_hugepagesz=1G hugepagesz=1G hugepages=4
```

Note:

CPU支持的hugepage大小可以从Intel架构上的CPU标志位确定。如果存在pse，则支持2M个hugepages，如果page1gb存在，
在IBM Power架构中，支持的hugepage大小为16MB和16GB。

Note: 对于64位程序，如果平台支持，建议使用1GB的hugepages。

在双插槽NUMA的系统上，在启动时预留的hugepage数目通常在两个插槽之间评分（假设两个插槽上都有足够的内存）。

有关这些和其他内核选项的信息，请参阅Linux源代码目录中/kernel-parameter.txt文件。

特例:

对于2MB页面，还可以在系统启动之后再分配，通过向 `/sys/devices/` 目录下的nr_hugepages文件写入hugepage数目来实现。 对于单节点系统，使用的命令如下（假设需要1024个页）:

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

在NUMA设备中，分配应该明确指定在哪个节点上:

```
echo 1024 > /sys/devices/system/node/node0/hugepages/hugepages-2048kB/nr_hugepages
echo 1024 > /sys/devices/system/node/node1/hugepages/hugepages-2048kB/nr_hugepages
```

Note: 对于1G页面，系统启动之后无法预留页面内存。

DPDK 使用 Hugepages

一旦预留了hugepage内存，为了使内存可用于DPDK，请执行以下步骤:

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

通过将一下命令添加到 `/etc/fstab` 文件中，安装点可以在重启时永久保存:

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

对于1GB内存，页面大小必须在安装选项中指定:

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

Linux 环境中 Xen Domain0 支持

现有的内存管理实现是基于Linux内核的hugepage机制。在Xen虚拟机管理程序中，对于DomainU客户端的支持意味着DPDK程序与客户一样正常工作。

但是，Domain0不支持hugepages。为了解决这个限制，添加了一个新的内核模块rte_dom0_mm用于方便内存的分配和映射，通过 **IOCTL** (分配) 和 **MMAP** (映射).

DPDK 中使能 Xen Dom0模式

默认情况下，DPDK构建时禁止使用 Xen Dom0 模式。要支持 Xen Dom0，CONFIG_RTE_LIBRTE_XEN_DOM0设置应该改为 “y”，编译时弃用该模式。

此外，为了允许接收错误套接字ID，CONFIG_RTE_EAL_ALLOW_INV_SOCKET_ID也必须设置为 “y”。

加载 DPDK rte_dom0_mm 模块

要在Xen Dom0下运行任何DPDK应用程序，必须使用rsv_memsize选项将 `rte_dom0_mm` 模块加载到运行的内核中。该模块位于DPDK目标目录的kmod子目录中。应该使用insmod命令加载此模块，如下所示:

```
sudo insmod kmod/rte_dom0_mm.ko rsv_memsize=X
```

X的值不能大于4096(MB).

配置内存用于DPDK使用

在加载rte_dom0_mm.ko内核模块之后，用户必须配置DPDK使用的内存大小。这也是通过将内存大小写入到目录 `/sys/devices/` 下的文件memsize中来实现的。 使用以下命令(假设需要2048MB):

```
echo 2048 > /sys/kernel/mm/dom0-mm/memsize-mB/memsize
```

用户还可以使用下面命令检查已经使用了多少内存:

```
cat /sys/kernel/mm/dom0-mm/memsize-mB/memsize_rsvd
```

Xen Domain0 不支持NUMA配置，因此 `--socket-mem` 命令选项对Xen Domain0无效。

Note: memsize的值不能大于rsv_memsize。

在 Xen Domain0上运行DPDK程序

要在Xen Domain0上运行DPDK程序，需要一个额外的命令行选项 `--xen-dom0`。

1.3 使用源码编译DPDK目标文件

Note: 这个过程的部分工作可以通过章节 使用脚本快速构建 描述的脚本来实现。

1.3.1 安装DPDK及源码

首先，解压文件并进入到DPDK源文件根目录下：

```
tar xJf dpdk-<version>.tar.xz
cd dpdk-<version>
```

DPDK源文件由几个目录组成:

- lib: DPDK 库文件
- drivers: DPDK 轮询驱动源文件
- app: DPDK 应用程序 (自动测试)源文件
- examples: DPDK 应用例程
- config, buildtools, mk: 框架相关的makefile、脚本及配置文件

1.3.2 DPDK目标环境安装

DPDK目标文件的格式为:

```
ARCH-MACHINE-EXECENV-TOOLCHAIN
```

其中:

- ARCH 可以是: i686, x86_64, ppc_64
- MACHINE 可以是: native, power8
- EXECENV 可以是: linuxapp, bsdapp
- TOOLCHAIN 可以是: gcc, icc

目标文件取决于运行环境是32位还是64位设备。可以在DPDK的 /config 目录中找到可用的目标，不能使用defconfig_前缀。

Note:

配置文件根据 RTE_MACHINE 优化级别不同分别提供。在配置文件内部，RTE_MACHINE 配置为 native， 意味着已编译的软件被调整到其构建的平台上。有关此设置的更多信息，请参阅 *DPDK* 编程指南。

当使用Intel® C++ 编译器 (icc)时，对64位和32位,需要使用以下命令进行调整。 注意，shell脚本会更新 $PATH 值，因此不能再同一个会话中执行。 此外，还应该检查编译器的安装目录，因为可能不同。

```
source /opt/intel/bin/iccvars.sh intel64
source /opt/intel/bin/iccvars.sh ia32
```

在顶级目录中使用 make install T=<target> 来生成目标文件。

例如，为了使用icc编译生成64位目标文件，运行如下命令:

```
make install T=x86_64-native-linuxapp-icc
```

为了使用gcc编译生成32位目标文件，命令如下:

```
make install T=i686-native-linuxapp-gcc
```

如果仅仅只是生成目标文件，并不运行，比如，配置文件改变需要重新编译，使用 make config T=<target> 命令:

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

Warning: 任何需要运行的内核模块，如 `igb_uio`, `kni`, 必须在与目标文件编译相同的内核下进行编译。 如果DPDK未在目标设备上构建，则应使用 `RTE_KERNELDIR` 环境变量将编译指向要在目标机上使用的内核版本的副本（交叉编译的内核版本）。

创建目标环境之后，用户可以移动到目标环境目录，并继续更改代码并编译。用户还可以通过编辑build目录中的.config文件对DPDK配置进行修改。 (这是顶级目录中defconfig文件的本地副本)。

```
cd x86_64-native-linuxapp-gcc
vi .config
make
```

此外，make clean命令可以用于删除任何现有的编译文件，以便后续完整、干净地重新编译代码。

1.3.3 Browsing the Installed DPDK Environment Target

一旦目标文件本创建，它就包含了构建客户应用程序所需的DPDK环境的所有库，包括轮询驱动程序和头文件。 此外，test和testpmd应用程序构建在build/app目录下，可以用于测试。 还有一个kmod目录，存放可能需要加载的内核模块。

1.3.4 加载模块启动DPDK环境需要的UIO功能

要运行任何的DPDK应用程序，需要将合适的uio模块线加载到当前内核中。在多数情况下，Linux内核包含了标准的 `uio_pci_generic` 模块就可以提供uio能力。 该模块可以使用命令加载

```
sudo modprobe uio_pci_generic
```

区别于 `uio_pci_generic` ，DPDK提供了一个igb_uio模块（可以在kmod目录下找到）。可以通过如下方式加载:

```
sudo modprobe uio
sudo insmod kmod/igb_uio.ko
```

Note: 对于一下不支持传统中断的设备，例如虚拟功能（VF）设备，必须使用 `igb_uio` 来替代 `uio_pci_generic` 模块。

由于DPDK 1.7版本提供VFIO支持，所以，对于支持VFIO的平台，可选则UIO，也可以不用。

1.3.5 加载VFIO模块

DPDK程序选择使用VFIO时，需要加载 `vfio-pci` 模块:

```
sudo modprobe vfio-pci
```

注意，要使用VFIO，首先，你的平台内核版本必须支持VFIO功能。 Linux内核从3.6.0版本之后就一直包含VFIO模块，通常是默认存在的。不够请查询发行文档以确认是否存在。

此外，要使用VFIO，内核和BIOS都必须支持，并配置为使用IO虚拟化 (如 Intel® VT-d)。

为了保证非特权用户运行DPDK时能够正确操作VFIO，还应设置正确的权限。这可以通过DPDK的配置脚本(dpdk-setup.sh文件位于usertools目录中)。

1.3.6 网络端口绑定/解绑定到内核去顶模块

从版本1.4开始，DPDK应用程序不再自动解除所有网络端口与原先内核驱动模块的绑定关系。 相反的，DPDK程序在运行前，需要将所要使用的端口绑定到 uio_pci_generic, igb_uio 或 vfio-pci 模块上。 任何Linux内核本身控制的端口无法被DPDK PMD驱动所使用。

Warning: 默认情况下，DPDK将在启动时不再自动解绑定内核模块与端口的关系。DPDK应用程序使用的任何端口必须与Linux无关，并绑定到 uio_pci_generic, igb_uio 或 vfio-pci 模块上。

将端口从Linux内核解绑，然后绑定到 uio_pci_generic, igb_uio 或 vfio-pci 模块上供DPDK使用，可以使用脚本dpdk_nic _bind.py（位于usertools目录下）。 这个工具可以用于提供当前系统上网络接口的状态图，绑定或解绑定来自不同内核模块的接口。 以下是脚本如何使用的一些实例。通过使用 --help or --usage 选项调用脚本，可以获得脚本的完整描述与帮助信息。 请注意，要将接口绑定到uio或vfio的话，需要先将这两个模块加载到内核，再运行 dpdk-devbind.py 脚本。

Warning:

由于VFIO的工作方式，设备是否可用VFIO是有明确限制的。大部分是由IOMMU组的功能决定的。
任何的虚拟设备可以独立使用VFIO，但是物理设备则要求将所有端口绑定到VFIO，或者其中一些绑定到VFIO，而其他端口不能绑定到任何其他驱动程序。

如果你的设备位于PCI-to-PCI桥之后，桥接器将成为设备所在的IOMMU组的一部分。因此，桥接驱动程序也应该从端口解绑定。

Warning: 虽然任何用户都可以运行dpdk-devbind.py脚本来查看网络接口的状态，但是绑定和解绑定则需要root权限。

查看系统中所有网络接口的状态:

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

Network devices using DPDK-compatible driver
============================================
0000:82:00.0 '82599EB 10-GbE NIC' drv=uio_pci_generic unused=ixgbe
0000:82:00.1 '82599EB 10-GbE NIC' drv=uio_pci_generic unused=ixgbe

Network devices using kernel driver
===================================
0000:04:00.0 'I350 1-GbE NIC' if=em0  drv=igb unused=uio_pci_generic *Active*
0000:04:00.1 'I350 1-GbE NIC' if=eth1 drv=igb unused=uio_pci_generic
0000:04:00.2 'I350 1-GbE NIC' if=eth2 drv=igb unused=uio_pci_generic
0000:04:00.3 'I350 1-GbE NIC' if=eth3 drv=igb unused=uio_pci_generic

Other network devices
=====================
<none>
```

绑定设备 eth1,‘‘04:00.1‘‘, 到 uio_pci_generic 驱动:

```
./usertools/dpdk-devbind.py --bind=uio_pci_generic 04:00.1
```

或者

```
./usertools/dpdk-devbind.py --bind=uio_pci_generic eth1
```

恢复设备 82:00.0 到Linux内核绑定状态：

```
./usertools/dpdk-devbind.py --bind=ixgbe 82:00.0
```

1.4 编译和运行简单应用程序

本章介绍如何在DPDK环境下编译和运行应用程序。还指出应用程序的存储位置。

Note: 此过程的部分操作也可以使用脚本来完成。参考 使用脚本快速构建 章节描述。

1.4.1 编译一个简单应用程序

一个DPDK目标环境创建完成时(如 x86_64-native-linuxapp-gcc)，它包含编译一个应用程序所需要的全部库和头文件。

当在Linux* 交叉环境中编译应用程序时，以下变量需要预先导出：

- RTE_SDK - 指向DPDK安装目录。
- RTE_TARGET - 指向DPDK目标环境目录。

以下是创建 helloworld 应用程序实例，该实例将在DPDK Linux环境中运行。 这个实例可以在目录 ${RTE_SDK}/examples 找到。

该目录包含 main.c 文件。该文件与DPDK目标环境中的库结合使用时，调用各种函数初始化DPDK环境，然后，为每个要使用的core启动一个入口点（调度应用程序） 。 默认情况下，二进制文件存储在build目录中。

```
cd examples/helloworld/
export RTE_SDK=$HOME/DPDK
export RTE_TARGET=x86_64-native-linuxapp-gcc

make
    CC main.o
    LD helloworld
    INSTALL-APP helloworld
    INSTALL-MAP helloworld.map

ls build/app
    helloworld helloworld.map
```

Note: 在上面的例子中， helloworld 是在DPDK的目录结构下的。 当然，也可以将其放在DPDK目录之外，以保证DPDK的结构不变。 下面的例子， helloworld 应用程序被复制到一个新的目录下。

```
export RTE_SDK=/home/user/DPDK
cp -r $(RTE_SDK)/examples/helloworld my_rte_app
cd my_rte_app/
export RTE_TARGET=x86_64-native-linuxapp-gcc

make
```

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

1.4.2 运行一个简单的应用程序

Warning: UIO驱动和hugepage必须在程序运行前设置好。

Warning: 应用程序使用的任何端口，必须绑定到合适的内核驱动模块上，如章节 网络端口绑定/解绑定到内核去顶模块 描述的那样。

应用程序与DPDK目标环境的环境抽象层（EAL）库相关联，该库提供了所有DPDK程序通用的一些选项。

以下是EAL提供的一些选项列表：

```
./rte-app -c COREMASK [-n NUM] [-b <domain:bus:devid.func>] \
          [--socket-mem=MB,...] [-m MB] [-r NUM] [-v] [--file-prefix] \
          [--proc-type <primary|secondary|auto>] [-- xen-dom0]
```

选项描述如下：

- `-c COREMASK`: 要运行的内核的十六进制掩码。注意，平台之间编号可能不同，需要事先确定。
- `-n NUM`: 每个处理器插槽的内存通道数目。
- `-b <domain:bus:devid.func>`: 端口黑名单，避免EAL使用指定的PCI设备。
- `--use-device`: 仅使用指定的以太网设备。使用逗号分隔 `[domain:]bus:devid.func` 值，不能与 `-b` 选项一起使用。
- `--socket-mem`: 从特定插槽上的hugepage分配内存。
- `-m MB`: 内存从hugepage分配，不管处理器插槽。建议使用 `--socket-mem` 而非这个选项。
- `-r NUM`: 内存数量。
- `-v`: 显示启动时的版本信息。
- `--huge-dir`: 挂载hugetlbfs的目录。
- `--file-prefix`: 用于hugepage文件名的前缀文本。
- `--proc-type`: 程序实例的类型。
- `--xen-dom0`: 支持在Xen Domain0上运行，但不具有hugetlbfs的程序。
- `--vmware-tsc-map`: 使用VMware TSC 映射而不是本地RDTSC。
- `--base-virtaddr`: 指定基本虚拟地址。
- `--vfio-intr`: 指定要由VFIO使用的中断类型。(如果不支持VFIO，则配置无效)。

其中 `-c` 是强制性的，其他为可选配置。

将DPDK应用程序二进制文件拷贝到目标设备，按照如下命令运行(我们假设每个平台处理器有4个内存通道，并且存在core0～3用于运行程序):

```
./helloworld -c f -n 4
```

Note: 选项 `--proc-type` 和 `--file-prefix` 用于运行多个DPDK进程。请参阅 “多应用程序实例” 章节及 *DPDK* 编程指南 获取更多细节。

应用程序使用的逻辑Core

对于DPDK应用程序，coremask参数始终是必须的。掩码的每个位对应于Linux提供的逻辑core ID。 由于这些逻辑core的编号，以及他们在NUMA插槽上的映射可能因平台而异，因此建议在选择每种情况下使用的coremaks时，都要考虑每个平台的core布局。

在DPDK程序初始化EAL层时，将显示要使用的逻辑core及其插槽位置。可以通过读取 `/proc/cpuinfo` 文件来获取系统上所有core的信息。例如执行 cat `/proc/cpuinfo`。 列出来的physical id 属性表示其所属的CPU插槽。当使用了其他处理器来了解逻辑core到插槽的映射时，这些信息很有用。

Note: 可以使用另一个Linux工具 `lstopo` 来获取逻辑core布局的图形化信息。在Fedora Linux上, 可以通过如下命令安装并运行工具:

```
sudo yum install hwloc
./lstopo
```

Warning: 逻辑core在不同的电路板上可能不同，在应用程序使用coremaks时需要先确定。

应用程序使用的Hugepage内存

当运行应用程序时，建议使用的内存与hugepage预留的内存一致。如果运行时没有 `-m` 或 `--socket-mem` 参数传入，这由DPDK应用程序在启动时自动完成。

如果通过显示传入 `-m` 或 `--socket-mem` 值，但是请求的内存超过了该值，应用程序将执行失败。 但是，如果用户请求的内存小于预留的hugepage-memory，应用程序也会失败，特别是当使用了 `-m` 选项的时候。因为，假设系统在插槽0和插槽1上有1024个预留的2MB页面，如果用户请求128 MB的内存，可能存在64个页不符合要求的情况:

- 内核只能在插槽1中将hugepage-memory提供给应用程序。在这种情况下，如果应用程序尝试创建一个插槽0中的对象，例如ring或者内存池，那么将执行失败 为了避免这个问题，建议使用 `--socket-mem` 选项替代 `-m` 选项。
- 这些页面可能位于物理内存中的任意位置，尽管DPDK EAL将尝试在连续的内存块中分配内存，但是页面可能是不连续的。在这种情况下，应用程序无法分配大内存。

使用socket-mem选项可以为特定的插槽请求特定大小的内存。通过提供 `--socket-mem` 标志和每个插槽需要的内存数量来实现的，如 `--socket-mem=0,512` 用于在插槽1上预留512MB内存。 类似的，在4插槽系统上，如果只能在插槽0和2上分配1GB内存，则可以使用参数‘‘–socket-mem=1024,0,1024‘‘ 来实现。 如果DPDK无法在每个插槽上分配足够的内存，则EAL初始化失败。

1.4.3 其他示例程序

其他的一些示例程序包含在${RTE_SDK}/examples 目录下。这些示例程序可以使用本手册前面部分所述的

方法进行构建运行。另外，请参阅 *DPDK*示例程序用户指南 了解应用程序的描述、编译和执行的具体说明以及代码解释。

1.4.4 附加的测试程序

此外，还有两个在创建库时构建的应用程序。这些源文件位于 DPDK/app目录下，称为test和testpmd程序。创建库之后，可以在build目录中找到。

- test程序为DPDK中的各种功能提供具体的测试。
- testpmd程序提供了许多不同的数据包吞吐测试，例如，在Intel® 82599 10 Gigabit Ethernet Controller中如何使用Flow Director。

1.5 启用附加功能

1.5.1 高精度事件定时器 (HPET) 功能

BIOS 支持

要使用HPET功能时，必须先在平台BIOS上开启高精度定时器。否则，默认情况下使用时间戳计数器(TSC) 。 通常情况下，起机时按 F2 可以访问BIOS。然后用户可以导航到HPET选项。 在Crystal Forest平台BIOS上，路径为：**Advanced -> PCH-IO Configuration -> High Precision Timer ->** (如果需要，将Disabled改为 Enabled)。

在已经起机的系统上，可以使用以下命令来检查HPET是否启用

```
grep hpet /proc/timer_list
```

如果没有条目，则必须在BIOS中启用HPET，镔铁重新启动系统。

Linux 内核支持

DPDK通过将定时器计数器映射到进程地址空间来使用平台的HPET功能，因此，要求开启 `HPET_MMAP` 系统内核配置选项。

> **Warning:** 在Fedora或者其他常见的Linux发行版本（如Ubuntu）中，默认不会启用 `HPET_MMAP` 选项。要重新编译启动此选项的内核，请参阅发行版本的相关说明。

DPDK 中使能 HPET

默认情况下，DPDK配置文件中是禁用HPET功能的。要使用HPET，需要将 `CONFIG_RTE_LIBEAL_USE_HPET` 设置为 `y` 来开启编译。

对于那些使用 `rte_get_hpet_cycles()` 及 `rte_get_hpet_hz()` API接口的应用程序，并且选择了HPET作为rte_timer库的默认时钟源，需要在初始化时调用 `rte_eal_hpet_init()` API。 这个API调用将保证HPET可用，如果HPET不可用(例如，内核没有开启 `HPET_MMAP` 使能)，则向程序返回一个错误值。 如果HPET在运行时不可用，应用程序可以方便的采取其他措施。

Note: 对于那些仅需要普通定时器API，而不是HPET定时器的应用程序，建议使用 `rte_get_timer_cycles()` 和 `rte_get_timer_hz()` API调用，而不是HPET API。这些通用的API兼容TSC和HPET时钟源，具体时钟源则取决于应用程序是否调用 "rte_eal_hpet_init()"初始化，以及运行时系统上可用的时钟。

1.5.2 没有Root权限情况下运行DPDK应用程序

虽然DPDK应用程序直接使用了网络端口及其他硬件资源，但通过许多小的权限调整，可以允许除root权限之外的用户运行这些应用程序。为了保证普通的Linux用户也可以运行这些程序，需要调整如下Linux文件系统权限:

- 所有用于hugepage挂载点的文件和目录，如 `/mnt/huge`
- `/dev` 中的UIO设备文件，如 `/dev/uio0`, `/dev/uio1` 等
- UIO系统配置和源文件，如 `uio0`:

```
/sys/class/uio/uio0/device/config
/sys/class/uio/uio0/device/resource*
```

- 如果要使用HPET，那么 `/dev/hpet` 目录也要修改

Note: 在某些Linux 安装中， `/dev/hugepages` 也是默认创建hugepage挂载点的文件。

1.5.3 电源管理和节能功能

如果要使用DPDK的电源管理功能，必须在平台BIOS中启用增强的Intel SpeedStep® Technology。否则，sys文件夹下 `/sys/devices/system/cpu/cpu0/cpufreq` 将不存在，不能使用基于CPU频率的电源管理。请参阅相关的BIOS文档以确定如何访问这些设置。

例如，在某些Intel参考平台上，开启Enhanced Intel SpeedStep® Technology 的路径为:

```
Advanced
  -> Processor Configuration
  -> Enhanced Intel SpeedStep® Tech
```

此外，C3 和 C6 也应该使能以支持电源管理。C3 和 C6 的配置路径为:

```
Advanced
  -> Processor Configuration
  -> Processor C3 Advanced
  -> Processor Configuration
  -> Processor C6
```

1.5.4 使用 Linux Core 隔离来减少上下文切换

虽然DPDK应用程序使用的线程固定在系统的逻辑核上，但Linux调度程序也可以在这些核上运行其他任务。为了防止在这些核上运行额外的工作负载，可以使用 `isolcpus` Linux 内核参数来将其与通用的Linux调度程序隔离开来。

例如，如果DPDK应用程序要在逻辑核2，4，6上运行，应将以下内容添加到内核参数表中:

```
isolcpus=2,4,6
```

1.5.5 加载 DPDK KNI 内核模块

要运行DPDK Kernel NIC Interface (KNI) 应用程序，需要将一个额外的内核模块(kni模块)加载到内核中。 该模块位于DPDK目录kmod子目录中。与 igb_uio 模块加载类似，(假设当前目录就是DPDK目录):

```
insmod kmod/rte_kni.ko
```

Note: 相关的详细信息，可以参阅 “Kernel NIC Interface Sample Application” 章节和 *DPDK* 示例程序用户指南 。

1.5.6 Linux IOMMU Pass-Through使用Intel® VT-d运行DPDK

要在Linux内核中启用Intel® VT-d，必须配置一系列内核选项，包括:

- IOMMU_SUPPORT
- IOMMU_API
- INTEL_IOMMU

另外，要使用Intel® VT-d运行DPDK，使用 igb_uio 驱动时必须携带 iommu=pt 参数。 这使得主机可以直接通过DMA重映射查找。 另外，如果内核中没有设置 INTEL_IOMMU_DEFAULT_ON 参数，那么也必须使用 intel_iommu=on 参数。这可以确保 Intel IOMMU 被正确初始化。

请注意，对于‘‘igb_uio‘‘ 驱动程序，使用 iommu = pt 是必须de ，vfio-pci 驱动程序实际上可以同时使用 iommu = pt 和 iommu = on 。

1.5.7 40G NIC上的小包处理高性能

由于在最新版本中可能提供用于性能提升的固件修复，因此最好进行固件更新以获取更高的性能。 请和 Intel’s Network Division 工程师联系以进行固件更新。 用户可以参考DPDK版本发行说明，以使用 i40e 驱动程序识别NIC的已验证固件版本。

使用16B大小的RX描述符

由于 i40e PMD 支持16B和32B的RX描述符，而16B大小的描述符可以帮助小型数据包提供性能，因此，配置文件中 CONFIG_RTE_LIBRTE_I40E_16BYTE_RX_DESC 更改为使用16B大小的描述符。

高性能和每数据包延迟权衡

由于硬件设计，每个数据包描述符回写都需要NIC内部的中断信号。中断的最小间隔可以在编译时通过配置文件中的 CONFIG_RTE_LIBRTE_I40E_ITR_INTERVAL 指定。 虽然有默认配置，但是该配置可以由用户自行调整，这取决于用户所关心的内容，整体性能或者每数据包延迟。

1.6 使用脚本快速构建

usertools目录中的dpdk-setup.sh脚本，向用户提供了快速执行如下任务功能:

- 构建DPDK库
- 加载/卸载DPDK IGB_UIO内核模块
- 加载/卸载VFIO内核模块
- 加载/卸载DPDK KNI内核模块
- 创建/删除NUMA 或 non-NUMA平台的hugepages
- 查看网络端口状态和预留给DPDK应用程序使用的端口
- 设置非root用户使用VFIO的权限
- 运行test和testpmd应用程序
- 查看meminfo中的hugepages
- 列出在 `/mnt/huge` 中的hugepages
- 删除内置的DPDK库

对于其中一个EAL目标，一旦完成了这些步骤，用户就可以编译自己的在EAL库中链接的应用程序来创建DPDK映像。

1.6.1 脚本组织

dpdk-setup.sh脚本在逻辑上组织成用户按顺序执行的一系列步骤。每个步骤都提供了许多选项来指导用户完成所需的任务。以下是每个步骤的简单介绍:

Step 1: Build DPDK Libraries

最开始，用户必须指定tagert的类型以便编译正确的库。

如本入门指南前面的章节描述，用户必须在此之前就安装好所有的库、模块、更新和编译器。

Step 2: Setup Environment

用户需要配置Linux* 环境以支持DPDK应用程序的运行。 可以为NUMA 或non-NUMA系统分配Hugepages。 任何原来已经存在的hugepages将被删除。 也可以在此步骤中插入所需的DPDK内核模块，并且可以将网络端口绑定到此模块供DPDK使用。

Step 3: Run an Application

一旦执行了其他步骤，用户就可以运行test程序。该程序允许用户为DPDK运行一系列功能测试。也可以运行支持数据包接收和发送的testpmd程序。

Step 4: Examining the System

此步骤提供了一些用于检查Hugepage映射状态的工具。

Step 5: System Cleanup

最后一步具有将系统恢复到原始状态的选项。

1.6.2 Use Cases

以下是使用dpdk-setup.sh的示例。脚本应该使用source命令运行。脚本中的某些选项在继续操作之前提示用户需要进一步的数据输入。

Warning: 必须与root全选运行dpdk-setup.sh。

```
source usertools/dpdk-setup.sh

------------------------------------------------------------------------

RTE_SDK exported as /home/user/rte

------------------------------------------------------------------------

Step 1: Select the DPDK environment to build

------------------------------------------------------------------------

[1] i686-native-linuxapp-gcc

[2] i686-native-linuxapp-icc

[3] ppc_64-power8-linuxapp-gcc

[4] x86_64-native-bsdapp-clang

[5] x86_64-native-bsdapp-gcc

[6] x86_64-native-linuxapp-clang

[7] x86_64-native-linuxapp-gcc

[8] x86_64-native-linuxapp-icc

------------------------------------------------------------------------

Step 2: Setup linuxapp environment

------------------------------------------------------------------------

[11] Insert IGB UIO module

[12] Insert VFIO module

[13] Insert KNI module

[14] Setup hugepage mappings for non-NUMA systems

[15] Setup hugepage mappings for NUMA systems

[16] Display current Ethernet device settings

[17] Bind Ethernet device to IGB UIO module

[18] Bind Ethernet device to VFIO module
```

```
[19] Setup VFIO permissions

------------------------------------------------------------------------------

Step 3: Run test application for linuxapp environment

------------------------------------------------------------------------------

[20] Run test application ($RTE_TARGET/app/test)

[21] Run testpmd application in interactive mode ($RTE_TARGET/app/testpmd)

------------------------------------------------------------------------------

Step 4: Other tools

------------------------------------------------------------------------------

[22] List hugepage info from /proc/meminfo

------------------------------------------------------------------------------

Step 5: Uninstall and system cleanup

------------------------------------------------------------------------------

[23] Uninstall all targets

[24] Unbind NICs from IGB UIO driver

[25] Remove IGB UIO module

[26] Remove VFIO module

[27] Remove KNI module

[28] Remove hugepage mappings

[29] Exit Script
```

Option:

以下选项演示了 “x86_64-native-linuxapp-gcc“ DPDK库的创建。

```
Option: 9

================== Installing x86_64-native-linuxapp-gcc

Configuration done
== Build lib
...
Build complete
RTE_TARGET exported as x86_64-native-linuxapp-gcc
```

以下选项用于启动DPDK UIO驱动程序。

```
Option: 25

Unloading any existing DPDK UIO module
Loading DPDK UIO module
```

以下选项演示了在NUMA系统中创建hugepage。为每个node分配1024个2MB的页。 应用程序应该使用 -m 4096 来启动，以便访问这两个内存区域。(如果没有 -m 选项，则自动完成)。

Note: 如果显示提示以删除临时文件，请输入'y'。

```
Option: 15

Removing currently reserved hugepages
mounting /mnt/huge and removing directory
Input the number of 2MB pages for each node
Example: to have 128MB of hugepages available per node,
enter '64' to reserve 64 * 2MB pages on each node
Number of pages for node0: 1024
Number of pages for node1: 1024
Reserving hugepages
Creating /mnt/huge and mounting as hugetlbfs
```

以下操作说明了启动测试应用程序以在单个core上运行

```
Option: 20

Enter hex bitmask of cores to execute test app on
Example: to execute app on cores 0 to 7, enter 0xff
bitmask: 0x01
Launching app
EAL: coremask set to 1
EAL: Detected lcore 0 on socket 0
...
EAL: Master core 0 is ready (tid=1b2ad720)
RTE>>
```

1.6.3 应用程序

一旦用户运行和dpdk-setup.sh脚本，构建了目标程序并且设置了hugepages，用户就可以继续构建和运行自己的应用程序或者源码中提供的示例。

/examples 目录中提供的示例程序为了解DPDK提供了很好的起点。 以下命令显示了helloworld应用程序的构建和运行方式。 按照4.2.1节，"应用程序使用的逻辑Core"描述，当选择用于应用程序的coremask时，需要确定平台的逻辑core的布局。

```
cd helloworld/
make
  CC main.o
  LD helloworld
  INSTALL-APP helloworld
  INSTALL-MAP helloworld.map

sudo ./build/app/helloworld -c 0xf -n 3
[sudo] password for rte:
```

```
EAL: coremask set to f
EAL: Detected lcore 0 as core 0 on socket 0
EAL: Detected lcore 1 as core 0 on socket 1
EAL: Detected lcore 2 as core 1 on socket 0
EAL: Detected lcore 3 as core 1 on socket 1
EAL: Setting up hugepage memory...
EAL: Ask a virtual area of 0x200000 bytes
EAL: Virtual area found at 0x7f0add800000 (size = 0x200000)
EAL: Ask a virtual area of 0x3d400000 bytes
EAL: Virtual area found at 0x7f0aa0200000 (size = 0x3d400000)
EAL: Ask a virtual area of 0x400000 bytes
EAL: Virtual area found at 0x7f0a9fc00000 (size = 0x400000)
EAL: Ask a virtual area of 0x400000 bytes
EAL: Virtual area found at 0x7f0a9f600000 (size = 0x400000)
EAL: Ask a virtual area of 0x400000 bytes
EAL: Virtual area found at 0x7f0a9f000000 (size = 0x400000)
EAL: Ask a virtual area of 0x800000 bytes
EAL: Virtual area found at 0x7f0a9e600000 (size = 0x800000)
EAL: Ask a virtual area of 0x800000 bytes
EAL: Virtual area found at 0x7f0a9dc00000 (size = 0x800000)
EAL: Ask a virtual area of 0x400000 bytes
EAL: Virtual area found at 0x7f0a9d600000 (size = 0x400000)
EAL: Ask a virtual area of 0x400000 bytes
EAL: Virtual area found at 0x7f0a9d000000 (size = 0x400000)
EAL: Ask a virtual area of 0x400000 bytes
EAL: Virtual area found at 0x7f0a9ca00000 (size = 0x400000)
EAL: Ask a virtual area of 0x200000 bytes
EAL: Virtual area found at 0x7f0a9c600000 (size = 0x200000)
EAL: Ask a virtual area of 0x200000 bytes
EAL: Virtual area found at 0x7f0a9c200000 (size = 0x200000)
EAL: Ask a virtual area of 0x3fc00000 bytes
EAL: Virtual area found at 0x7f0a5c400000 (size = 0x3fc00000)
EAL: Ask a virtual area of 0x200000 bytes
EAL: Virtual area found at 0x7f0a5c000000 (size = 0x200000)
EAL: Requesting 1024 pages of size 2MB from socket 0
EAL: Requesting 1024 pages of size 2MB from socket 1
EAL: Master core 0 is ready (tid=de25b700)
EAL: Core 1 is ready (tid=5b7fe700)
EAL: Core 3 is ready (tid=5a7fc700)
EAL: Core 2 is ready (tid=5affd700)
hello from core 1
hello from core 2
hello from core 3
hello from core 0
```

1.7 如何获取Intel平台上网卡的最佳性能

本文档一步一步教你如何在Intel平台上运行DPDK程序以获取最佳性能。

1.7.1 硬件及存储需求

为了获得最佳性能，请使用Intel Xeon级服务器系统，如Ivy Bridge，Haswell或更高版本。

确保每个内存通道至少插入一个内存DIMM，每个内存通道的内存大小至少为4GB。 **Note**: 这对性能有最直接的影响。

可以通过使用 `dmidecode` 来检查内存配置:

```
dmidecode -t memory | grep Locator

Locator: DIMM_A1
Bank Locator: NODE 1
Locator: DIMM_A2
Bank Locator: NODE 1
Locator: DIMM_B1
Bank Locator: NODE 1
Locator: DIMM_B2
Bank Locator: NODE 1
...
Locator: DIMM_G1
Bank Locator: NODE 2
Locator: DIMM_G2
Bank Locator: NODE 2
Locator: DIMM_H1
Bank Locator: NODE 2
Locator: DIMM_H2
Bank Locator: NODE 2
```

上面的示例输出显示共有8个通道，从 `A` 到 `H`，每个通道都有2个DIMM。

你也可以使用 `dmidecode` 来确定内存频率:

```
dmidecode -t memory | grep Speed

Speed: 2133 MHz
Configured Clock Speed: 2134 MHz
Speed: Unknown
Configured Clock Speed: Unknown
Speed: 2133 MHz
Configured Clock Speed: 2134 MHz
Speed: Unknown
...
Speed: 2133 MHz
Configured Clock Speed: 2134 MHz
Speed: Unknown
Configured Clock Speed: Unknown
Speed: 2133 MHz
Configured Clock Speed: 2134 MHz
Speed: Unknown
Configured Clock Speed: Unknown
```

输出显示2133 MHz（DDR4）和未知（不存在）的速度。这与先前的输出一致，表明每个通道都有一个存储。

网卡需求

使用 *DPDK supported <http://dpdk.org/doc/nics>* 描述的高端NIC，如Intel XL710 40GbE。

确保每个网卡已经更新最新版本的NVM/固件。

使用PCIe Gen3 插槽，如 Gen3 `x8` 或者 Gen3 `x16` ，因为PCIe Gen2 插槽不能提供2 x 10GbE或更高的带宽。可以使用 `lspci` 命令来检查PCI插槽的速率:

```
lspci -s 03:00.1 -vv | grep LnkSta

LnkSta: Speed 8GT/s, Width x8, TrErr- Train- SlotClk+ DLActive- ...
LnkSta2: Current De-emphasis Level: -6dB, EqualizationComplete+ ...
```

当将NIC插入PCI插槽时，需要查看屏幕输出，如 CPU0 或 CPU1，以指示连接的插槽。

同时应该注意NUMA，如果使用不同网卡的2个或更多端口，最好确保这些NIC在同一个CPU插槽上，下面进一步展示了如何确定这一点。

BIOS 设置

以下是关于BIOS设置的一些建议。不同的平台可能会有不同的名字，因此如下仅用于参考:

1. 开始之前，请考虑将所有BIOS设置为默认值
2. 禁用所有省电选项，如电源性能调整、CPU P-State, CPU C3 Report and CPU C6 Report。
3. 选择 **Performance** 作为CPU电源及性能策略。
4. 禁用Turbo Boost以确保性能缩放随着内核数量的增加而增加。
5. 将内存频率设置为最高可用的值，NOT auto。
6. 当测试NIC的物理功能时，禁用所有的虚拟化选项，如果要使用VFIO，请打开 `VT-d` if you wants to use VFIO.

Linux引导选项

以下是GRUB启动选项的一些建议配置:

1. 使用默认的grub文件作为起点
2. 通过grub配置保留1G的hugepage。例如，保留8个1G大小的页面:

   ```
   default_hugepagesz=1G hugepagesz=1G hugepages=8
   ```

3. 隔离将用于DPDK的CPU core.如:

   ```
   isolcpus=2,3,4,5,6,7,8
   ```

4. 如果要使用VFIO，请使用以下附加的grub参数:

   ```
   iommu=pt intel_iommu=on
   ```

1.7.2 运行DPDK前的配置

1. 构建目标文件，预留hugepage。参阅前面 *在 Linux 环境中使用 Hugepages* 描述。

   以下命令为具体过程:

   ```
   # Build DPDK target.
   cd dpdk_folder
   make install T=x86_64-native-linuxapp-gcc -j

   # Get the hugepage size.
   awk '/Hugepagesize/ {print $2}' /proc/meminfo
   ```

```
# Get the total huge page numbers.
awk '/HugePages_Total/ {print $2} ' /proc/meminfo

# Unmount the hugepages.
umount `awk '/hugetlbfs/ {print $2}' /proc/mounts`

# Create the hugepage mount folder.
mkdir -p /mnt/huge

# Mount to the specific folder.
mount -t hugetlbfs nodev /mnt/huge
```

2. 使用命令 `cpu_layout` 来检查CPU布局:

```
cd dpdk_folder

usertools/cpu_layout.py
```

   或者运行 `lscpu` 检查每个插槽上的core。

3. 检查NIC ID和插槽ID:

```
# 列出所有的网卡的PCI地址及设备ID.
lspci -nn | grep Eth
```

   例如，假设你的输入如下:

```
82:00.0 Ethernet [0200]: Intel XL710 for 40GbE QSFP+ [8086:1583]
82:00.1 Ethernet [0200]: Intel XL710 for 40GbE QSFP+ [8086:1583]
85:00.0 Ethernet [0200]: Intel XL710 for 40GbE QSFP+ [8086:1583]
85:00.1 Ethernet [0200]: Intel XL710 for 40GbE QSFP+ [8086:1583]
```

   检测PCI设备相关联的NUMA节点:

```
cat /sys/bus/pci/devices/0000\:xx\:00.x/numa_node
```

   通常的，`0x:00.x` 表示在插槽0，而 `8x:00.x` 表示在插槽1。 **Note**: 为了说去最佳性能，请保证core和NIC位于同一插槽中。 在上面的例子中 `85:00.0` 在插槽1，因此必须被插槽1上的core使用才能获得最佳性能。

4. 将测试端口绑定到DPDK兼容的驱动程序，如igb_uio。例如，将两个端口绑定到兼容DPDK的驱动程序并检查状态:

```
# 绑定端口 82:00.0 和 85:00.0 到DPDK驱动
./dpdk_folder/usertools/dpdk-devbind.py -b igb_uio 82:00.0 85:00.0

# 检查端口驱动状态
./dpdk_folder/usertools/dpdk-devbind.py --status
```

   运行 `dpdk-devbind.py --help` 以获取更多信息。

有关DPDK设置和Linux内核需求的更多信息，请参阅 使用源码编译*DPDK*目标文件 。

1.7.3 网卡最佳性能实践举例

以下是运行DPDK `l3fwd` 例程并获取最佳性能的例子。使用 Intel 服务平台和Intel XL710 NICs。 具体的40G

NIC配置请参阅i40e NIC指南。

本例场景是通过两个Intel XL710 40GbE端口获取最优性能。请参阅 Fig. 1.1 用于性能测试设置。

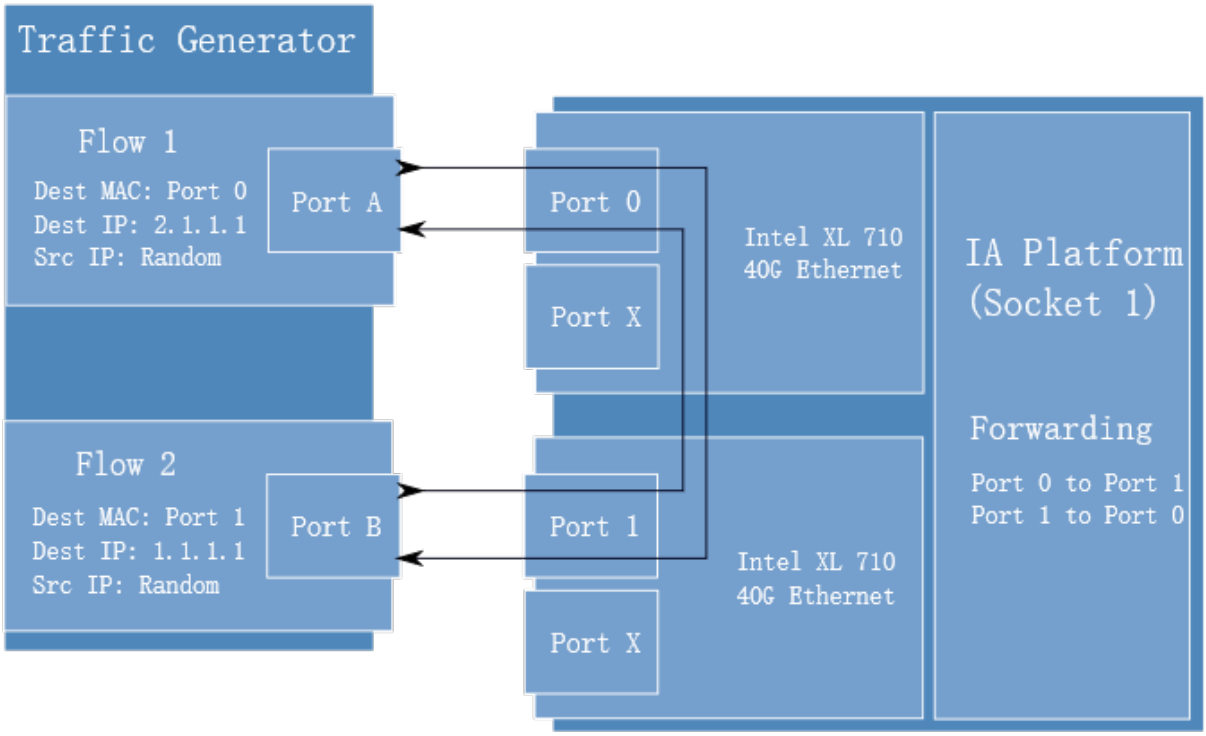


Fig. 1.1: 性能测试搭建

1. 将两个Intel XL710 NIC添加到平台，并使用每个卡一个端口来获得最佳性能。使用两个NIC的原因是克服PCIe Gen3的限制，因为它不能提供80G带宽。 对于两个40G端口，但两个不同的PCIe Gen3 x8插槽可以。 请参考上面的示例NIC输出，然后我们可以选择 `82:00.0` 及 `85:00.0` 作为测试端口:

```
82:00.0 Ethernet [0200]: Intel XL710 for 40GbE QSFP+ [8086:1583]
85:00.0 Ethernet [0200]: Intel XL710 for 40GbE QSFP+ [8086:1583]
```

2. 将端口连接到打流机，对于高速测试，最好有专用的打流设备。
3. 检测PCI设备的numa节点，并获取该插槽id上的core。 在本例中， `82:00.0` 和 `85:00.0` 都在插槽1上，插槽1上的core id为18-35 和 54-71。 Note: 不要在同一个core上使用两个逻辑核(e.g core18 有两个逻辑核core18 and core54)，而是使用来自不同core的两个逻辑核。
4. 将这两个端口绑定到igb_uio。
5. 对于XL710 40G 端口，我们需要至少两个队列来实现最佳性能，因此每个端口需要两个队列，每个队列将需要专用的CPU内核来接收/发送数据包。
6. 使用DPDK示例程序 `l3fwd` 做性能测试，两个端口进行双向转发，使用默认的lpm模式编译 `l3fwd sample`。
7. 运行l3fwd的命令如下所示:

```
./l3fwd -c 0x3c0000 -n 4 -w 82:00.0 -w 85:00.0 \
        -- -p 0x3 --config '(0,0,18),(0,1,19),(1,0,20),(1,1,21)'
```

命令表示应用程序使用（core18，port0，队列0），（core19，port0，队列1），（core20，port1，队列0），（core18，port1，队列1）。

8. 配置打流机用于发包
    - 创建流
    - 设置报文类型为Ethernet II type to 0x0800。

CHAPTER 2

Getting Started Guide for FreeBSD

2.1 Introduction

This document contains instructions for installing and configuring the Data Plane Development Kit (DPDK) software. It is designed to get customers up and running quickly and describes how to compile and run a DPDK application in a FreeBSD application (bsdapp) environment, without going deeply into detail.

For a comprehensive guide to installing and using FreeBSD, the following handbook is available from the FreeBSD Documentation Project: FreeBSD Handbook.

Note: The DPDK is now available as part of the FreeBSD ports collection. Installing via the ports collection infrastructure is now the recommended way to install the DPDK on FreeBSD, and is documented in the next chapter, *Installing DPDK from the Ports Collection*.

2.1.1 Documentation Roadmap

The following is a list of DPDK documents in the suggested reading order:

- **Release Notes** : Provides release-specific information, including supported features, limitations, fixed issues, known issues and so on. Also, provides the answers to frequently asked questions in FAQ format.
- **Getting Started Guide** (this document): Describes how to install and configure the DPDK; designed to get users up and running quickly with the software.
- **Programmer's Guide**: Describes:
  - The software architecture and how to use it (through examples), specifically in a Linux* application (linuxapp) environment
  - The content of the DPDK, the build system (including the commands that can be used in the root DPDK Makefile to build the development kit and an application) and guidelines for porting an application
  - Optimizations used in the software and those that should be considered for new development

A glossary of terms is also provided.

- **API Reference**: Provides detailed information about DPDK functions, data structures and other programming constructs.
- **Sample Applications User Guide**: Describes a set of sample applications. Each chapter describes a sample application that showcases specific functionality and provides instructions on how to compile, run and use the sample application.

2.2 Installing DPDK from the Ports Collection

The easiest way to get up and running with the DPDK on FreeBSD is to install it from the ports collection. Details of getting and using the ports collection are documented in the FreeBSD Handbook.

Note: Testing has been performed using FreeBSD 10.0-RELEASE (x86_64) and requires the installation of the kernel sources, which should be included during the installation of FreeBSD.

2.2.1 Installing the DPDK FreeBSD Port

On a system with the ports collection installed in `/usr/ports`, the DPDK can be installed using the commands:

```
cd /usr/ports/net/dpdk

make install
```

After the installation of the DPDK port, instructions will be printed on how to install the kernel modules required to use the DPDK. A more complete version of these instructions can be found in the sections *Loading the DPDK contigmem Module* and *Loading the DPDK nic_uio Module*. Normally, lines like those below would be added to the file `/boot/loader.conf`.

```
# Reserve 2 x 1G blocks of contiguous memory using contigmem driver:
hw.contigmem.num_buffers=2
hw.contigmem.buffer_size=1073741824
contigmem_load="YES"

# Identify NIC devices for DPDK apps to use and load nic_uio driver:
hw.nic_uio.bdfs="2:0:0,2:0:1"
nic_uio_load="YES"
```

2.2.2 Compiling and Running the Example Applications

When the DPDK has been installed from the ports collection it installs its example applications in `/usr/local/share/dpdk/examples` - also accessible via symlink as `/usr/local/share/examples/dpdk`. These examples can be compiled and run as described in *Compiling and Running Sample Applications*. In this case, the required environmental variables should be set as below:

- `RTE_SDK=/usr/local/share/dpdk`
- `RTE_TARGET=x86_64-native-bsdapp-clang`

Note: To install a copy of the DPDK compiled using gcc, please download the official DPDK package from http://dpdk.org/ and install manually using the instructions given in the next chapter, *Compiling the DPDK Target from Source*

An example application can therefore be copied to a user's home directory and compiled and run as below:

```
export RTE_SDK=/usr/local/share/dpdk

export RTE_TARGET=x86_64-native-bsdapp-clang

cp -r /usr/local/share/dpdk/examples/helloworld .

cd helloworld/

gmake
  CC main.o
  LD helloworld
  INSTALL-APP helloworld
  INSTALL-MAP helloworld.map

sudo ./build/helloworld -l 0-3 -n 2

EAL: Contigmem driver has 2 buffers, each of size 1GB
EAL: Sysctl reports 8 cpus
EAL: Detected lcore 0
EAL: Detected lcore 1
EAL: Detected lcore 2
EAL: Detected lcore 3
EAL: Support maximum 64 logical core(s) by configuration.
EAL: Detected 4 lcore(s)
EAL: Setting up physically contiguous memory...
EAL: Mapped memory segment 1 @ 0x802400000: len 1073741824
EAL: Mapped memory segment 2 @ 0x842400000: len 1073741824
EAL: WARNING: clock_gettime cannot use CLOCK_MONOTONIC_RAW and HPET
     is not available - clock timings may be less accurate.
EAL: TSC frequency is ~3569023 KHz
EAL: PCI scan found 24 devices
EAL: Master core 0 is ready (tid=0x802006400)
EAL: Core 1 is ready (tid=0x802006800)
EAL: Core 3 is ready (tid=0x802007000)
EAL: Core 2 is ready (tid=0x802006c00)
EAL: PCI device 0000:01:00.0 on NUMA socket 0
EAL:   probe driver: 8086:10fb rte_ixgbe_pmd
EAL:   PCI memory mapped at 0x80074a000
EAL:   PCI memory mapped at 0x8007ca000
EAL: PCI device 0000:01:00.1 on NUMA socket 0
EAL:   probe driver: 8086:10fb rte_ixgbe_pmd
EAL:   PCI memory mapped at 0x8007ce000
EAL:   PCI memory mapped at 0x80084e000
EAL: PCI device 0000:02:00.0 on NUMA socket 0
EAL:   probe driver: 8086:10fb rte_ixgbe_pmd
EAL:   PCI memory mapped at 0x800852000
EAL:   PCI memory mapped at 0x8008d2000
EAL: PCI device 0000:02:00.1 on NUMA socket 0
EAL:   probe driver: 8086:10fb rte_ixgbe_pmd
EAL:   PCI memory mapped at 0x801b3f000
EAL:   PCI memory mapped at 0x8008d6000
```

```
hello from core 1
hello from core 2
hello from core 3
hello from core 0
```

Note: To run a DPDK process as a non-root user, adjust the permissions on the `/dev/contigmem` and `/dev/uio device` nodes as described in section *Running DPDK Applications Without Root Privileges*

Note: For an explanation of the command-line parameters that can be passed to an DPDK application, see section *Running a Sample Application*.

2.3 Compiling the DPDK Target from Source

2.3.1 System Requirements

The DPDK and its applications require the GNU make system (gmake) to build on FreeBSD. Optionally, gcc may also be used in place of clang to build the DPDK, in which case it too must be installed prior to compiling the DPDK. The installation of these tools is covered in this section.

Compiling the DPDK requires the FreeBSD kernel sources, which should be included during the installation of FreeBSD on the development platform. The DPDK also requires the use of FreeBSD ports to compile and function.

To use the FreeBSD ports system, it is required to update and extract the FreeBSD ports tree by issuing the following commands:

```
portsnap fetch
portsnap extract
```

If the environment requires proxies for external communication, these can be set using:

```
setenv http_proxy <my_proxy_host>:<port>
setenv ftp_proxy <my_proxy_host>:<port>
```

The FreeBSD ports below need to be installed prior to building the DPDK. In general these can be installed using the following set of commands:

```
cd /usr/ports/<port_location>

make config-recursive

make install

make clean
```

Each port location can be found using:

```
whereis <port_name>
```

The ports required and their locations are as follows:

- dialog4ports: `/usr/ports/ports-mgmt/dialog4ports`

- GNU make(gmake): `/usr/ports/devel/gmake`
- coreutils: `/usr/ports/sysutils/coreutils`

For compiling and using the DPDK with gcc, the compiler must be installed from the ports collection:

- gcc: version 4.9 is recommended `/usr/ports/lang/gcc49`. Ensure that `CPU_OPTS` is selected (default is OFF).

When running the make config-recursive command, a dialog may be presented to the user. For the installation of the DPDK, the default options were used.

Note: To avoid multiple dialogs being presented to the user during make install, it is advisable before running the make install command to re-run the make config-recursive command until no more dialogs are seen.

2.3.2 Install the DPDK and Browse Sources

First, uncompress the archive and move to the DPDK source directory:

```
unzip DPDK-<version>.zip
cd DPDK-<version>
```

The DPDK is composed of several directories:

- lib: Source code of DPDK libraries
- app: Source code of DPDK applications (automatic tests)
- examples: Source code of DPDK applications
- config, buildtools, mk: Framework-related makefiles, scripts and configuration

2.3.3 Installation of the DPDK Target Environments

The format of a DPDK target is:

```
ARCH-MACHINE-EXECENV-TOOLCHAIN
```

Where:

- `ARCH` is: `x86_64`
- `MACHINE` is: `native`
- `EXECENV` is: `bsdapp`
- `TOOLCHAIN` is: `gcc | clang`

The configuration files for the DPDK targets can be found in the DPDK/config directory in the form of:

```
defconfig_ARCH-MACHINE-EXECENV-TOOLCHAIN
```

Note: Configuration files are provided with the `RTE_MACHINE` optimization level set. Within the configuration files, the `RTE_MACHINE` configuration value is set to native, which means that the compiled software is tuned for the platform on which it is built. For more information on this setting, and its possible values, see the *DPDK Programmers Guide*.

To make the target, use `gmake install T=<target>`.

For example to compile for FreeBSD use:

```
gmake install T=x86_64-native-bsdapp-clang
```

Note: If the compiler binary to be used does not correspond to that given in the TOOLCHAIN part of the target, the compiler command may need to be explicitly specified. For example, if compiling for gcc, where the gcc binary is called gcc4.9, the command would need to be `gmake install T=<target> CC=gcc4.9`.

2.3.4 Browsing the Installed DPDK Environment Target

Once a target is created, it contains all the libraries and header files for the DPDK environment that are required to build customer applications. In addition, the test and testpmd applications are built under the build/app directory, which may be used for testing. A kmod directory is also present that contains the kernel modules to install.

2.3.5 Loading the DPDK contigmem Module

To run a DPDK application, physically contiguous memory is required. In the absence of non-transparent superpages, the included sources for the contigmem kernel module provides the ability to present contiguous blocks of memory for the DPDK to use. The contigmem module must be loaded into the running kernel before any DPDK is run. The module is found in the kmod sub-directory of the DPDK target directory.

The amount of physically contiguous memory along with the number of physically contiguous blocks to be reserved by the module can be set at runtime prior to module loading using:

```
kenv hw.contigmem.num_buffers=n
kenv hw.contigmem.buffer_size=m
```

The kernel environment variables can also be specified during boot by placing the following in `/boot/loader.conf`:

```
hw.contigmem.num_buffers=n hw.contigmem.buffer_size=m
```

The variables can be inspected using the following command:

```
sysctl -a hw.contigmem
```

Where n is the number of blocks and m is the size in bytes of each area of contiguous memory. A default of two buffers of size 1073741824 bytes (1 Gigabyte) each is set during module load if they are not specified in the environment.

The module can then be loaded using kldload (assuming that the current directory is the DPDK target directory):

```
kldload ./kmod/contigmem.ko
```

It is advisable to include the loading of the contigmem module during the boot process to avoid issues with potential memory fragmentation during later system up time. This can be achieved by copying the module to the `/boot/kernel/` directory and placing the following into `/boot/loader.conf`:

```
contigmem_load="YES"
```

Note: The contigmem_load directive should be placed after any definitions of `hw.contigmem.num_buffers` and `hw.contigmem.buffer_size` if the default values are not to be used.

An error such as:

```
kldload: can't load ./x86_64-native-bsdapp-gcc/kmod/contigmem.ko:
         Exec format error
```

is generally attributed to not having enough contiguous memory available and can be verified via dmesg or `/var/log/messages`:

```
kernel: contigmalloc failed for buffer <n>
```

To avoid this error, reduce the number of buffers or the buffer size.

2.3.6 Loading the DPDK nic_uio Module

After loading the contigmem module, the `nic_uio` module must also be loaded into the running kernel prior to running any DPDK application. This module must be loaded using the kldload command as shown below (assuming that the current directory is the DPDK target directory).

```
kldload ./kmod/nic_uio.ko
```

Note: If the ports to be used are currently bound to a existing kernel driver then the `hw.nic_uio.bdfs sysctl` value will need to be set before loading the module. Setting this value is described in the next section below.

Currently loaded modules can be seen by using the `kldstat` command and a module can be removed from the running kernel by using `kldunload <module_name>`.

To load the module during boot, copy the `nic_uio` module to `/boot/kernel` and place the following into `/boot/loader.conf`:

```
nic_uio_load="YES"
```

Note: `nic_uio_load="YES"` must appear after the contigmem_load directive, if it exists.

By default, the `nic_uio` module will take ownership of network ports if they are recognized DPDK devices and are not owned by another module. However, since the FreeBSD kernel includes support, either built-in, or via a separate driver module, for most network card devices, it is likely that the ports to be used are already bound to a driver other than `nic_uio`. The following sub-section describe how to query and modify the device ownership of the ports to be used by DPDK applications.

Binding Network Ports to the nic_uio Module

Device ownership can be viewed using the pciconf -l command. The example below shows four Intel® 82599 network ports under `if_ixgbe` module ownership.

```
pciconf -l
ix0@pci0:1:0:0: class=0x020000 card=0x00038086 chip=0x10fb8086 rev=0x01 hdr=0x00
ix1@pci0:1:0:1: class=0x020000 card=0x00038086 chip=0x10fb8086 rev=0x01 hdr=0x00
```

```
ix2@pci0:2:0:0: class=0x020000 card=0x00038086 chip=0x10fb8086 rev=0x01 hdr=0x00
ix3@pci0:2:0:1: class=0x020000 card=0x00038086 chip=0x10fb8086 rev=0x01 hdr=0x00
```

The first column constitutes three components:

1. Device name: `ixN`
2. Unit name: `pci0`
3. Selector (Bus:Device:Function): `1:0:0`

Where no driver is associated with a device, the device name will be `none`.

By default, the FreeBSD kernel will include built-in drivers for the most common devices; a kernel rebuild would normally be required to either remove the drivers or configure them as loadable modules.

To avoid building a custom kernel, the `nic_uio` module can detach a network port from its current device driver. This is achieved by setting the `hw.nic_uio.bdfs` kernel environment variable prior to loading `nic_uio`, as follows:

```
hw.nic_uio.bdfs="b:d:f,b:d:f,..."
```

Where a comma separated list of selectors is set, the list must not contain any whitespace.

For example to re-bind `ix2@pci0:2:0:0` and `ix3@pci0:2:0:1` to the `nic_uio` module upon loading, use the following command:

```
kenv hw.nic_uio.bdfs="2:0:0,2:0:1"
```

The variable can also be specified during boot by placing the following into `/boot/loader.conf`, before the previously-described `nic_uio_load` line - as shown:

```
hw.nic_uio.bdfs="2:0:0,2:0:1"
nic_uio_load="YES"
```

Binding Network Ports Back to their Original Kernel Driver

If the original driver for a network port has been compiled into the kernel, it is necessary to reboot FreeBSD to restore the original device binding. Before doing so, update or remove the `hw.nic_uio.bdfs` in `/boot/loader.conf`.

If rebinding to a driver that is a loadable module, the network port binding can be reset without rebooting. To do so, unload both the target kernel module and the `nic_uio` module, modify or clear the `hw.nic_uio.bdfs` kernel environment (kenv) value, and reload the two drivers - first the original kernel driver, and then the `nic_uio driver`. Note: the latter does not need to be reloaded unless there are ports that are still to be bound to it.

Example commands to perform these steps are shown below:

```
kldunload nic_uio
kldunload <original_driver>

# To clear the value completely:
kenv -u hw.nic_uio.bdfs

# To update the list of ports to bind:
kenv hw.nic_uio.bdfs="b:d:f,b:d:f,..."

kldload <original_driver>

kldload nic_uio  # optional
```

2.4 Compiling and Running Sample Applications

The chapter describes how to compile and run applications in a DPDK environment. It also provides a pointer to where sample applications are stored.

2.4.1 Compiling a Sample Application

Once a DPDK target environment directory has been created (such as `x86_64-native-bsdapp-clang`), it contains all libraries and header files required to build an application.

When compiling an application in the FreeBSD environment on the DPDK, the following variables must be exported:

- `RTE_SDK` - Points to the DPDK installation directory.
- `RTE_TARGET` - Points to the DPDK target environment directory. For FreeBSD, this is the `x86_64-native-bsdapp-clang` or `x86_64-native-bsdapp-gcc` directory.

The following is an example of creating the `helloworld` application, which runs in the DPDK FreeBSD environment. While the example demonstrates compiling using gcc version 4.9, compiling with clang will be similar, except that the `CC=` parameter can probably be omitted. The `helloworld` example may be found in the `${RTE_SDK}/examples` directory.

The directory contains the `main.c` file. This file, when combined with the libraries in the DPDK target environment, calls the various functions to initialize the DPDK environment, then launches an entry point (dispatch application) for each core to be utilized. By default, the binary is generated in the build directory.

```
setenv RTE_SDK /home/user/DPDK
cd $(RTE_SDK)
cd examples/helloworld/
setenv RTE_SDK $HOME/DPDK
setenv RTE_TARGET x86_64-native-bsdapp-gcc

gmake CC=gcc49
  CC main.o
  LD helloworld
  INSTALL-APP helloworld
  INSTALL-MAP helloworld.map

ls build/app
  helloworld helloworld.map
```

Note: In the above example, `helloworld` was in the directory structure of the DPDK. However, it could have been located outside the directory structure to keep the DPDK structure intact. In the following case, the `helloworld` application is copied to a new directory as a new starting point.

```
setenv RTE_SDK /home/user/DPDK
cp -r $(RTE_SDK)/examples/helloworld my_rte_app
cd my_rte_app/
setenv RTE_TARGET x86_64-native-bsdapp-gcc

gmake CC=gcc49
  CC main.o
  LD helloworld
  INSTALL-APP helloworld
  INSTALL-MAP helloworld.map
```

2.4.2 Running a Sample Application

1. The `contigmem` and `nic_uio` modules must be set up prior to running an application.
2. Any ports to be used by the application must be already bound to the `nic_uio` module, as described in section *Binding Network Ports to the nic_uio Module*, prior to running the application. The application is linked with the DPDK target environment's Environment Abstraction Layer (EAL) library, which provides some options that are generic to every DPDK application.

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

```
./rte-app -l CORELIST [-n NUM] [-b <domain:bus:devid.func>] \
          [-r NUM] [-v] [--proc-type <primary|secondary|auto>]
```

Note: EAL has a common interface between all operating systems and is based on the Linux notation for PCI devices. For example, a FreeBSD device selector of `pci0:2:0:1` is referred to as `02:00.1` in EAL.

The EAL options for FreeBSD are as follows:

- `-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 list of cores to use instead of a core mask.
- `-n NUM`: Number of memory channels per processor socket.
- `-b <domain:bus:devid.func>`: Blacklisting of ports; prevent EAL from using specified PCI device (multiple `-b` options are allowed).
- `--use-device`: Use the specified Ethernet device(s) only. Use comma-separate `[domain:]bus:devid.func` values. Cannot be used with `-b` option.
- `-r NUM`: Number of memory ranks.
- `-v`: Display version information on startup.
- `--proc-type`: The type of process instance.

Other options, specific to Linux and are not supported under FreeBSD are as follows:

- `socket-mem`: Memory to allocate from hugepages on specific sockets.
- `--huge-dir`: The directory where hugetlbfs is mounted.
- `--file-prefix`: The prefix text used for hugepage filenames.
- `-m MB`: Memory to allocate from hugepages, regardless of processor socket. It is recommended that `--socket-mem` be used instead of this option.

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

Copy the DPDK application binary to your target, then run the application as follows (assuming the platform has four memory channels, and that cores 0-3 are present and are to be used for running the application):

```
./helloworld -l 0-3 -n 4
```

Note: The `--proc-type` and `--file-prefix` EAL options are used for running multiple DPDK processes. See the "Multi-process Sample Application" chapter in the *DPDK Sample Applications User Guide and the DPDK Programmers Guide* for more details.

2.4.3 Running DPDK Applications Without Root Privileges

Although applications using the DPDK use network ports and other hardware resources directly, with a number of small permission adjustments, it is possible to run these applications as a user other than "root". To do so, the ownership, or permissions, on the following file system objects should be adjusted to ensure that the user account being used to run the DPDK application has access to them:

- The userspace-io device files in `/dev`, for example, `/dev/uio0`, `/dev/uio1`, and so on
- The userspace contiguous memory device: `/dev/contigmem`

Note: Please refer to the DPDK Release Notes for supported applications.

CHAPTER 3

Sample Applications User Guides

3.1 Introduction

This document describes the sample applications that are included in the Data Plane Development Kit (DPDK). Each chapter describes a sample application that showcases specific functionality and provides instructions on how to compile, run and use the sample application.

3.1.1 Documentation Roadmap

The following is a list of DPDK documents in suggested reading order:

- **Release Notes** : Provides release-specific information, including supported features, limitations, fixed issues, known issues and so on. Also, provides the answers to frequently asked questions in FAQ format.
- **Getting Started Guides** : Describes how to install and configure the DPDK software for your operating system; designed to get users up and running quickly with the software.
- **Programmer's Guide:** Describes:
  - The software architecture and how to use it (through examples), specifically in a Linux* application (linuxapp) environment.
  - The content of the DPDK, the build system (including the commands that can be used in the root DPDK Makefile to build the development kit and an application) and guidelines for porting an application.
  - Optimizations used in the software and those that should be considered for new development

A glossary of terms is also provided.

- **API Reference** : Provides detailed information about DPDK functions, data structures and other programming constructs.
- **Sample Applications User Guide** : Describes a set of sample applications. Each chapter describes a sample application that showcases specific functionality and provides instructions on how to compile, run and use the sample application.

3.2 Command Line Sample Application

This chapter describes the Command Line sample application that is part of the Data Plane Development Kit (DPDK).

3.2.1 Overview

The Command Line sample application is a simple application that demonstrates the use of the command line interface in the DPDK. This application is a readline-like interface that can be used to debug a DPDK application, in a Linux* application environment.

Note: The rte_cmdline library should not be used in production code since it is not validated to the same standard as other DPDK libraries. See also the "rte_cmdline library should not be used in production code due to limited testing" item in the "Known Issues" section of the Release Notes.

The Command Line sample application supports some of the features of the GNU readline library such as, completion, cut/paste and some other special bindings that make configuration and debug faster and easier.

The application shows how the rte_cmdline application can be extended to handle a list of objects. There are three simple commands:

- add obj_name IP: Add a new object with an IP/IPv6 address associated to it.
- del obj_name: Delete the specified object.
- show obj_name: Show the IP associated with the specified object.

Note: To terminate the application, use **Ctrl-d**.

3.2.2 Compiling the Application

1. Go to example directory:

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}/examples/cmdline
```

2. Set the target (a default target is used if not specified). For example:

```
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

   Refer to the *DPDK Getting Started Guide* for possible RTE_TARGET values.

3. Build the application:

```
make
```

3.2.3 Running the Application

To run the application in linuxapp environment, issue the following command:

```
$ ./build/cmdline -l 0-3 -n 4
```

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

3.2.4 Explanation

The following sections provide some explanation of the code.

EAL Initialization and cmdline Start

The first task is the initialization of the Environment Abstraction Layer (EAL). This is achieved as follows:

```
int main(int argc, char **argv)
{
    ret = rte_eal_init(argc, argv);
    if (ret < 0)
        rte_panic("Cannot init EAL\n");
```

Then, a new command line object is created and started to interact with the user through the console:

```
cl = cmdline_stdin_new(main_ctx, "example> ");
cmdline_interact(cl);
cmdline_stdin_exit(cl);
```

The cmd line_interact() function returns when the user types **Ctrl-d** and in this case, the application exits.

Defining a cmdline Context

A cmdline context is a list of commands that are listed in a NULL-terminated table, for example:

```
cmdline_parse_ctx_t main_ctx[] = {
    (cmdline_parse_inst_t *) &cmd_obj_del_show,
    (cmdline_parse_inst_t *) &cmd_obj_add,
    (cmdline_parse_inst_t *) &cmd_help,
     NULL,
};
```

Each command (of type cmdline_parse_inst_t) is defined statically. It contains a pointer to a callback function that is executed when the command is parsed, an opaque pointer, a help string and a list of tokens in a NULL-terminated table.

The rte_cmdline application provides a list of pre-defined token types:

- String Token: Match a static string, a list of static strings or any string.
- Number Token: Match a number that can be signed or unsigned, from 8-bit to 32-bit.
- IP Address Token: Match an IPv4 or IPv6 address or network.
- Ethernet* Address Token: Match a MAC address.

In this example, a new token type obj_list is defined and implemented in the parse_obj_list.c and parse_obj_list.h files.

For example, the cmd_obj_del_show command is defined as shown below:

```
struct cmd_obj_add_result {
    cmdline_fixed_string_t action;
    cmdline_fixed_string_t name;
```

```
    struct object *obj;
};

static void cmd_obj_del_show_parsed(void *parsed_result, struct cmdline *cl,␣
↪attribute ((unused)) void *data)
{
   /* ... */
}

cmdline_parse_token_string_t cmd_obj_action = TOKEN_STRING_INITIALIZER(struct cmd_obj_
↪del_show_result, action, "show#del");

parse_token_obj_list_t cmd_obj_obj = TOKEN_OBJ_LIST_INITIALIZER(struct cmd_obj_del_
↪show_result, obj, &global_obj_list);

cmdline_parse_inst_t cmd_obj_del_show = {
    .f = cmd_obj_del_show_parsed, /* function to call */
    .data = NULL,  /* 2nd arg of func */
    .help_str = "Show/del an object",
    .tokens = { /* token list, NULL terminated */
        (void *)&cmd_obj_action,
        (void *)&cmd_obj_obj,
         NULL,
    },
};
```

This command is composed of two tokens:

- The first token is a string token that can be show or del.
- The second token is an object that was previously added using the add command in the global_obj_list variable.

Once the command is parsed, the rte_cmdline application fills a cmd_obj_del_show_result structure. A pointer to this structure is given as an argument to the callback function and can be used in the body of this function.

3.3 Ethtool Sample Application

The Ethtool sample application shows an implementation of an ethtool-like API and provides a console environment that allows its use to query and change Ethernet card parameters. The sample is based upon a simple L2 frame reflector.

3.3.1 Compiling the Application

To compile the application:

1. Go to the sample application directory:

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}/examples/ethtool
```

2. Set the target (a default target is used if not specified). For example:

```
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

   See the *DPDK Getting Started Guide* for possible RTE_TARGET values.

3. Build the application:

```
make
```

3.3.2 Running the Application

The application requires an available core for each port, plus one. The only available options are the standard ones for the EAL:

```
./ethtool-app/ethtool-app/${RTE_TARGET}/ethtool [EAL options]
```

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

3.3.3 Using the application

The application is console-driven using the cmdline DPDK interface:

```
EthApp>
```

From this interface the available commands and descriptions of what they do as as follows:

- `drvinfo`: Print driver info
- `eeprom`: Dump EEPROM to file
- `link`: Print port link states
- `macaddr`: Gets/sets MAC address
- `mtu`: Set NIC MTU
- `open`: Open port
- `pause`: Get/set port pause state
- `portstats`: Print port statistics
- `regs`: Dump port register(s) to file
- `ringparam`: Get/set ring parameters
- `rxmode`: Toggle port Rx mode
- `stop`: Stop port
- `validate`: Check that given MAC address is valid unicast address
- `vlan`: Add/remove VLAN id
- `quit`: Exit program

3.3.4 Explanation

The sample program has two parts: A background *packet reflector* that runs on a slave core, and a foreground *Ethtool Shell* that runs on the master core. These are described below.

Packet Reflector

The background packet reflector is intended to demonstrate basic packet processing on NIC ports controlled by the Ethtool shim. Each incoming MAC frame is rewritten so that it is returned to the sender, using the port in question's own MAC address as the source address, and is then sent out on the same port.

Ethtool Shell

The foreground part of the Ethtool sample is a console-based interface that accepts commands as described in *using the application*. Individual call-back functions handle the detail associated with each command, which make use of the functions defined in the *Ethtool interface* to the DPDK functions.

3.3.5 Ethtool interface

The Ethtool interface is built as a separate library, and implements the following functions:

- `rte_ethtool_get_drvinfo()`
- `rte_ethtool_get_regs_len()`
- `rte_ethtool_get_regs()`
- `rte_ethtool_get_link()`
- `rte_ethtool_get_eeprom_len()`
- `rte_ethtool_get_eeprom()`
- `rte_ethtool_set_eeprom()`
- `rte_ethtool_get_pauseparam()`
- `rte_ethtool_set_pauseparam()`
- `rte_ethtool_net_open()`
- `rte_ethtool_net_stop()`
- `rte_ethtool_net_get_mac_addr()`
- `rte_ethtool_net_set_mac_addr()`
- `rte_ethtool_net_validate_addr()`
- `rte_ethtool_net_change_mtu()`
- `rte_ethtool_net_get_stats64()`
- `rte_ethtool_net_vlan_rx_add_vid()`
- `rte_ethtool_net_vlan_rx_kill_vid()`
- `rte_ethtool_net_set_rx_mode()`
- `rte_ethtool_get_ringparam()`
- `rte_ethtool_set_ringparam()`

3.4 Exception Path Sample Application

The Exception Path sample application is a simple example that demonstrates the use of the DPDK to set up an exception path for packets to go through the Linux* kernel. This is done by using virtual TAP network interfaces. These can be read from and written to by the DPDK application and appear to the kernel as a standard network interface.

3.4.1 Overview

The application creates two threads for each NIC port being used. One thread reads from the port and writes the data unmodified to a thread-specific TAP interface. The second thread reads from a TAP interface and writes the data unmodified to the NIC port.

The packet flow through the exception path application is as shown in the following figure.

Fig. 3.1: Packet Flow

To make throughput measurements, kernel bridges must be setup to forward data between the bridges appropriately.

3.4.2 Compiling the Application

1. Go to example directory:

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}/examples/exception_path
```

2. Set the target (a default target will be used if not specified). For example:

```
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

This application is intended as a linuxapp only. See the *DPDK Getting Started Guide* for possible RTE_TARGET values.

1. Build the application:

```
make
```

3.4.3 Running the Application

The application requires a number of command line options:

```
.build/exception_path [EAL options] -- -p PORTMASK -i IN_CORES -o OUT_CORES
```

where:

- -p PORTMASK: A hex bitmask of ports to use
- -i IN_CORES: A hex bitmask of cores which read from NIC
- -o OUT_CORES: A hex bitmask of cores which write to NIC

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

The number of bits set in each bitmask must be the same. The coremask -c or the corelist -l parameter of the EAL options should include IN_CORES and OUT_CORES. The same bit must not be set in IN_CORES and OUT_CORES. The affinities between ports and cores are set beginning with the least significant bit of each mask, that is, the port represented by the lowest bit in PORTMASK is read from by the core represented by the lowest bit in IN_CORES, and written to by the core represented by the lowest bit in OUT_CORES.

For example to run the application with two ports and four cores:

```
./build/exception_path -l 0-3 -n 4 -- -p 3 -i 3 -o c
```

Getting Statistics

While the application is running, statistics on packets sent and received can be displayed by sending the SIGUSR1 signal to the application from another terminal:

```
killall -USR1 exception_path
```

The statistics can be reset by sending a SIGUSR2 signal in a similar way.

3.4.4 Explanation

The following sections provide some explanation of the code.

Initialization

Setup of the mbuf pool, driver and queues is similar to the setup done in the *L2 Forwarding Sample Application (in Real and Virtualized Environments)*. In addition, the TAP interfaces must also be created. A TAP interface is created for each lcore that is being used. The code for creating the TAP interface is as follows:

```
/*
 *   Create a tap network interface, or use existing one with same name.
 *   If name[0]='\0' then a name is automatically assigned and returned in name.
 */

static int tap_create(char *name)
{
    struct ifreq ifr;
    int fd, ret;

    fd = open("/dev/net/tun", O_RDWR);
    if (fd < 0)
        return fd;

    memset(&ifr, 0, sizeof(ifr));

    /* TAP device without packet information */

    ifr.ifr_flags = IFF_TAP | IFF_NO_PI;
    if (name && *name)
        rte_snprinf(ifr.ifr_name, IFNAMSIZ, name);

    ret = ioctl(fd, TUNSETIFF, (void *) &ifr);
```

```
    if (ret < 0) {
        close(fd);
        return ret;

    }

    if (name)
        snprintf(name, IFNAMSIZ, ifr.ifr_name);

    return fd;
}
```

The other step in the initialization process that is unique to this sample application is the association of each port with two cores:

- One core to read from the port and write to a TAP interface
- A second core to read from a TAP interface and write to the port

This is done using an array called port_ids[], which is indexed by the lcore IDs. The population of this array is shown below:

```
 tx_port = 0;
 rx_port = 0;

 RTE_LCORE_FOREACH(i) {
     if (input_cores_mask & (1ULL << i)) {
         /* Skip ports that are not enabled */
         while ((ports_mask & (1 << rx_port)) == 0) {
             rx_port++;
             if (rx_port > (sizeof(ports_mask) * 8))
                 goto fail; /* not enough ports */
         }
         port_ids[i] = rx_port++;
     } else if (output_cores_mask & (1ULL << i)) {
         /* Skip ports that are not enabled */
         while ((ports_mask & (1 << tx_port)) == 0) {
             tx_port++;
             if (tx_port > (sizeof(ports_mask) * 8))
                goto fail; /* not enough ports */
         }
         port_ids[i] = tx_port++;
     }
}
```

Packet Forwarding

After the initialization steps are complete, the main_loop() function is run on each lcore. This function first checks the lcore_id against the user provided input_cores_mask and output_cores_mask to see if this core is reading from or writing to a TAP interface.

For the case that reads from a NIC port, the packet reception is the same as in the L2 Forwarding sample application (see *Receive, Process and Transmit Packets*). The packet transmission is done by calling write() with the file descriptor of the appropriate TAP interface and then explicitly freeing the mbuf back to the pool.

```
/* Loop forever reading from NIC and writing to tap */

for (;;) {
    struct rte_mbuf *pkts_burst[PKT_BURST_SZ];
    unsigned i;

    const unsigned nb_rx = rte_eth_rx_burst(port_ids[lcore_id], 0, pkts_burst, PKT_
↪BURST_SZ);

    lcore_stats[lcore_id].rx += nb_rx;

    for (i = 0; likely(i < nb_rx); i++) {
        struct rte_mbuf *m = pkts_burst[i];
        int ret = write(tap_fd, rte_pktmbuf_mtod(m, void*),

        rte_pktmbuf_data_len(m));
        rte_pktmbuf_free(m);
        if (unlikely(ret<0))
            lcore_stats[lcore_id].dropped++;
        else
            lcore_stats[lcore_id].tx++;
    }
}
```

For the other case that reads from a TAP interface and writes to a NIC port, packets are retrieved by doing a read() from the file descriptor of the appropriate TAP interface. This fills in the data into the mbuf, then other fields are set manually. The packet can then be transmitted as normal.

```
/* Loop forever reading from tap and writing to NIC */

for (;;) {
    int ret;
    struct rte_mbuf *m = rte_pktmbuf_alloc(pktmbuf_pool);

    if (m == NULL)
        continue;

    ret = read(tap_fd, m->pkt.data, MAX_PACKET_SZ); lcore_stats[lcore_id].rx++;
    if (unlikely(ret < 0)) {
        FATAL_ERROR("Reading from %s interface failed", tap_name);
    }

    m->pkt.nb_segs = 1;
    m->pkt.next = NULL;
    m->pkt.data_len = (uint16_t)ret;

    ret = rte_eth_tx_burst(port_ids[lcore_id], 0, &m, 1);
    if (unlikely(ret < 1)) {
        rte_pktmuf_free(m);
        lcore_stats[lcore_id].dropped++;
    }
    else {
        lcore_stats[lcore_id].tx++;
    }
}
```

To set up loops for measuring throughput, TAP interfaces can be connected using bridging. The steps to do this are described in the section that follows.

Managing TAP Interfaces and Bridges

The Exception Path sample application creates TAP interfaces with names of the format tap_dpdk_nn, where nn is the lcore ID. These TAP interfaces need to be configured for use:

```
ifconfig tap_dpdk_00 up
```

To set up a bridge between two interfaces so that packets sent to one interface can be read from another, use the brctl tool:

```
brctl addbr "br0"
brctl addif br0 tap_dpdk_00
brctl addif br0 tap_dpdk_03
ifconfig br0 up
```

The TAP interfaces created by this application exist only when the application is running, so the steps above need to be repeated each time the application is run. To avoid this, persistent TAP interfaces can be created using openvpn:

```
openvpn --mktun --dev tap_dpdk_00
```

If this method is used, then the steps above have to be done only once and the same TAP interfaces can be reused each time the application is run. To remove bridges and persistent TAP interfaces, the following commands are used:

```
ifconfig br0 down
brctl delbr br0
openvpn --rmtun --dev tap_dpdk_00
```

3.5 Hello World Sample Application

The Hello World sample application is an example of the simplest DPDK application that can be written. The application simply prints an "helloworld" message on every enabled lcore.

3.5.1 Compiling the Application

1. Go to the example directory:

   ```
   export RTE_SDK=/path/to/rte_sdk
   cd ${RTE_SDK}/examples/helloworld
   ```

2. Set the target (a default target is used if not specified). For example:

   ```
   export RTE_TARGET=x86_64-native-linuxapp-gcc
   ```

   See the *DPDK Getting Started* Guide for possible RTE_TARGET values.

3. Build the application:

   ```
   make
   ```

3.5.2 Running the Application

To run the example in a linuxapp environment:

```
$ ./build/helloworld -l 0-3 -n 4
```

Refer to *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

3.5.3 Explanation

The following sections provide some explanation of code.

EAL Initialization

The first task is to initialize the Environment Abstraction Layer (EAL). This is done in the main() function using the following code:

```
int

main(int argc, char **argv)

{
    ret = rte_eal_init(argc, argv);
    if (ret < 0)
        rte_panic("Cannot init EAL\n");
```

This call finishes the initialization process that was started before main() is called (in case of a Linuxapp environment). The argc and argv arguments are provided to the rte_eal_init() function. The value returned is the number of parsed arguments.

Starting Application Unit Lcores

Once the EAL is initialized, the application is ready to launch a function on an lcore. In this example, lcore_hello() is called on every available lcore. The following is the definition of the function:

```
static int
lcore_hello( attribute ((unused)) void *arg)
{
    unsigned lcore_id;

    lcore_id = rte_lcore_id();
    printf("hello from core %u\n", lcore_id);
    return 0;
}
```

The code that launches the function on each lcore is as follows:

```
/* call lcore_hello() on every slave lcore */

RTE_LCORE_FOREACH_SLAVE(lcore_id) {
   rte_eal_remote_launch(lcore_hello, NULL, lcore_id);
}

/* call it on master lcore too */

lcore_hello(NULL);
```

The following code is equivalent and simpler:

```
rte_eal_mp_remote_launch(lcore_hello, NULL, CALL_MASTER);
```

Refer to the *DPDK API Reference* for detailed information on the rte_eal_mp_remote_launch() function.

3.6 Basic Forwarding Sample Application

The Basic Forwarding sample application is a simple *skeleton* example of a forwarding application.

It is intended as a demonstration of the basic components of a DPDK forwarding application. For more detailed implementations see the L2 and L3 forwarding sample applications.

3.6.1 Compiling the Application

To compile the application export the path to the DPDK source tree and go to the example directory:

```
export RTE_SDK=/path/to/rte_sdk

cd ${RTE_SDK}/examples/skeleton
```

Set the target, for example:

```
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

See the *DPDK Getting Started* Guide for possible `RTE_TARGET` values.

Build the application as follows:

```
make
```

3.6.2 Running the Application

To run the example in a `linuxapp` environment:

```
./build/basicfwd -l 1 -n 4
```

Refer to *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

3.6.3 Explanation

The following sections provide an explanation of the main components of the code.

All DPDK library functions used in the sample code are prefixed with `rte_` and are explained in detail in the *DPDK API Documentation*.

The Main Function

The `main()` function performs the initialization and calls the execution threads for each lcore.

The first task is to initialize the Environment Abstraction Layer (EAL). The `argc` and `argv` arguments are provided to the `rte_eal_init()` function. The value returned is the number of parsed arguments:

```
int ret = rte_eal_init(argc, argv);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Error with EAL initialization\n");
```

The `main()` also allocates a mempool to hold the mbufs (Message Buffers) used by the application:

```
mbuf_pool = rte_mempool_create("MBUF_POOL",
                               NUM_MBUFS * nb_ports,
                               MBUF_SIZE,
                               MBUF_CACHE_SIZE,
                               sizeof(struct rte_pktmbuf_pool_private),
                               rte_pktmbuf_pool_init, NULL,
                               rte_pktmbuf_init,      NULL,
                               rte_socket_id(),
                               0);
```

Mbufs are the packet buffer structure used by DPDK. They are explained in detail in the "Mbuf Library" section of the *DPDK Programmer's Guide*.

The `main()` function also initializes all the ports using the user defined `port_init()` function which is explained in the next section:

```
for (portid = 0; portid < nb_ports; portid++) {
    if (port_init(portid, mbuf_pool) != 0) {
        rte_exit(EXIT_FAILURE,
                 "Cannot init port %" PRIu8 "\n", portid);
    }
}
```

Once the initialization is complete, the application is ready to launch a function on an lcore. In this example `lcore_main()` is called on a single lcore.

```
lcore_main();
```

The `lcore_main()` function is explained below.

The Port Initialization Function

The main functional part of the port initialization used in the Basic Forwarding application is shown below:

```
static inline int
port_init(uint8_t port, struct rte_mempool *mbuf_pool)
{
    struct rte_eth_conf port_conf = port_conf_default;
    const uint16_t rx_rings = 1, tx_rings = 1;
    struct ether_addr addr;
    int retval;
    uint16_t q;

    if (port >= rte_eth_dev_count())
```

```
        return -1;

    /* Configure the Ethernet device. */
    retval = rte_eth_dev_configure(port, rx_rings, tx_rings, &port_conf);
    if (retval != 0)
        return retval;

    /* Allocate and set up 1 RX queue per Ethernet port. */
    for (q = 0; q < rx_rings; q++) {
        retval = rte_eth_rx_queue_setup(port, q, RX_RING_SIZE,
                rte_eth_dev_socket_id(port), NULL, mbuf_pool);
        if (retval < 0)
            return retval;
    }

    /* Allocate and set up 1 TX queue per Ethernet port. */
    for (q = 0; q < tx_rings; q++) {
        retval = rte_eth_tx_queue_setup(port, q, TX_RING_SIZE,
                rte_eth_dev_socket_id(port), NULL);
        if (retval < 0)
            return retval;
    }

    /* Start the Ethernet port. */
    retval = rte_eth_dev_start(port);
    if (retval < 0)
        return retval;

    /* Enable RX in promiscuous mode for the Ethernet device. */
    rte_eth_promiscuous_enable(port);

    return 0;
}
```

The Ethernet ports are configured with default settings using the `rte_eth_dev_configure()` function and the `port_conf_default` struct:

```
static const struct rte_eth_conf port_conf_default = {
    .rxmode = { .max_rx_pkt_len = ETHER_MAX_LEN }
};
```

For this example the ports are set up with 1 RX and 1 TX queue using the `rte_eth_rx_queue_setup()` and `rte_eth_tx_queue_setup()` functions.

The Ethernet port is then started:

```
retval  = rte_eth_dev_start(port);
```

Finally the RX port is set in promiscuous mode:

```
rte_eth_promiscuous_enable(port);
```

The Lcores Main

As we saw above the `main()` function calls an application function on the available lcores. For the Basic Forwarding application the lcore function looks like the following:

```
static __attribute__((noreturn)) void
lcore_main(void)
{
    const uint8_t nb_ports = rte_eth_dev_count();
    uint8_t port;

    /*
     * Check that the port is on the same NUMA node as the polling thread
     * for best performance.
     */
    for (port = 0; port < nb_ports; port++)
        if (rte_eth_dev_socket_id(port) > 0 &&
                rte_eth_dev_socket_id(port) !=
                        (int)rte_socket_id())
            printf("WARNING, port %u is on remote NUMA node to "
                    "polling thread.\n\tPerformance will "
                    "not be optimal.\n", port);

    printf("\nCore %u forwarding packets. [Ctrl+C to quit]\n",
            rte_lcore_id());

    /* Run until the application is quit or killed. */
    for (;;) {
        /*
         * Receive packets on a port and forward them on the paired
         * port. The mapping is 0 -> 1, 1 -> 0, 2 -> 3, 3 -> 2, etc.
         */
        for (port = 0; port < nb_ports; port++) {

            /* Get burst of RX packets, from first port of pair. */
            struct rte_mbuf *bufs[BURST_SIZE];
            const uint16_t nb_rx = rte_eth_rx_burst(port, 0,
                    bufs, BURST_SIZE);

            if (unlikely(nb_rx == 0))
                continue;

            /* Send burst of TX packets, to second port of pair. */
            const uint16_t nb_tx = rte_eth_tx_burst(port ^ 1, 0,
                    bufs, nb_rx);

            /* Free any unsent packets. */
            if (unlikely(nb_tx < nb_rx)) {
                uint16_t buf;
                for (buf = nb_tx; buf < nb_rx; buf++)
                    rte_pktmbuf_free(bufs[buf]);
            }
        }
    }
}
```

The main work of the application is done within the loop:

```
for (;;) {
    for (port = 0; port < nb_ports; port++) {

        /* Get burst of RX packets, from first port of pair. */
        struct rte_mbuf *bufs[BURST_SIZE];
```

```
        const uint16_t nb_rx = rte_eth_rx_burst(port, 0,
                bufs, BURST_SIZE);

        if (unlikely(nb_rx == 0))
            continue;

        /* Send burst of TX packets, to second port of pair. */
        const uint16_t nb_tx = rte_eth_tx_burst(port ^ 1, 0,
                bufs, nb_rx);

        /* Free any unsent packets. */
        if (unlikely(nb_tx < nb_rx)) {
            uint16_t buf;
            for (buf = nb_tx; buf < nb_rx; buf++)
                rte_pktmbuf_free(bufs[buf]);
        }
    }
}
```

Packets are received in bursts on the RX ports and transmitted in bursts on the TX ports. The ports are grouped in pairs with a simple mapping scheme using the an XOR on the port number:

```
0 -> 1
1 -> 0

2 -> 3
3 -> 2

etc.
```

The `rte_eth_tx_burst()` function frees the memory buffers of packets that are transmitted. If packets fail to transmit, `(nb_tx < nb_rx)`, then they must be freed explicitly using `rte_pktmbuf_free()`.

The forwarding loop can be interrupted and the application closed using `Ctrl-C`.

3.7 RX/TX Callbacks Sample Application

The RX/TX Callbacks sample application is a packet forwarding application that demonstrates the use of user defined callbacks on received and transmitted packets. The application performs a simple latency check, using callbacks, to determine the time packets spend within the application.

In the sample application a user defined callback is applied to all received packets to add a timestamp. A separate callback is applied to all packets prior to transmission to calculate the elapsed time, in CPU cycles.

3.7.1 Compiling the Application

To compile the application export the path to the DPDK source tree and go to the example directory:

```
export RTE_SDK=/path/to/rte_sdk

cd ${RTE_SDK}/examples/rxtx_callbacks
```

Set the target, for example:

```
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

See the *DPDK Getting Started* Guide for possible `RTE_TARGET` values.

The callbacks feature requires that the `CONFIG_RTE_ETHDEV_RXTX_CALLBACKS` setting is on in the `config/common_` config file that applies to the target. This is generally on by default:

```
CONFIG_RTE_ETHDEV_RXTX_CALLBACKS=y
```

Build the application as follows:

```
make
```

3.7.2 Running the Application

To run the example in a `linuxapp` environment:

```
./build/rxtx_callbacks -l 1 -n 4
```

Refer to *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

3.7.3 Explanation

The `rxtx_callbacks` application is mainly a simple forwarding application based on the *Basic Forwarding Sample Application*. See that section of the documentation for more details of the forwarding part of the application.

The sections below explain the additional RX/TX callback code.

The Main Function

The `main()` function performs the application initialization and calls the execution threads for each lcore. This function is effectively identical to the `main()` function explained in *Basic Forwarding Sample Application*.

The `lcore_main()` function is also identical.

The main difference is in the user defined `port_init()` function where the callbacks are added. This is explained in the next section:

The Port Initialization Function

The main functional part of the port initialization is shown below with comments:

```
static inline int
port_init(uint8_t port, struct rte_mempool *mbuf_pool)
{
    struct rte_eth_conf port_conf = port_conf_default;
    const uint16_t rx_rings = 1, tx_rings = 1;
    struct ether_addr addr;
    int retval;
    uint16_t q;

    if (port >= rte_eth_dev_count())
        return -1;
```

```
    /* Configure the Ethernet device. */
    retval = rte_eth_dev_configure(port, rx_rings, tx_rings, &port_conf);
    if (retval != 0)
        return retval;

    /* Allocate and set up 1 RX queue per Ethernet port. */
    for (q = 0; q < rx_rings; q++) {
        retval = rte_eth_rx_queue_setup(port, q, RX_RING_SIZE,
                rte_eth_dev_socket_id(port), NULL, mbuf_pool);
        if (retval < 0)
            return retval;
    }

    /* Allocate and set up 1 TX queue per Ethernet port. */
    for (q = 0; q < tx_rings; q++) {
        retval = rte_eth_tx_queue_setup(port, q, TX_RING_SIZE,
                rte_eth_dev_socket_id(port), NULL);
        if (retval < 0)
            return retval;
    }

    /* Start the Ethernet port. */
    retval = rte_eth_dev_start(port);
    if (retval < 0)
        return retval;

    /* Enable RX in promiscuous mode for the Ethernet device. */
    rte_eth_promiscuous_enable(port);


    /* Add the callbacks for RX and TX.*/
    rte_eth_add_rx_callback(port, 0, add_timestamps, NULL);
    rte_eth_add_tx_callback(port, 0, calc_latency, NULL);

    return 0;
}
```

The RX and TX callbacks are added to the ports/queues as function pointers:

```
rte_eth_add_rx_callback(port, 0, add_timestamps, NULL);
rte_eth_add_tx_callback(port, 0, calc_latency,   NULL);
```

More than one callback can be added and additional information can be passed to callback function pointers as a `void*`. In the examples above `NULL` is used.

The `add_timestamps()` and `calc_latency()` functions are explained below.

The add_timestamps() Callback

The `add_timestamps()` callback is added to the RX port and is applied to all packets received:

```
static uint16_t
add_timestamps(uint8_t port __rte_unused, uint16_t qidx __rte_unused,
        struct rte_mbuf **pkts, uint16_t nb_pkts, void *_ __rte_unused)
{
    unsigned i;
```

```
    uint64_t now = rte_rdtsc();

    for (i = 0; i < nb_pkts; i++)
        pkts[i]->udata64 = now;

    return nb_pkts;
}
```

The DPDK function `rte_rdtsc()` is used to add a cycle count timestamp to each packet (see the *cycles* section of the *DPDK API Documentation* for details).

The calc_latency() Callback

The `calc_latency()` callback is added to the TX port and is applied to all packets prior to transmission:

```
static uint16_t
calc_latency(uint8_t port __rte_unused, uint16_t qidx __rte_unused,
        struct rte_mbuf **pkts, uint16_t nb_pkts, void *_ __rte_unused)
{
    uint64_t cycles = 0;
    uint64_t now = rte_rdtsc();
    unsigned i;

    for (i = 0; i < nb_pkts; i++)
        cycles += now - pkts[i]->udata64;

    latency_numbers.total_cycles += cycles;
    latency_numbers.total_pkts   += nb_pkts;

    if (latency_numbers.total_pkts > (100 * 1000 * 1000ULL)) {
        printf("Latency = %"PRIu64" cycles\n",
                latency_numbers.total_cycles / latency_numbers.total_pkts);

        latency_numbers.total_cycles = latency_numbers.total_pkts = 0;
    }

    return nb_pkts;
}
```

The `calc_latency()` function accumulates the total number of packets and the total number of cycles used. Once more than 100 million packets have been transmitted the average cycle count per packet is printed out and the counters are reset.

3.8 IP Fragmentation Sample Application

The IPv4 Fragmentation application is a simple example of packet processing using the Data Plane Development Kit (DPDK). The application does L3 forwarding with IPv4 and IPv6 packet fragmentation.

3.8.1 Overview

The application demonstrates the use of zero-copy buffers for packet fragmentation. The initialization and run-time paths are very similar to those of the *L2 Forwarding Sample Application (in Real and Virtualized Environments)*. This guide highlights the differences between the two applications.

There are three key differences from the L2 Forwarding sample application:

- The first difference is that the IP Fragmentation sample application makes use of indirect buffers.
- The second difference is that the forwarding decision is taken based on information read from the input packet's IP header.
- The third difference is that the application differentiates between IP and non-IP traffic by means of offload flags.

The Longest Prefix Match (LPM for IPv4, LPM6 for IPv6) table is used to store/lookup an outgoing port number, associated with that IP address. Any unmatched packets are forwarded to the originating port.

By default, input frame sizes up to 9.5 KB are supported. Before forwarding, the input IP packet is fragmented to fit into the "standard" Ethernet* v2 MTU (1500 bytes).

3.8.2 Building the Application

To build the application:

1. Go to the sample application directory:

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}/examples/ip_fragmentation
```

2. Set the target (a default target is used if not specified). For example:

```
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

See the *DPDK Getting Started Guide* for possible RTE_TARGET values.

1. Build the application:

```
make
```

3.8.3 Running the Application

The LPM object is created and loaded with the pre-configured entries read from global l3fwd_ipv4_route_array and l3fwd_ipv6_route_array tables. For each input packet, the packet forwarding decision (that is, the identification of the output interface for the packet) is taken as a result of LPM lookup. If the IP packet size is greater than default output MTU, then the input packet is fragmented and several fragments are sent via the output interface.

Application usage:

```
./build/ip_fragmentation [EAL options] -- -p PORTMASK [-q NQ]
```

where:

- -p PORTMASK is a hexadecimal bitmask of ports to configure
- -q NQ is the number of queue (=ports) per lcore (the default is 1)

To run the example in linuxapp environment with 2 lcores (2,4) over 2 ports(0,2) with 1 RX queue per lcore:

```
./build/ip_fragmentation -l 2,4 -n 3 -- -p 5
EAL: coremask set to 14
EAL: Detected lcore 0 on socket 0
EAL: Detected lcore 1 on socket 1
EAL: Detected lcore 2 on socket 0
EAL: Detected lcore 3 on socket 1
```

```
EAL: Detected lcore 4 on socket 0
...

Initializing port 0 on lcore 2... Address:00:1B:21:76:FA:2C, rxq=0 txq=2,0 txq=4,1
done: Link Up - speed 10000 Mbps - full-duplex
Skipping disabled port 1
Initializing port 2 on lcore 4... Address:00:1B:21:5C:FF:54, rxq=0 txq=2,0 txq=4,1
done: Link Up - speed 10000 Mbps - full-duplex
Skipping disabled port 3IP_FRAG: Socket 0: adding route 100.10.0.0/16 (port 0)
IP_FRAG: Socket 0: adding route 100.20.0.0/16 (port 1)
...
IP_FRAG: Socket 0: adding route 0101:0101:0101:0101:0101:0101:0101:0101/48 (port 0)
IP_FRAG: Socket 0: adding route 0201:0101:0101:0101:0101:0101:0101:0101/48 (port 1)
...
IP_FRAG: entering main loop on lcore 4
IP_FRAG: -- lcoreid=4 portid=2
IP_FRAG: entering main loop on lcore 2
IP_FRAG: -- lcoreid=2 portid=0
```

To run the example in linuxapp environment with 1 lcore (4) over 2 ports(0,2) with 2 RX queues per lcore:

```
./build/ip_fragmentation -l 4 -n 3 -- -p 5 -q 2
```

To test the application, flows should be set up in the flow generator that match the values in the l3fwd_ipv4_route_array and/or l3fwd_ipv6_route_array table.

The default l3fwd_ipv4_route_array table is:

```
struct l3fwd_ipv4_route l3fwd_ipv4_route_array[] = {
    {IPv4(100, 10, 0, 0), 16, 0},
    {IPv4(100, 20, 0, 0), 16, 1},
    {IPv4(100, 30, 0, 0), 16, 2},
    {IPv4(100, 40, 0, 0), 16, 3},
    {IPv4(100, 50, 0, 0), 16, 4},
    {IPv4(100, 60, 0, 0), 16, 5},
    {IPv4(100, 70, 0, 0), 16, 6},
    {IPv4(100, 80, 0, 0), 16, 7},
};
```

The default l3fwd_ipv6_route_array table is:

```
struct l3fwd_ipv6_route l3fwd_ipv6_route_array[] = {
    {{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 0},
    {{2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 1},
    {{3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 2},
    {{4, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 3},
    {{5, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 4},
    {{6, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 5},
    {{7, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 6},
    {{8, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 7},
};
```

For example, for the input IPv4 packet with destination address: 100.10.1.1 and packet length 9198 bytes, seven IPv4 packets will be sent out from port #0 to the destination address 100.10.1.1: six of those packets will have length 1500 bytes and one packet will have length 318 bytes. IP Fragmentation sample application provides basic NUMA support in that all the memory structures are allocated on all sockets that have active lcores on them.

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Ab-

straction Layer (EAL) options.

3.9 IPv4 Multicast Sample Application

The IPv4 Multicast application is a simple example of packet processing using the Data Plane Development Kit (DPDK). The application performs L3 multicasting.

3.9.1 Overview

The application demonstrates the use of zero-copy buffers for packet forwarding. The initialization and run-time paths are very similar to those of the *L2 Forwarding Sample Application (in Real and Virtualized Environments)*. This guide highlights the differences between the two applications. There are two key differences from the L2 Forwarding sample application:

- The IPv4 Multicast sample application makes use of indirect buffers.
- The forwarding decision is taken based on information read from the input packet's IPv4 header.

The lookup method is the Four-byte Key (FBK) hash-based method. The lookup table is composed of pairs of destination IPv4 address (the FBK) and a port mask associated with that IPv4 address.

For convenience and simplicity, this sample application does not take IANA-assigned multicast addresses into account, but instead equates the last four bytes of the multicast group (that is, the last four bytes of the destination IP address) with the mask of ports to multicast packets to. Also, the application does not consider the Ethernet addresses; it looks only at the IPv4 destination address for any given packet.

3.9.2 Building the Application

To compile the application:

1. Go to the sample application directory:

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}/examples/ipv4_multicast
```

2. Set the target (a default target is used if not specified). For example:

```
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

See the *DPDK Getting Started Guide* for possible RTE_TARGET values.

1. Build the application:

```
make
```

Note: The compiled application is written to the build subdirectory. To have the application written to a different location, the O=/path/to/build/directory option may be specified in the make command.

3.9.3 Running the Application

The application has a number of command line options:

```
./build/ipv4_multicast [EAL options] -- -p PORTMASK [-q NQ]
```

where,

- -p PORTMASK: Hexadecimal bitmask of ports to configure
- -q NQ: determines the number of queues per lcore

Note: Unlike the basic L2/L3 Forwarding sample applications, NUMA support is not provided in the IPv4 Multicast sample application.

Typically, to run the IPv4 Multicast sample application, issue the following command (as root):

```
./build/ipv4_multicast -l 0-3 -n 3 -- -p 0x3 -q 1
```

In this command:

- The -c option enables cores 0, 1, 2 and 3
- The -n option specifies 3 memory channels
- The -p option enables ports 0 and 1
- The -q option assigns 1 queue to each lcore

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

3.9.4 Explanation

The following sections provide some explanation of the code. As mentioned in the overview section, the initialization and run-time paths are very similar to those of the *L2 Forwarding Sample Application (in Real and Virtualized Environments)*. The following sections describe aspects that are specific to the IPv4 Multicast sample application.

Memory Pool Initialization

The IPv4 Multicast sample application uses three memory pools. Two of the pools are for indirect buffers used for packet duplication purposes. Memory pools for indirect buffers are initialized differently from the memory pool for direct buffers:

```
packet_pool = rte_pktmbuf_pool_create("packet_pool", NB_PKT_MBUF, 32,
                    0, PKT_MBUF_DATA_SIZE, rte_socket_id());
header_pool = rte_pktmbuf_pool_create("header_pool", NB_HDR_MBUF, 32,
                    0, HDR_MBUF_DATA_SIZE, rte_socket_id());
clone_pool = rte_pktmbuf_pool_create("clone_pool", NB_CLONE_MBUF, 32,
                    0, 0, rte_socket_id());
```

The reason for this is because indirect buffers are not supposed to hold any packet data and therefore can be initialized with lower amount of reserved memory for each buffer.

Hash Initialization

The hash object is created and loaded with the pre-configured entries read from a global array:

```
static int

init_mcast_hash(void)
{
    uint32_t i;
    mcast_hash_params.socket_id = rte_socket_id();

    mcast_hash = rte_fbk_hash_create(&mcast_hash_params);
    if (mcast_hash == NULL){
        return -1;
    }

    for (i = 0; i < N_MCAST_GROUPS; i ++){
        if (rte_fbk_hash_add_key(mcast_hash, mcast_group_table[i].ip, mcast_group_
↪table[i].port_mask) < 0) {
                    return -1;
        }
    }
    return 0;
}
```

Forwarding

All forwarding is done inside the mcast_forward() function. Firstly, the Ethernet* header is removed from the packet and the IPv4 address is extracted from the IPv4 header:

```
/* Remove the Ethernet header from the input packet */

iphdr = (struct ipv4_hdr *)rte_pktmbuf_adj(m, sizeof(struct ether_hdr));
RTE_ASSERT(iphdr != NULL);
dest_addr = rte_be_to_cpu_32(iphdr->dst_addr);
```

Then, the packet is checked to see if it has a multicast destination address and if the routing table has any ports assigned to the destination address:

```
if (!IS_IPV4_MCAST(dest_addr) ||
    (hash = rte_fbk_hash_lookup(mcast_hash, dest_addr)) <= 0 ||
    (port_mask = hash & enabled_port_mask) == 0) {
        rte_pktmbuf_free(m);
        return;
}
```

Then, the number of ports in the destination portmask is calculated with the help of the bitcnt() function:

```
/* Get number of bits set. */

static inline uint32_t bitcnt(uint32_t v)
{
    uint32_t n;

    for (n = 0; v != 0; v &= v - 1, n++)
        ;
    return n;
}
```

This is done to determine which forwarding algorithm to use. This is explained in more detail in the next section.

Thereafter, a destination Ethernet address is constructed:

```
/* construct destination Ethernet address */

dst_eth_addr = ETHER_ADDR_FOR_IPV4_MCAST(dest_addr);
```

Since Ethernet addresses are also part of the multicast process, each outgoing packet carries the same destination Ethernet address. The destination Ethernet address is constructed from the lower 23 bits of the multicast group OR-ed with the Ethernet address 01:00:5e:00:00:00, as per RFC 1112:

```
#define ETHER_ADDR_FOR_IPV4_MCAST(x) \
    (rte_cpu_to_be_64(0x01005e000000ULL | ((x) & 0x7fffff)) >> 16)
```

Then, packets are dispatched to the destination ports according to the portmask associated with a multicast group:

```
for (port = 0; use_clone != port_mask; port_mask >>= 1, port++) {
    /* Prepare output packet and send it out. */

    if ((port_mask & 1) != 0) {
        if (likely ((mc = mcast_out_pkt(m, use_clone)) != NULL))
            mcast_send_pkt(mc, &dst_eth_addr.as_addr, qconf, port);
        else if (use_clone == 0)
             rte_pktmbuf_free(m);
   }
}
```

The actual packet transmission is done in the mcast_send_pkt() function:

```
static inline void mcast_send_pkt(struct rte_mbuf *pkt, struct ether_addr *dest_addr,␣
↪struct lcore_queue_conf *qconf, uint8_t port)
{
    struct ether_hdr *ethdr;
    uint16_t len;

    /* Construct Ethernet header. */

    ethdr = (struct ether_hdr *)rte_pktmbuf_prepend(pkt, (uint16_t) sizeof(*ethdr));

    RTE_ASSERT(ethdr != NULL);

    ether_addr_copy(dest_addr, &ethdr->d_addr);
    ether_addr_copy(&ports_eth_addr[port], &ethdr->s_addr);
    ethdr->ether_type = rte_be_to_cpu_16(ETHER_TYPE_IPv4);

    /* Put new packet into the output queue */

    len = qconf->tx_mbufs[port].len;
    qconf->tx_mbufs[port].m_table[len] = pkt;
    qconf->tx_mbufs[port].len = ++len;

    /* Transmit packets */

    if (unlikely(MAX_PKT_BURST == len))
        send_burst(qconf, port);
}
```

Buffer Cloning

This is the most important part of the application since it demonstrates the use of zero- copy buffer cloning. There are two approaches for creating the outgoing packet and although both are based on the data zero-copy idea, there are some differences in the detail.

The first approach creates a clone of the input packet, for example, walk though all segments of the input packet and for each of segment, create a new buffer and attach that new buffer to the segment (refer to rte_pktmbuf_clone() in the rte_mbuf library for more details). A new buffer is then allocated for the packet header and is prepended to the cloned buffer.

The second approach does not make a clone, it just increments the reference counter for all input packet segment, allocates a new buffer for the packet header and prepends it to the input packet.

Basically, the first approach reuses only the input packet's data, but creates its own copy of packet's metadata. The second approach reuses both input packet's data and metadata.

The advantage of first approach is that each outgoing packet has its own copy of the metadata, so we can safely modify the data pointer of the input packet. That allows us to skip creation if the output packet is for the last destination port and instead modify input packet's header in place. For example, for N destination ports, we need to invoke mcast_out_pkt() (N-1) times.

The advantage of the second approach is that there is less work to be done for each outgoing packet, that is, the "clone" operation is skipped completely. However, there is a price to pay. The input packet's metadata must remain intact, so for N destination ports, we need to invoke mcast_out_pkt() (N) times.

Therefore, for a small number of outgoing ports (and segments in the input packet), first approach is faster. As the number of outgoing ports (and/or input segments) grows, the second approach becomes more preferable.

Depending on the number of segments or the number of ports in the outgoing portmask, either the first (with cloning) or the second (without cloning) approach is taken:

```
use_clone = (port_num <= MCAST_CLONE_PORTS && m->pkt.nb_segs <= MCAST_CLONE_SEGS);
```

It is the mcast_out_pkt() function that performs the packet duplication (either with or without actually cloning the buffers):

```
static inline struct rte_mbuf *mcast_out_pkt(struct rte_mbuf *pkt, int use_clone)
{
    struct rte_mbuf *hdr;

    /* Create new mbuf for the header. */

    if (unlikely ((hdr = rte_pktmbuf_alloc(header_pool)) == NULL))
        return NULL;

    /* If requested, then make a new clone packet. */

    if (use_clone != 0 && unlikely ((pkt = rte_pktmbuf_clone(pkt, clone_pool)) ==␣
↪NULL)) {
        rte_pktmbuf_free(hdr);
        return NULL;
    }

    /* prepend new header */

    hdr->pkt.next = pkt;

    /* update header's fields */
```

```
    hdr->pkt.pkt_len = (uint16_t)(hdr->pkt.data_len + pkt->pkt.pkt_len);
    hdr->pkt.nb_segs = (uint8_t)(pkt->pkt.nb_segs + 1);

    /* copy metadata from source packet */

    hdr->pkt.in_port = pkt->pkt.in_port;
    hdr->pkt.vlan_macip = pkt->pkt.vlan_macip;
    hdr->pkt.hash = pkt->pkt.hash;
    hdr->ol_flags = pkt->ol_flags;
    rte_mbuf_sanity_check(hdr, RTE_MBUF_PKT, 1);

    return hdr;
}
```

3.10 IP Reassembly Sample Application

The L3 Forwarding application is a simple example of packet processing using the DPDK. The application performs L3 forwarding with reassembly for fragmented IPv4 and IPv6 packets.

3.10.1 Overview

The application demonstrates the use of the DPDK libraries to implement packet forwarding with reassembly for IPv4 and IPv6 fragmented packets. The initialization and run- time paths are very similar to those of the *L2 Forwarding Sample Application (in Real and Virtualized Environments)*. The main difference from the L2 Forwarding sample application is that it reassembles fragmented IPv4 and IPv6 packets before forwarding. The maximum allowed size of reassembled packet is 9.5 KB.

There are two key differences from the L2 Forwarding sample application:

- The first difference is that the forwarding decision is taken based on information read from the input packet's IP header.
- The second difference is that the application differentiates between IP and non-IP traffic by means of offload flags.

3.10.2 The Longest Prefix Match (LPM for IPv4, LPM6 for IPv6) table is used to store/lookup an outgoing port number, associated with that IPv4 address. Any unmatched packets are forwarded to the originating port.Compiling the Application

To compile the application:

1. Go to the sample application directory:

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}/examples/ip_reassembly
```

1. Set the target (a default target is used if not specified). For example:

```
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

See the *DPDK Getting Started Guide* for possible RTE_TARGET values.

1. Build the application:

```
make
```

3.10.3 Running the Application

The application has a number of command line options:

```
./build/ip_reassembly [EAL options] -- -p PORTMASK [-q NQ] [--maxflows=FLOWS>] [--
↪flowttl=TTL[(s|ms)]]
```

where:

- -p PORTMASK: Hexadecimal bitmask of ports to configure
- -q NQ: Number of RX queues per lcore
- –maxflows=FLOWS: determines maximum number of active fragmented flows (1-65535). Default value: 4096.
- –flowttl=TTL[(s|ms)]: determines maximum Time To Live for fragmented packet. If all fragments of the packet wouldn't appear within given time-out, then they are considered as invalid and will be dropped. Valid range is 1ms - 3600s. Default value: 1s.

To run the example in linuxapp environment with 2 lcores (2,4) over 2 ports(0,2) with 1 RX queue per lcore:

```
./build/ip_reassembly -l 2,4 -n 3 -- -p 5
EAL: coremask set to 14
EAL: Detected lcore 0 on socket 0
EAL: Detected lcore 1 on socket 1
EAL: Detected lcore 2 on socket 0
EAL: Detected lcore 3 on socket 1
EAL: Detected lcore 4 on socket 0
...

Initializing port 0 on lcore 2... Address:00:1B:21:76:FA:2C, rxq=0 txq=2,0 txq=4,1
done: Link Up - speed 10000 Mbps - full-duplex
Skipping disabled port 1
Initializing port 2 on lcore 4... Address:00:1B:21:5C:FF:54, rxq=0 txq=2,0 txq=4,1
done: Link Up - speed 10000 Mbps - full-duplex
Skipping disabled port 3IP_FRAG: Socket 0: adding route 100.10.0.0/16 (port 0)
IP_RSMBL: Socket 0: adding route 100.20.0.0/16 (port 1)
...

IP_RSMBL: Socket 0: adding route 0101:0101:0101:0101:0101:0101:0101:0101/48 (port 0)
IP_RSMBL: Socket 0: adding route 0201:0101:0101:0101:0101:0101:0101:0101/48 (port 1)
...

IP_RSMBL: entering main loop on lcore 4
IP_RSMBL: -- lcoreid=4 portid=2
IP_RSMBL: entering main loop on lcore 2
IP_RSMBL: -- lcoreid=2 portid=0
```

To run the example in linuxapp environment with 1 lcore (4) over 2 ports(0,2) with 2 RX queues per lcore:

```
./build/ip_reassembly -l 4 -n 3 -- -p 5 -q 2
```

To test the application, flows should be set up in the flow generator that match the values in the l3fwd_ipv4_route_array and/or l3fwd_ipv6_route_array table.

Please note that in order to test this application, the traffic generator should be generating valid fragmented IP packets. For IPv6, the only supported case is when no other extension headers other than fragment extension header are present in the packet.

The default l3fwd_ipv4_route_array table is:

```
struct l3fwd_ipv4_route l3fwd_ipv4_route_array[] = {
    {IPv4(100, 10, 0, 0), 16, 0},
    {IPv4(100, 20, 0, 0), 16, 1},
    {IPv4(100, 30, 0, 0), 16, 2},
    {IPv4(100, 40, 0, 0), 16, 3},
    {IPv4(100, 50, 0, 0), 16, 4},
    {IPv4(100, 60, 0, 0), 16, 5},
    {IPv4(100, 70, 0, 0), 16, 6},
    {IPv4(100, 80, 0, 0), 16, 7},
};
```

The default l3fwd_ipv6_route_array table is:

```
struct l3fwd_ipv6_route l3fwd_ipv6_route_array[] = {
    {{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 0},
    {{2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 1},
    {{3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 2},
    {{4, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 3},
    {{5, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 4},
    {{6, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 5},
    {{7, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 6},
    {{8, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 7},
};
```

For example, for the fragmented input IPv4 packet with destination address: 100.10.1.1, a reassembled IPv4 packet be sent out from port #0 to the destination address 100.10.1.1 once all the fragments are collected.

3.10.4 Explanation

The following sections provide some explanation of the sample application code. As mentioned in the overview section, the initialization and run-time paths are very similar to those of the *L2 Forwarding Sample Application (in Real and Virtualized Environments)*. The following sections describe aspects that are specific to the IP reassemble sample application.

IPv4 Fragment Table Initialization

This application uses the rte_ip_frag library. Please refer to Programmer's Guide for more detailed explanation of how to use this library. Fragment table maintains information about already received fragments of the packet. Each IP packet is uniquely identified by triple <Source IP address>, <Destination IP address>, <ID>. To avoid lock contention, each RX queue has its own Fragment Table, e.g. the application can't handle the situation when different fragments of the same packet arrive through different RX queues. Each table entry can hold information about packet consisting of up to RTE_LIBRTE_IP_FRAG_MAX_FRAGS fragments.

```
frag_cycles = (rte_get_tsc_hz() + MS_PER_S - 1) / MS_PER_S * max_flow_ttl;

if ((qconf->frag_tbl[queue] = rte_ip_frag_tbl_create(max_flow_num, IPV4_FRAG_TBL_
↪BUCKET_ENTRIES, max_flow_num, frag_cycles, socket)) == NULL)
{
    RTE_LOG(ERR, IP_RSMBL, "ip_frag_tbl_create(%u) on " "lcore: %u for queue: %u␣
↪failed\n",  max_flow_num, lcore, queue);
```

```
    return -1;
}
```

Mempools Initialization

The reassembly application demands a lot of mbuf's to be allocated. At any given time up to (2 * max_flow_num * RTE_LIBRTE_IP_FRAG_MAX_FRAGS * <maximum number of mbufs per packet>) can be stored inside Fragment Table waiting for remaining fragments. To keep mempool size under reasonable limits and to avoid situation when one RX queue can starve other queues, each RX queue uses its own mempool.

```
nb_mbuf = RTE_MAX(max_flow_num, 2UL * MAX_PKT_BURST) * RTE_LIBRTE_IP_FRAG_MAX_FRAGS;
nb_mbuf *= (port_conf.rxmode.max_rx_pkt_len + BUF_SIZE - 1) / BUF_SIZE;
nb_mbuf *= 2; /* ipv4 and ipv6 */
nb_mbuf += RTE_TEST_RX_DESC_DEFAULT + RTE_TEST_TX_DESC_DEFAULT;
nb_mbuf = RTE_MAX(nb_mbuf, (uint32_t)NB_MBUF);

snprintf(buf, sizeof(buf), "mbuf_pool_%u_%u", lcore, queue);

if ((rxq->pool = rte_mempool_create(buf, nb_mbuf, MBUF_SIZE, 0, sizeof(struct rte_
↪pktmbuf_pool_private), rte_pktmbuf_pool_init, NULL,
    rte_pktmbuf_init, NULL, socket, MEMPOOL_F_SP_PUT | MEMPOOL_F_SC_GET)) == NULL) {

        RTE_LOG(ERR, IP_RSMBL, "mempool_create(%s) failed", buf);
        return -1;
}
```

Packet Reassembly and Forwarding

For each input packet, the packet forwarding operation is done by the l3fwd_simple_forward() function. If the packet is an IPv4 or IPv6 fragment, then it calls rte_ipv4_reassemble_packet() for IPv4 packets, or rte_ipv6_reassemble_packet() for IPv6 packets. These functions either return a pointer to valid mbuf that contains reassembled packet, or NULL (if the packet can't be reassembled for some reason). Then l3fwd_simple_forward() continues with the code for the packet forwarding decision (that is, the identification of the output interface for the packet) and actual transmit of the packet.

The rte_ipv4_reassemble_packet() or rte_ipv6_reassemble_packet() are responsible for:

1. Searching the Fragment Table for entry with packet's <IP Source Address, IP Destination Address, Packet ID>
2. If the entry is found, then check if that entry already timed-out. If yes, then free all previously received fragments, and remove information about them from the entry.
3. If no entry with such key is found, then try to create a new one by one of two ways:
   (a) Use as empty entry
   (b) Delete a timed-out entry, free mbufs associated with it mbufs and store a new entry with specified key in it.
4. Update the entry with new fragment information and check if a packet can be reassembled (the packet's entry contains all fragments).
   (a) If yes, then, reassemble the packet, mark table's entry as empty and return the reassembled mbuf to the caller.
   (b) If no, then just return a NULL to the caller.

If at any stage of packet processing a reassembly function encounters an error (can't insert new entry into the Fragment table, or invalid/timed-out fragment), then it will free all associated with the packet fragments, mark the table entry as invalid and return NULL to the caller.

Debug logging and Statistics Collection

The RTE_LIBRTE_IP_FRAG_TBL_STAT controls statistics collection for the IP Fragment Table. This macro is disabled by default. To make ip_reassembly print the statistics to the standard output, the user must send either an USR1, INT or TERM signal to the process. For all of these signals, the ip_reassembly process prints Fragment table statistics for each RX queue, plus the INT and TERM will cause process termination as usual.

3.11 Kernel NIC Interface Sample Application

The Kernel NIC Interface (KNI) is a DPDK control plane solution that allows userspace applications to exchange packets with the kernel networking stack. To accomplish this, DPDK userspace applications use an IOCTL call to request the creation of a KNI virtual device in the Linux* kernel. The IOCTL call provides interface information and the DPDK's physical address space, which is re-mapped into the kernel address space by the KNI kernel loadable module that saves the information to a virtual device context. The DPDK creates FIFO queues for packet ingress and egress to the kernel module for each device allocated.

The KNI kernel loadable module is a standard net driver, which upon receiving the IOCTL call access the DPDK's FIFO queue to receive/transmit packets from/to the DPDK userspace application. The FIFO queues contain pointers to data packets in the DPDK. This:

- Provides a faster mechanism to interface with the kernel net stack and eliminates system calls
- Facilitates the DPDK using standard Linux* userspace net tools (tcpdump, ftp, and so on)
- Eliminate the copy_to_user and copy_from_user operations on packets.

The Kernel NIC Interface sample application is a simple example that demonstrates the use of the DPDK to create a path for packets to go through the Linux* kernel. This is done by creating one or more kernel net devices for each of the DPDK ports. The application allows the use of standard Linux tools (ethtool, ifconfig, tcpdump) with the DPDK ports and also the exchange of packets between the DPDK application and the Linux* kernel.

3.11.1 Overview

The Kernel NIC Interface sample application uses two threads in user space for each physical NIC port being used, and allocates one or more KNI device for each physical NIC port with kernel module's support. For a physical NIC port, one thread reads from the port and writes to KNI devices, and another thread reads from KNI devices and writes the data unmodified to the physical NIC port. It is recommended to configure one KNI device for each physical NIC port. If configured with more than one KNI devices for a physical NIC port, it is just for performance testing, or it can work together with VMDq support in future.

The packet flow through the Kernel NIC Interface application is as shown in the following figure.

3.11.2 Compiling the Application

Compile the application as follows:

1. Go to the example directory:

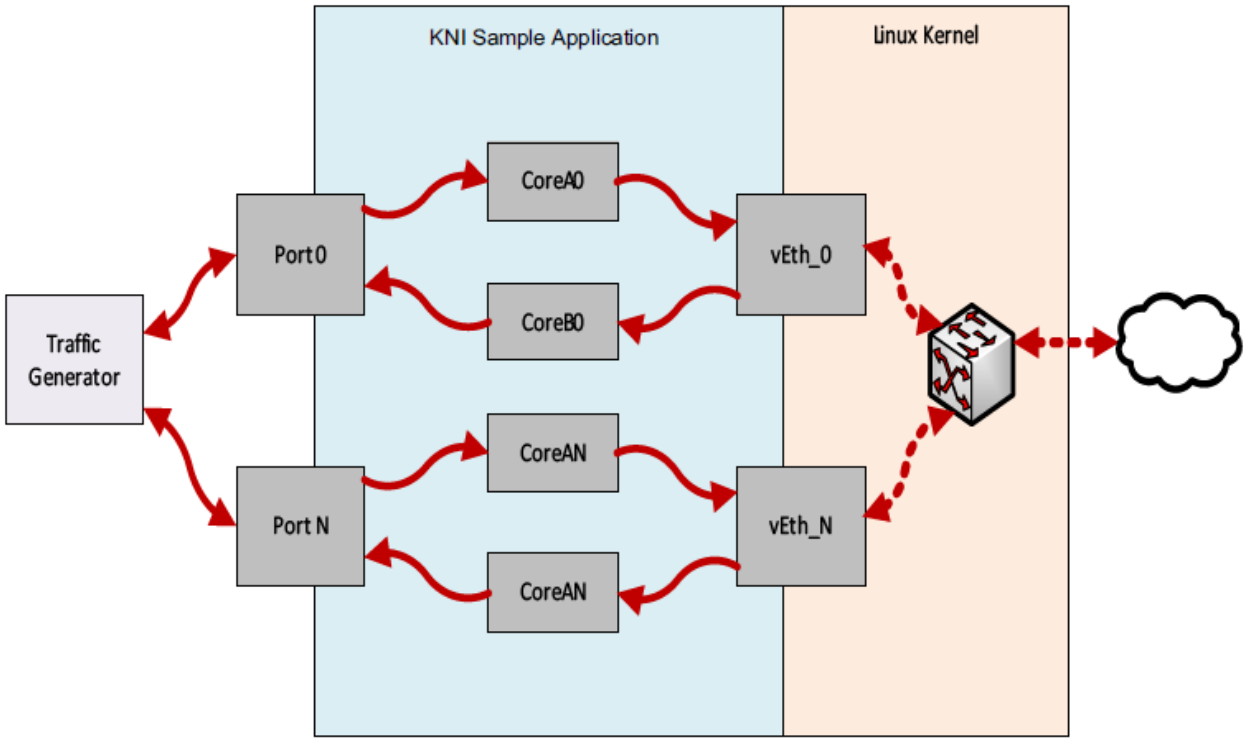


Fig. 3.2: Kernel NIC Application Packet Flow

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}/examples/kni
```

2. Set the target (a default target is used if not specified)

Note: This application is intended as a linuxapp only.

```
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

3. Build the application:

```
make
```

3.11.3 Loading the Kernel Module

Loading the KNI kernel module without any parameter is the typical way a DPDK application gets packets into and out of the kernel net stack. This way, only one kernel thread is created for all KNI devices for packet receiving in kernel side:

```
#insmod rte_kni.ko
```

Pinning the kernel thread to a specific core can be done using a taskset command such as following:

```
#taskset -p 100000 `pgrep --fl kni_thread | awk '{print $1}'`
```

This command line tries to pin the specific kni_thread on the 20th lcore (lcore numbering starts at 0), which means it needs to check if that lcore is available on the board. This command must be sent after the application has been launched, as insmod does not start the kni thread.

For optimum performance, the lcore in the mask must be selected to be on the same socket as the lcores used in the KNI application.

To provide flexibility of performance, the kernel module of the KNI, located in the kmod sub-directory of the DPDK target directory, can be loaded with parameter of kthread_mode as follows:

- #insmod rte_kni.ko kthread_mode=single

  This mode will create only one kernel thread for all KNI devices for packet receiving in kernel side. By default, it is in this single kernel thread mode. It can set core affinity for this kernel thread by using Linux command taskset.

- #insmod rte_kni.ko kthread_mode =multiple

  This mode will create a kernel thread for each KNI device for packet receiving in kernel side. The core affinity of each kernel thread is set when creating the KNI device. The lcore ID for each kernel thread is provided in the command line of launching the application. Multiple kernel thread mode can provide scalable higher performance.

To measure the throughput in a loopback mode, the kernel module of the KNI, located in the kmod sub-directory of the DPDK target directory, can be loaded with parameters as follows:

- #insmod rte_kni.ko lo_mode=lo_mode_fifo

  This loopback mode will involve ring enqueue/dequeue operations in kernel space.

- #insmod rte_kni.ko lo_mode=lo_mode_fifo_skb

  This loopback mode will involve ring enqueue/dequeue operations and sk buffer copies in kernel space.

3.11.4 Running the Application

The application requires a number of command line options:

```
kni [EAL options] -- -P -p PORTMASK --config="(port,lcore_rx,lcore_tx[,lcore_kthread,.
↪..])[,port,lcore_rx,lcore_tx[,lcore_kthread,...]]"
```

Where:

- -P: Set all ports to promiscuous mode so that packets are accepted regardless of the packet's Ethernet MAC destination address. Without this option, only packets with the Ethernet MAC destination address set to the Ethernet address of the port are accepted.

- -p PORTMASK: Hexadecimal bitmask of ports to configure.

- –config="(port,lcore_rx, lcore_tx[,lcore_kthread, ...]) [, port,lcore_rx, lcore_tx[,lcore_kthread, ...]]": Determines which lcores of RX, TX, kernel thread are mapped to which ports.

Refer to *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

The -c coremask or -l corelist parameter of the EAL options should include the lcores indicated by the lcore_rx and lcore_tx, but does not need to include lcores indicated by lcore_kthread as they are used to pin the kernel thread on. The -p PORTMASK parameter should include the ports indicated by the port in –config, neither more nor less.

The lcore_kthread in –config can be configured none, one or more lcore IDs. In multiple kernel thread mode, if configured none, a KNI device will be allocated for each port, while no specific lcore affinity will be set for its kernel thread. If configured one or more lcore IDs, one or more KNI devices will be allocated for each port, while specific

lcore affinity will be set for its kernel thread. In single kernel thread mode, if configured none, a KNI device will be allocated for each port. If configured one or more lcore IDs, one or more KNI devices will be allocated for each port while no lcore affinity will be set as there is only one kernel thread for all KNI devices.

For example, to run the application with two ports served by six lcores, one lcore of RX, one lcore of TX, and one lcore of kernel thread for each port:

```
./build/kni -l 4-7 -n 4 -- -P -p 0x3 -config="(0,4,6,8),(1,5,7,9)"
```

3.11.5 KNI Operations

Once the KNI application is started, one can use different Linux* commands to manage the net interfaces. If more than one KNI devices configured for a physical port, only the first KNI device will be paired to the physical device. Operations on other KNI devices will not affect the physical port handled in user space application.

Assigning an IP address:

```
#ifconfig vEth0_0 192.168.0.1
```

Displaying the NIC registers:

```
#ethtool -d vEth0_0
```

Dumping the network traffic:

```
#tcpdump -i vEth0_0
```

When the DPDK userspace application is closed, all the KNI devices are deleted from Linux*.

3.11.6 Explanation

The following sections provide some explanation of code.

Initialization

Setup of mbuf pool, driver and queues is similar to the setup done in the *L2 Forwarding Sample Application (in Real and Virtualized Environments)*.. In addition, one or more kernel NIC interfaces are allocated for each of the configured ports according to the command line parameters.

The code for allocating the kernel NIC interfaces for a specific port is as follows:

```
static int
kni_alloc(uint8_t port_id)
{
    uint8_t i;
    struct rte_kni *kni;
    struct rte_kni_conf conf;
    struct kni_port_params **params = kni_port_params_array;

    if (port_id >= RTE_MAX_ETHPORTS || !params[port_id])
        return -1;

    params[port_id]->nb_kni = params[port_id]->nb_lcore_k ? params[port_id]->nb_
↪lcore_k : 1;
```

```
    for (i = 0; i < params[port_id]->nb_kni; i++) {

        /* Clear conf at first */

        memset(&conf, 0, sizeof(conf));
        if (params[port_id]->nb_lcore_k) {
            snprintf(conf.name, RTE_KNI_NAMESIZE, "vEth%u_%u", port_id, i);
            conf.core_id = params[port_id]->lcore_k[i];
            conf.force_bind = 1;
        } else
            snprintf(conf.name, RTE_KNI_NAMESIZE, "vEth%u", port_id);
            conf.group_id = (uint16_t)port_id;
            conf.mbuf_size = MAX_PACKET_SZ;

            /*
             *   The first KNI device associated to a port
             *   is the master, for multiple kernel thread
             *   environment.
             */

            if (i == 0) {
                struct rte_kni_ops ops;
                struct rte_eth_dev_info dev_info;

                memset(&dev_info, 0, sizeof(dev_info)); rte_eth_dev_info_get(port_id,
↪ &dev_info);

                conf.addr = dev_info.pci_dev->addr;
                conf.id = dev_info.pci_dev->id;

                memset(&ops, 0, sizeof(ops));

                ops.port_id = port_id;
                ops.change_mtu = kni_change_mtu;
                ops.config_network_if = kni_config_network_interface;

                kni = rte_kni_alloc(pktmbuf_pool, &conf, &ops);
            } else
                kni = rte_kni_alloc(pktmbuf_pool, &conf, NULL);

            if (!kni)
                rte_exit(EXIT_FAILURE, "Fail to create kni for "
                        "port: %d\n", port_id);

            params[port_id]->kni[i] = kni;
        }
    return 0;
}
```

The other step in the initialization process that is unique to this sample application is the association of each port with lcores for RX, TX and kernel threads.

- One lcore to read from the port and write to the associated one or more KNI devices
- Another lcore to read from one or more KNI devices and write to the port
- Other lcores for pinning the kernel threads on one by one

This is done by using the‘kni_port_params_array[]‘ array, which is indexed by the port ID. The code is as follows:

```
static int
parse_config(const char *arg)
{
    const char *p, *p0 = arg;
    char s[256], *end;
    unsigned size;
    enum fieldnames {
        FLD_PORT = 0,
        FLD_LCORE_RX,
        FLD_LCORE_TX,
        _NUM_FLD = KNI_MAX_KTHREAD + 3,
    };
    int i, j, nb_token;
    char *str_fld[_NUM_FLD];
    unsigned long int_fld[_NUM_FLD];
    uint8_t port_id, nb_kni_port_params = 0;

    memset(&kni_port_params_array, 0, sizeof(kni_port_params_array));

    while (((p = strchr(p0, '(')) != NULL) && nb_kni_port_params < RTE_MAX_ETHPORTS) {
        p++;
        if ((p0 = strchr(p, ')')) == NULL)
            goto fail;

        size = p0 - p;

        if (size >= sizeof(s)) {
            printf("Invalid config parameters\n");
            goto fail;
        }

        snprintf(s, sizeof(s), "%.*s", size, p);
        nb_token = rte_strsplit(s, sizeof(s), str_fld, _NUM_FLD, ',');

        if (nb_token <= FLD_LCORE_TX) {
            printf("Invalid config parameters\n");
            goto fail;
        }

        for (i = 0; i < nb_token; i++) {
            errno = 0;
            int_fld[i] = strtoul(str_fld[i], &end, 0);
            if (errno != 0 || end == str_fld[i]) {
                printf("Invalid config parameters\n");
                goto fail;
            }
        }

        i = 0;
        port_id = (uint8_t)int_fld[i++];

        if (port_id >= RTE_MAX_ETHPORTS) {
            printf("Port ID %u could not exceed the maximum %u\n", port_id, RTE_MAX_
↪ETHPORTS);
            goto fail;
        }

        if (kni_port_params_array[port_id]) {
```

```
            printf("Port %u has been configured\n", port_id);
            goto fail;
        }

        kni_port_params_array[port_id] = (struct kni_port_params*)rte_zmalloc("KNI_
↪port_params", sizeof(struct kni_port_params), RTE_CACHE_LINE_SIZE);
        kni_port_params_array[port_id]->port_id = port_id;
        kni_port_params_array[port_id]->lcore_rx = (uint8_t)int_fld[i++];
        kni_port_params_array[port_id]->lcore_tx = (uint8_t)int_fld[i++];

        if (kni_port_params_array[port_id]->lcore_rx >= RTE_MAX_LCORE || kni_port_
↪params_array[port_id]->lcore_tx >= RTE_MAX_LCORE) {
            printf("lcore_rx %u or lcore_tx %u ID could not "
                    "exceed the maximum %u\n",
                    kni_port_params_array[port_id]->lcore_rx, kni_port_params_
↪array[port_id]->lcore_tx, RTE_MAX_LCORE);
            goto fail;
       }

    for (j = 0; i < nb_token && j < KNI_MAX_KTHREAD; i++, j++)
        kni_port_params_array[port_id]->lcore_k[j] = (uint8_t)int_fld[i];
        kni_port_params_array[port_id]->nb_lcore_k = j;
    }

    print_config();

    return 0;

fail:

    for (i = 0; i < RTE_MAX_ETHPORTS; i++) {
        if (kni_port_params_array[i]) {
            rte_free(kni_port_params_array[i]);
            kni_port_params_array[i] = NULL;
        }
    }

    return -1;

}
```

Packet Forwarding

After the initialization steps are completed, the main_loop() function is run on each lcore. This function first checks the lcore_id against the user provided lcore_rx and lcore_tx to see if this lcore is reading from or writing to kernel NIC interfaces.

For the case that reads from a NIC port and writes to the kernel NIC interfaces, the packet reception is the same as in L2 Forwarding sample application (see *Receive, Process and Transmit Packets*). The packet transmission is done by sending mbufs into the kernel NIC interfaces by rte_kni_tx_burst(). The KNI library automatically frees the mbufs after the kernel successfully copied the mbufs.

```
/**
 *   Interface to burst rx and enqueue mbufs into rx_q
 */
```

```
static void
kni_ingress(struct kni_port_params *p)
{
    uint8_t i, nb_kni, port_id;
    unsigned nb_rx, num;
    struct rte_mbuf *pkts_burst[PKT_BURST_SZ];

    if (p == NULL)
        return;

    nb_kni = p->nb_kni;
    port_id = p->port_id;

    for (i = 0; i < nb_kni; i++) {
        /* Burst rx from eth */
        nb_rx = rte_eth_rx_burst(port_id, 0, pkts_burst, PKT_BURST_SZ);
        if (unlikely(nb_rx > PKT_BURST_SZ)) {
            RTE_LOG(ERR, APP, "Error receiving from eth\n");
            return;
        }

        /* Burst tx to kni */
        num = rte_kni_tx_burst(p->kni[i], pkts_burst, nb_rx);
        kni_stats[port_id].rx_packets += num;
        rte_kni_handle_request(p->kni[i]);

        if (unlikely(num < nb_rx)) {
            /* Free mbufs not tx to kni interface */
            kni_burst_free_mbufs(&pkts_burst[num], nb_rx - num);
            kni_stats[port_id].rx_dropped += nb_rx - num;
        }
    }
}
```

For the other case that reads from kernel NIC interfaces and writes to a physical NIC port, packets are retrieved by reading mbufs from kernel NIC interfaces by *rte_kni_rx_burst()*. The packet transmission is the same as in the L2 Forwarding sample application (see *Receive, Process and Transmit Packets*).

```
/**
 *   Interface to dequeue mbufs from tx_q and burst tx
 */

static void

kni_egress(struct kni_port_params *p)
{
    uint8_t i, nb_kni, port_id;
    unsigned nb_tx, num;
    struct rte_mbuf *pkts_burst[PKT_BURST_SZ];

    if (p == NULL)
        return;

    nb_kni = p->nb_kni;
    port_id = p->port_id;

    for (i = 0; i < nb_kni; i++) {
        /* Burst rx from kni */
```

```
        num = rte_kni_rx_burst(p->kni[i], pkts_burst, PKT_BURST_SZ);
        if (unlikely(num > PKT_BURST_SZ)) {
            RTE_LOG(ERR, APP, "Error receiving from KNI\n");
            return;
        }

        /* Burst tx to eth */

        nb_tx = rte_eth_tx_burst(port_id, 0, pkts_burst, (uint16_t)num);

        kni_stats[port_id].tx_packets += nb_tx;

        if (unlikely(nb_tx < num)) {
            /* Free mbufs not tx to NIC */
            kni_burst_free_mbufs(&pkts_burst[nb_tx], num - nb_tx);
            kni_stats[port_id].tx_dropped += num - nb_tx;
        }
    }
}
```

Callbacks for Kernel Requests

To execute specific PMD operations in user space requested by some Linux* commands, callbacks must be implemented and filled in the struct rte_kni_ops structure. Currently, setting a new MTU and configuring the network interface (up/ down) are supported.

```
static struct rte_kni_ops kni_ops = {
    .change_mtu = kni_change_mtu,
    .config_network_if = kni_config_network_interface,
};

/* Callback for request of changing MTU */

static int
kni_change_mtu(uint8_t port_id, unsigned new_mtu)
{
    int ret;
    struct rte_eth_conf conf;

    if (port_id >= rte_eth_dev_count()) {
        RTE_LOG(ERR, APP, "Invalid port id %d\n", port_id);
        return -EINVAL;
    }

    RTE_LOG(INFO, APP, "Change MTU of port %d to %u\n", port_id, new_mtu);

    /* Stop specific port */

    rte_eth_dev_stop(port_id);

    memcpy(&conf, &port_conf, sizeof(conf));

    /* Set new MTU */

    if (new_mtu > ETHER_MAX_LEN)
        conf.rxmode.jumbo_frame = 1;
```

```
    else
        conf.rxmode.jumbo_frame = 0;

    /* mtu + length of header + length of FCS = max pkt length */

    conf.rxmode.max_rx_pkt_len = new_mtu + KNI_ENET_HEADER_SIZE + KNI_ENET_FCS_SIZE;

    ret = rte_eth_dev_configure(port_id, 1, 1, &conf);
    if (ret < 0) {
        RTE_LOG(ERR, APP, "Fail to reconfigure port %d\n", port_id);
        return ret;
    }

    /* Restart specific port */

    ret = rte_eth_dev_start(port_id);
    if (ret < 0) {
         RTE_LOG(ERR, APP, "Fail to restart port %d\n", port_id);
        return ret;
    }

    return 0;
}

/* Callback for request of configuring network interface up/down */

static int
kni_config_network_interface(uint8_t port_id, uint8_t if_up)
{
    int ret = 0;

    if (port_id >= rte_eth_dev_count() || port_id >= RTE_MAX_ETHPORTS) {
        RTE_LOG(ERR, APP, "Invalid port id %d\n", port_id);
        return -EINVAL;
    }

    RTE_LOG(INFO, APP, "Configure network interface of %d %s\n",

    port_id, if_up ? "up" : "down");

    if (if_up != 0) {
        /* Configure network interface up */
        rte_eth_dev_stop(port_id);
        ret = rte_eth_dev_start(port_id);
    } else /* Configure network interface down */
        rte_eth_dev_stop(port_id);

    if (ret < 0)
        RTE_LOG(ERR, APP, "Failed to start port %d\n", port_id);
    return ret;
}
```

3.12 Keep Alive Sample Application

The Keep Alive application is a simple example of a heartbeat/watchdog for packet processing cores. It demonstrates how to detect 'failed' DPDK cores and notify a fault management entity of this failure. Its purpose is to ensure the failure of the core does not result in a fault that is not detectable by a management entity.

3.12.1 Overview

The application demonstrates how to protect against 'silent outages' on packet processing cores. A Keep Alive Monitor Agent Core (master) monitors the state of packet processing cores (worker cores) by dispatching pings at a regular time interval (default is 5ms) and monitoring the state of the cores. Cores states are: Alive, MIA, Dead or Buried. MIA indicates a missed ping, and Dead indicates two missed pings within the specified time interval. When a core is Dead, a callback function is invoked to restart the packet processing core; A real life application might use this callback function to notify a higher level fault management entity of the core failure in order to take the appropriate corrective action.

Note: Only the worker cores are monitored. A local (on the host) mechanism or agent to supervise the Keep Alive Monitor Agent Core DPDK core is required to detect its failure.

Note: This application is based on the *L2 Forwarding Sample Application (in Real and Virtualized Environments)*. As such, the initialization and run-time paths are very similar to those of the L2 forwarding application.

3.12.2 Compiling the Application

To compile the application:

1. Go to the sample application directory:

```
export RTE_SDK=/path/to/rte_sdk cd ${RTE_SDK}/examples/keep_alive
```

2. Set the target (a default target is used if not specified). For example:

```
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

   See the *DPDK Getting Started Guide* for possible RTE_TARGET values.

3. Build the application:

```
make
```

3.12.3 Running the Application

The application has a number of command line options:

```
./build/l2fwd-keepalive [EAL options] \
        -- -p PORTMASK [-q NQ] [-K PERIOD] [-T PERIOD]
```

where,

- `p PORTMASK`: A hexadecimal bitmask of the ports to configure
- `q NQ`: A number of queues (=ports) per lcore (default is 1)
- `K PERIOD`: Heartbeat check period in ms(5ms default; 86400 max)

- `T PERIOD`: statistics will be refreshed each PERIOD seconds (0 to disable, 10 default, 86400 maximum).

To run the application in linuxapp environment with 4 lcores, 16 ports 8 RX queues per lcore and a ping interval of 10ms, issue the command:

```
./build/l2fwd-keepalive -l 0-3 -n 4 -- -q 8 -p ffff -K 10
```

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

3.12.4 Explanation

The following sections provide some explanation of the The Keep-Alive/'Liveliness' conceptual scheme. As mentioned in the overview section, the initialization and run-time paths are very similar to those of the *L2 Forwarding Sample Application (in Real and Virtualized Environments)*.

The Keep-Alive/'Liveliness' conceptual scheme:

- A Keep- Alive Agent Runs every N Milliseconds.
- DPDK Cores respond to the keep-alive agent.
- If keep-alive agent detects time-outs, it notifies the fault management entity through a callback function.

The following sections provide some explanation of the code aspects that are specific to the Keep Alive sample application.

The keepalive functionality is initialized with a struct rte_keepalive and the callback function to invoke in the case of a timeout.

```
rte_global_keepalive_info = rte_keepalive_create(&dead_core, NULL);
if (rte_global_keepalive_info == NULL)
    rte_exit(EXIT_FAILURE, "keepalive_create() failed");
```

The function that issues the pings keepalive_dispatch_pings() is configured to run every check_period milliseconds.

```
if (rte_timer_reset(&hb_timer,
        (check_period * rte_get_timer_hz()) / 1000,
        PERIODICAL,
        rte_lcore_id(),
        &rte_keepalive_dispatch_pings,
        rte_global_keepalive_info
        ) != 0 )
    rte_exit(EXIT_FAILURE, "Keepalive setup failure.\n");
```

The rest of the initialization and run-time path follows the same paths as the the L2 forwarding application. The only addition to the main processing loop is the mark alive functionality and the example random failures.

```
rte_keepalive_mark_alive(&rte_global_keepalive_info);
cur_tsc = rte_rdtsc();

/* Die randomly within 7 secs for demo purposes.. */
if (cur_tsc - tsc_initial > tsc_lifetime)
break;
```

The rte_keepalive_mark_alive function simply sets the core state to alive.

```
static inline void
rte_keepalive_mark_alive(struct rte_keepalive *keepcfg)
{
    keepcfg->state_flags[rte_lcore_id()] = ALIVE;
}
```

3.13 L2 Forwarding with Crypto Sample Application

The L2 Forwarding with Crypto (l2fwd-crypto) sample application is a simple example of packet processing using the Data Plane Development Kit (DPDK), in conjunction with the Cryptodev library.

3.13.1 Overview

The L2 Forwarding with Crypto sample application performs a crypto operation (cipher/hash) specified by the user from command line (or using the default values), with a crypto device capable of doing that operation, for each packet that is received on a RX_PORT and performs L2 forwarding. The destination port is the adjacent port from the enabled portmask, that is, if the first four ports are enabled (portmask 0xf), ports 0 and 1 forward into each other, and ports 2 and 3 forward into each other. Also, the MAC addresses are affected as follows:

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

3.13.2 Compiling the Application

1. Go to the example directory:

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}/examples/l2fwd-crypto
```

2. Set the target (a default target is used if not specified). For example:

```
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

   See the DPDK Getting Started Guide for possible RTE_TARGET values.

3. Build the application:

```
make
```

3.13.3 Running the Application

The application requires a number of command line options:

```
./build/l2fwd-crypto [EAL options] -- [-p PORTMASK] [-q NQ] [-s] [-T PERIOD] /
[--cdev_type HW/SW/ANY] [--chain HASH_CIPHER/CIPHER_HASH/CIPHER_ONLY/HASH_ONLY] /
[--cipher_algo ALGO] [--cipher_op ENCRYPT/DECRYPT] [--cipher_key KEY] /
[--cipher_key_random_size SIZE] [--iv IV] [--iv_random_size SIZE] /
[--auth_algo ALGO] [--auth_op GENERATE/VERIFY] [--auth_key KEY] /
[--auth_key_random_size SIZE] [--aad AAD] [--aad_random_size SIZE] /
[--digest size SIZE] [--sessionless]
```

where,

- p PORTMASK: A hexadecimal bitmask of the ports to configure (default is all the ports)
- q NQ: A number of queues (=ports) per lcore (default is 1)
- s: manage all ports from single core
- T PERIOD: statistics will be refreshed each PERIOD seconds
  (0 to disable, 10 default, 86400 maximum)
- cdev_type: select preferred crypto device type: HW, SW or anything (ANY)
  (default is ANY)
- chain: select the operation chaining to perform: Cipher->Hash (CIPHER_HASH),
  Hash->Cipher (HASH_CIPHER), Cipher (CIPHER_ONLY), Hash(HASH_ONLY)
  (default is Cipher->Hash)
- cipher_algo: select the ciphering algorithm (default is aes-cbc)
- cipher_op: select the ciphering operation to perform: ENCRYPT or DECRYPT
  (default is ENCRYPT)
- cipher_key: set the ciphering key to be used. Bytes has to be separated with ":"
- cipher_key_random_size: set the size of the ciphering key,
  which will be generated randomly.
  Note that if –cipher_key is used, this will be ignored.
- iv: set the IV to be used. Bytes has to be separated with ":"
- iv_random_size: set the size of the IV, which will be generated randomly.
  Note that if –iv is used, this will be ignored.
- auth_algo: select the authentication algorithm (default is sha1-hmac)
- cipher_op: select the authentication operation to perform: GENERATE or VERIFY
  (default is GENERATE)
- auth_key: set the authentication key to be used. Bytes has to be separated with ":"
- auth_key_random_size: set the size of the authentication key,
  which will be generated randomly.
  Note that if –auth_key is used, this will be ignored.
- aad: set the AAD to be used. Bytes has to be separated with ":"
- aad_random_size: set the size of the AAD, which will be generated randomly.
  Note that if –aad is used, this will be ignored.
- digest_size: set the size of the digest to be generated/verified.
- sessionless: no crypto session will be created.

The application requires that crypto devices capable of performing the specified crypto operation are available on application initialization. This means that HW crypto device/s must be bound to a DPDK driver or a SW crypto device/s (virtual crypto PMD) must be created (using –vdev).

To run the application in linuxapp environment with 2 lcores, 2 ports and 2 crypto devices, issue the command:

```
$ ./build/l2fwd-crypto -l 0-1 -n 4 --vdev "cryptodev_aesni_mb_pmd" \
--vdev "cryptodev_aesni_mb_pmd" -- -p 0x3 --chain CIPHER_HASH \
--cipher_op ENCRYPT --cipher_algo aes-cbc \
--cipher_key 00:01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f \
--auth_op GENERATE --auth_algo aes-xcbc-mac \
--auth_key 10:11:12:13:14:15:16:17:18:19:1a:1b:1c:1d:1e:1f
```

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

3.13.4 Explanation

The L2 forward with Crypto application demonstrates the performance of a crypto operation on a packet received on a RX PORT before forwarding it to a TX PORT.

The following figure illustrates a sample flow of a packet in the application, from reception until transmission.

Fig. 3.3: Encryption flow Through the L2 Forwarding with Crypto Application

The following sections provide some explanation of the application.

Crypto operation specification

All the packets received in all the ports get transformed by the crypto device/s (ciphering and/or authentication). The crypto operation to be performed on the packet is parsed from the command line (go to "Running the Application section for all the options).

If no parameter is passed, the default crypto operation is:

- Encryption with AES-CBC with 128 bit key.
- Authentication with SHA1-HMAC (generation).
- Keys, IV and AAD are generated randomly.

There are two methods to pass keys, IV and ADD from the command line:

- Passing the full key, separated bytes by ":":

```
--cipher_key 00:11:22:33:44
```

- Passing the size, so key is generated randomly:

```
--cipher_key_random_size 16
```

Note: If full key is passed (first method) and the size is passed as well (second method), the latter will be ignored.

Size of these keys are checked (regardless the method), before starting the app, to make sure that it is supported by the crypto devices.

Crypto device initialization

Once the encryption operation is defined, crypto devices are initialized. The crypto devices must be either bound to a DPDK driver (if they are physical devices) or created using the EAL option –vdev (if they are virtual devices), when running the application.

The initialize_cryptodevs() function performs the device initialization. It iterates through the list of the available crypto devices and check which ones are capable of performing the operation. Each device has a set of capabilities associated with it, which are stored in the device info structure, so the function checks if the operation is within the structure of each device.

The following code checks if the device supports the specified cipher algorithm (similar for the authentication algorithm):

```
/* Check if device supports cipher algo */
i = 0;
opt_cipher_algo = options->cipher_xform.cipher.algo;
cap = &dev_info.capabilities[i];
while (cap->op != RTE_CRYPTO_OP_TYPE_UNDEFINED) {
        cap_cipher_algo = cap->sym.cipher.algo;
        if (cap->sym.xform_type ==
                        RTE_CRYPTO_SYM_XFORM_CIPHER) {
                if (cap_cipher_algo == opt_cipher_algo) {
                        if (check_type(options, &dev_info) == 0)
                                break;
                }
        }
        cap = &dev_info.capabilities[++i];
}
```

If a capable crypto device is found, key sizes are checked to see if they are supported (cipher key and IV for the ciphering):

```
/*
 * Check if length of provided cipher key is supported
 * by the algorithm chosen.
 */
if (options->ckey_param) {
        if (check_supported_size(
                        options->cipher_xform.cipher.key.length,
                        cap->sym.cipher.key_size.min,
                        cap->sym.cipher.key_size.max,
                        cap->sym.cipher.key_size.increment)
                                != 0) {
                printf("Unsupported cipher key length\n");
                return -1;
        }
/*
 * Check if length of the cipher key to be randomly generated
 * is supported by the algorithm chosen.
 */
} else if (options->ckey_random_size != -1) {
        if (check_supported_size(options->ckey_random_size,
                        cap->sym.cipher.key_size.min,
                        cap->sym.cipher.key_size.max,
                        cap->sym.cipher.key_size.increment)
                                != 0) {
                printf("Unsupported cipher key length\n");
                return -1;
        }
        options->cipher_xform.cipher.key.length =
                                options->ckey_random_size;
/* No size provided, use minimum size. */
} else
        options->cipher_xform.cipher.key.length =
```

```
                        cap->sym.cipher.key_size.min;
```

After all the checks, the device is configured and it is added to the crypto device list.

Note: The number of crypto devices that supports the specified crypto operation must be at least the number of ports to be used.

Session creation

The crypto operation has a crypto session associated to it, which contains information such as the transform chain to perform (e.g. ciphering then hashing), pointers to the keys, lengths... etc.

This session is created and is later attached to the crypto operation:

```
static struct rte_cryptodev_sym_session *
initialize_crypto_session(struct l2fwd_crypto_options *options,
                uint8_t cdev_id)
{
        struct rte_crypto_sym_xform *first_xform;

        if (options->xform_chain == L2FWD_CRYPTO_CIPHER_HASH) {
                first_xform = &options->cipher_xform;
                first_xform->next = &options->auth_xform;
        } else if (options->xform_chain == L2FWD_CRYPTO_HASH_CIPHER) {
                first_xform = &options->auth_xform;
                first_xform->next = &options->cipher_xform;
        } else if (options->xform_chain == L2FWD_CRYPTO_CIPHER_ONLY) {
                first_xform = &options->cipher_xform;
        } else {
                first_xform = &options->auth_xform;
        }

        /* Setup Cipher Parameters */
        return rte_cryptodev_sym_session_create(cdev_id, first_xform);
}

...

port_cparams[i].session = initialize_crypto_session(options,
                              port_cparams[i].dev_id);
```

Crypto operation creation

Given N packets received from a RX PORT, N crypto operations are allocated and filled:

```
if (nb_rx) {
/*
 * If we can't allocate a crypto_ops, then drop
 * the rest of the burst and dequeue and
 * process the packets to free offload structs
 */
if (rte_crypto_op_bulk_alloc(
                l2fwd_crypto_op_pool,
                RTE_CRYPTO_OP_TYPE_SYMMETRIC,
                ops_burst, nb_rx) !=
                                nb_rx) {
```

```
        for (j = 0; j < nb_rx; j++)
                rte_pktmbuf_free(pkts_burst[i]);

        nb_rx = 0;
}
```

After filling the crypto operation (including session attachment), the mbuf which will be transformed is attached to it:

```
op->sym->m_src = m;
```

Since no destination mbuf is set, the source mbuf will be overwritten after the operation is done (in-place).

Crypto operation enqueuing/dequeuing

Once the operation has been created, it has to be enqueued in one of the crypto devices. Before doing so, for performance reasons, the operation stays in a buffer. When the buffer has enough operations (MAX_PKT_BURST), they are enqueued in the device, which will perform the operation at that moment:

```
static int
l2fwd_crypto_enqueue(struct rte_crypto_op *op,
                struct l2fwd_crypto_params *cparams)
{
        unsigned lcore_id, len;
        struct lcore_queue_conf *qconf;

        lcore_id = rte_lcore_id();

        qconf = &lcore_queue_conf[lcore_id];
        len = qconf->op_buf[cparams->dev_id].len;
        qconf->op_buf[cparams->dev_id].buffer[len] = op;
        len++;

        /* enough ops to be sent */
        if (len == MAX_PKT_BURST) {
                l2fwd_crypto_send_burst(qconf, MAX_PKT_BURST, cparams);
                len = 0;
        }

        qconf->op_buf[cparams->dev_id].len = len;
        return 0;
}

...

static int
l2fwd_crypto_send_burst(struct lcore_queue_conf *qconf, unsigned n,
                struct l2fwd_crypto_params *cparams)
{
        struct rte_crypto_op **op_buffer;
        unsigned ret;

        op_buffer = (struct rte_crypto_op **)
                        qconf->op_buf[cparams->dev_id].buffer;

        ret = rte_cryptodev_enqueue_burst(cparams->dev_id,
                        cparams->qp_id, op_buffer, (uint16_t) n);
```

```
        crypto_statistics[cparams->dev_id].enqueued += ret;
        if (unlikely(ret < n)) {
                crypto_statistics[cparams->dev_id].errors += (n - ret);
                do {
                        rte_pktmbuf_free(op_buffer[ret]->sym->m_src);
                        rte_crypto_op_free(op_buffer[ret]);
                } while (++ret < n);
        }

        return 0;
}
```

After this, the operations are dequeued from the device, and the transformed mbuf is extracted from the operation. Then, the operation is freed and the mbuf is forwarded as it is done in the L2 forwarding application.

```
/* Dequeue packets from Crypto device */
do {
        nb_rx = rte_cryptodev_dequeue_burst(
                        cparams->dev_id, cparams->qp_id,
                        ops_burst, MAX_PKT_BURST);

        crypto_statistics[cparams->dev_id].dequeued +=
                        nb_rx;

        /* Forward crypto'd packets */
        for (j = 0; j < nb_rx; j++) {
                m = ops_burst[j]->sym->m_src;

                rte_crypto_op_free(ops_burst[j]);
                l2fwd_simple_forward(m, portid);
        }
} while (nb_rx == MAX_PKT_BURST);
```

3.14 L2 Forwarding Sample Application (in Real and Virtualized Environments) with core load statistics.

The L2 Forwarding sample application is a simple example of packet processing using the Data Plane Development Kit (DPDK) which also takes advantage of Single Root I/O Virtualization (SR-IOV) features in a virtualized environment.

Note: This application is a variation of L2 Forwarding sample application. It demonstrate possible scheme of job stats library usage therefore some parts of this document is identical with original L2 forwarding application.

3.14.1 Overview

The L2 Forwarding sample application, which can operate in real and virtualized environments, performs L2 forwarding for each packet that is received. The destination port is the adjacent port from the enabled portmask, that is, if the first four ports are enabled (portmask 0xf), ports 1 and 2 forward into each other, and ports 3 and 4 forward into each other. Also, the MAC addresses are affected as follows:

- The source MAC address is replaced by the TX port MAC address
- The destination MAC address is replaced by 02:00:00:00:00:TX_PORT_ID

This application can be used to benchmark performance using a traffic-generator, as shown in the Fig. 3.4.

The application can also be used in a virtualized environment as shown in Fig. 3.5.

The L2 Forwarding application can also be used as a starting point for developing a new application based on the DPDK.

Fig. 3.4: Performance Benchmark Setup (Basic Environment)

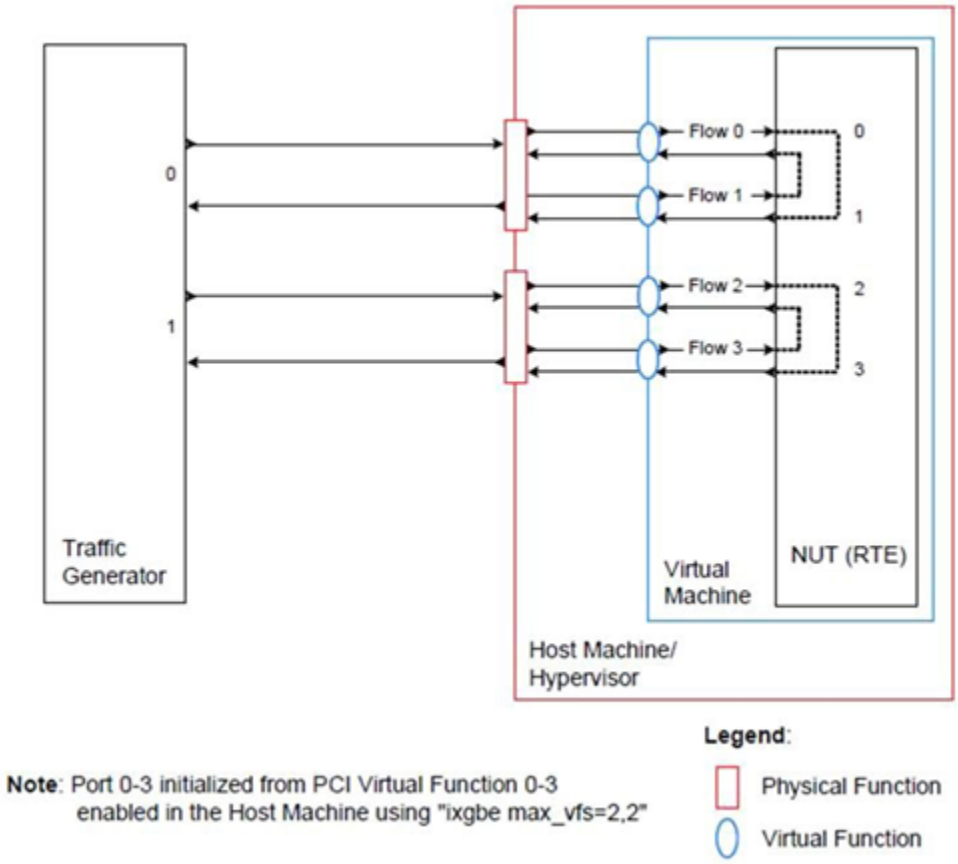


Fig. 3.5: Performance Benchmark Setup (Virtualized Environment)

Virtual Function Setup Instructions

This application can use the virtual function available in the system and therefore can be used in a virtual machine without passing through the whole Network Device into a guest machine in a virtualized scenario. The virtual functions can be enabled in the host machine or the hypervisor with the respective physical function driver.

For example, in a Linux* host machine, it is possible to enable a virtual function using the following command:

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

This command enables two Virtual Functions on each of Physical Function of the NIC, with two physical ports in the PCI configuration space. It is important to note that enabled Virtual Function 0 and 2 would belong to Physical

Function 0 and Virtual Function 1 and 3 would belong to Physical Function 1, in this case enabling a total of four Virtual Functions.

3.14.2 Compiling the Application

1. Go to the example directory:

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}/examples/l2fwd-jobstats
```

2. Set the target (a default target is used if not specified). For example:

```
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

   See the DPDK Getting Started Guide for possible RTE_TARGET values.

3. Build the application:

```
make
```

3.14.3 Running the Application

The application requires a number of command line options:

```
./build/l2fwd-jobstats [EAL options] -- -p PORTMASK [-q NQ] [-l]
```

where,

- p PORTMASK: A hexadecimal bitmask of the ports to configure
- q NQ: A number of queues (=ports) per lcore (default is 1)
- l: Use locale thousands separator when formatting big numbers.

To run the application in linuxapp environment with 4 lcores, 16 ports, 8 RX queues per lcore and thousands separator printing, issue the command:

```
$ ./build/l2fwd-jobstats -l 0-3 -n 4 -- -q 8 -p ffff -l
```

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

3.14.4 Explanation

The following sections provide some explanation of the code.

Command Line Arguments

The L2 Forwarding sample application takes specific parameters, in addition to Environment Abstraction Layer (EAL) arguments (see *Running the Application*). The preferred way to parse parameters is to use the getopt() function, since it is part of a well-defined and portable library.

The parsing of arguments is done in the l2fwd_parse_args() function. The method of argument parsing is not described here. Refer to the *glibc getopt(3)* man page for details.

EAL arguments are parsed first, then application-specific arguments. This is done at the beginning of the main() function:

```
/* init EAL */

ret = rte_eal_init(argc, argv);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Invalid EAL arguments\n");

argc -= ret;
argv += ret;

/* parse application arguments (after the EAL ones) */

ret = l2fwd_parse_args(argc, argv);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Invalid L2FWD arguments\n");
```

Mbuf Pool Initialization

Once the arguments are parsed, the mbuf pool is created. The mbuf pool contains a set of mbuf objects that will be used by the driver and the application to store network packet data:

```
/* create the mbuf pool */
l2fwd_pktmbuf_pool = rte_pktmbuf_pool_create("mbuf_pool", NB_MBUF,
            MEMPOOL_CACHE_SIZE, 0, RTE_MBUF_DEFAULT_BUF_SIZE,
            rte_socket_id());

if (l2fwd_pktmbuf_pool == NULL)
    rte_exit(EXIT_FAILURE, "Cannot init mbuf pool\n");
```

The rte_mempool is a generic structure used to handle pools of objects. In this case, it is necessary to create a pool that will be used by the driver. The number of allocated pkt mbufs is NB_MBUF, with a data room size of RTE_MBUF_DEFAULT_BUF_SIZE each. A per-lcore cache of MEMPOOL_CACHE_SIZE mbufs is kept. The memory is allocated in rte_socket_id() socket, but it is possible to extend this code to allocate one mbuf pool per socket.

The rte_pktmbuf_pool_create() function uses the default mbuf pool and mbuf initializers, respectively rte_pktmbuf_pool_init() and rte_pktmbuf_init(). An advanced application may want to use the mempool API to create the mbuf pool with more control.

Driver Initialization

The main part of the code in the main() function relates to the initialization of the driver. To fully understand this code, it is recommended to study the chapters that related to the Poll Mode Driver in the *DPDK Programmer's Guide* and the *DPDK API Reference*.

```
nb_ports = rte_eth_dev_count();

if (nb_ports == 0)
    rte_exit(EXIT_FAILURE, "No Ethernet ports - bye\n");

/* reset l2fwd_dst_ports */

for (portid = 0; portid < RTE_MAX_ETHPORTS; portid++)
```

```
    l2fwd_dst_ports[portid] = 0;

last_port = 0;

/*
 * Each logical core is assigned a dedicated TX queue on each port.
 */
for (portid = 0; portid < nb_ports; portid++) {
    /* skip ports that are not enabled */
    if ((l2fwd_enabled_port_mask & (1 << portid)) == 0)
       continue;

    if (nb_ports_in_mask % 2) {
        l2fwd_dst_ports[portid] = last_port;
        l2fwd_dst_ports[last_port] = portid;
    }
    else
       last_port = portid;

    nb_ports_in_mask++;

    rte_eth_dev_info_get((uint8_t) portid, &dev_info);
}
```

The next step is to configure the RX and TX queues. For each port, there is only one RX queue (only one lcore is able to poll a given port). The number of TX queues depends on the number of available lcores. The rte_eth_dev_configure() function is used to configure the number of queues for a port:

```
ret = rte_eth_dev_configure((uint8_t)portid, 1, 1, &port_conf);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Cannot configure device: "
        "err=%d, port=%u\n",
        ret, portid);
```

The global configuration is stored in a static structure:

```
static const struct rte_eth_conf port_conf = {
    .rxmode = {
        .split_hdr_size = 0,
        .header_split = 0,   /**< Header Split disabled */
        .hw_ip_checksum = 0, /**< IP checksum offload disabled */
        .hw_vlan_filter = 0, /**< VLAN filtering disabled */
        .jumbo_frame = 0,    /**< Jumbo Frame Support disabled */
        .hw_strip_crc= 0,    /**< CRC stripped by hardware */
    },

    .txmode = {
        .mq_mode = ETH_DCB_NONE
    },
};
```

RX Queue Initialization

The application uses one lcore to poll one or several ports, depending on the -q option, which specifies the number of queues per lcore.

For example, if the user specifies -q 4, the application is able to poll four ports with one lcore. If there are 16 ports on the target (and if the portmask argument is -p ffff), the application will need four lcores to poll all the ports.

```
ret = rte_eth_rx_queue_setup(portid, 0, nb_rxd,
            rte_eth_dev_socket_id(portid),
            NULL,
            l2fwd_pktmbuf_pool);

if (ret < 0)
    rte_exit(EXIT_FAILURE, "rte_eth_rx_queue_setup:err=%d, port=%u\n",
            ret, (unsigned) portid);
```

The list of queues that must be polled for a given lcore is stored in a private structure called struct lcore_queue_conf.

```
struct lcore_queue_conf {
    unsigned n_rx_port;
    unsigned rx_port_list[MAX_RX_QUEUE_PER_LCORE];
    truct mbuf_table tx_mbufs[RTE_MAX_ETHPORTS];

    struct rte_timer rx_timers[MAX_RX_QUEUE_PER_LCORE];
    struct rte_jobstats port_fwd_jobs[MAX_RX_QUEUE_PER_LCORE];

    struct rte_timer flush_timer;
    struct rte_jobstats flush_job;
    struct rte_jobstats idle_job;
    struct rte_jobstats_context jobs_context;

    rte_atomic16_t stats_read_pending;
    rte_spinlock_t lock;
} __rte_cache_aligned;
```

Values of struct lcore_queue_conf:

- n_rx_port and rx_port_list[] are used in the main packet processing loop (see Section *Receive, Process and Transmit Packets* later in this chapter).
- rx_timers and flush_timer are used to ensure forced TX on low packet rate.
- flush_job, idle_job and jobs_context are librte_jobstats objects used for managing l2fwd jobs.
- stats_read_pending and lock are used during job stats read phase.

TX Queue Initialization

Each lcore should be able to transmit on any port. For every port, a single TX queue is initialized.

```
/* init one TX queue on each port */

fflush(stdout);
ret = rte_eth_tx_queue_setup(portid, 0, nb_txd,
        rte_eth_dev_socket_id(portid),
        NULL);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "rte_eth_tx_queue_setup:err=%d, port=%u\n",
            ret, (unsigned) portid);
```

Jobs statistics initialization

There are several statistics objects available:

- Flush job statistics

```
rte_jobstats_init(&qconf->flush_job, "flush", drain_tsc, drain_tsc,
        drain_tsc, 0);

rte_timer_init(&qconf->flush_timer);
ret = rte_timer_reset(&qconf->flush_timer, drain_tsc, PERIODICAL,
            lcore_id, &l2fwd_flush_job, NULL);

if (ret < 0) {
    rte_exit(1, "Failed to reset flush job timer for lcore %u: %s",
                lcore_id, rte_strerror(-ret));
}
```

- Statistics per RX port

```
rte_jobstats_init(job, name, 0, drain_tsc, 0, MAX_PKT_BURST);
rte_jobstats_set_update_period_function(job, l2fwd_job_update_cb);

rte_timer_init(&qconf->rx_timers[i]);
ret = rte_timer_reset(&qconf->rx_timers[i], 0, PERIODICAL, lcore_id,
        l2fwd_fwd_job, (void *)(uintptr_t)i);

if (ret < 0) {
    rte_exit(1, "Failed to reset lcore %u port %u job timer: %s",
                lcore_id, qconf->rx_port_list[i], rte_strerror(-ret));
}
```

Following parameters are passed to rte_jobstats_init():

- 0 as minimal poll period
- drain_tsc as maximum poll period
- MAX_PKT_BURST as desired target value (RX burst size)

Main loop

The forwarding path is reworked comparing to original L2 Forwarding application. In the l2fwd_main_loop() function three loops are placed.

```
for (;;) {
    rte_spinlock_lock(&qconf->lock);

    do {
        rte_jobstats_context_start(&qconf->jobs_context);

        /* Do the Idle job:
         * - Read stats_read_pending flag
         * - check if some real job need to be executed
         */
        rte_jobstats_start(&qconf->jobs_context, &qconf->idle_job);

        do {
            uint8_t i;
```

```
            uint64_t now = rte_get_timer_cycles();

            need_manage = qconf->flush_timer.expire < now;
            /* Check if we was esked to give a stats. */
            stats_read_pending =
                    rte_atomic16_read(&qconf->stats_read_pending);
            need_manage |= stats_read_pending;

            for (i = 0; i < qconf->n_rx_port && !need_manage; i++)
                need_manage = qconf->rx_timers[i].expire < now;

        } while (!need_manage);
        rte_jobstats_finish(&qconf->idle_job, qconf->idle_job.target);

        rte_timer_manage();
        rte_jobstats_context_finish(&qconf->jobs_context);
    } while (likely(stats_read_pending == 0));

    rte_spinlock_unlock(&qconf->lock);
    rte_pause();
}
```

First infinite for loop is to minimize impact of stats reading. Lock is only locked/unlocked when asked.

Second inner while loop do the whole jobs management. When any job is ready, the use rte_timer_manage() is used to call the job handler. In this place functions l2fwd_fwd_job() and l2fwd_flush_job() are called when needed. Then rte_jobstats_context_finish() is called to mark loop end - no other jobs are ready to execute. By this time stats are ready to be read and if stats_read_pending is set, loop breaks allowing stats to be read.

Third do-while loop is the idle job (idle stats counter). Its only purpose is monitoring if any job is ready or stats job read is pending for this lcore. Statistics from this part of code is considered as the headroom available for additional processing.

Receive, Process and Transmit Packets

The main task of l2fwd_fwd_job() function is to read ingress packets from the RX queue of particular port and forward it. This is done using the following code:

```
total_nb_rx = rte_eth_rx_burst((uint8_t) portid, 0, pkts_burst,
        MAX_PKT_BURST);

for (j = 0; j < total_nb_rx; j++) {
    m = pkts_burst[j];
    rte_prefetch0(rte_pktmbuf_mtod(m, void *));
    l2fwd_simple_forward(m, portid);
}
```

Packets are read in a burst of size MAX_PKT_BURST. Then, each mbuf in the table is processed by the l2fwd_simple_forward() function. The processing is very simple: process the TX port from the RX port, then replace the source and destination MAC addresses.

The rte_eth_rx_burst() function writes the mbuf pointers in a local table and returns the number of available mbufs in the table.

After first read second try is issued.

```
if (total_nb_rx == MAX_PKT_BURST) {
    const uint16_t nb_rx = rte_eth_rx_burst((uint8_t) portid, 0, pkts_burst,
            MAX_PKT_BURST);

    total_nb_rx += nb_rx;
    for (j = 0; j < nb_rx; j++) {
        m = pkts_burst[j];
        rte_prefetch0(rte_pktmbuf_mtod(m, void *));
        l2fwd_simple_forward(m, portid);
    }
}
```

This second read is important to give job stats library a feedback how many packets was processed.

```
/* Adjust period time in which we are running here. */
if (rte_jobstats_finish(job, total_nb_rx) != 0) {
    rte_timer_reset(&qconf->rx_timers[port_idx], job->period, PERIODICAL,
            lcore_id, l2fwd_fwd_job, arg);
}
```

To maximize performance exactly MAX_PKT_BURST is expected (the target value) to be read for each l2fwd_fwd_job() call. If total_nb_rx is smaller than target value job->period will be increased. If it is greater the period will be decreased.

Note: In the following code, one line for getting the output port requires some explanation.

During the initialization process, a static array of destination ports (l2fwd_dst_ports[]) is filled such that for each source port, a destination port is assigned that is either the next or previous enabled port from the portmask. Naturally, the number of ports in the portmask must be even, otherwise, the application exits.

```
static void
l2fwd_simple_forward(struct rte_mbuf *m, unsigned portid)
{
    struct ether_hdr *eth;
    void *tmp;
    unsigned dst_port;

    dst_port = l2fwd_dst_ports[portid];

    eth = rte_pktmbuf_mtod(m, struct ether_hdr *);

    /* 02:00:00:00:00:xx */

    tmp = &eth->d_addr.addr_bytes[0];

    *((uint64_t *)tmp) = 0x000000000002 + ((uint64_t) dst_port << 40);

    /* src addr */

    ether_addr_copy(&l2fwd_ports_eth_addr[dst_port], &eth->s_addr);

    l2fwd_send_packet(m, (uint8_t) dst_port);
}
```

Then, the packet is sent using the l2fwd_send_packet (m, dst_port) function. For this test application, the processing is exactly the same for all packets arriving on the same RX port. Therefore, it would have been possible to call the

l2fwd_send_burst() function directly from the main loop to send all the received packets on the same TX port, using the burst-oriented send function, which is more efficient.

However, in real-life applications (such as, L3 routing), packet N is not necessarily forwarded on the same port as packet N-1. The application is implemented to illustrate that, so the same approach can be reused in a more complex application.

The l2fwd_send_packet() function stores the packet in a per-lcore and per-txport table. If the table is full, the whole packets table is transmitted using the l2fwd_send_burst() function:

```
/* Send the packet on an output interface */

static int
l2fwd_send_packet(struct rte_mbuf *m, uint8_t port)
{
    unsigned lcore_id, len;
    struct lcore_queue_conf *qconf;

    lcore_id = rte_lcore_id();
    qconf = &lcore_queue_conf[lcore_id];
    len = qconf->tx_mbufs[port].len;
    qconf->tx_mbufs[port].m_table[len] = m;
    len++;

    /* enough pkts to be sent */

    if (unlikely(len == MAX_PKT_BURST)) {
        l2fwd_send_burst(qconf, MAX_PKT_BURST, port);
        len = 0;
    }

    qconf->tx_mbufs[port].len = len; return 0;
}
```

To ensure that no packets remain in the tables, the flush job exists. The l2fwd_flush_job() is called periodically to for each lcore draining TX queue of each port. This technique introduces some latency when there are not many packets to send, however it improves performance:

```
static void
l2fwd_flush_job(__rte_unused struct rte_timer *timer, __rte_unused void *arg)
{
    uint64_t now;
    unsigned lcore_id;
    struct lcore_queue_conf *qconf;
    struct mbuf_table *m_table;
    uint8_t portid;

    lcore_id = rte_lcore_id();
    qconf = &lcore_queue_conf[lcore_id];

    rte_jobstats_start(&qconf->jobs_context, &qconf->flush_job);

    now = rte_get_timer_cycles();
    lcore_id = rte_lcore_id();
    qconf = &lcore_queue_conf[lcore_id];
    for (portid = 0; portid < RTE_MAX_ETHPORTS; portid++) {
        m_table = &qconf->tx_mbufs[portid];
        if (m_table->len == 0 || m_table->next_flush_time <= now)
            continue;
```

```
        l2fwd_send_burst(qconf, portid);
    }


    /* Pass target to indicate that this job is happy of time interval
     * in which it was called. */
    rte_jobstats_finish(&qconf->flush_job, qconf->flush_job.target);
}
```

3.15 L2 Forwarding Sample Application (in Real and Virtualized Environments)

The L2 Forwarding sample application is a simple example of packet processing using the Data Plane Development Kit (DPDK) which also takes advantage of Single Root I/O Virtualization (SR-IOV) features in a virtualized environment.

Note: Please note that previously a separate L2 Forwarding in Virtualized Environments sample application was used, however, in later DPDK versions these sample applications have been merged.

3.15.1 Overview

The L2 Forwarding sample application, which can operate in real and virtualized environments, performs L2 forwarding for each packet that is received on an RX_PORT. The destination port is the adjacent port from the enabled portmask, that is, if the first four ports are enabled (portmask 0xf), ports 1 and 2 forward into each other, and ports 3 and 4 forward into each other. Also, if MAC addresses updating is enabled, the MAC addresses are affected as follows:

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

This application can be used to benchmark performance using a traffic-generator, as shown in the Fig. 3.6, or in a virtualized environment as shown in Fig. 3.7.

Fig. 3.6: Performance Benchmark Setup (Basic Environment)

This application may be used for basic VM to VM communication as shown in Fig. 3.8, when MAC addresses updating is disabled.

The L2 Forwarding application can also be used as a starting point for developing a new application based on the DPDK.

Virtual Function Setup Instructions

This application can use the virtual function available in the system and therefore can be used in a virtual machine without passing through the whole Network Device into a guest machine in a virtualized scenario. The virtual functions can be enabled in the host machine or the hypervisor with the respective physical function driver.

For example, in a Linux* host machine, it is possible to enable a virtual function using the following command:

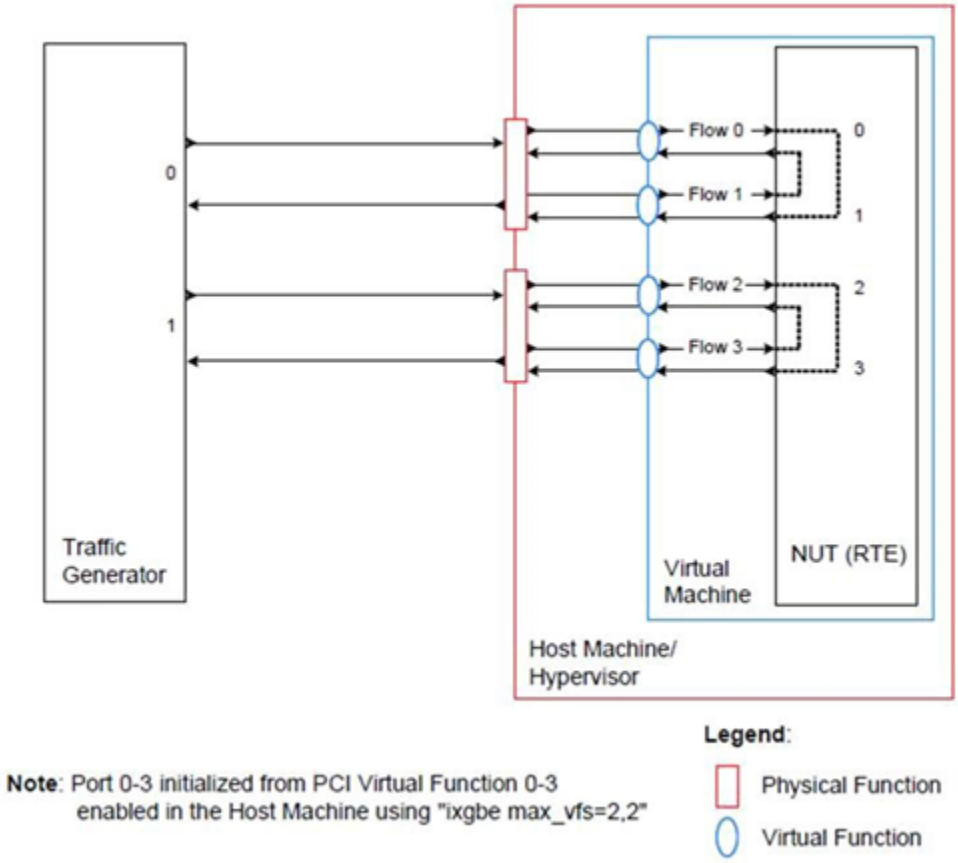


Fig. 3.7: Performance Benchmark Setup (Virtualized Environment)

Fig. 3.8: Virtual Machine to Virtual Machine communication.

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

This command enables two Virtual Functions on each of Physical Function of the NIC, with two physical ports in the PCI configuration space. It is important to note that enabled Virtual Function 0 and 2 would belong to Physical Function 0 and Virtual Function 1 and 3 would belong to Physical Function 1, in this case enabling a total of four Virtual Functions.

3.15.2 Compiling the Application

1. Go to the example directory:

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}/examples/l2fwd
```

2. Set the target (a default target is used if not specified). For example:

```
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

See the DPDK Getting Started Guide for possible RTE_TARGET values.

3. Build the application:

```
make
```

3.15.3 Running the Application

The application requires a number of command line options:

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

where,

- p PORTMASK: A hexadecimal bitmask of the ports to configure
- q NQ: A 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 4 -- -q 8 -p ffff
```

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

3.15.4 Explanation

The following sections provide some explanation of the code.

Command Line Arguments

The L2 Forwarding sample application takes specific parameters, in addition to Environment Abstraction Layer (EAL) arguments. The preferred way to parse parameters is to use the getopt() function, since it is part of a well-defined and portable library.

The parsing of arguments is done in the l2fwd_parse_args() function. The method of argument parsing is not described here. Refer to the *glibc getopt(3)* man page for details.

EAL arguments are parsed first, then application-specific arguments. This is done at the beginning of the main() function:

```
/* init EAL */

ret = rte_eal_init(argc, argv);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Invalid EAL arguments\n");

argc -= ret;
argv += ret;

/* parse application arguments (after the EAL ones) */

ret = l2fwd_parse_args(argc, argv);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Invalid L2FWD arguments\n");
```

Mbuf Pool Initialization

Once the arguments are parsed, the mbuf pool is created. The mbuf pool contains a set of mbuf objects that will be used by the driver and the application to store network packet data:

```
/* create the mbuf pool */

l2fwd_pktmbuf_pool = rte_pktmbuf_pool_create("mbuf_pool", NB_MBUF,
    MEMPOOL_CACHE_SIZE, 0, RTE_MBUF_DEFAULT_BUF_SIZE,
    rte_socket_id());

if (l2fwd_pktmbuf_pool == NULL)
    rte_panic("Cannot init mbuf pool\n");
```

The rte_mempool is a generic structure used to handle pools of objects. In this case, it is necessary to create a pool that will be used by the driver. The number of allocated pkt mbufs is NB_MBUF, with a data room size of RTE_MBUF_DEFAULT_BUF_SIZE each. A per-lcore cache of 32 mbufs is kept. The memory is allocated in NUMA socket 0, but it is possible to extend this code to allocate one mbuf pool per socket.

The rte_pktmbuf_pool_create() function uses the default mbuf pool and mbuf initializers, respectively rte_pktmbuf_pool_init() and rte_pktmbuf_init(). An advanced application may want to use the mempool API to create the mbuf pool with more control.

Driver Initialization

The main part of the code in the main() function relates to the initialization of the driver. To fully understand this code, it is recommended to study the chapters that related to the Poll Mode Driver in the *DPDK Programmer's Guide* - Rel 1.4 EAR and the *DPDK API Reference*.

```
if (rte_eal_pci_probe() < 0)
    rte_exit(EXIT_FAILURE, "Cannot probe PCI\n");

nb_ports = rte_eth_dev_count();

if (nb_ports == 0)
    rte_exit(EXIT_FAILURE, "No Ethernet ports - bye\n");

/* reset l2fwd_dst_ports */

for (portid = 0; portid < RTE_MAX_ETHPORTS; portid++)
    l2fwd_dst_ports[portid] = 0;

last_port = 0;

/*
 * Each logical core is assigned a dedicated TX queue on each port.
 */

for (portid = 0; portid < nb_ports; portid++) {
    /* skip ports that are not enabled */

    if ((l2fwd_enabled_port_mask & (1 << portid)) == 0)
       continue;

    if (nb_ports_in_mask % 2) {
        l2fwd_dst_ports[portid] = last_port;
        l2fwd_dst_ports[last_port] = portid;
    }
    else
       last_port = portid;

    nb_ports_in_mask++;

    rte_eth_dev_info_get((uint8_t) portid, &dev_info);
}
```

Observe that:

- rte_igb_pmd_init_all() simultaneously registers the driver as a PCI driver and as an Ethernet* Poll Mode Driver.
- rte_eal_pci_probe() parses the devices on the PCI bus and initializes recognized devices.

The next step is to configure the RX and TX queues. For each port, there is only one RX queue (only one lcore is able to poll a given port). The number of TX queues depends on the number of available lcores. The rte_eth_dev_configure() function is used to configure the number of queues for a port:

```
ret = rte_eth_dev_configure((uint8_t)portid, 1, 1, &port_conf);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Cannot configure device: "
        "err=%d, port=%u\n",
        ret, portid);
```

The global configuration is stored in a static structure:

```
static const struct rte_eth_conf port_conf = {
    .rxmode = {
        .split_hdr_size = 0,
        .header_split = 0,   /**< Header Split disabled */
```

```
        .hw_ip_checksum = 0, /**< IP checksum offload disabled */
        .hw_vlan_filter = 0, /**< VLAN filtering disabled */
        .jumbo_frame = 0,    /**< Jumbo Frame Support disabled */
        .hw_strip_crc= 0,    /**< CRC stripped by hardware */
    },

    .txmode = {
        .mq_mode = ETH_DCB_NONE
    },
};
```

RX Queue Initialization

The application uses one lcore to poll one or several ports, depending on the -q option, which specifies the number of queues per lcore.

For example, if the user specifies -q 4, the application is able to poll four ports with one lcore. If there are 16 ports on the target (and if the portmask argument is -p ffff), the application will need four lcores to poll all the ports.

```
ret = rte_eth_rx_queue_setup((uint8_t) portid, 0, nb_rxd, SOCKET0, &rx_conf, l2fwd_
↪pktmbuf_pool);
if (ret < 0)

    rte_exit(EXIT_FAILURE, "rte_eth_rx_queue_setup: "
        "err=%d, port=%u\n",
        ret, portid);
```

The list of queues that must be polled for a given lcore is stored in a private structure called struct lcore_queue_conf.

```
struct lcore_queue_conf {
    unsigned n_rx_port;
    unsigned rx_port_list[MAX_RX_QUEUE_PER_LCORE];
    struct mbuf_table tx_mbufs[L2FWD_MAX_PORTS];
} rte_cache_aligned;

struct lcore_queue_conf lcore_queue_conf[RTE_MAX_LCORE];
```

The values n_rx_port and rx_port_list[] are used in the main packet processing loop (see *Receive, Process and Transmit Packets*).

The global configuration for the RX queues is stored in a static structure:

```
static const struct rte_eth_rxconf rx_conf = {
    .rx_thresh = {
        .pthresh = RX_PTHRESH,
        .hthresh = RX_HTHRESH,
        .wthresh = RX_WTHRESH,
    },
};
```

TX Queue Initialization

Each lcore should be able to transmit on any port. For every port, a single TX queue is initialized.

```
/* init one TX queue on each port */

fflush(stdout);

ret = rte_eth_tx_queue_setup((uint8_t) portid, 0, nb_txd, rte_eth_dev_socket_
↪id(portid), &tx_conf);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "rte_eth_tx_queue_setup:err=%d, port=%u\n", ret,␣
↪(unsigned) portid);
```

The global configuration for TX queues is stored in a static structure:

```
static const struct rte_eth_txconf tx_conf = {
    .tx_thresh = {
        .pthresh = TX_PTHRESH,
        .hthresh = TX_HTHRESH,
        .wthresh = TX_WTHRESH,
    },
    .tx_free_thresh = RTE_TEST_TX_DESC_DEFAULT + 1, /* disable feature */
};
```

Receive, Process and Transmit Packets

In the l2fwd_main_loop() function, the main task is to read ingress packets from the RX queues. This is done using the following code:

```
/*
 * Read packet from RX queues
 */

for (i = 0; i < qconf->n_rx_port; i++) {
    portid = qconf->rx_port_list[i];
    nb_rx = rte_eth_rx_burst((uint8_t) portid, 0,  pkts_burst, MAX_PKT_BURST);

    for (j = 0; j < nb_rx; j++) {
        m = pkts_burst[j];
        rte_prefetch0[rte_pktmbuf_mtod(m, void *)); l2fwd_simple_forward(m, portid);
    }
}
```

Packets are read in a burst of size MAX_PKT_BURST. The rte_eth_rx_burst() function writes the mbuf pointers in a local table and returns the number of available mbufs in the table.

Then, each mbuf in the table is processed by the l2fwd_simple_forward() function. The processing is very simple: process the TX port from the RX port, then replace the source and destination MAC addresses if MAC addresses updating is enabled.

Note: In the following code, one line for getting the output port requires some explanation.

During the initialization process, a static array of destination ports (l2fwd_dst_ports[]) is filled such that for each source port, a destination port is assigned that is either the next or previous enabled port from the portmask. Naturally, the number of ports in the portmask must be even, otherwise, the application exits.

```
static void
l2fwd_simple_forward(struct rte_mbuf *m, unsigned portid)
```

```
{
    struct ether_hdr *eth;
    void *tmp;
    unsigned dst_port;

    dst_port = l2fwd_dst_ports[portid];

    eth = rte_pktmbuf_mtod(m, struct ether_hdr *);

    /* 02:00:00:00:00:xx */

    tmp = &eth->d_addr.addr_bytes[0];

    *((uint64_t *)tmp) = 0x000000000002 + ((uint64_t) dst_port << 40);

    /* src addr */

    ether_addr_copy(&l2fwd_ports_eth_addr[dst_port], &eth->s_addr);

    l2fwd_send_packet(m, (uint8_t) dst_port);
}
```

Then, the packet is sent using the l2fwd_send_packet (m, dst_port) function. For this test application, the processing is exactly the same for all packets arriving on the same RX port. Therefore, it would have been possible to call the l2fwd_send_burst() function directly from the main loop to send all the received packets on the same TX port, using the burst-oriented send function, which is more efficient.

However, in real-life applications (such as, L3 routing), packet N is not necessarily forwarded on the same port as packet N-1. The application is implemented to illustrate that, so the same approach can be reused in a more complex application.

The l2fwd_send_packet() function stores the packet in a per-lcore and per-txport table. If the table is full, the whole packets table is transmitted using the l2fwd_send_burst() function:

```
/* Send the packet on an output interface */

static int
l2fwd_send_packet(struct rte_mbuf *m, uint8_t port)
{
    unsigned lcore_id, len;
    struct lcore_queue_conf *qconf;

    lcore_id = rte_lcore_id();
    qconf = &lcore_queue_conf[lcore_id];
    len = qconf->tx_mbufs[port].len;
    qconf->tx_mbufs[port].m_table[len] = m;
    len++;

    /* enough pkts to be sent */

    if (unlikely(len == MAX_PKT_BURST)) {
        l2fwd_send_burst(qconf, MAX_PKT_BURST, port);
        len = 0;
    }

    qconf->tx_mbufs[port].len = len; return 0;
}
```

To ensure that no packets remain in the tables, each lcore does a draining of TX queue in its main loop. This technique introduces some latency when there are not many packets to send, however it improves performance:

```
cur_tsc = rte_rdtsc();

/*
 *   TX burst queue drain
 */

diff_tsc = cur_tsc - prev_tsc;

if (unlikely(diff_tsc > drain_tsc)) {
    for (portid = 0; portid < RTE_MAX_ETHPORTS; portid++) {
        if (qconf->tx_mbufs[portid].len == 0)
            continue;

        l2fwd_send_burst(&lcore_queue_conf[lcore_id], qconf->tx_mbufs[portid].len,␣
↪(uint8_t) portid);

        qconf->tx_mbufs[portid].len = 0;
    }

    /* if timer is enabled */

    if (timer_period > 0) {
        /* advance the timer */

        timer_tsc += diff_tsc;

        /* if timer has reached its timeout */

        if (unlikely(timer_tsc >= (uint64_t) timer_period)) {
            /* do this only on master core */

            if (lcore_id == rte_get_master_lcore()) {
                print_stats();

                /* reset the timer */
                timer_tsc = 0;
            }
        }
    }

    prev_tsc = cur_tsc;
}
```

3.16 L2 Forwarding Sample Application with Cache Allocation Technology (CAT)

Basic Forwarding sample application is a simple *skeleton* example of a forwarding application. It has been extended to make use of CAT via extended command line options and linking against the libpqos library.

It is intended as a demonstration of the basic components of a DPDK forwarding application and use of the libpqos library to program CAT. For more detailed implementations see the L2 and L3 forwarding sample applications.

CAT and Code Data Prioritization (CDP) features allow management of the CPU's last level cache. CAT introduces

classes of service (COS) that are essentially bitmasks. In current CAT implementations, a bit in a COS bitmask corresponds to one cache way in last level cache. A CPU core is always assigned to one of the CAT classes. By programming CPU core assignment and COS bitmasks, applications can be given exclusive, shared, or mixed access to the CPU's last level cache. CDP extends CAT so that there are two bitmasks per COS, one for data and one for code. The number of classes and number of valid bits in a COS bitmask is CPU model specific and COS bitmasks need to be contiguous. Sample code calls this bitmask `cbm` or capacity bitmask. By default, after reset, all CPU cores are assigned to COS 0 and all classes are programmed to allow fill into all cache ways. CDP is off by default.

For more information about CAT please see:

- https://github.com/01org/intel-cmt-cat

White paper demonstrating example use case:

- Increasing Platform Determinism with Platform Quality of Service for the Data Plane Development Kit

3.16.1 Compiling the Application

Requires `libpqos` from Intel's intel-cmt-cat software package hosted on GitHub repository. For installation notes, please see `README` file.

GIT:

- https://github.com/01org/intel-cmt-cat

To compile the application export the path to PQoS lib and the DPDK source tree and go to the example directory:

```
export PQOS_INSTALL_PATH=/path/to/libpqos
export RTE_SDK=/path/to/rte_sdk

cd ${RTE_SDK}/examples/l2fwd-cat
```

Set the target, for example:

```
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

See the *DPDK Getting Started* Guide for possible `RTE_TARGET` values.

Build the application as follows:

```
make
```

3.16.2 Running the Application

To run the example in a `linuxapp` environment and enable CAT on cpus 0-2:

```
./build/l2fwd-cat -l 1 -n 4 -- --l3ca="0x3@(0-2)"
```

or to enable CAT and CDP on cpus 1,3:

```
./build/l2fwd-cat -l 1 -n 4 -- --l3ca="(0x00C00,0x00300)@(1,3)"
```

If CDP is not supported it will fail with following error message:

```
PQOS: CDP requested but not supported.
PQOS: Requested CAT configuration is not valid!
PQOS: Shutting down PQoS library...
```

```
EAL: Error - exiting with code: 1
  Cause: PQOS: L3CA init failed!
```

The option to enable CAT is:

- `--l3ca='<common_cbm@cpus>[,<(code_cbm,data_cbm)@cpus>...]'`:

  where `cbm` stands for capacity bitmask and must be expressed in hexadecimal form.

  `common_cbm` is a single mask, for a CDP enabled system, a group of two masks (`code_cbm` and `data_cbm`) is used.

  `(` and `)` are necessary if it's a group.

  `cpus` could be a single digit/range or a group and must be expressed in decimal form.

  `(` and `)` are necessary if it's a group.

  e.g. `--l3ca='0x00F00@(1,3),0x0FF00@(4-6),0xF0000@7'`

  - cpus 1 and 3 share its 4 ways with cpus 4, 5 and 6;
  - cpus 4, 5 and 6 share half (4 out of 8 ways) of its L3 with cpus 1 and 3;
  - cpus 4, 5 and 6 have exclusive access to 4 out of 8 ways;
  - cpu 7 has exclusive access to all of its 4 ways;

  e.g. `--l3ca='(0x00C00,0x00300)@(1,3)'` for CDP enabled system

  - cpus 1 and 3 have access to 2 ways for code and 2 ways for data, code and data ways are not overlapping.

Refer to *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

To reset or list CAT configuration and control CDP please use `pqos` tool from Intel's intel-cmt-cat software package.

To enabled or disable CDP:

```
sudo ./pqos -S cdp-on

sudo ./pqos -S cdp-off
```

to reset CAT configuration:

```
sudo ./pqos -R
```

to list CAT config:

```
sudo ./pqos -s
```

For more info about `pqos` tool please see its man page or intel-cmt-cat wiki.

3.16.3 Explanation

The following sections provide an explanation of the main components of the code.

All DPDK library functions used in the sample code are prefixed with `rte_` and are explained in detail in the *DPDK API Documentation*.

The Main Function

The `main()` function performs the initialization and calls the execution threads for each lcore.

The first task is to initialize the Environment Abstraction Layer (EAL). The `argc` and `argv` arguments are provided to the `rte_eal_init()` function. The value returned is the number of parsed arguments:

```
int ret = rte_eal_init(argc, argv);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Error with EAL initialization\n");
```

The next task is to initialize the PQoS library and configure CAT. The `argc` and `argv` arguments are provided to the `cat_init()` function. The value returned is the number of parsed arguments:

```
int ret = cat_init(argc, argv);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "PQOS: L3CA init failed!\n");
```

`cat_init()` is a wrapper function which parses the command, validates the requested parameters and configures CAT accordingly.

Parsing of command line arguments is done in `parse_args(...)`. libpqos is then initialized with the `pqos_init(...)` call. Next, libpqos is queried for system CPU information and L3CA capabilities via `pqos_cap_get(...)` and `pqos_cap_get_type(..., PQOS_CAP_TYPE_L3CA, ...)` calls. When all capability and topology information is collected, the requested CAT configuration is validated. A check is then performed (on per socket basis) for a sufficient number of un-associated COS. COS are selected and configured via the `pqos_l3ca_set(...)` call. Finally, COS are associated to relevant CPUs via `pqos_l3ca_assoc_set(...)` calls.

`atexit(...)` is used to register `cat_exit(...)` to be called on a clean exit. `cat_exit(...)` performs a simple CAT clean-up, by associating COS 0 to all involved CPUs via `pqos_l3ca_assoc_set(...)` calls.

3.17 L3 Forwarding Sample Application

The L3 Forwarding application is a simple example of packet processing using the DPDK. The application performs L3 forwarding.

3.17.1 Overview

The application demonstrates the use of the hash and LPM libraries in the DPDK to implement packet forwarding. The initialization and run-time paths are very similar to those of the *L2 Forwarding Sample Application (in Real and Virtualized Environments)*. The main difference from the L2 Forwarding sample application is that the forwarding decision is made based on information read from the input packet.

The lookup method is either hash-based or LPM-based and is selected at run time. When the selected lookup method is hash-based, a hash object is used to emulate the flow classification stage. The hash object is used in correlation with a flow table to map each input packet to its flow at runtime.

The hash lookup key is represented by a DiffServ 5-tuple composed of the following fields read from the input packet: Source IP Address, Destination IP Address, Protocol, Source Port and Destination Port. The ID of the output interface for the input packet is read from the identified flow table entry. The set of flows used by the application is statically configured and loaded into the hash at initialization time. When the selected lookup method is LPM based, an LPM object is used to emulate the forwarding stage for IPv4 packets. The LPM object is used as the routing table to identify the next hop for each input packet at runtime.

The LPM lookup key is represented by the Destination IP Address field read from the input packet. The ID of the output interface for the input packet is the next hop returned by the LPM lookup. The set of LPM rules used by the application is statically configured and loaded into the LPM object at initialization time.

In the sample application, hash-based forwarding supports IPv4 and IPv6. LPM-based forwarding supports IPv4 only.

3.17.2 Compiling the Application

To compile the application:

1. Go to the sample application directory:

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}/examples/l3fwd
```

2. Set the target (a default target is used if not specified). For example:

```
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

   See the *DPDK Getting Started Guide* for possible RTE_TARGET values.

3. Build the application:

```
make
```

3.17.3 Running the Application

The application has a number of command line options:

```
./l3fwd [EAL options] -- -p PORTMASK
                         [-P]
                         [-E]
                         [-L]
                         --config(port,queue,lcore)[,(port,queue,lcore)]
                         [--eth-dest=X,MM:MM:MM:MM:MM:MM]
                         [--enable-jumbo [--max-pkt-len PKTLEN]]
                         [--no-numa]
                         [--hash-entry-num]
                         [--ipv6]
                         [--parse-ptype]
```

Where,

- `-p PORTMASK`: Hexadecimal bitmask of ports to configure
- `-P`: Optional, sets all ports to promiscuous mode so that packets are accepted regardless of the packet's Ethernet MAC destination address. Without this option, only packets with the Ethernet MAC destination address set to the Ethernet address of the port are accepted.
- `-E`: Optional, enable exact match.
- `-L`: Optional, enable longest prefix match.
- `--config (port,queue,lcore)[,(port,queue,lcore)]`: Determines which queues from which ports are mapped to which cores.
- `--eth-dest=X,MM:MM:MM:MM:MM:MM`: Optional, ethernet destination for port X.
- `--enable-jumbo`: Optional, enables jumbo frames.

- `--max-pkt-len`: Optional, under the premise of enabling jumbo, maximum packet length in decimal (64-9600).
- `--no-numa`: Optional, disables numa awareness.
- `--hash-entry-num`: Optional, specifies the hash entry number in hexadecimal to be setup.
- `--ipv6`: Optional, set if running ipv6 packets.
- `--parse-ptype`: Optional, set to use software to analyze packet type. Without this option, hardware will check the packet type.

For example, consider a dual processor socket platform with 8 physical cores, where cores 0-7 and 16-23 appear on socket 0, while cores 8-15 and 24-31 appear on socket 1.

To enable L3 forwarding between two ports, assuming that both ports are in the same socket, using two cores, cores 1 and 2, (which are in the same socket too), use the following command:

```
./build/l3fwd -l 1,2 -n 4 -- -p 0x3 --config="(0,0,1),(1,0,2)"
```

In this command:

- The -l option enables cores 1, 2
- The -p option enables ports 0 and 1
- The –config option enables one queue on each port and maps each (port,queue) pair to a specific core. The following table shows the mapping in this example:

Port	Queue	lcore	Description
0	0	1	Map queue 0 from port 0 to lcore 1.
1	0	2	Map queue 0 from port 1 to lcore 2.

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

3.17.4 Explanation

The following sections provide some explanation of the sample application code. As mentioned in the overview section, the initialization and run-time paths are very similar to those of the *L2 Forwarding Sample Application (in Real and Virtualized Environments)*. The following sections describe aspects that are specific to the L3 Forwarding sample application.

Hash Initialization

The hash object is created and loaded with the pre-configured entries read from a global array, and then generate the expected 5-tuple as key to keep consistence with those of real flow for the convenience to execute hash performance test on 4M/8M/16M flows.

Note: The Hash initialization will setup both ipv4 and ipv6 hash table, and populate the either table depending on the value of variable ipv6. To support the hash performance test with up to 8M single direction flows/16M bi-direction flows, populate_ipv4_many_flow_into_table() function will populate the hash table with specified hash table entry number(default 4M).

Note: Value of global variable ipv6 can be specified with –ipv6 in the command line. Value of global variable hash_entry_number, which is used to specify the total hash entry number for all used ports in hash performance test, can be specified with –hash-entry-num VALUE in command line, being its default value 4.

```
#if (APP_LOOKUP_METHOD == APP_LOOKUP_EXACT_MATCH)

    static void
    setup_hash(int socketid)
    {
        // ...

        if (hash_entry_number != HASH_ENTRY_NUMBER_DEFAULT) {
            if (ipv6 == 0) {
                /* populate the ipv4 hash */
                populate_ipv4_many_flow_into_table(ipv4_l3fwd_lookup_struct[socketid],
↪ hash_entry_number);
            } else {
                /* populate the ipv6 hash */
                populate_ipv6_many_flow_into_table( ipv6_l3fwd_lookup_
↪struct[socketid], hash_entry_number);
            }
        } else
            if (ipv6 == 0) {
                /* populate the ipv4 hash */
                populate_ipv4_few_flow_into_table(ipv4_l3fwd_lookup_struct[socketid]);
            } else {
                /* populate the ipv6 hash */
                populate_ipv6_few_flow_into_table(ipv6_l3fwd_lookup_struct[socketid]);
            }
        }
    }
#endif
```

LPM Initialization

The LPM object is created and loaded with the pre-configured entries read from a global array.

```
#if (APP_LOOKUP_METHOD == APP_LOOKUP_LPM)

static void
setup_lpm(int socketid)
{
    unsigned i;
    int ret;
    char s[64];

    /* create the LPM table */

    snprintf(s, sizeof(s), "IPV4_L3FWD_LPM_%d", socketid);

    ipv4_l3fwd_lookup_struct[socketid] = rte_lpm_create(s, socketid, IPV4_L3FWD_LPM_
↪MAX_RULES, 0);

    if (ipv4_l3fwd_lookup_struct[socketid] == NULL)
        rte_exit(EXIT_FAILURE, "Unable to create the l3fwd LPM table"
```

```
            " on socket %d\n", socketid);

    /* populate the LPM table */

    for (i = 0; i < IPV4_L3FWD_NUM_ROUTES; i++) {
        /* skip unused ports */

        if ((1 << ipv4_l3fwd_route_array[i].if_out & enabled_port_mask) == 0)
            continue;

        ret = rte_lpm_add(ipv4_l3fwd_lookup_struct[socketid], ipv4_l3fwd_route_
↪array[i].ip,
                                ipv4_l3fwd_route_array[i].depth, ipv4_l3fwd_route_
↪array[i].if_out);

        if (ret < 0) {
            rte_exit(EXIT_FAILURE, "Unable to add entry %u to the "
                    "l3fwd LPM table on socket %d\n", i, socketid);
        }

        printf("LPM: Adding route 0x%08x / %d (%d)\n",
            (unsigned)ipv4_l3fwd_route_array[i].ip, ipv4_l3fwd_route_array[i].depth,␣
↪ipv4_l3fwd_route_array[i].if_out);
    }
}
#endif
```

Packet Forwarding for Hash-based Lookups

For each input packet, the packet forwarding operation is done by the l3fwd_simple_forward() or simple_ipv4_fwd_4pkts() function for IPv4 packets or the simple_ipv6_fwd_4pkts() function for IPv6 packets. The l3fwd_simple_forward() function provides the basic functionality for both IPv4 and IPv6 packet forwarding for any number of burst packets received, and the packet forwarding decision (that is, the identification of the output interface for the packet) for hash-based lookups is done by the get_ipv4_dst_port() or get_ipv6_dst_port() function. The get_ipv4_dst_port() function is shown below:

```
static inline uint8_t
get_ipv4_dst_port(void *ipv4_hdr, uint8_t portid, lookup_struct_t *ipv4_l3fwd_lookup_
↪struct)
{
    int ret = 0;
    union ipv4_5tuple_host key;

    ipv4_hdr = (uint8_t *)ipv4_hdr + offsetof(struct ipv4_hdr, time_to_live);

    m128i data = _mm_loadu_si128(( m128i*)(ipv4_hdr));

    /* Get 5 tuple: dst port, src port, dst IP address, src IP address and protocol */

    key.xmm = _mm_and_si128(data, mask0);

    /* Find destination port */

    ret = rte_hash_lookup(ipv4_l3fwd_lookup_struct, (const void *)&key);

    return (uint8_t)((ret < 0)? portid : ipv4_l3fwd_out_if[ret]);
```

```
}
```

The get_ipv6_dst_port() function is similar to the get_ipv4_dst_port() function.

The simple_ipv4_fwd_4pkts() and simple_ipv6_fwd_4pkts() function are optimized for continuous 4 valid ipv4 and ipv6 packets, they leverage the multiple buffer optimization to boost the performance of forwarding packets with the exact match on hash table. The key code snippet of simple_ipv4_fwd_4pkts() is shown below:

```
static inline void
simple_ipv4_fwd_4pkts(struct rte_mbuf* m[4], uint8_t portid, struct lcore_conf *qconf)
{
    // ...

    data[0] = _mm_loadu_si128(( m128i*)(rte_pktmbuf_mtod(m[0], unsigned char *) +␣
↪sizeof(struct ether_hdr) + offsetof(struct ipv4_hdr, time_to_live)));
    data[1] = _mm_loadu_si128(( m128i*)(rte_pktmbuf_mtod(m[1], unsigned char *) +␣
↪sizeof(struct ether_hdr) + offsetof(struct ipv4_hdr, time_to_live)));
    data[2] = _mm_loadu_si128(( m128i*)(rte_pktmbuf_mtod(m[2], unsigned char *) +␣
↪sizeof(struct ether_hdr) + offsetof(struct ipv4_hdr, time_to_live)));
    data[3] = _mm_loadu_si128(( m128i*)(rte_pktmbuf_mtod(m[3], unsigned char *) +␣
↪sizeof(struct ether_hdr) + offsetof(struct ipv4_hdr, time_to_live)));

    key[0].xmm = _mm_and_si128(data[0], mask0);
    key[1].xmm = _mm_and_si128(data[1], mask0);
    key[2].xmm = _mm_and_si128(data[2], mask0);
    key[3].xmm = _mm_and_si128(data[3], mask0);

    const void *key_array[4] = {&key[0], &key[1], &key[2],&key[3]};

    rte_hash_lookup_bulk(qconf->ipv4_lookup_struct, &key_array[0], 4, ret);

    dst_port[0] = (ret[0] < 0)? portid:ipv4_l3fwd_out_if[ret[0]];
    dst_port[1] = (ret[1] < 0)? portid:ipv4_l3fwd_out_if[ret[1]];
    dst_port[2] = (ret[2] < 0)? portid:ipv4_l3fwd_out_if[ret[2]];
    dst_port[3] = (ret[3] < 0)? portid:ipv4_l3fwd_out_if[ret[3]];

    // ...
}
```

The simple_ipv6_fwd_4pkts() function is similar to the simple_ipv4_fwd_4pkts() function.

Known issue: IP packets with extensions or IP packets which are not TCP/UDP cannot work well at this mode.

Packet Forwarding for LPM-based Lookups

For each input packet, the packet forwarding operation is done by the l3fwd_simple_forward() function, but the packet forwarding decision (that is, the identification of the output interface for the packet) for LPM-based lookups is done by the get_ipv4_dst_port() function below:

```
static inline uint8_t
get_ipv4_dst_port(struct ipv4_hdr *ipv4_hdr, uint8_t portid, lookup_struct_t *ipv4_
↪l3fwd_lookup_struct)
{
    uint8_t next_hop;

    return (uint8_t) ((rte_lpm_lookup(ipv4_l3fwd_lookup_struct, rte_be_to_cpu_32(ipv4_
↪hdr->dst_addr), &next_hop) == 0)? next_hop : portid);
```

```
}
```

3.18 L3 Forwarding with Power Management Sample Application

3.18.1 Introduction

The L3 Forwarding with Power Management application is an example of power-aware packet processing using the DPDK. The application is based on existing L3 Forwarding sample application, with the power management algorithms to control the P-states and C-states of the Intel processor via a power management library.

3.18.2 Overview

The application demonstrates the use of the Power libraries in the DPDK to implement packet forwarding. The initialization and run-time paths are very similar to those of the *L3 Forwarding Sample Application*. The main difference from the L3 Forwarding sample application is that this application introduces power-aware optimization algorithms by leveraging the Power library to control P-state and C-state of processor based on packet load.

The DPDK includes poll-mode drivers to configure Intel NIC devices and their receive (Rx) and transmit (Tx) queues. The design principle of this PMD is to access the Rx and Tx descriptors directly without any interrupts to quickly receive, process and deliver packets in the user space.

In general, the DPDK executes an endless packet processing loop on dedicated IA cores that include the following steps:

- Retrieve input packets through the PMD to poll Rx queue
- Process each received packet or provide received packets to other processing cores through software queues
- Send pending output packets to Tx queue through the PMD

In this way, the PMD achieves better performance than a traditional interrupt-mode driver, at the cost of keeping cores active and running at the highest frequency, hence consuming the maximum power all the time. However, during the period of processing light network traffic, which happens regularly in communication infrastructure systems due to well-known "tidal effect", the PMD is still busy waiting for network packets, which wastes a lot of power.

Processor performance states (P-states) are the capability of an Intel processor to switch between different supported operating frequencies and voltages. If configured correctly, according to system workload, this feature provides power savings. CPUFreq is the infrastructure provided by the Linux* kernel to control the processor performance state capability. CPUFreq supports a user space governor that enables setting frequency via manipulating the virtual file device from a user space application. The Power library in the DPDK provides a set of APIs for manipulating a virtual file device to allow user space application to set the CPUFreq governor and set the frequency of specific cores.

This application includes a P-state power management algorithm to generate a frequency hint to be sent to CPUFreq. The algorithm uses the number of received and available Rx packets on recent polls to make a heuristic decision to scale frequency up/down. Specifically, some thresholds are checked to see whether a specific core running an DPDK polling thread needs to increase frequency a step up based on the near to full trend of polled Rx queues. Also, it decreases frequency a step if packet processed per loop is far less than the expected threshold or the thread's sleeping time exceeds a threshold.

C-States are also known as sleep states. They allow software to put an Intel core into a low power idle state from which it is possible to exit via an event, such as an interrupt. However, there is a tradeoff between the power consumed in the idle state and the time required to wake up from the idle state (exit latency). Therefore, as you go into deeper C-states, the power consumed is lower but the exit latency is increased. Each C-state has a target residency. It is essential that when entering into a C-state, the core remains in this C-state for at least as long as the target residency in order to fully realize the benefits of entering the C-state. CPUIdle is the infrastructure provide by the Linux kernel to control the

processor C-state capability. Unlike CPUFreq, CPUIdle does not provide a mechanism that allows the application to change C-state. It actually has its own heuristic algorithms in kernel space to select target C-state to enter by executing privileged instructions like HLT and MWAIT, based on the speculative sleep duration of the core. In this application, we introduce a heuristic algorithm that allows packet processing cores to sleep for a short period if there is no Rx packet received on recent polls. In this way, CPUIdle automatically forces the corresponding cores to enter deeper C-states instead of always running to the C0 state waiting for packets.

Note: To fully demonstrate the power saving capability of using C-states, it is recommended to enable deeper C3 and C6 states in the BIOS during system boot up.

3.18.3 Compiling the Application

To compile the application:

1. Go to the sample application directory:

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}/examples/l3fwd-power
```

2. Set the target (a default target is used if not specified). For example:

```
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

   See the *DPDK Getting Started Guide* for possible RTE_TARGET values.

3. Build the application:

```
make
```

3.18.4 Running the Application

The application has a number of command line options:

```
./build/l3fwd_power [EAL options] -- -p PORTMASK [-P]  --config(port,queue,lcore)[,
↪(port,queue,lcore)] [--enable-jumbo [--max-pkt-len PKTLEN]] [--no-numa]
```

where,

- -p PORTMASK: Hexadecimal bitmask of ports to configure
- -P: Sets all ports to promiscuous mode so that packets are accepted regardless of the packet's Ethernet MAC destination address. Without this option, only packets with the Ethernet MAC destination address set to the Ethernet address of the port are accepted.
- –config (port,queue,lcore)[,(port,queue,lcore)]: determines which queues from which ports are mapped to which cores.
- –enable-jumbo: optional, enables jumbo frames
- –max-pkt-len: optional, maximum packet length in decimal (64-9600)
- –no-numa: optional, disables numa awareness

See *L3 Forwarding Sample Application* for details. The L3fwd-power example reuses the L3fwd command line options.

3.18.5 Explanation

The following sections provide some explanation of the sample application code. As mentioned in the overview section, the initialization and run-time paths are identical to those of the L3 forwarding application. The following sections describe aspects that are specific to the L3 Forwarding with Power Management sample application.

Power Library Initialization

The Power library is initialized in the main routine. It changes the P-state governor to userspace for specific cores that are under control. The Timer library is also initialized and several timers are created later on, responsible for checking if it needs to scale down frequency at run time by checking CPU utilization statistics.

Note: Only the power management related initialization is shown.

```
int main(int argc, char **argv)
{
    struct lcore_conf *qconf;
    int ret;
    unsigned nb_ports;
    uint16_t queueid;
    unsigned lcore_id;
    uint64_t hz;
    uint32_t n_tx_queue, nb_lcores;
    uint8_t portid, nb_rx_queue, queue, socketid;

    // ...

    /* init RTE timer library to be used to initialize per-core timers */

    rte_timer_subsystem_init();

    // ...


    /* per-core initialization */

    for (lcore_id = 0; lcore_id < RTE_MAX_LCORE; lcore_id++) {
        if (rte_lcore_is_enabled(lcore_id) == 0)
            continue;

        /* init power management library for a specified core */

        ret = rte_power_init(lcore_id);
        if (ret)
            rte_exit(EXIT_FAILURE, "Power management library "
                "initialization failed on core%d\n", lcore_id);

        /* init timer structures for each enabled lcore */

        rte_timer_init(&power_timers[lcore_id]);

        hz = rte_get_hpet_hz();

        rte_timer_reset(&power_timers[lcore_id], hz/TIMER_NUMBER_PER_SECOND, SINGLE,␣
↪lcore_id, power_timer_cb, NULL);
```

```
        // ...
    }

    // ...
}
```

Monitoring Loads of Rx Queues

In general, the polling nature of the DPDK prevents the OS power management subsystem from knowing if the network load is actually heavy or light. In this sample, sampling network load work is done by monitoring received and available descriptors on NIC Rx queues in recent polls. Based on the number of returned and available Rx descriptors, this example implements algorithms to generate frequency scaling hints and speculative sleep duration, and use them to control P-state and C-state of processors via the power management library. Frequency (P-state) control and sleep state (C-state) control work individually for each logical core, and the combination of them contributes to a power efficient packet processing solution when serving light network loads.

The rte_eth_rx_burst() function and the newly-added rte_eth_rx_queue_count() function are used in the endless packet processing loop to return the number of received and available Rx descriptors. And those numbers of specific queue are passed to P-state and C-state heuristic algorithms to generate hints based on recent network load trends.

Note: Only power control related code is shown.

```
static
attribute ((noreturn)) int main_loop( attribute ((unused)) void *dummy)
{
    // ...

    while (1) {
    // ...

    /**
     * Read packet from RX queues
     */

    lcore_scaleup_hint = FREQ_CURRENT;
    lcore_rx_idle_count = 0;

    for (i = 0; i < qconf->n_rx_queue; ++i)
    {
        rx_queue = &(qconf->rx_queue_list[i]);
        rx_queue->idle_hint = 0;
        portid = rx_queue->port_id;
        queueid = rx_queue->queue_id;

        nb_rx = rte_eth_rx_burst(portid, queueid, pkts_burst, MAX_PKT_BURST);
        stats[lcore_id].nb_rx_processed += nb_rx;

        if (unlikely(nb_rx == 0)) {
            /**
             * no packet received from rx queue, try to
             * sleep for a while forcing CPU enter deeper
             * C states.
             */
```

```
            rx_queue->zero_rx_packet_count++;

            if (rx_queue->zero_rx_packet_count <= MIN_ZERO_POLL_COUNT)
                continue;

            rx_queue->idle_hint = power_idle_heuristic(rx_queue->zero_rx_packet_
↪count);
            lcore_rx_idle_count++;
        } else {
            rx_ring_length = rte_eth_rx_queue_count(portid, queueid);

            rx_queue->zero_rx_packet_count = 0;

            /**
             * do not scale up frequency immediately as
             * user to kernel space communication is costly
             * which might impact packet I/O for received
             * packets.
             */

            rx_queue->freq_up_hint = power_freq_scaleup_heuristic(lcore_id, rx_ring_
↪length);
        }

        /* Prefetch and forward packets */

        // ...
    }

    if (likely(lcore_rx_idle_count != qconf->n_rx_queue)) {
        for (i = 1, lcore_scaleup_hint = qconf->rx_queue_list[0].freq_up_hint; i <␣
↪qconf->n_rx_queue; ++i) {
            x_queue = &(qconf->rx_queue_list[i]);

            if (rx_queue->freq_up_hint > lcore_scaleup_hint)

                lcore_scaleup_hint = rx_queue->freq_up_hint;
        }

        if (lcore_scaleup_hint == FREQ_HIGHEST)

            rte_power_freq_max(lcore_id);

        else if (lcore_scaleup_hint == FREQ_HIGHER)
            rte_power_freq_up(lcore_id);
        } else {
            /**
             *  All Rx queues empty in recent consecutive polls,
             *  sleep in a conservative manner, meaning sleep as
             * less as possible.
             */

            for (i = 1, lcore_idle_hint = qconf->rx_queue_list[0].idle_hint; i <␣
↪qconf->n_rx_queue; ++i) {
                rx_queue = &(qconf->rx_queue_list[i]);
                if (rx_queue->idle_hint < lcore_idle_hint)
                    lcore_idle_hint = rx_queue->idle_hint;
```

```
            }

            if ( lcore_idle_hint < SLEEP_GEAR1_THRESHOLD)
                /**
                 *   execute "pause" instruction to avoid context
                 *   switch for short sleep.
                 */
                rte_delay_us(lcore_idle_hint);
            else
                /* long sleep force ruining thread to suspend */
                usleep(lcore_idle_hint);

            stats[lcore_id].sleep_time += lcore_idle_hint;
        }
    }
}
```

P-State Heuristic Algorithm

The power_freq_scaleup_heuristic() function is responsible for generating a frequency hint for the specified logical core according to available descriptor number returned from rte_eth_rx_queue_count(). On every poll for new packets, the length of available descriptor on an Rx queue is evaluated, and the algorithm used for frequency hinting is as follows:

- If the size of available descriptors exceeds 96, the maximum frequency is hinted.
- If the size of available descriptors exceeds 64, a trend counter is incremented by 100.
- If the length of the ring exceeds 32, the trend counter is incremented by 1.
- When the trend counter reached 10000 the frequency hint is changed to the next higher frequency.

Note: The assumption is that the Rx queue size is 128 and the thresholds specified above must be adjusted accordingly based on actual hardware Rx queue size, which are configured via the rte_eth_rx_queue_setup() function.

In general, a thread needs to poll packets from multiple Rx queues. Most likely, different queue have different load, so they would return different frequency hints. The algorithm evaluates all the hints and then scales up frequency in an aggressive manner by scaling up to highest frequency as long as one Rx queue requires. In this way, we can minimize any negative performance impact.

On the other hand, frequency scaling down is controlled in the timer callback function. Specifically, if the sleep times of a logical core indicate that it is sleeping more than 25% of the sampling period, or if the average packet per iteration is less than expectation, the frequency is decreased by one step.

C-State Heuristic Algorithm

Whenever recent rte_eth_rx_burst() polls return 5 consecutive zero packets, an idle counter begins incrementing for each successive zero poll. At the same time, the function power_idle_heuristic() is called to generate speculative sleep duration in order to force logical to enter deeper sleeping C-state. There is no way to control C- state directly, and the CPUIdle subsystem in OS is intelligent enough to select C-state to enter based on actual sleep period time of giving logical core. The algorithm has the following sleeping behavior depending on the idle counter:

- If idle count less than 100, the counter value is used as a microsecond sleep value through rte_delay_us() which execute pause instructions to avoid costly context switch but saving power at the same time.

- If idle count is between 100 and 999, a fixed sleep interval of 100 μs is used. A 100 μs sleep interval allows the core to enter the C1 state while keeping a fast response time in case new traffic arrives.
- If idle count is greater than 1000, a fixed sleep value of 1 ms is used until the next timer expiration is used. This allows the core to enter the C3/C6 states.

Note: The thresholds specified above need to be adjusted for different Intel processors and traffic profiles.

If a thread polls multiple Rx queues and different queue returns different sleep duration values, the algorithm controls the sleep time in a conservative manner by sleeping for the least possible time in order to avoid a potential performance impact.

3.19 L3 Forwarding with Access Control Sample Application

The L3 Forwarding with Access Control application is a simple example of packet processing using the DPDK. The application performs a security check on received packets. Packets that are in the Access Control List (ACL), which is loaded during initialization, are dropped. Others are forwarded to the correct port.

3.19.1 Overview

The application demonstrates the use of the ACL library in the DPDK to implement access control and packet L3 forwarding. The application loads two types of rules at initialization:

- Route information rules, which are used for L3 forwarding
- Access Control List (ACL) rules that blacklist (or block) packets with a specific characteristic

When packets are received from a port, the application extracts the necessary information from the TCP/IP header of the received packet and performs a lookup in the rule database to figure out whether the packets should be dropped (in the ACL range) or forwarded to desired ports. The initialization and run-time paths are similar to those of the *L3 Forwarding Sample Application*. However, there are significant differences in the two applications. For example, the original L3 forwarding application uses either LPM or an exact match algorithm to perform forwarding port lookup, while this application uses the ACL library to perform both ACL and route entry lookup. The following sections provide more detail.

Classification for both IPv4 and IPv6 packets is supported in this application. The application also assumes that all the packets it processes are TCP/UDP packets and always extracts source/destination port information from the packets.

Tuple Packet Syntax

The application implements packet classification for the IPv4/IPv6 5-tuple syntax specifically. The 5-tuple syntax consist of a source IP address, a destination IP address, a source port, a destination port and a protocol identifier. The fields in the 5-tuple syntax have the following formats:

- **Source IP address and destination IP address** : Each is either a 32-bit field (for IPv4), or a set of 4 32-bit fields (for IPv6) represented by a value and a mask length. For example, an IPv4 range of 192.168.1.0 to 192.168.1.255 could be represented by a value = [192, 168, 1, 0] and a mask length = 24.
- **Source port and destination port** : Each is a 16-bit field, represented by a lower start and a higher end. For example, a range of ports 0 to 8192 could be represented by lower = 0 and higher = 8192.
- **Protocol identifier** : An 8-bit field, represented by a value and a mask, that covers a range of values. To verify that a value is in the range, use the following expression: "(VAL & mask) == value"

The trick in how to represent a range with a mask and value is as follows. A range can be enumerated in binary numbers with some bits that are never changed and some bits that are dynamically changed. Set those bits that dynamically changed in mask and value with 0. Set those bits that never changed in the mask with 1, in value with number expected. For example, a range of 6 to 7 is enumerated as 0b110 and 0b111. Bit 1-7 are bits never changed and bit 0 is the bit dynamically changed. Therefore, set bit 0 in mask and value with 0, set bits 1-7 in mask with 1, and bits 1-7 in value with number 0b11. So, mask is 0xfe, value is 0x6.

Note: The library assumes that each field in the rule is in LSB or Little Endian order when creating the database. It internally converts them to MSB or Big Endian order. When performing a lookup, the library assumes the input is in MSB or Big Endian order.

Access Rule Syntax

In this sample application, each rule is a combination of the following:

- 5-tuple field: This field has a format described in Section.
- priority field: A weight to measure the priority of the rules. The rule with the higher priority will ALWAYS be returned if the specific input has multiple matches in the rule database. Rules with lower priority will NEVER be returned in any cases.
- userdata field: A user-defined field that could be any value. It can be the forwarding port number if the rule is a route table entry or it can be a pointer to a mapping address if the rule is used for address mapping in the NAT application. The key point is that it is a useful reserved field for user convenience.

ACL and Route Rules

The application needs to acquire ACL and route rules before it runs. Route rules are mandatory, while ACL rules are optional. To simplify the complexity of the priority field for each rule, all ACL and route entries are assumed to be in the same file. To read data from the specified file successfully, the application assumes the following:

- Each rule occupies a single line.
- Only the following four rule line types are valid in this application:
- ACL rule line, which starts with a leading character '@'
- Route rule line, which starts with a leading character 'R'
- Comment line, which starts with a leading character '#'
- Empty line, which consists of a space, form-feed ('f'), newline ('n'), carriage return ('r'), horizontal tab ('t'), or vertical tab ('v').

Other lines types are considered invalid.

- Rules are organized in descending order of priority, which means rules at the head of the file always have a higher priority than those further down in the file.
- A typical IPv4 ACL rule line should have a format as shown below:

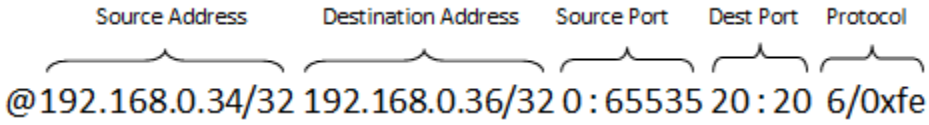


Fig. 3.9: A typical IPv4 ACL rule

IPv4 addresses are specified in CIDR format as specified in RFC 4632. They consist of the dot notation for the address and a prefix length separated by '/'. For example, 192.168.0.34/32, where the address is 192.168.0.34 and the prefix length is 32.

Ports are specified as a range of 16-bit numbers in the format MIN:MAX, where MIN and MAX are the inclusive minimum and maximum values of the range. The range 0:65535 represents all possible ports in a range. When MIN and MAX are the same value, a single port is represented, for example, 20:20.

The protocol identifier is an 8-bit value and a mask separated by '/'. For example: 6/0xfe matches protocol values 6 and 7.

- Route rules start with a leading character 'R' and have the same format as ACL rules except an extra field at the tail that indicates the forwarding port number.

Rules File Example

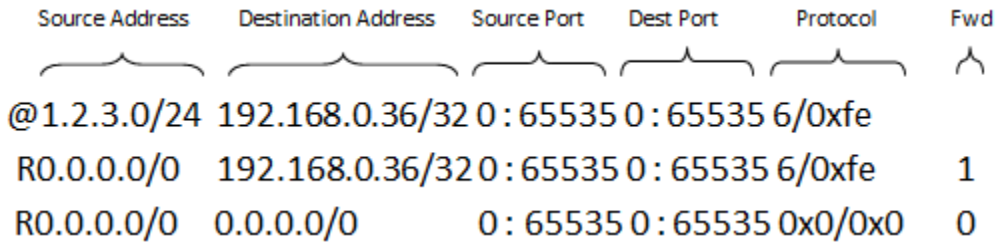


Fig. 3.10: Rules example

Each rule is explained as follows:

- Rule 1 (the first line) tells the application to drop those packets with source IP address = [1.2.3.*], destination IP address = [192.168.0.36], protocol = [6]/[7]
- Rule 2 (the second line) is similar to Rule 1, except the source IP address is ignored. It tells the application to forward packets with destination IP address = [192.168.0.36], protocol = [6]/[7], destined to port 1.
- Rule 3 (the third line) tells the application to forward all packets to port 0. This is something like a default route entry.

As described earlier, the application assume rules are listed in descending order of priority, therefore Rule 1 has the highest priority, then Rule 2, and finally, Rule 3 has the lowest priority.

Consider the arrival of the following three packets:

- Packet 1 has source IP address = [1.2.3.4], destination IP address = [192.168.0.36], and protocol = [6]
- Packet 2 has source IP address = [1.2.4.4], destination IP address = [192.168.0.36], and protocol = [6]
- Packet 3 has source IP address = [1.2.3.4], destination IP address = [192.168.0.36], and protocol = [8]

Observe that:

- Packet 1 matches all of the rules
- Packet 2 matches Rule 2 and Rule 3
- Packet 3 only matches Rule 3

For priority reasons, Packet 1 matches Rule 1 and is dropped. Packet 2 matches Rule 2 and is forwarded to port 1. Packet 3 matches Rule 3 and is forwarded to port 0.

For more details on the rule file format, please refer to rule_ipv4.db and rule_ipv6.db files (inside <RTE_SDK>/examples/l3fwd-acl/).

Application Phases

Once the application starts, it transitions through three phases:

- **Initialization Phase** - Perform the following tasks:
- Parse command parameters. Check the validity of rule file(s) name(s), number of logical cores, receive and transmit queues. Bind ports, queues and logical cores. Check ACL search options, and so on.
- Call Environmental Abstraction Layer (EAL) and Poll Mode Driver (PMD) functions to initialize the environment and detect possible NICs. The EAL creates several threads and sets affinity to a specific hardware thread CPU based on the configuration specified by the command line arguments.
- Read the rule files and format the rules into the representation that the ACL library can recognize. Call the ACL library function to add the rules into the database and compile them as a trie of pattern sets. Note that application maintains a separate AC contexts for IPv4 and IPv6 rules.
- **Runtime Phase** - Process the incoming packets from a port. Packets are processed in three steps:
  - Retrieval: Gets a packet from the receive queue. Each logical core may process several queues for different ports. This depends on the configuration specified by command line arguments.
  - Lookup: Checks that the packet type is supported (IPv4/IPv6) and performs a 5-tuple lookup over corresponding AC context. If an ACL rule is matched, the packets will be dropped and return back to step 1. If a route rule is matched, it indicates the packet is not in the ACL list and should be forwarded. If there is no matches for the packet, then the packet is dropped.
  - Forwarding: Forwards the packet to the corresponding port.
- **Final Phase** - Perform the following tasks:

  Calls the EAL, PMD driver and ACL library to free resource, then quits.

3.19.2 Compiling the Application

To compile the application:

1. Go to the sample application directory:

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}/examples/l3fwd-acl
```

2. Set the target (a default target is used if not specified). For example:

```
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

   See the *DPDK IPL Getting Started Guide* for possible RTE_TARGET values.

3. Build the application:

```
make
```

3.19.3 Running the Application

The application has a number of command line options:

```
./build/l3fwd-acl [EAL options] -- -p PORTMASK [-P] --config(port,queue,lcore)[,(port,
↪queue,lcore)] --rule_ipv4 FILENAME rule_ipv6 FILENAME [--scalar] [--enable-jumbo [--
↪max-pkt-len PKTLEN]] [--no-numa]
```

where,

- -p PORTMASK: Hexadecimal bitmask of ports to configure
- -P: Sets all ports to promiscuous mode so that packets are accepted regardless of the packet's Ethernet MAC destination address. Without this option, only packets with the Ethernet MAC destination address set to the Ethernet address of the port are accepted.
- –config (port,queue,lcore)[,(port,queue,lcore)]: determines which queues from which ports are mapped to which cores
- –rule_ipv4 FILENAME: Specifies the IPv4 ACL and route rules file
- –rule_ipv6 FILENAME: Specifies the IPv6 ACL and route rules file
- –scalar: Use a scalar function to perform rule lookup
- –enable-jumbo: optional, enables jumbo frames
- –max-pkt-len: optional, maximum packet length in decimal (64-9600)
- –no-numa: optional, disables numa awareness

For example, consider a dual processor socket platform with 8 physical cores, where cores 0-7 and 16-23 appear on socket 0, while cores 8-15 and 24-31 appear on socket 1.

To enable L3 forwarding between two ports, assuming that both ports are in the same socket, using two cores, cores 1 and 2, (which are in the same socket too), use the following command:

```
./build/l3fwd-acl -l 1,2 -n 4 -- -p 0x3 --config="(0,0,1),(1,0,2)" --rule_ipv4="./
↪rule_ipv4.db" -- rule_ipv6="./rule_ipv6.db" --scalar
```

In this command:

- The -l option enables cores 1, 2
- The -p option enables ports 0 and 1
- The –config option enables one queue on each port and maps each (port,queue) pair to a specific core. The following table shows the mapping in this example:

Port	Queue	lcore	Description
0	0	1	Map queue 0 from port 0 to lcore 1.
1	0	2	Map queue 0 from port 1 to lcore 2.

- The –rule_ipv4 option specifies the reading of IPv4 rules sets from the ./ rule_ipv4.db file.
- The –rule_ipv6 option specifies the reading of IPv6 rules sets from the ./ rule_ipv6.db file.
- The –scalar option specifies the performing of rule lookup with a scalar function.

3.19.4 Explanation

The following sections provide some explanation of the sample application code. The aspects of port, device and CPU configuration are similar to those of the *L3 Forwarding Sample Application*. The following sections describe aspects that are specific to L3 forwarding with access control.

Parse Rules from File

As described earlier, both ACL and route rules are assumed to be saved in the same file. The application parses the rules from the file and adds them to the database by calling the ACL library function. It ignores empty and comment

lines, and parses and validates the rules it reads. If errors are detected, the application exits with messages to identify the errors encountered.

The application needs to consider the userdata and priority fields. The ACL rules save the index to the specific rules in the userdata field, while route rules save the forwarding port number. In order to differentiate the two types of rules, ACL rules add a signature in the userdata field. As for the priority field, the application assumes rules are organized in descending order of priority. Therefore, the code only decreases the priority number with each rule it parses.

Setting Up the ACL Context

For each supported AC rule format (IPv4 5-tuple, IPv6 6-tuple) application creates a separate context handler from the ACL library for each CPU socket on the board and adds parsed rules into that context.

Note, that for each supported rule type, application needs to calculate the expected offset of the fields from the start of the packet. That's why only packets with fixed IPv4/ IPv6 header are supported. That allows to perform ACL classify straight over incoming packet buffer - no extra protocol field retrieval need to be performed.

Subsequently, the application checks whether NUMA is enabled. If it is, the application records the socket IDs of the CPU cores involved in the task.

Finally, the application creates contexts handler from the ACL library, adds rules parsed from the file into the database and build an ACL trie. It is important to note that the application creates an independent copy of each database for each socket CPU involved in the task to reduce the time for remote memory access.

3.20 L3 Forwarding in a Virtualization Environment Sample Application

The L3 Forwarding in a Virtualization Environment sample application is a simple example of packet processing using the DPDK. The application performs L3 forwarding that takes advantage of Single Root I/O Virtualization (SR-IOV) features in a virtualized environment.

3.20.1 Overview

The application demonstrates the use of the hash and LPM libraries in the DPDK to implement packet forwarding. The initialization and run-time paths are very similar to those of the *L3 Forwarding Sample Application*. The forwarding decision is taken based on information read from the input packet.

The lookup method is either hash-based or LPM-based and is selected at compile time. When the selected lookup method is hash-based, a hash object is used to emulate the flow classification stage. The hash object is used in correlation with the flow table to map each input packet to its flow at runtime.

The hash lookup key is represented by the DiffServ 5-tuple composed of the following fields read from the input packet: Source IP Address, Destination IP Address, Protocol, Source Port and Destination Port. The ID of the output interface for the input packet is read from the identified flow table entry. The set of flows used by the application is statically configured and loaded into the hash at initialization time. When the selected lookup method is LPM based, an LPM object is used to emulate the forwarding stage for IPv4 packets. The LPM object is used as the routing table to identify the next hop for each input packet at runtime.

The LPM lookup key is represented by the Destination IP Address field read from the input packet. The ID of the output interface for the input packet is the next hop returned by the LPM lookup. The set of LPM rules used by the application is statically configured and loaded into the LPM object at the initialization time.

Note: Please refer to *Virtual Function Setup Instructions* for virtualized test case setup.

3.20.2 Compiling the Application

To compile the application:

1. Go to the sample application directory:

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}/examples/l3fwd-vf
```

2. Set the target (a default target is used if not specified). For example:

```
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

   See the *DPDK Getting Started Guide* for possible RTE_TARGET values.

3. Build the application:

```
make
```

Note: The compiled application is written to the build subdirectory. To have the application written to a different location, the O=/path/to/build/directory option may be specified in the make command.

3.20.3 Running the Application

The application has a number of command line options:

```
./build/l3fwd-vf [EAL options] -- -p PORTMASK  --config(port,queue,lcore)[,(port,
↪queue,lcore)] [--no-numa]
```

where,

- –p PORTMASK: Hexadecimal bitmask of ports to configure
- –config (port,queue,lcore)[,(port,queue,lcore]: determines which queues from which ports are mapped to which cores
- –no-numa: optional, disables numa awareness

For example, consider a dual processor socket platform with 8 physical cores, where cores 0-7 and 16-23 appear on socket 0, while cores 8-15 and 24-31 appear on socket 1.

To enable L3 forwarding between two ports, assuming that both ports are in the same socket, using two cores, cores 1 and 2, (which are in the same socket too), use the following command:

```
./build/l3fwd-vf -l 1,2 -n 4 -- -p 0x3 --config="(0,0,1),(1,0,2)"
```

In this command:

- The -l option enables cores 1 and 2
- The -p option enables ports 0 and 1

- The –config option enables one queue on each port and maps each (port,queue) pair to a specific core. The following table shows the mapping in this example:

Port	Queue	lcore	Description
0	0	1	Map queue 0 from port 0 to lcore 1
1	0	2	Map queue 0 from port 1 to lcore 2

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

3.20.4 Explanation

The operation of this application is similar to that of the basic L3 Forwarding Sample Application. See *Explanation* for more information.

3.21 Link Status Interrupt Sample Application

The Link Status Interrupt sample application is a simple example of packet processing using the Data Plane Development Kit (DPDK) that demonstrates how network link status changes for a network port can be captured and used by a DPDK application.

3.21.1 Overview

The Link Status Interrupt sample application registers a user space callback for the link status interrupt of each port and performs L2 forwarding for each packet that is received on an RX_PORT. The following operations are performed:

- RX_PORT and TX_PORT are paired with available ports one-by-one according to the core mask
- The source MAC address is replaced by the TX_PORT MAC address
- The destination MAC address is replaced by 02:00:00:00:00:TX_PORT_ID

This application can be used to demonstrate the usage of link status interrupt and its user space callbacks and the behavior of L2 forwarding each time the link status changes.

3.21.2 Compiling the Application

1. Go to the example directory:

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}/examples/link_status_interrupt
```

2. Set the target (a default target is used if not specified). For example:

```
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

   See the *DPDK Getting Started Guide* for possible RTE_TARGET values.

3. Build the application:

```
make
```

Note: The compiled application is written to the build subdirectory. To have the application written to a different location, the O=/path/to/build/directory option may be specified on the make command line.

3.21.3 Running the Application

The application requires a number of command line options:

```
./build/link_status_interrupt [EAL options] -- -p PORTMASK [-q NQ][-T PERIOD]
```

where,

- -p PORTMASK: A hexadecimal bitmask of the ports to configure
- -q NQ: A number of queues (=ports) per lcore (default is 1)
- -T PERIOD: statistics will be refreshed each PERIOD seconds (0 to disable, 10 default)

To run the application in a linuxapp environment with 4 lcores, 4 memory channels, 16 ports and 8 RX queues per lcore, issue the command:

```
$ ./build/link_status_interrupt -l 0-3 -n 4-- -q 8 -p ffff
```

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

3.21.4 Explanation

The following sections provide some explanation of the code.

Command Line Arguments

The Link Status Interrupt sample application takes specific parameters, in addition to Environment Abstraction Layer (EAL) arguments (see Section *Running the Application*).

Command line parsing is done in the same way as it is done in the L2 Forwarding Sample Application. See *Command Line Arguments* for more information.

Mbuf Pool Initialization

Mbuf pool initialization is done in the same way as it is done in the L2 Forwarding Sample Application. See *Mbuf Pool Initialization* for more information.

Driver Initialization

The main part of the code in the main() function relates to the initialization of the driver. To fully understand this code, it is recommended to study the chapters that related to the Poll Mode Driver in the *DPDK Programmer's Guide and the DPDK API Reference*.

```
if (rte_eal_pci_probe() < 0)
    rte_exit(EXIT_FAILURE, "Cannot probe PCI\n");

nb_ports = rte_eth_dev_count();
if (nb_ports == 0)
    rte_exit(EXIT_FAILURE, "No Ethernet ports - bye\n");

/*
 * Each logical core is assigned a dedicated TX queue on each port.
 */

for (portid = 0; portid < nb_ports; portid++) {
    /* skip ports that are not enabled */

    if ((lsi_enabled_port_mask & (1 << portid)) == 0)
        continue;

    /* save the destination port id */

    if (nb_ports_in_mask % 2) {
        lsi_dst_ports[portid] = portid_last;
        lsi_dst_ports[portid_last] = portid;
    }
    else
        portid_last = portid;

    nb_ports_in_mask++;

    rte_eth_dev_info_get((uint8_t) portid, &dev_info);
}
```

Observe that:

- rte_eal_pci_probe() parses the devices on the PCI bus and initializes recognized devices.

The next step is to configure the RX and TX queues. For each port, there is only one RX queue (only one lcore is able to poll a given port). The number of TX queues depends on the number of available lcores. The rte_eth_dev_configure() function is used to configure the number of queues for a port:

```
ret = rte_eth_dev_configure((uint8_t) portid, 1, 1, &port_conf);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Cannot configure device: err=%d, port=%u\n", ret, portid);
```

The global configuration is stored in a static structure:

```
static const struct rte_eth_conf port_conf = {
    .rxmode = {
        .split_hdr_size = 0,
        .header_split = 0,   /**< Header Split disabled */
        .hw_ip_checksum = 0, /**< IP checksum offload disabled */
        .hw_vlan_filter = 0, /**< VLAN filtering disabled */
        .hw_strip_crc= 0,    /**< CRC stripped by hardware */
    },
    .txmode = {},
    .intr_conf = {
        .lsc = 1, /**< link status interrupt feature enabled */
    },
};
```

Configuring lsc to 0 (the default) disables the generation of any link status change interrupts in kernel space and no user space interrupt event is received. The public interface rte_eth_link_get() accesses the NIC registers directly to update the link status. Configuring lsc to non-zero enables the generation of link status change interrupts in kernel space when a link status change is present and calls the user space callbacks registered by the application. The public interface rte_eth_link_get() just reads the link status in a global structure that would be updated in the interrupt host thread only.

Interrupt Callback Registration

The application can register one or more callbacks to a specific port and interrupt event. An example callback function that has been written as indicated below.

```
static void
lsi_event_callback(uint8_t port_id, enum rte_eth_event_type type, void *param)
{
    struct rte_eth_link link;

    RTE_SET_USED(param);

    printf("\n\nIn registered callback...\n");

    printf("Event type: %s\n", type == RTE_ETH_EVENT_INTR_LSC ? "LSC interrupt" :
↪"unknown event");

    rte_eth_link_get_nowait(port_id, &link);

    if (link.link_status) {
        printf("Port %d Link Up - speed %u Mbps - %s\n\n", port_id, (unsigned)link.
↪link_speed,
               (link.link_duplex == ETH_LINK_FULL_DUPLEX) ? ("full-duplex") : ("half-
↪duplex"));
    } else
        printf("Port %d Link Down\n\n", port_id);
}
```

This function is called when a link status interrupt is present for the right port. The port_id indicates which port the interrupt applies to. The type parameter identifies the interrupt event type, which currently can be RTE_ETH_EVENT_INTR_LSC only, but other types can be added in the future. The param parameter is the address of the parameter for the callback. This function should be implemented with care since it will be called in the interrupt host thread, which is different from the main thread of its caller.

The application registers the lsi_event_callback and a NULL parameter to the link status interrupt event on each port:

```
rte_eth_dev_callback_register((uint8_t)portid, RTE_ETH_EVENT_INTR_LSC, lsi_event_
↪callback, NULL);
```

This registration can be done only after calling the rte_eth_dev_configure() function and before calling any other function. If lsc is initialized with 0, the callback is never called since no interrupt event would ever be present.

RX Queue Initialization

The application uses one lcore to poll one or several ports, depending on the -q option, which specifies the number of queues per lcore.

For example, if the user specifies -q 4, the application is able to poll four ports with one lcore. If there are 16 ports on the target (and if the portmask argument is -p ffff), the application will need four lcores to poll all the ports.

```
ret = rte_eth_rx_queue_setup((uint8_t) portid, 0, nb_rxd, SOCKET0, &rx_conf, lsi_
↪pktmbuf_pool);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "rte_eth_rx_queue_setup: err=%d, port=%u\n", ret, portid);
```

The list of queues that must be polled for a given lcore is stored in a private structure called struct lcore_queue_conf.

```
struct lcore_queue_conf {
    unsigned n_rx_port;
    unsigned rx_port_list[MAX_RX_QUEUE_PER_LCORE]; unsigned tx_queue_id;
    struct mbuf_table tx_mbufs[LSI_MAX_PORTS];
} rte_cache_aligned;

struct lcore_queue_conf lcore_queue_conf[RTE_MAX_LCORE];
```

The n_rx_port and rx_port_list[] fields are used in the main packet processing loop (see *Receive, Process and Transmit Packets*).

The global configuration for the RX queues is stored in a static structure:

```
static const struct rte_eth_rxconf rx_conf = {
    .rx_thresh = {
        .pthresh = RX_PTHRESH,
        .hthresh = RX_HTHRESH,
        .wthresh = RX_WTHRESH,
    },
};
```

TX Queue Initialization

Each lcore should be able to transmit on any port. For every port, a single TX queue is initialized.

```
/* init one TX queue logical core on each port */

fflush(stdout);

ret = rte_eth_tx_queue_setup(portid, 0, nb_txd, rte_eth_dev_socket_id(portid), &tx_
↪conf);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "rte_eth_tx_queue_setup: err=%d,port=%u\n", ret,␣
↪(unsigned) portid);
```

The global configuration for TX queues is stored in a static structure:

```
static const struct rte_eth_txconf tx_conf = {
    .tx_thresh = {
        .pthresh = TX_PTHRESH,
        .hthresh = TX_HTHRESH,
        .wthresh = TX_WTHRESH,
    },
    .tx_free_thresh = RTE_TEST_TX_DESC_DEFAULT + 1, /* disable feature */
};
```

Receive, Process and Transmit Packets

In the lsi_main_loop() function, the main task is to read ingress packets from the RX queues. This is done using the following code:

```
/*
 *   Read packet from RX queues
 */

for (i = 0; i < qconf->n_rx_port; i++) {
    portid = qconf->rx_port_list[i];
    nb_rx = rte_eth_rx_burst((uint8_t) portid, 0, pkts_burst, MAX_PKT_BURST);
    port_statistics[portid].rx += nb_rx;

    for (j = 0; j < nb_rx; j++) {
        m = pkts_burst[j];
        rte_prefetch0(rte_pktmbuf_mtod(m, void *));
        lsi_simple_forward(m, portid);
    }
}
```

Packets are read in a burst of size MAX_PKT_BURST. The rte_eth_rx_burst() function writes the mbuf pointers in a local table and returns the number of available mbufs in the table.

Then, each mbuf in the table is processed by the lsi_simple_forward() function. The processing is very simple: processes the TX port from the RX port and then replaces the source and destination MAC addresses.

Note: In the following code, the two lines for calculating the output port require some explanation. If portId is even, the first line does nothing (as portid & 1 will be 0), and the second line adds 1. If portId is odd, the first line subtracts one and the second line does nothing. Therefore, 0 goes to 1, and 1 to 0, 2 goes to 3 and 3 to 2, and so on.

```
static void
lsi_simple_forward(struct rte_mbuf *m, unsigned portid)
{
    struct ether_hdr *eth;
    void *tmp;
    unsigned dst_port = lsi_dst_ports[portid];

    eth = rte_pktmbuf_mtod(m, struct ether_hdr *);

    /* 02:00:00:00:00:xx */

    tmp = &eth->d_addr.addr_bytes[0];

    *((uint64_t *)tmp) = 0x000000000002 + (dst_port << 40);

    /* src addr */
    ether_addr_copy(&lsi_ports_eth_addr[dst_port], &eth->s_addr);

    lsi_send_packet(m, dst_port);
}
```

Then, the packet is sent using the lsi_send_packet(m, dst_port) function. For this test application, the processing is exactly the same for all packets arriving on the same RX port. Therefore, it would have been possible to call the lsi_send_burst() function directly from the main loop to send all the received packets on the same TX port using the burst-oriented send function, which is more efficient.

However, in real-life applications (such as, L3 routing), packet N is not necessarily forwarded on the same port as packet N-1. The application is implemented to illustrate that so the same approach can be reused in a more complex application.

The lsi_send_packet() function stores the packet in a per-lcore and per-txport table. If the table is full, the whole packets table is transmitted using the lsi_send_burst() function:

```
/* Send the packet on an output interface */

static int
lsi_send_packet(struct rte_mbuf *m, uint8_t port)
{
    unsigned lcore_id, len;
    struct lcore_queue_conf *qconf;

    lcore_id = rte_lcore_id();
    qconf = &lcore_queue_conf[lcore_id];
    len = qconf->tx_mbufs[port].len;
    qconf->tx_mbufs[port].m_table[len] = m;
    len++;

    /* enough pkts to be sent */

    if (unlikely(len == MAX_PKT_BURST)) {
        lsi_send_burst(qconf, MAX_PKT_BURST, port);
        len = 0;
    }
    qconf->tx_mbufs[port].len = len;

    return 0;
}
```

To ensure that no packets remain in the tables, each lcore does a draining of the TX queue in its main loop. This technique introduces some latency when there are not many packets to send. However, it improves performance:

```
 cur_tsc = rte_rdtsc();

 /*
  *    TX burst queue drain
  */

 diff_tsc = cur_tsc - prev_tsc;

 if (unlikely(diff_tsc > drain_tsc)) {
     /* this could be optimized (use queueid instead of * portid), but it is not␣
↪called so often */

     for (portid = 0; portid < RTE_MAX_ETHPORTS; portid++) {
         if (qconf->tx_mbufs[portid].len == 0)
             continue;

         lsi_send_burst(&lcore_queue_conf[lcore_id],
         qconf->tx_mbufs[portid].len, (uint8_t) portid);
         qconf->tx_mbufs[portid].len = 0;
     }

     /* if timer is enabled */

     if (timer_period > 0) {
```

```
        /* advance the timer */

        timer_tsc += diff_tsc;

        /* if timer has reached its timeout */

        if (unlikely(timer_tsc >= (uint64_t) timer_period)) {
            /* do this only on master core */

            if (lcore_id == rte_get_master_lcore()) {
                print_stats();

                /* reset the timer */
                timer_tsc = 0;
            }
        }
    }
    prev_tsc = cur_tsc;
}
```

3.22 Load Balancer Sample Application

The Load Balancer sample application demonstrates the concept of isolating the packet I/O task from the application-specific workload. Depending on the performance target, a number of logical cores (lcores) are dedicated to handle the interaction with the NIC ports (I/O lcores), while the rest of the lcores are dedicated to performing the application processing (worker lcores). The worker lcores are totally oblivious to the intricacies of the packet I/O activity and use the NIC-agnostic interface provided by software rings to exchange packets with the I/O cores.

3.22.1 Overview

The architecture of the Load Balance application is presented in the following figure.

For the sake of simplicity, the diagram illustrates a specific case of two I/O RX and two I/O TX lcores off loading the packet I/O overhead incurred by four NIC ports from four worker cores, with each I/O lcore handling RX/TX for two NIC ports.

I/O RX Logical Cores

Each I/O RX lcore performs packet RX from its assigned NIC RX rings and then distributes the received packets to the worker threads. The application allows each I/O RX lcore to communicate with any of the worker threads, therefore each (I/O RX lcore, worker lcore) pair is connected through a dedicated single producer - single consumer software ring.

The worker lcore to handle the current packet is determined by reading a predefined 1-byte field from the input packet:

worker_id = packet[load_balancing_field] % n_workers

Since all the packets that are part of the same traffic flow are expected to have the same value for the load balancing field, this scheme also ensures that all the packets that are part of the same traffic flow are directed to the same worker lcore (flow affinity) in the same order they enter the system (packet ordering).

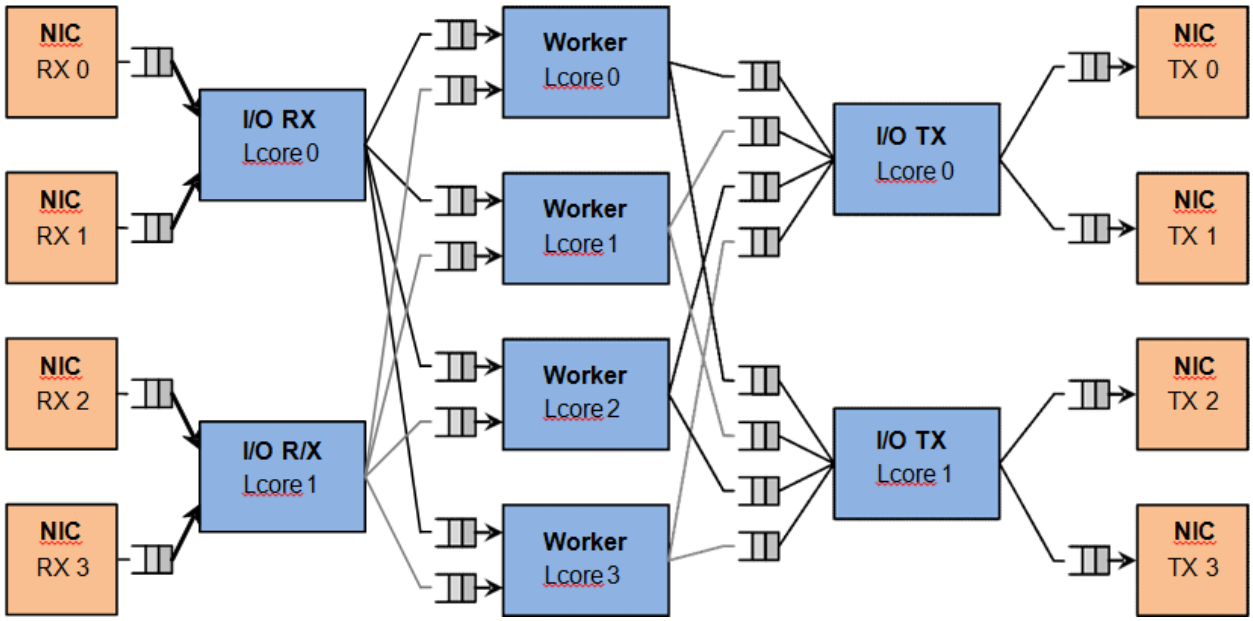


Fig. 3.11: Load Balancer Application Architecture

I/O TX Logical Cores

Each I/O lcore owns the packet TX for a predefined set of NIC ports. To enable each worker thread to send packets to any NIC TX port, the application creates a software ring for each (worker lcore, NIC TX port) pair, with each I/O TX core handling those software rings that are associated with NIC ports that it handles.

Worker Logical Cores

Each worker lcore reads packets from its set of input software rings and routes them to the NIC ports for transmission by dispatching them to output software rings. The routing logic is LPM based, with all the worker threads sharing the same LPM rules.

3.22.2 Compiling the Application

The sequence of steps used to build the application is:

1. Export the required environment variables:

```
export RTE_SDK=<Path to the DPDK installation folder>
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

2. Build the application executable file:

```
cd ${RTE_SDK}/examples/load_balancer
make
```

   For more details on how to build the DPDK libraries and sample applications, please refer to the *DPDK Getting Started Guide*.

3.22.3 Running the Application

To successfully run the application, the command line used to start the application has to be in sync with the traffic flows configured on the traffic generator side.

For examples of application command lines and traffic generator flows, please refer to the DPDK Test Report. For more details on how to set up and run the sample applications provided with DPDK package, please refer to the *DPDK Getting Started Guide*.

3.22.4 Explanation

Application Configuration

The application run-time configuration is done through the application command line parameters. Any parameter that is not specified as mandatory is optional, with the default value hard-coded in the main.h header file from the application folder.

The list of application command line parameters is listed below:

1. –rx "(PORT, QUEUE, LCORE), ...": The list of NIC RX ports and queues handled by the I/O RX lcores. This parameter also implicitly defines the list of I/O RX lcores. This is a mandatory parameter.
2. –tx "(PORT, LCORE), ... ": The list of NIC TX ports handled by the I/O TX lcores. This parameter also implicitly defines the list of I/O TX lcores. This is a mandatory parameter.
3. –w "LCORE, ...": The list of the worker lcores. This is a mandatory parameter.
4. –lpm "IP / PREFIX => PORT; ...": The list of LPM rules used by the worker lcores for packet forwarding. This is a mandatory parameter.
5. –rsz "A, B, C, D": Ring sizes:
   (a) A = The size (in number of buffer descriptors) of each of the NIC RX rings read by the I/O RX lcores.
   (b) B = The size (in number of elements) of each of the software rings used by the I/O RX lcores to send packets to worker lcores.
   (c) C = The size (in number of elements) of each of the software rings used by the worker lcores to send packets to I/O TX lcores.
   (d) D = The size (in number of buffer descriptors) of each of the NIC TX rings written by I/O TX lcores.
6. –bsz "(A, B), (C, D), (E, F)": Burst sizes:
   (a) A = The I/O RX lcore read burst size from NIC RX.
   (b) B = The I/O RX lcore write burst size to the output software rings.
   (c) C = The worker lcore read burst size from the input software rings.
   (d) D = The worker lcore write burst size to the output software rings.
   (e) E = The I/O TX lcore read burst size from the input software rings.
   (f) F = The I/O TX lcore write burst size to the NIC TX.
7. –pos-lb POS: The position of the 1-byte field within the input packet used by the I/O RX lcores to identify the worker lcore for the current packet. This field needs to be within the first 64 bytes of the input packet.

The infrastructure of software rings connecting I/O lcores and worker lcores is built by the application as a result of the application configuration provided by the user through the application command line parameters.

A specific lcore performing the I/O RX role for a specific set of NIC ports can also perform the I/O TX role for the same or a different set of NIC ports. A specific lcore cannot perform both the I/O role (either RX or TX) and the worker role during the same session.

Example:

```
./load_balancer -l 3-7 -n 4 -- --rx "(0,0,3),(1,0,3)" --tx "(0,3),(1,3)" --w "4,5,6,7
↪" --lpm "1.0.0.0/24=>0; 1.0.1.0/24=>1;" --pos-lb 29
```

There is a single I/O lcore (lcore 3) that handles RX and TX for two NIC ports (ports 0 and 1) that handles packets to/from four worker lcores (lcores 4, 5, 6 and 7) that are assigned worker IDs 0 to 3 (worker ID for lcore 4 is 0, for lcore 5 is 1, for lcore 6 is 2 and for lcore 7 is 3).

Assuming that all the input packets are IPv4 packets with no VLAN label and the source IP address of the current packet is A.B.C.D, the worker lcore for the current packet is determined by byte D (which is byte 29). There are two LPM rules that are used by each worker lcore to route packets to the output NIC ports.

The following table illustrates the packet flow through the system for several possible traffic flows:

Flow #	Source IP Address	Destination IP Address	Worker ID (Worker lcore)	Output NIC Port
1	0.0.0.0	1.0.0.1	0 (4)	0
2	0.0.0.1	1.0.1.2	1 (5)	1
3	0.0.0.14	1.0.0.3	2 (6)	0
4	0.0.0.15	1.0.1.4	3 (7)	1

NUMA Support

The application has built-in performance enhancements for the NUMA case:

1. One buffer pool per each CPU socket.
2. One LPM table per each CPU socket.
3. Memory for the NIC RX or TX rings is allocated on the same socket with the lcore handling the respective ring.

In the case where multiple CPU sockets are used in the system, it is recommended to enable at least one lcore to fulfill the I/O role for the NIC ports that are directly attached to that CPU socket through the PCI Express* bus. It is always recommended to handle the packet I/O with lcores from the same CPU socket as the NICs.

Depending on whether the I/O RX lcore (same CPU socket as NIC RX), the worker lcore and the I/O TX lcore (same CPU socket as NIC TX) handling a specific input packet, are on the same or different CPU sockets, the following run-time scenarios are possible:

1. AAA: The packet is received, processed and transmitted without going across CPU sockets.
2. AAB: The packet is received and processed on socket A, but as it has to be transmitted on a NIC port connected to socket B, the packet is sent to socket B through software rings.
3. ABB: The packet is received on socket A, but as it has to be processed by a worker lcore on socket B, the packet is sent to socket B through software rings. The packet is transmitted by a NIC port connected to the same CPU socket as the worker lcore that processed it.
4. ABC: The packet is received on socket A, it is processed by an lcore on socket B, then it has to be transmitted out by a NIC connected to socket C. The performance price for crossing the CPU socket boundary is paid twice for this packet.

3.23 Server-Node EFD Sample Application

This sample application demonstrates the use of EFD library as a flow-level load balancer, for more information about the EFD Library please refer to the DPDK programmer's guide.

This sample application is a variant of the *client-server sample application* where a specific target node is specified for every and each flow (not in a round-robin fashion as the original load balancing sample application).

3.23.1 Overview

The architecture of the EFD flow-based load balancer sample application is presented in the following figure.

Fig. 3.12: Using EFD as a Flow-Level Load Balancer

As shown in Fig. 3.12, the sample application consists of a front-end node (server) using the EFD library to create a load-balancing table for flows, for each flow a target backend worker node is specified. The EFD table does not store the flow key (unlike a regular hash table), and hence, it can individually load-balance millions of flows (number of targets * maximum number of flows fit in a flow table per target) while still fitting in CPU cache.

It should be noted that although they are referred to as nodes, the frontend server and worker nodes are processes running on the same platform.

Front-end Server

Upon initializing, the frontend server node (process) creates a flow distributor table (based on the EFD library) which is populated with flow information and its intended target node.

The sample application assigns a specific target node_id (process) for each of the IP destination addresses as follows:

```
node_id = i % num_nodes; /* Target node id is generated */
ip_dst = rte_cpu_to_be_32(i); /* Specific ip destination address is
                                 assigned to this target node */
```

then the pair of <key,target> is inserted into the flow distribution table.

The main loop of the server process receives a burst of packets, then for each packet, a flow key (IP destination address) is extracted. The flow distributor table is looked up and the target node id is returned. Packets are then enqueued to the specified target node id.

It should be noted that flow distributor table is not a membership test table. I.e. if the key has already been inserted the target node id will be correct, but for new keys the flow distributor table will return a value (which can be valid).

Backend Worker Nodes

Upon initializing, the worker node (process) creates a flow table (a regular hash table that stores the key default size 1M flows) which is populated with only the flow information that is serviced at this node. This flow key is essential to point out new keys that have not been inserted before.

The worker node's main loop is simply receiving packets then doing a hash table lookup. If a match occurs then statistics are updated for flows serviced by this node. If no match is found in the local hash table then this indicates that this is a new flow, which is dropped.

3.23.2 Compiling the Application

The sequence of steps used to build the application is:

1. Export the required environment variables:

```
export RTE_SDK=/path/to/rte_sdk
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

2. Build the application executable file:

```
cd ${RTE_SDK}/examples/server_node_efd/
make
```

   For more details on how to build the DPDK libraries and sample applications, please refer to the *DPDK Getting Started Guide*.

3.23.3 Running the Application

The application has two binaries to be run: the front-end server and the back-end node.

The frontend server (server) has the following command line options:

```
./server [EAL options] -- -p PORTMASK -n NUM_NODES -f NUM_FLOWS
```

Where,

- `-p PORTMASK`: Hexadecimal bitmask of ports to configure
- `-n NUM_NODES`: Number of back-end nodes that will be used
- `-f NUM_FLOWS`: Number of flows to be added in the EFD table (1 million, by default)

The back-end node (node) has the following command line options:

```
./node [EAL options] -- -n NODE_ID
```

Where,

- `-n NODE_ID`: Node ID, which cannot be equal or higher than NUM_MODES

First, the server app must be launched, with the number of nodes that will be run. Once it has been started, the node instances can be run, with different NODE_ID. These instances have to be run as secondary processes, with `--proc-type=secondary` in the EAL options, which will attach to the primary process memory, and therefore, they can access the queues created by the primary process to distribute packets.

To successfully run the application, the command line used to start the application has to be in sync with the traffic flows configured on the traffic generator side.

For examples of application command lines and traffic generator flows, please refer to the DPDK Test Report. For more details on how to set up and run the sample applications provided with DPDK package, please refer to the *DPDK Getting Started Guide for Linux* and *DPDK Getting Started Guide for FreeBSD*.

3.23.4 Explanation

As described in previous sections, there are two processes in this example.

The first process, the front-end server, creates and populates the EFD table, which is used to distribute packets to nodes, which the number of flows specified in the command line (1 million, by default).

```
static void
create_efd_table(void)
{
    uint8_t socket_id = rte_socket_id();

    /* create table */
    efd_table = rte_efd_create("flow table", num_flows * 2, sizeof(uint32_t),
                    1 << socket_id, socket_id);

    if (efd_table == NULL)
        rte_exit(EXIT_FAILURE, "Problem creating the flow table\n");
}

static void
populate_efd_table(void)
{
    unsigned int i;
    int32_t ret;
    uint32_t ip_dst;
    uint8_t socket_id = rte_socket_id();
    uint64_t node_id;

    /* Add flows in table */
    for (i = 0; i < num_flows; i++) {
        node_id = i % num_nodes;

        ip_dst = rte_cpu_to_be_32(i);
        ret = rte_efd_update(efd_table, socket_id,
                        (void *)&ip_dst, (efd_value_t)node_id);
        if (ret < 0)
            rte_exit(EXIT_FAILURE, "Unable to add entry %u in "
                                "EFD table\n", i);
    }

    printf("EFD table: Adding 0x%x keys\n", num_flows);
}
```

After initialization, packets are received from the enabled ports, and the IPv4 address from the packets is used as a key to look up in the EFD table, which tells the node where the packet has to be distributed.

```
static void
process_packets(uint32_t port_num __rte_unused, struct rte_mbuf *pkts[],
        uint16_t rx_count, unsigned int socket_id)
{
    uint16_t i;
    uint8_t node;
    efd_value_t data[EFD_BURST_MAX];
    const void *key_ptrs[EFD_BURST_MAX];

    struct ipv4_hdr *ipv4_hdr;
    uint32_t ipv4_dst_ip[EFD_BURST_MAX];

    for (i = 0; i < rx_count; i++) {
        /* Handle IPv4 header.*/
        ipv4_hdr = rte_pktmbuf_mtod_offset(pkts[i], struct ipv4_hdr *,
                sizeof(struct ether_hdr));
        ipv4_dst_ip[i] = ipv4_hdr->dst_addr;
        key_ptrs[i] = (void *)&ipv4_dst_ip[i];
```

```
    }

    rte_efd_lookup_bulk(efd_table, socket_id, rx_count,
                (const void **) key_ptrs, data);
    for (i = 0; i < rx_count; i++) {
        node = (uint8_t) ((uintptr_t)data[i]);

        if (node >= num_nodes) {
            /*
             * Node is out of range, which means that
             * flow has not been inserted
             */
            flow_dist_stats.drop++;
            rte_pktmbuf_free(pkts[i]);
        } else {
            flow_dist_stats.distributed++;
            enqueue_rx_packet(node, pkts[i]);
        }
    }

    for (i = 0; i < num_nodes; i++)
        flush_rx_queue(i);
}
```

The burst of packets received is enqueued in temporary buffers (per node), and enqueued in the shared ring between the server and the node. After this, a new burst of packets is received and this process is repeated infinitely.

```
static void
flush_rx_queue(uint16_t node)
{
    uint16_t j;
    struct node *cl;

    if (cl_rx_buf[node].count == 0)
        return;

    cl = &nodes[node];
    if (rte_ring_enqueue_bulk(cl->rx_q, (void **)cl_rx_buf[node].buffer,
            cl_rx_buf[node].count, NULL) != cl_rx_buf[node].count){
        for (j = 0; j < cl_rx_buf[node].count; j++)
            rte_pktmbuf_free(cl_rx_buf[node].buffer[j]);
        cl->stats.rx_drop += cl_rx_buf[node].count;
    } else
        cl->stats.rx += cl_rx_buf[node].count;

    cl_rx_buf[node].count = 0;
}
```

The second process, the back-end node, receives the packets from the shared ring with the server and send them out, if they belong to the node.

At initialization, it attaches to the server process memory, to have access to the shared ring, parameters and statistics.

```
rx_ring = rte_ring_lookup(get_rx_queue_name(node_id));
if (rx_ring == NULL)
    rte_exit(EXIT_FAILURE, "Cannot get RX ring - "
            "is server process running?\n");
```

```
mp = rte_mempool_lookup(PKTMBUF_POOL_NAME);
if (mp == NULL)
    rte_exit(EXIT_FAILURE, "Cannot get mempool for mbufs\n");

mz = rte_memzone_lookup(MZ_SHARED_INFO);
if (mz == NULL)
    rte_exit(EXIT_FAILURE, "Cannot get port info structure\n");
info = mz->addr;
tx_stats = &(info->tx_stats[node_id]);
filter_stats = &(info->filter_stats[node_id]);
```

Then, the hash table that contains the flows that will be handled by the node is created and populated.

```
static struct rte_hash *
create_hash_table(const struct shared_info *info)
{
    uint32_t num_flows_node = info->num_flows / info->num_nodes;
    char name[RTE_HASH_NAMESIZE];
    struct rte_hash *h;

    /* create table */
    struct rte_hash_parameters hash_params = {
        .entries = num_flows_node * 2, /* table load = 50% */
        .key_len = sizeof(uint32_t), /* Store IPv4 dest IP address */
        .socket_id = rte_socket_id(),
        .hash_func_init_val = 0,
    };

    snprintf(name, sizeof(name), "hash_table_%d", node_id);
    hash_params.name = name;
    h = rte_hash_create(&hash_params);

    if (h == NULL)
        rte_exit(EXIT_FAILURE,
                "Problem creating the hash table for node %d\n",
                node_id);
    return h;
}

static void
populate_hash_table(const struct rte_hash *h, const struct shared_info *info)
{
    unsigned int i;
    int32_t ret;
    uint32_t ip_dst;
    uint32_t num_flows_node = 0;
    uint64_t target_node;

    /* Add flows in table */
    for (i = 0; i < info->num_flows; i++) {
        target_node = i % info->num_nodes;
        if (target_node != node_id)
            continue;

        ip_dst = rte_cpu_to_be_32(i);

        ret = rte_hash_add_key(h, (void *) &ip_dst);
        if (ret < 0)
```

```
            rte_exit(EXIT_FAILURE, "Unable to add entry %u "
                    "in hash table\n", i);
        else
            num_flows_node++;

    }

    printf("Hash table: Adding 0x%x keys\n", num_flows_node);
}
```

After initialization, packets are dequeued from the shared ring (from the server) and, like in the server process, the IPv4 address from the packets is used as a key to look up in the hash table. If there is a hit, packet is stored in a buffer, to be eventually transmitted in one of the enabled ports. If key is not there, packet is dropped, since the flow is not handled by the node.

```
static inline void
handle_packets(struct rte_hash *h, struct rte_mbuf **bufs, uint16_t num_packets)
{
    struct ipv4_hdr *ipv4_hdr;
    uint32_t ipv4_dst_ip[PKT_READ_SIZE];
    const void *key_ptrs[PKT_READ_SIZE];
    unsigned int i;
    int32_t positions[PKT_READ_SIZE] = {0};

    for (i = 0; i < num_packets; i++) {
        /* Handle IPv4 header.*/
        ipv4_hdr = rte_pktmbuf_mtod_offset(bufs[i], struct ipv4_hdr *,
                sizeof(struct ether_hdr));
        ipv4_dst_ip[i] = ipv4_hdr->dst_addr;
        key_ptrs[i] = &ipv4_dst_ip[i];
    }
    /* Check if packets belongs to any flows handled by this node */
    rte_hash_lookup_bulk(h, key_ptrs, num_packets, positions);

    for (i = 0; i < num_packets; i++) {
        if (likely(positions[i] >= 0)) {
            filter_stats->passed++;
            transmit_packet(bufs[i]);
        } else {
            filter_stats->drop++;
            /* Drop packet, as flow is not handled by this node */
            rte_pktmbuf_free(bufs[i]);
        }
    }
}
```

Finally, note that both processes updates statistics, such as transmitted, received and dropped packets, which are shown and refreshed by the server app.

```
static void
do_stats_display(void)
{
    unsigned int i, j;
    const char clr[] = {27, '[', '2', 'J', '\0'};
    const char topLeft[] = {27, '[', '1', ';', '1', 'H', '\0'};
    uint64_t port_tx[RTE_MAX_ETHPORTS], port_tx_drop[RTE_MAX_ETHPORTS];
    uint64_t node_tx[MAX_NODES], node_tx_drop[MAX_NODES];
```

```
    /* to get TX stats, we need to do some summing calculations */
    memset(port_tx, 0, sizeof(port_tx));
    memset(port_tx_drop, 0, sizeof(port_tx_drop));
    memset(node_tx, 0, sizeof(node_tx));
    memset(node_tx_drop, 0, sizeof(node_tx_drop));

    for (i = 0; i < num_nodes; i++) {
        const struct tx_stats *tx = &info->tx_stats[i];

        for (j = 0; j < info->num_ports; j++) {
            const uint64_t tx_val = tx->tx[info->id[j]];
            const uint64_t drop_val = tx->tx_drop[info->id[j]];

            port_tx[j] += tx_val;
            port_tx_drop[j] += drop_val;
            node_tx[i] += tx_val;
            node_tx_drop[i] += drop_val;
        }
    }

    /* Clear screen and move to top left */
    printf("%s%s", clr, topLeft);

    printf("PORTS\n");
    printf("-----\n");
    for (i = 0; i < info->num_ports; i++)
        printf("Port %u: '%s'\t", (unsigned int)info->id[i],
                get_printable_mac_addr(info->id[i]));
    printf("\n\n");
    for (i = 0; i < info->num_ports; i++) {
        printf("Port %u - rx: %9"PRIu64"\t"
                "tx: %9"PRIu64"\n",
                (unsigned int)info->id[i], info->rx_stats.rx[i],
                port_tx[i]);
    }

    printf("\nSERVER\n");
    printf("-----\n");
    printf("distributed: %9"PRIu64", drop: %9"PRIu64"\n",
            flow_dist_stats.distributed, flow_dist_stats.drop);

    printf("\nNODES\n");
    printf("-------\n");
    for (i = 0; i < num_nodes; i++) {
        const unsigned long long rx = nodes[i].stats.rx;
        const unsigned long long rx_drop = nodes[i].stats.rx_drop;
        const struct filter_stats *filter = &info->filter_stats[i];

        printf("Node %2u - rx: %9llu, rx_drop: %9llu\n"
                "            tx: %9"PRIu64", tx_drop: %9"PRIu64"\n"
                "            filter_passed: %9"PRIu64", "
                "filter_drop: %9"PRIu64"\n",
                i, rx, rx_drop, node_tx[i], node_tx_drop[i],
                filter->passed, filter->drop);
    }

    printf("\n");
}
```

3.24 Multi-process Sample Application

This chapter describes the example applications for multi-processing that are included in the DPDK.

3.24.1 Example Applications

Building the Sample Applications

The multi-process example applications are built in the same way as other sample applications, and as documented in the *DPDK Getting Started Guide*. To build all the example applications:

1. Set RTE_SDK and go to the example directory:

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}/examples/multi_process
```

2. Set the target (a default target will be used if not specified). For example:

```
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

   See the *DPDK Getting Started Guide* for possible RTE_TARGET values.

3. Build the applications:

```
make
```

Note: If just a specific multi-process application needs to be built, the final make command can be run just in that application's directory, rather than at the top-level multi-process directory.

Basic Multi-process Example

The examples/simple_mp folder in the DPDK release contains a basic example application to demonstrate how two DPDK processes can work together using queues and memory pools to share information.

Running the Application

To run the application, start one copy of the simple_mp binary in one terminal, passing at least two cores in the coremask/corelist, as follows:

```
./build/simple_mp -l 0-1 -n 4 --proc-type=primary
```

For the first DPDK process run, the proc-type flag can be omitted or set to auto, since all DPDK processes will default to being a primary instance, meaning they have control over the hugepage shared memory regions. The process should start successfully and display a command prompt as follows:

```
$ ./build/simple_mp -l 0-1 -n 4 --proc-type=primary
EAL: coremask set to 3
EAL: Detected lcore 0 on socket 0
EAL: Detected lcore 1 on socket 0
EAL: Detected lcore 2 on socket 0
EAL: Detected lcore 3 on socket 0
```

```
...

EAL: Requesting 2 pages of size 1073741824
EAL: Requesting 768 pages of size 2097152
EAL: Ask a virtual area of 0x40000000 bytes
EAL: Virtual area found at 0x7ff200000000 (size = 0x40000000)
...

EAL: check igb_uio module
EAL: check module finished
EAL: Master core 0 is ready (tid=54e41820)
EAL: Core 1 is ready (tid=53b32700)

Starting core 1

simple_mp >
```

To run the secondary process to communicate with the primary process, again run the same binary setting at least two cores in the coremask/corelist:

```
./build/simple_mp -l 2-3 -n 4 --proc-type=secondary
```

When running a secondary process such as that shown above, the proc-type parameter can again be specified as auto. However, omitting the parameter altogether will cause the process to try and start as a primary rather than secondary process.

Once the process type is specified correctly, the process starts up, displaying largely similar status messages to the primary instance as it initializes. Once again, you will be presented with a command prompt.

Once both processes are running, messages can be sent between them using the send command. At any stage, either process can be terminated using the quit command.

```
EAL: Master core 10 is ready (tid=b5f89820)           EAL: Master core 8 is ready␣
↪(tid=864a3820)
EAL: Core 11 is ready (tid=84ffe700)                  EAL: Core 9 is ready␣
↪(tid=85995700)
Starting core 11                                      Starting core 9
simple_mp > send hello_secondary                      simple_mp > core 9: Received
↪'hello_secondary'
simple_mp > core 11: Received 'hello_primary'         simple_mp > send hello_primary
simple_mp > quit                                      simple_mp > quit
```

Note: If the primary instance is terminated, the secondary instance must also be shut-down and restarted after the primary. This is necessary because the primary instance will clear and reset the shared memory regions on startup, invalidating the secondary process's pointers. The secondary process can be stopped and restarted without affecting the primary process.

How the Application Works

The core of this example application is based on using two queues and a single memory pool in shared memory. These three objects are created at startup by the primary process, since the secondary process cannot create objects in memory as it cannot reserve memory zones, and the secondary process then uses lookup functions to attach to these objects as it starts up.

```
if (rte_eal_process_type() == RTE_PROC_PRIMARY){
    send_ring = rte_ring_create(_PRI_2_SEC, ring_size, SOCKET0, flags);
    recv_ring = rte_ring_create(_SEC_2_PRI, ring_size, SOCKET0, flags);
    message_pool = rte_mempool_create(_MSG_POOL, pool_size, string_size, pool_cache,␣
↪priv_data_sz, NULL, NULL, NULL, NULL, SOCKET0, flags);
} else {
    recv_ring = rte_ring_lookup(_PRI_2_SEC);
    send_ring = rte_ring_lookup(_SEC_2_PRI);
    message_pool = rte_mempool_lookup(_MSG_POOL);
}
```

Note, however, that the named ring structure used as send_ring in the primary process is the recv_ring in the secondary process.

Once the rings and memory pools are all available in both the primary and secondary processes, the application simply dedicates two threads to sending and receiving messages respectively. The receive thread simply dequeues any messages on the receive ring, prints them, and frees the buffer space used by the messages back to the memory pool. The send thread makes use of the command-prompt library to interactively request user input for messages to send. Once a send command is issued by the user, a buffer is allocated from the memory pool, filled in with the message contents, then enqueued on the appropriate rte_ring.

Symmetric Multi-process Example

The second example of DPDK multi-process support demonstrates how a set of processes can run in parallel, with each process performing the same set of packet- processing operations. (Since each process is identical in functionality to the others, we refer to this as symmetric multi-processing, to differentiate it from asymmetric multi- processing - such as a client-server mode of operation seen in the next example, where different processes perform different tasks, yet co-operate to form a packet-processing system.) The following diagram shows the data-flow through the application, using two processes.

As the diagram shows, each process reads packets from each of the network ports in use. RSS is used to distribute incoming packets on each port to different hardware RX queues. Each process reads a different RX queue on each port and so does not contend with any other process for that queue access. Similarly, each process writes outgoing packets to a different TX queue on each port.

Running the Application

As with the simple_mp example, the first instance of the symmetric_mp process must be run as the primary instance, though with a number of other application- specific parameters also provided after the EAL arguments. These additional parameters are:

- -p <portmask>, where portmask is a hexadecimal bitmask of what ports on the system are to be used. For example: -p 3 to use ports 0 and 1 only.
- –num-procs <N>, where N is the total number of symmetric_mp instances that will be run side-by-side to perform packet processing. This parameter is used to configure the appropriate number of receive queues on each network port.
- –proc-id <n>, where n is a numeric value in the range 0 <= n < N (number of processes, specified above). This identifies which symmetric_mp instance is being run, so that each process can read a unique receive queue on each network port.

The secondary symmetric_mp instances must also have these parameters specified, and the first two must be the same as those passed to the primary instance, or errors result.

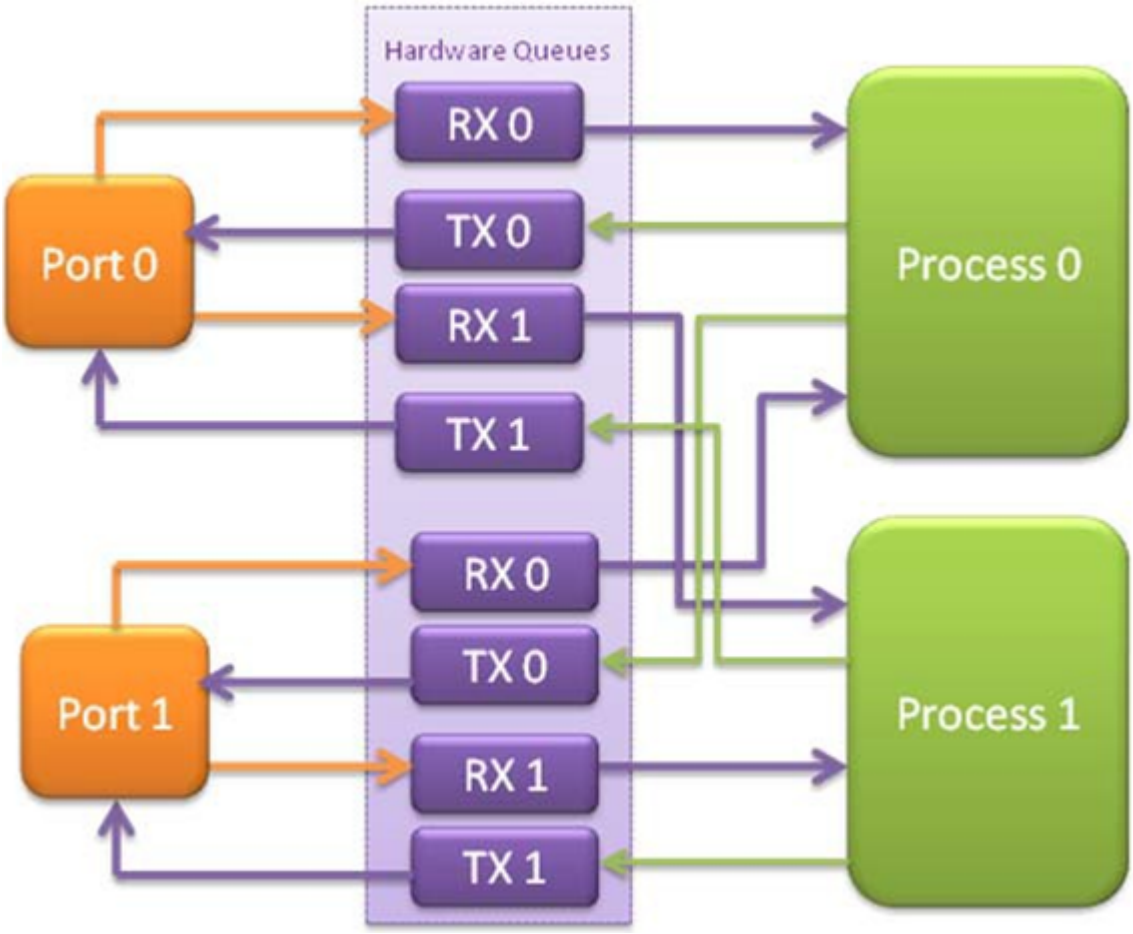


Fig. 3.13: Example Data Flow in a Symmetric Multi-process Application

For example, to run a set of four symmetric_mp instances, running on lcores 1-4, all performing level-2 forwarding of packets between ports 0 and 1, the following commands can be used (assuming run as root):

```
# ./build/symmetric_mp -l 1 -n 4 --proc-type=auto -- -p 3 --num-procs=4 --proc-id=0
# ./build/symmetric_mp -l 2 -n 4 --proc-type=auto -- -p 3 --num-procs=4 --proc-id=1
# ./build/symmetric_mp -l 3 -n 4 --proc-type=auto -- -p 3 --num-procs=4 --proc-id=2
# ./build/symmetric_mp -l 4 -n 4 --proc-type=auto -- -p 3 --num-procs=4 --proc-id=3
```

Note: In the above example, the process type can be explicitly specified as primary or secondary, rather than auto. When using auto, the first process run creates all the memory structures needed for all processes - irrespective of whether it has a proc-id of 0, 1, 2 or 3.

Note: For the symmetric multi-process example, since all processes work in the same manner, once the hugepage shared memory and the network ports are initialized, it is not necessary to restart all processes if the primary instance dies. Instead, that process can be restarted as a secondary, by explicitly setting the proc-type to secondary on the command line. (All subsequent instances launched will also need this explicitly specified, as auto-detection will detect no primary processes running and therefore attempt to re-initialize shared memory.)

How the Application Works

The initialization calls in both the primary and secondary instances are the same for the most part, calling the rte_eal_init(), 1 G and 10 G driver initialization and then rte_eal_pci_probe() functions. Thereafter, the initialization done depends on whether the process is configured as a primary or secondary instance.

In the primary instance, a memory pool is created for the packet mbufs and the network ports to be used are initialized - the number of RX and TX queues per port being determined by the num-procs parameter passed on the command-line. The structures for the initialized network ports are stored in shared memory and therefore will be accessible by the secondary process as it initializes.

```
if (num_ports & 1)
   rte_exit(EXIT_FAILURE, "Application must use an even number of ports\n");

for(i = 0; i < num_ports; i++){
    if(proc_type == RTE_PROC_PRIMARY)
        if (smp_port_init(ports[i], mp, (uint16_t)num_procs) < 0)
            rte_exit(EXIT_FAILURE, "Error initializing ports\n");
}
```

In the secondary instance, rather than initializing the network ports, the port information exported by the primary process is used, giving the secondary process access to the hardware and software rings for each network port. Similarly, the memory pool of mbufs is accessed by doing a lookup for it by name:

```
mp = (proc_type == RTE_PROC_SECONDARY) ? rte_mempool_lookup(_SMP_MBUF_POOL) : rte_
↪mempool_create(_SMP_MBUF_POOL, NB_MBUFS, MBUF_SIZE, ... )
```

Once this initialization is complete, the main loop of each process, both primary and secondary, is exactly the same - each process reads from each port using the queue corresponding to its proc-id parameter, and writes to the corresponding transmit queue on the output port.

Client-Server Multi-process Example

The third example multi-process application included with the DPDK shows how one can use a client-server type multi-process design to do packet processing. In this example, a single server process performs the packet reception from the ports being used and distributes these packets using round-robin ordering among a set of client processes, which perform the actual packet processing. In this case, the client applications just perform level-2 forwarding of packets by sending each packet out on a different network port.

The following diagram shows the data-flow through the application, using two client processes.

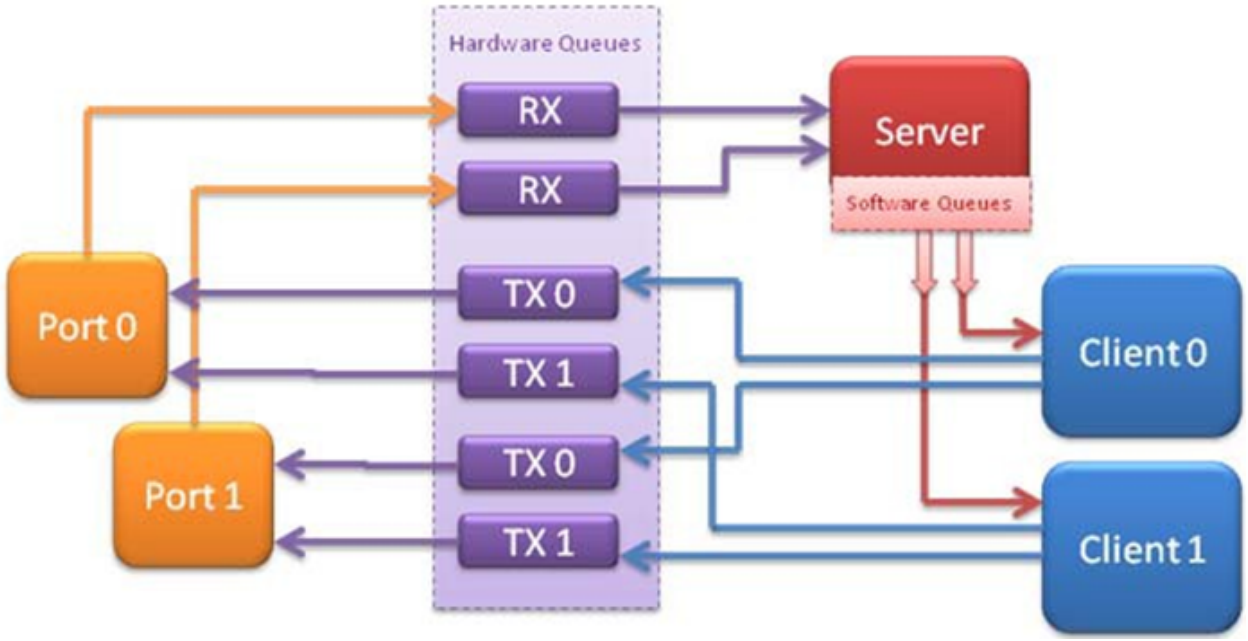


Fig. 3.14: Example Data Flow in a Client-Server Symmetric Multi-process Application

Running the Application

The server process must be run initially as the primary process to set up all memory structures for use by the clients. In addition to the EAL parameters, the application- specific parameters are:

- -p <portmask >, where portmask is a hexadecimal bitmask of what ports on the system are to be used. For example: -p 3 to use ports 0 and 1 only.
- -n <num-clients>, where the num-clients parameter is the number of client processes that will process the packets received by the server application.

Note: In the server process, a single thread, the master thread, that is, the lowest numbered lcore in the core-mask/corelist, performs all packet I/O. If a coremask/corelist is specified with more than a single lcore bit set in it, an additional lcore will be used for a thread to periodically print packet count statistics.

Since the server application stores configuration data in shared memory, including the network ports to be used, the only application parameter needed by a client process is its client instance ID. Therefore, to run a server application on lcore 1 (with lcore 2 printing statistics) along with two client processes running on lcores 3 and 4, the following commands could be used:

```
# ./mp_server/build/mp_server -l 1-2 -n 4 -- -p 3 -n 2
# ./mp_client/build/mp_client -l 3 -n 4 --proc-type=auto -- -n 0
# ./mp_client/build/mp_client -l 4 -n 4 --proc-type=auto -- -n 1
```

Note: If the server application dies and needs to be restarted, all client applications also need to be restarted, as there is no support in the server application for it to run as a secondary process. Any client processes that need restarting can be restarted without affecting the server process.

How the Application Works

The server process performs the network port and data structure initialization much as the symmetric multi-process application does when run as primary. One additional enhancement in this sample application is that the server process stores its port configuration data in a memory zone in hugepage shared memory. This eliminates the need for the client processes to have the portmask parameter passed into them on the command line, as is done for the symmetric multi-process application, and therefore eliminates mismatched parameters as a potential source of errors.

In the same way that the server process is designed to be run as a primary process instance only, the client processes are designed to be run as secondary instances only. They have no code to attempt to create shared memory objects. Instead, handles to all needed rings and memory pools are obtained via calls to rte_ring_lookup() and rte_mempool_lookup(). The network ports for use by the processes are obtained by loading the network port drivers and probing the PCI bus, which will, as in the symmetric multi-process example, automatically get access to the network ports using the settings already configured by the primary/server process.

Once all applications are initialized, the server operates by reading packets from each network port in turn and distributing those packets to the client queues (software rings, one for each client process) in round-robin order. On the client side, the packets are read from the rings in as big of bursts as possible, then routed out to a different network port. The routing used is very simple. All packets received on the first NIC port are transmitted back out on the second port and vice versa. Similarly, packets are routed between the 3rd and 4th network ports and so on. The sending of packets is done by writing the packets directly to the network ports; they are not transferred back via the server process.

In both the server and the client processes, outgoing packets are buffered before being sent, so as to allow the sending of multiple packets in a single burst to improve efficiency. For example, the client process will buffer packets to send, until either the buffer is full or until we receive no further packets from the server.

Master-slave Multi-process Example

The fourth example of DPDK multi-process support demonstrates a master-slave model that provide the capability of application recovery if a slave process crashes or meets unexpected conditions. In addition, it also demonstrates the floating process, which can run among different cores in contrast to the traditional way of binding a process/thread to a specific CPU core, using the local cache mechanism of mempool structures.

This application performs the same functionality as the L2 Forwarding sample application, therefore this chapter does not cover that part but describes functionality that is introduced in this multi-process example only. Please refer to *L2 Forwarding Sample Application (in Real and Virtualized Environments)* for more information.

Unlike previous examples where all processes are started from the command line with input arguments, in this example, only one process is spawned from the command line and that process creates other processes. The following section describes this in more detail.

Master-slave Process Models

The process spawned from the command line is called the *master process* in this document. A process created by the master is called a *slave process*. The application has only one master process, but could have multiple slave processes.

Once the master process begins to run, it tries to initialize all the resources such as memory, CPU cores, driver, ports, and so on, as the other examples do. Thereafter, it creates slave processes, as shown in the following figure.

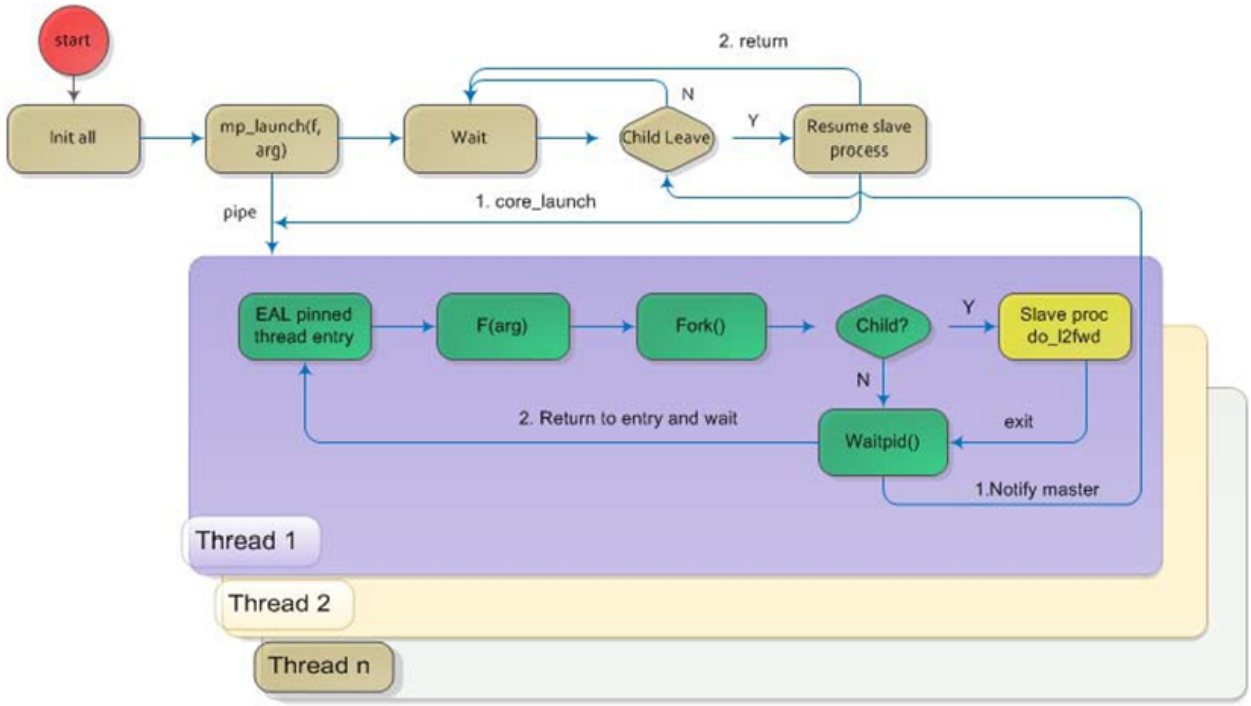


Fig. 3.15: Master-slave Process Workflow

The master process calls the rte_eal_mp_remote_launch() EAL function to launch an application function for each pinned thread through the pipe. Then, it waits to check if any slave processes have exited. If so, the process tries to re-initialize the resources that belong to that slave and launch them in the pinned thread entry again. The following section describes the recovery procedures in more detail.

For each pinned thread in EAL, after reading any data from the pipe, it tries to call the function that the application specified. In this master specified function, a fork() call creates a slave process that performs the L2 forwarding task. Then, the function waits until the slave exits, is killed or crashes. Thereafter, it notifies the master of this event and returns. Finally, the EAL pinned thread waits until the new function is launched.

After discussing the master-slave model, it is necessary to mention another issue, global and static variables.

For multiple-thread cases, all global and static variables have only one copy and they can be accessed by any thread if applicable. So, they can be used to sync or share data among threads.

In the previous examples, each process has separate global and static variables in memory and are independent of each other. If it is necessary to share the knowledge, some communication mechanism should be deployed, such as, memzone, ring, shared memory, and so on. The global or static variables are not a valid approach to share data among processes. For variables in this example, on the one hand, the slave process inherits all the knowledge of these variables after being created by the master. On the other hand, other processes cannot know if one or more processes modifies them after slave creation since that is the nature of a multiple process address space. But this does not mean that these variables cannot be used to share or sync data; it depends on the use case. The following are the possible use cases:

1. The master process starts and initializes a variable and it will never be changed after slave processes created. This case is OK.
2. After the slave processes are created, the master or slave cores need to change a variable, but other processes do not need to know the change. This case is also OK.
3. After the slave processes are created, the master or a slave needs to change a variable. In the meantime, one or more other process needs to be aware of the change. In this case, global and static variables cannot be used to share knowledge. Another communication mechanism is needed. A simple approach without lock protection can be a heap buffer allocated by rte_malloc or mem zone.

Slave Process Recovery Mechanism

Before talking about the recovery mechanism, it is necessary to know what is needed before a new slave instance can run if a previous one exited.

When a slave process exits, the system returns all the resources allocated for this process automatically. However, this does not include the resources that were allocated by the DPDK. All the hardware resources are shared among the processes, which include memzone, mempool, ring, a heap buffer allocated by the rte_malloc library, and so on. If the new instance runs and the allocated resource is not returned, either resource allocation failed or the hardware resource is lost forever.

When a slave process runs, it may have dependencies on other processes. They could have execution sequence orders; they could share the ring to communicate; they could share the same port for reception and forwarding; they could use lock structures to do exclusive access in some critical path. What happens to the dependent process(es) if the peer leaves? The consequence are varied since the dependency cases are complex. It depends on what the processed had shared. However, it is necessary to notify the peer(s) if one slave exited. Then, the peer(s) will be aware of that and wait until the new instance begins to run.

Therefore, to provide the capability to resume the new slave instance if the previous one exited, it is necessary to provide several mechanisms:

1. Keep a resource list for each slave process. Before a slave process run, the master should prepare a resource list. After it exits, the master could either delete the allocated resources and create new ones, or re-initialize those for use by the new instance.
2. Set up a notification mechanism for slave process exit cases. After the specific slave leaves, the master should be notified and then help to create a new instance. This mechanism is provided in Section *Master-slave Process Models*.
3. Use a synchronization mechanism among dependent processes. The master should have the capability to stop or kill slave processes that have a dependency on the one that has exited. Then, after the new instance of exited slave process begins to run, the dependency ones could resume or run from the start. The example sends a STOP command to slave processes dependent on the exited one, then they will exit. Thereafter, the master creates new instances for the exited slave processes.

The following diagram describes slave process recovery.

Floating Process Support

When the DPDK application runs, there is always a -c option passed in to indicate the cores that are enabled. Then, the DPDK creates a thread for each enabled core. By doing so, it creates a 1:1 mapping between the enabled core and each thread. The enabled core always has an ID, therefore, each thread has a unique core ID in the DPDK execution environment. With the ID, each thread can easily access the structures or resources exclusively belonging to it without using function parameter passing. It can easily use the rte_lcore_id() function to get the value in every function that is called.

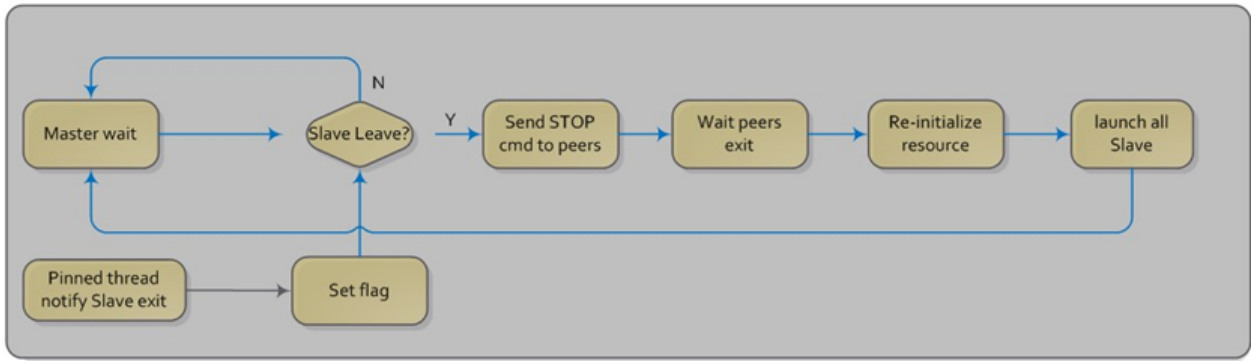


Fig. 3.16: Slave Process Recovery Process Flow

For threads/processes not created in that way, either pinned to a core or not, they will not own a unique ID and the rte_lcore_id() function will not work in the correct way. However, sometimes these threads/processes still need the unique ID mechanism to do easy access on structures or resources. For example, the DPDK mempool library provides a local cache mechanism (refer to 本地缓存) for fast element allocation and freeing. If using a non-unique ID or a fake one, a race condition occurs if two or more threads/ processes with the same core ID try to use the local cache.

Therefore, unused core IDs from the passing of parameters with the -c option are used to organize the core ID allocation array. Once the floating process is spawned, it tries to allocate a unique core ID from the array and release it on exit.

A natural way to spawn a floating process is to use the fork() function and allocate a unique core ID from the unused core ID array. However, it is necessary to write new code to provide a notification mechanism for slave exit and make sure the process recovery mechanism can work with it.

To avoid producing redundant code, the Master-Slave process model is still used to spawn floating processes, then cancel the affinity to specific cores. Besides that, clear the core ID assigned to the DPDK spawning a thread that has a 1:1 mapping with the core mask. Thereafter, get a new core ID from the unused core ID allocation array.

Run the Application

This example has a command line similar to the L2 Forwarding sample application with a few differences.

To run the application, start one copy of the l2fwd_fork binary in one terminal. Unlike the L2 Forwarding example, this example requires at least three cores since the master process will wait and be accountable for slave process recovery. The command is as follows:

```
#./build/l2fwd_fork -l 2-4 -n 4 -- -p 3 -f
```

This example provides another -f option to specify the use of floating process. If not specified, the example will use a pinned process to perform the L2 forwarding task.

To verify the recovery mechanism, proceed as follows: First, check the PID of the slave processes:

```
#ps -fe | grep l2fwd_fork
root 5136 4843 29 11:11 pts/1 00:00:05 ./build/l2fwd_fork
root 5145 5136 98 11:11 pts/1 00:00:11 ./build/l2fwd_fork
root 5146 5136 98 11:11 pts/1 00:00:11 ./build/l2fwd_fork
```

Then, kill one of the slaves:

```
#kill -9 5145
```

After 1 or 2 seconds, check whether the slave has resumed:

```
#ps -fe | grep l2fwd_fork
root 5136 4843 3 11:11 pts/1 00:00:06 ./build/l2fwd_fork
root 5247 5136 99 11:14 pts/1 00:00:01 ./build/l2fwd_fork
root 5248 5136 99 11:14 pts/1 00:00:01 ./build/l2fwd_fork
```

It can also monitor the traffic generator statics to see whether slave processes have resumed.

Explanation

As described in previous sections, not all global and static variables need to change to be accessible in multiple processes; it depends on how they are used. In this example, the statics info on packets dropped/forwarded/received count needs to be updated by the slave process, and the master needs to see the update and print them out. So, it needs to allocate a heap buffer using rte_zmalloc. In addition, if the -f option is specified, an array is needed to store the allocated core ID for the floating process so that the master can return it after a slave has exited accidentally.

```
static int
l2fwd_malloc_shared_struct(void)
{
    port_statistics = rte_zmalloc("port_stat", sizeof(struct l2fwd_port_statistics) *␣
↪RTE_MAX_ETHPORTS, 0);

    if (port_statistics == NULL)
        return -1;

    /* allocate mapping_id array */

    if (float_proc) {
        int i;

        mapping_id = rte_malloc("mapping_id", sizeof(unsigned) * RTE_MAX_LCORE, 0);
        if (mapping_id == NULL)
            return -1;

        for (i = 0 ;i < RTE_MAX_LCORE; i++)
            mapping_id[i] = INVALID_MAPPING_ID;

    }
    return 0;
}
```

For each slave process, packets are received from one port and forwarded to another port that another slave is operating on. If the other slave exits accidentally, the port it is operating on may not work normally, so the first slave cannot forward packets to that port. There is a dependency on the port in this case. So, the master should recognize the dependency. The following is the code to detect this dependency:

```
for (portid = 0; portid < nb_ports; portid++) {
    /* skip ports that are not enabled */

    if ((l2fwd_enabled_port_mask & (1 << portid)) == 0)
        continue;

    /* Find pair ports' lcores */

    find_lcore = find_pair_lcore = 0;
    pair_port = l2fwd_dst_ports[portid];
```

```
    for (i = 0; i < RTE_MAX_LCORE; i++) {
        if (!rte_lcore_is_enabled(i))
            continue;

        for (j = 0; j < lcore_queue_conf[i].n_rx_port;j++) {
            if (lcore_queue_conf[i].rx_port_list[j] == portid) {
                lcore = i;
                find_lcore = 1;
                break;
            }

            if (lcore_queue_conf[i].rx_port_list[j] == pair_port) {
                pair_lcore = i;
                find_pair_lcore = 1;
                break;
            }
        }

        if (find_lcore && find_pair_lcore)
            break;
    }

    if (!find_lcore || !find_pair_lcore)
        rte_exit(EXIT_FAILURE, "Not find port=%d pair\\n", portid);

    printf("lcore %u and %u paired\\n", lcore, pair_lcore);

    lcore_resource[lcore].pair_id = pair_lcore;
    lcore_resource[pair_lcore].pair_id = lcore;
}
```

Before launching the slave process, it is necessary to set up the communication channel between the master and slave so that the master can notify the slave if its peer process with the dependency exited. In addition, the master needs to register a callback function in the case where a specific slave exited.

```
for (i = 0; i < RTE_MAX_LCORE; i++) {
    if (lcore_resource[i].enabled) {
        /* Create ring for master and slave communication */

        ret = create_ms_ring(i);
        if (ret != 0)
            rte_exit(EXIT_FAILURE, "Create ring for lcore=%u failed",i);

        if (flib_register_slave_exit_notify(i,slave_exit_cb) != 0)
            rte_exit(EXIT_FAILURE, "Register master_trace_slave_exit failed");
    }
}
```

After launching the slave process, the master waits and prints out the port statics periodically. If an event indicating that a slave process exited is detected, it sends the STOP command to the peer and waits until it has also exited. Then, it tries to clean up the execution environment and prepare new resources. Finally, the new slave instance is launched.

```
while (1) {
    sleep(1);
    cur_tsc = rte_rdtsc();
    diff_tsc = cur_tsc - prev_tsc;
```

```
    /* if timer is enabled */

    if (timer_period > 0) {
        /* advance the timer */
        timer_tsc += diff_tsc;

        /* if timer has reached its timeout */
        if (unlikely(timer_tsc >= (uint64_t) timer_period)) {
            print_stats();

            /* reset the timer */
            timer_tsc = 0;
        }
    }

    prev_tsc = cur_tsc;

    /* Check any slave need restart or recreate */

    rte_spinlock_lock(&res_lock);

    for (i = 0; i < RTE_MAX_LCORE; i++) {
        struct lcore_resource_struct *res = &lcore_resource[i];
        struct lcore_resource_struct *pair = &lcore_resource[res->pair_id];

        /* If find slave exited, try to reset pair */

        if (res->enabled && res->flags && pair->enabled) {
            if (!pair->flags) {
                master_sendcmd_with_ack(pair->lcore_id, CMD_STOP);
                rte_spinlock_unlock(&res_lock);
                sleep(1);
                rte_spinlock_lock(&res_lock);
                if (pair->flags)
                    continue;
            }

            if (reset_pair(res->lcore_id, pair->lcore_id) != 0)
                rte_exit(EXIT_FAILURE, "failed to reset slave");

            res->flags = 0;
            pair->flags = 0;
        }
    }
    rte_spinlock_unlock(&res_lock);
}
```

When the slave process is spawned and starts to run, it checks whether the floating process option is applied. If so, it clears the affinity to a specific core and also sets the unique core ID to 0. Then, it tries to allocate a new core ID. Since the core ID has changed, the resource allocated by the master cannot work, so it remaps the resource to the new core ID slot.

```
static int
l2fwd_launch_one_lcore( attribute ((unused)) void *dummy)
{
    unsigned lcore_id = rte_lcore_id();

    if (float_proc) {
```

```
        unsigned flcore_id;

        /* Change it to floating process, also change it's lcore_id */

        clear_cpu_affinity();

        RTE_PER_LCORE(_lcore_id) = 0;

        /* Get a lcore_id */

        if (flib_assign_lcore_id() < 0 ) {
            printf("flib_assign_lcore_id failed\n");
            return -1;
        }

        flcore_id = rte_lcore_id();

        /* Set mapping id, so master can return it after slave exited */

        mapping_id[lcore_id] = flcore_id;
        printf("Org lcore_id = %u, cur lcore_id = %u\n",lcore_id, flcore_id);
        remapping_slave_resource(lcore_id, flcore_id);
    }

    l2fwd_main_loop();

    /* return lcore_id before return */
    if (float_proc) {
        flib_free_lcore_id(rte_lcore_id());
        mapping_id[lcore_id] = INVALID_MAPPING_ID;
    }
    return 0;
}
```

3.25 QoS Metering Sample Application

The QoS meter sample application is an example that demonstrates the use of DPDK to provide QoS marking and metering, as defined by RFC2697 for Single Rate Three Color Marker (srTCM) and RFC 2698 for Two Rate Three Color Marker (trTCM) algorithm.

3.25.1 Overview

The application uses a single thread for reading the packets from the RX port, metering, marking them with the appropriate color (green, yellow or red) and writing them to the TX port.

A policing scheme can be applied before writing the packets to the TX port by dropping or changing the color of the packet in a static manner depending on both the input and output colors of the packets that are processed by the meter.

The operation mode can be selected as compile time out of the following options:

- Simple forwarding
- srTCM color blind
- srTCM color aware

- srTCM color blind
- srTCM color aware

Please refer to RFC2697 and RFC2698 for details about the srTCM and trTCM configurable parameters (CIR, CBS and EBS for srTCM; CIR, PIR, CBS and PBS for trTCM).

The color blind modes are functionally equivalent with the color-aware modes when all the incoming packets are colored as green.

3.25.2 Compiling the Application

1. Go to the example directory:

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}/examples/qos_meter
```

2. Set the target (a default target is used if not specified):

Note: This application is intended as a linuxapp only.

```
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

3. Build the application:

```
make
```

3.25.3 Running the Application

The application execution command line is as below:

```
./qos_meter [EAL options] -- -p PORTMASK
```

The application is constrained to use a single core in the EAL core mask and 2 ports only in the application port mask (first port from the port mask is used for RX and the other port in the core mask is used for TX).

Refer to *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

3.25.4 Explanation

Selecting one of the metering modes is done with these defines:

```
#define APP_MODE_FWD    0
#define APP_MODE_SRTCM_COLOR_BLIND  1
#define APP_MODE_SRTCM_COLOR_AWARE  2
#define APP_MODE_TRTCM_COLOR_BLIND  3
#define APP_MODE_TRTCM_COLOR_AWARE  4

#define APP_MODE  APP_MODE_SRTCM_COLOR_BLIND
```

To simplify debugging (for example, by using the traffic generator RX side MAC address based packet filtering feature), the color is defined as the LSB byte of the destination MAC address.

The traffic meter parameters are configured in the application source code with following default values:

```
struct rte_meter_srtcm_params app_srtcm_params[] = {

    {.cir = 1000000 * 46, .cbs = 2048, .ebs = 2048},

};

struct rte_meter_trtcm_params app_trtcm_params[] = {

    {.cir = 1000000 * 46, .pir = 1500000 * 46, .cbs = 2048, .pbs = 2048},

};
```

Assuming the input traffic is generated at line rate and all packets are 64 bytes Ethernet frames (IPv4 packet size of 46 bytes) and green, the expected output traffic should be marked as shown in the following table:

Table 3.1: Output Traffic Marking

Mode	Green (Mpps)	Yellow (Mpps)	Red (Mpps)
srTCM blind	1	1	12.88
srTCM color	1	1	12.88
trTCM blind	1	0.5	13.38
trTCM color	1	0.5	13.38
FWD	14.88	0	0

To set up the policing scheme as desired, it is necessary to modify the main.h source file, where this policy is implemented as a static structure, as follows:

```
int policer_table[e_RTE_METER_COLORS][e_RTE_METER_COLORS] =
{
   { GREEN, RED, RED},
   { DROP, YELLOW, RED},
   { DROP, DROP, RED}
};
```

Where rows indicate the input color, columns indicate the output color, and the value that is stored in the table indicates the action to be taken for that particular case.

There are four different actions:

- GREEN: The packet's color is changed to green.
- YELLOW: The packet's color is changed to yellow.
- RED: The packet's color is changed to red.
- DROP: The packet is dropped.

In this particular case:

- Every packet which input and output color are the same, keeps the same color.
- Every packet which color has improved is dropped (this particular case can't happen, so these values will not be used).
- For the rest of the cases, the color is changed to red.

3.26 QoS Scheduler Sample Application

The QoS sample application demonstrates the use of the DPDK to provide QoS scheduling.

3.26.1 Overview

The architecture of the QoS scheduler application is shown in the following figure.

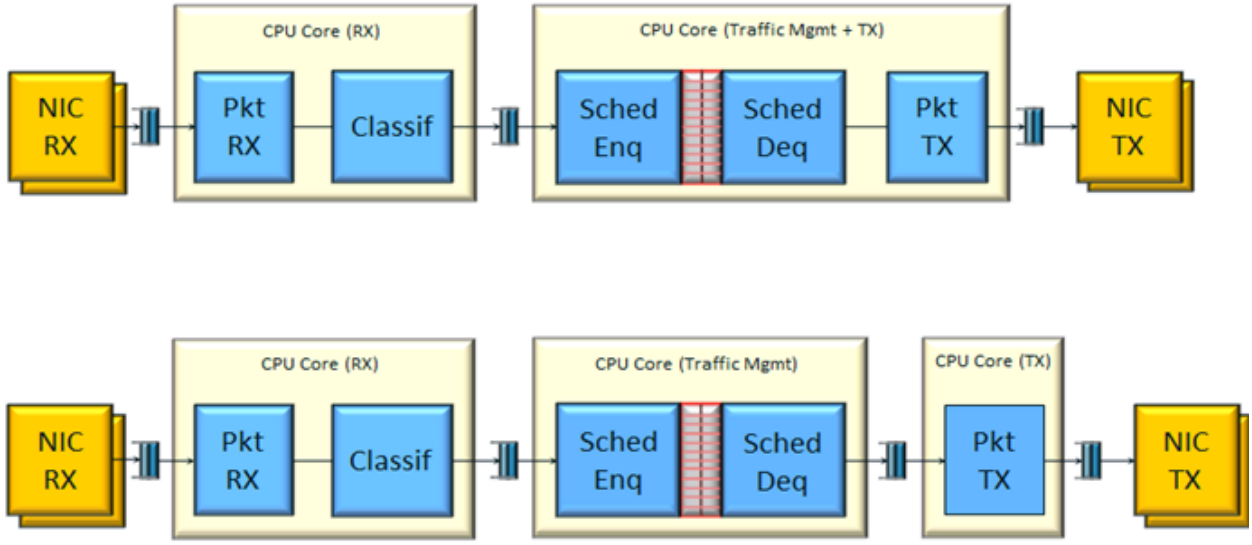


Fig. 3.17: QoS Scheduler Application Architecture

There are two flavors of the runtime execution for this application, with two or three threads per each packet flow configuration being used. The RX thread reads packets from the RX port, classifies the packets based on the double VLAN (outer and inner) and the lower two bytes of the IP destination address and puts them into the ring queue. The worker thread dequeues the packets from the ring and calls the QoS scheduler enqueue/dequeue functions. If a separate TX core is used, these are sent to the TX ring. Otherwise, they are sent directly to the TX port. The TX thread, if present, reads from the TX ring and write the packets to the TX port.

3.26.2 Compiling the Application

To compile the application:

1. Go to the sample application directory:

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}/examples/qos_sched
```

2. Set the target (a default target is used if not specified). For example:

Note: This application is intended as a linuxapp only.

```
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

3. Build the application:

```
make
```

Note: To get statistics on the sample app using the command line interface as described in the next section, DPDK must be compiled defining *CONFIG_RTE_SCHED_COLLECT_STATS*, which can be done by changing the configuration file for the specific target to be compiled.

3.26.3 Running the Application

Note: In order to run the application, a total of at least 4 G of huge pages must be set up for each of the used sockets (depending on the cores in use).

The application has a number of command line options:

```
./qos_sched [EAL options] -- <APP PARAMS>
```

Mandatory application parameters include:

- –pfc "RX PORT, TX PORT, RX LCORE, WT LCORE, TX CORE": Packet flow configuration. Multiple pfc entities can be configured in the command line, having 4 or 5 items (if TX core defined or not).

Optional application parameters include:

- -i: It makes the application to start in the interactive mode. In this mode, the application shows a command line that can be used for obtaining statistics while scheduling is taking place (see interactive mode below for more information).
- –mst n: Master core index (the default value is 1).
- –rsz "A, B, C": Ring sizes:
- A = Size (in number of buffer descriptors) of each of the NIC RX rings read by the I/O RX lcores (the default value is 128).
- B = Size (in number of elements) of each of the software rings used by the I/O RX lcores to send packets to worker lcores (the default value is 8192).
- C = Size (in number of buffer descriptors) of each of the NIC TX rings written by worker lcores (the default value is 256)
- –bsz "A, B, C, D": Burst sizes
- A = I/O RX lcore read burst size from the NIC RX (the default value is 64)
- B = I/O RX lcore write burst size to the output software rings, worker lcore read burst size from input software rings,QoS enqueue size (the default value is 64)
- C = QoS dequeue size (the default value is 32)
- D = Worker lcore write burst size to the NIC TX (the default value is 64)
- –msz M: Mempool size (in number of mbufs) for each pfc (default 2097152)
- –rth "A, B, C": The RX queue threshold parameters
- A = RX prefetch threshold (the default value is 8)
- B = RX host threshold (the default value is 8)

- C = RX write-back threshold (the default value is 4)
- –tth “A, B, C”: TX queue threshold parameters
- A = TX prefetch threshold (the default value is 36)
- B = TX host threshold (the default value is 0)
- C = TX write-back threshold (the default value is 0)
- –cfg FILE: Profile configuration to load

Refer to *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

The profile configuration file defines all the port/subport/pipe/traffic class/queue parameters needed for the QoS scheduler configuration.

The profile file has the following format:

```
; port configuration [port]

frame overhead = 24
number of subports per port = 1
number of pipes per subport = 4096
queue sizes = 64 64 64 64

; Subport configuration

[subport 0]
tb rate = 1250000000; Bytes per second
tb size = 1000000; Bytes
tc 0 rate = 1250000000;     Bytes per second
tc 1 rate = 1250000000;     Bytes per second
tc 2 rate = 1250000000;     Bytes per second
tc 3 rate = 1250000000;     Bytes per second
tc period = 10;             Milliseconds
tc oversubscription period = 10;     Milliseconds

pipe 0-4095 = 0;        These pipes are configured with pipe profile 0

; Pipe configuration

[pipe profile 0]
tb rate = 305175; Bytes per second
tb size = 1000000; Bytes

tc 0 rate = 305175; Bytes per second
tc 1 rate = 305175; Bytes per second
tc 2 rate = 305175; Bytes per second
tc 3 rate = 305175; Bytes per second
tc period = 40; Milliseconds

tc 0 oversubscription weight = 1
tc 1 oversubscription weight = 1
tc 2 oversubscription weight = 1
tc 3 oversubscription weight = 1

tc 0 wrr weights = 1 1 1 1
tc 1 wrr weights = 1 1 1 1
tc 2 wrr weights = 1 1 1 1
```

```
tc 3 wrr weights = 1 1 1 1

; RED params per traffic class and color (Green / Yellow / Red)

[red]
tc 0 wred min = 48 40 32
tc 0 wred max = 64 64 64
tc 0 wred inv prob = 10 10 10
tc 0 wred weight = 9 9 9

tc 1 wred min = 48 40 32
tc 1 wred max = 64 64 64
tc 1 wred inv prob = 10 10 10
tc 1 wred weight = 9 9 9

tc 2 wred min = 48 40 32
tc 2 wred max = 64 64 64
tc 2 wred inv prob = 10 10 10
tc 2 wred weight = 9 9 9

tc 3 wred min = 48 40 32
tc 3 wred max = 64 64 64
tc 3 wred inv prob = 10 10 10
tc 3 wred weight = 9 9 9
```

Interactive mode

These are the commands that are currently working under the command line interface:

- Control Commands
- –quit: Quits the application.
- General Statistics
  - stats app: Shows a table with in-app calculated statistics.
  - stats port X subport Y: For a specific subport, it shows the number of packets that went through the scheduler properly and the number of packets that were dropped. The same information is shown in bytes. The information is displayed in a table separating it in different traffic classes.
  - stats port X subport Y pipe Z: For a specific pipe, it shows the number of packets that went through the scheduler properly and the number of packets that were dropped. The same information is shown in bytes. This information is displayed in a table separating it in individual queues.
- Average queue size

All of these commands work the same way, averaging the number of packets throughout a specific subset of queues.

Two parameters can be configured for this prior to calling any of these commands:

- qavg n X: n is the number of times that the calculation will take place. Bigger numbers provide higher accuracy. The default value is 10.
- qavg period X: period is the number of microseconds that will be allowed between each calculation. The default value is 100.

The commands that can be used for measuring average queue size are:

- qavg port X subport Y: Show average queue size per subport.

- qavg port X subport Y tc Z: Show average queue size per subport for a specific traffic class.
- qavg port X subport Y pipe Z: Show average queue size per pipe.
- qavg port X subport Y pipe Z tc A: Show average queue size per pipe for a specific traffic class.
- qavg port X subport Y pipe Z tc A q B: Show average queue size of a specific queue.

Example

The following is an example command with a single packet flow configuration:

```
./qos_sched -l 1,5,7 -n 4 -- --pfc "3,2,5,7" --cfg ./profile.cfg
```

This example uses a single packet flow configuration which creates one RX thread on lcore 5 reading from port 3 and a worker thread on lcore 7 writing to port 2.

Another example with 2 packet flow configurations using different ports but sharing the same core for QoS scheduler is given below:

```
./qos_sched -l 1,2,6,7 -n 4 -- --pfc "3,2,2,6,7" --pfc "1,0,2,6,7" --cfg ./profile.cfg
```

Note that independent cores for the packet flow configurations for each of the RX, WT and TX thread are also supported, providing flexibility to balance the work.

The EAL coremask/corelist is constrained to contain the default mastercore 1 and the RX, WT and TX cores only.

3.26.4 Explanation

The Port/Subport/Pipe/Traffic Class/Queue are the hierarchical entities in a typical QoS application:

- A subport represents a predefined group of users.
- A pipe represents an individual user/subscriber.
- A traffic class is the representation of a different traffic type with a specific loss rate, delay and jitter requirements; such as data voice, video or data transfers.
- A queue hosts packets from one or multiple connections of the same type belonging to the same user.

The traffic flows that need to be configured are application dependent. This application classifies based on the QinQ double VLAN tags and the IP destination address as indicated in the following table.

Table 3.2: Entity Types

Level Name	Siblings per Parent	QoS Functional Description	Selected By
Port	•	Ethernet port	Physical port
Subport	Config (8)	Traffic shaped (token bucket)	Outer VLAN tag
Pipe	Config (4k)	Traffic shaped (token bucket)	Inner VLAN tag
Traffic Class	4	TCs of the same pipe services in strict priority	Destination IP address (0.0.X.0)
Queue	4	Queue of the same TC serviced in WRR	Destination IP address (0.0.0.X)

Please refer to the "QoS Scheduler" chapter in the *DPDK Programmer's Guide* for more information about these parameters.

3.27 Quota and Watermark Sample Application

The Quota and Watermark sample application is a simple example of packet processing using Data Plane Development Kit (DPDK) that showcases the use of a quota as the maximum number of packets enqueue/dequeue at a time and low and high thresholds, or watermarks, to signal low and high ring usage respectively.

Additionally, it shows how the thresholds can be used to feedback congestion notifications to data producers by temporarily stopping processing overloaded rings and sending Ethernet flow control frames.

This sample application is split in two parts:

- qw - The core quota and watermark sample application
- qwctl - A command line tool to alter quota and watermarks while qw is running

3.27.1 Overview

The Quota and Watermark sample application performs forwarding for each packet that is received on a given port. The destination port is the adjacent port from the enabled port mask, that is, if the first four ports are enabled (port mask 0xf), ports 0 and 1 forward into each other, and ports 2 and 3 forward into each other. The MAC addresses of the forwarded Ethernet frames are not affected.

Internally, packets are pulled from the ports by the master logical core and put on a variable length processing pipeline, each stage of which being connected by rings, as shown in Fig. 3.18.

An adjustable quota value controls how many packets are being moved through the pipeline per enqueue and dequeue. Adjustable threshold values associated with the rings control a back-off mechanism that tries to prevent the pipeline from being overloaded by:

- Stopping enqueuing on rings for which the usage has crossed the high watermark threshold
- Sending Ethernet pause frames
- Only resuming enqueuing on a ring once its usage goes below a global low watermark threshold

This mechanism allows congestion notifications to go up the ring pipeline and eventually lead to an Ethernet flow control frame being send to the source.

On top of serving as an example of quota and watermark usage, this application can be used to benchmark ring based processing pipelines performance using a traffic- generator, as shown in Fig. 3.19.

3.27.2 Compiling the Application

1. Go to the example directory:

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}/examples/quota_watermark
```

2. Set the target (a default target is used if not specified). For example:

```
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

   See the *DPDK Getting Started Guide* for possible RTE_TARGET values.

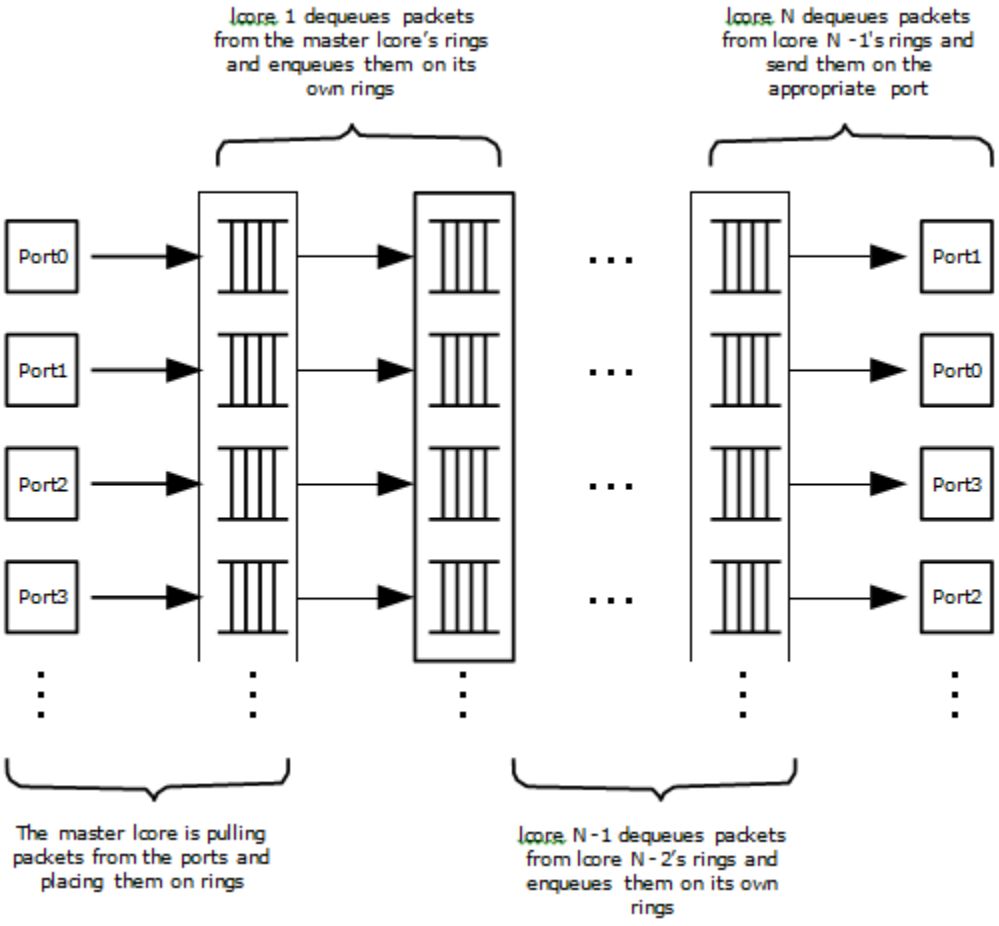


Fig. 3.18: Pipeline Overview

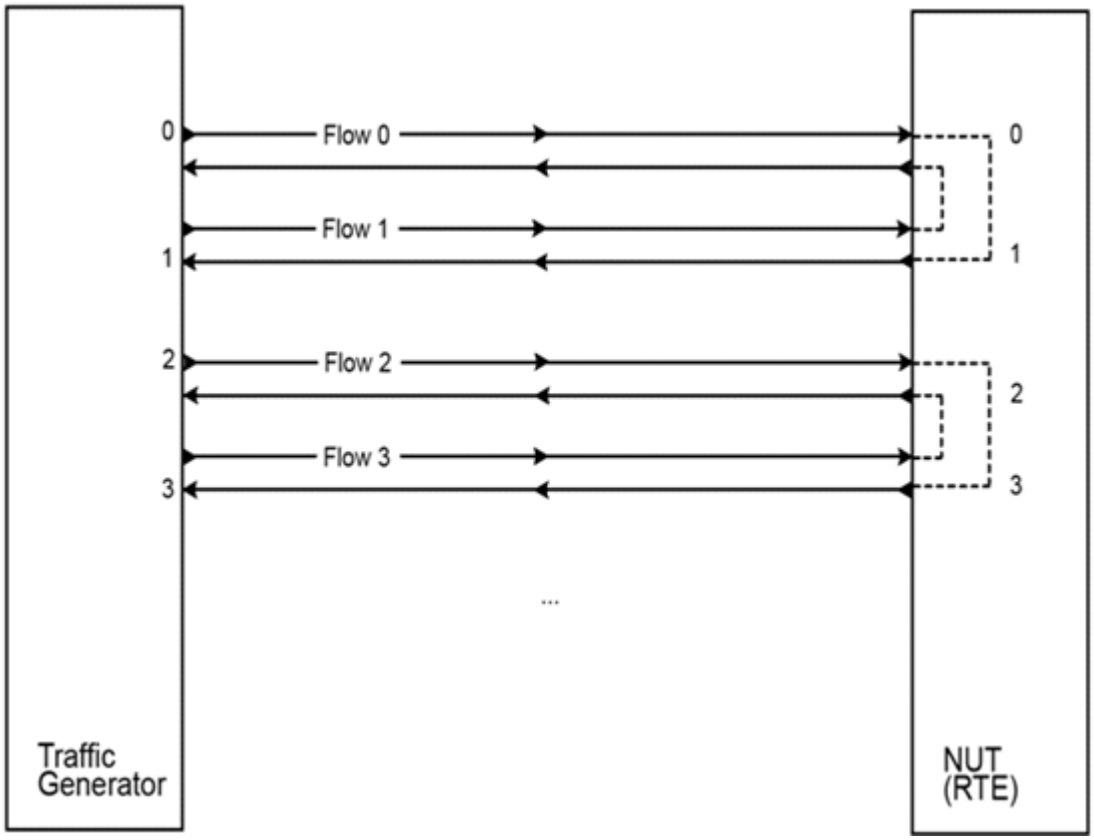


Fig. 3.19: Ring-based Processing Pipeline Performance Setup

3. Build the application:

```
make
```

3.27.3 Running the Application

The core application, qw, has to be started first.

Once it is up and running, one can alter quota and watermarks while it runs using the control application, qwctl.

Running the Core Application

The application requires a single command line option:

```
./qw/build/qw [EAL options] -- -p PORTMASK
```

where,

-p PORTMASK: A hexadecimal bitmask of the ports to configure

To run the application in a linuxapp environment with four logical cores and ports 0 and 2, issue the following command:

```
./qw/build/qw -l 0-3 -n 4 -- -p 5
```

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

Running the Control Application

The control application requires a number of command line options:

```
./qwctl/build/qwctl [EAL options] --proc-type=secondary
```

The –proc-type=secondary option is necessary for the EAL to properly initialize the control application to use the same huge pages as the core application and thus be able to access its rings.

To run the application in a linuxapp environment on logical core 0, issue the following command:

```
./qwctl/build/qwctl -l 0 -n 4 --proc-type=secondary
```

Refer to the *DPDK Getting Started* Guide for general information on running applications and the Environment Abstraction Layer (EAL) options.

qwctl is an interactive command line that let the user change variables in a running instance of qw. The help command gives a list of available commands:

```
$ qwctl > help
```

3.27.4 Code Overview

The following sections provide a quick guide to the application's source code.

Core Application - qw

EAL and Drivers Setup

The EAL arguments are parsed at the beginning of the main() function:

```
ret = rte_eal_init(argc, argv);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Cannot initialize EAL\n");

argc -= ret;
argv += ret;
```

Then, a call to init_dpdk(), defined in init.c, is made to initialize the poll mode drivers:

```
void
init_dpdk(void)
{
    int ret;

    /* Bind the drivers to usable devices */

    ret = rte_eal_pci_probe();
    if (ret < 0)
        rte_exit(EXIT_FAILURE, "rte_eal_pci_probe(): error %d\n", ret);

    if (rte_eth_dev_count() < 2)
        rte_exit(EXIT_FAILURE, "Not enough Ethernet port available\n");
}
```

To fully understand this code, it is recommended to study the chapters that relate to the *Poll Mode Driver* in the *DPDK Getting Started Guide* and the *DPDK API Reference*.

Shared Variables Setup

The quota and high and low watermark shared variables are put into an rte_memzone using a call to setup_shared_variables():

```
void
setup_shared_variables(void)
{
       const struct rte_memzone *qw_memzone;

       qw_memzone = rte_memzone_reserve(QUOTA_WATERMARK_MEMZONE_NAME,
                       3 * sizeof(int), rte_socket_id(), 0);
       if (qw_memzone == NULL)
               rte_exit(EXIT_FAILURE, "%s\n", rte_strerror(rte_errno));

       quota = qw_memzone->addr;
       low_watermark = (unsigned int *) qw_memzone->addr + 1;
       high_watermark = (unsigned int *) qw_memzone->addr + 2;
}
```

These three variables are initialized to a default value in main() and can be changed while qw is running using the qwctl control program.

Application Arguments

The qw application only takes one argument: a port mask that specifies which ports should be used by the application. At least two ports are needed to run the application and there should be an even number of ports given in the port mask.

The port mask parsing is done in parse_qw_args(), defined in args.c.

Mbuf Pool Initialization

Once the application's arguments are parsed, an mbuf pool is created. It contains a set of mbuf objects that are used by the driver and the application to store network packets:

```
/* Create a pool of mbuf to store packets */
mbuf_pool = rte_pktmbuf_pool_create("mbuf_pool", MBUF_PER_POOL, 32, 0,
                                    MBUF_DATA_SIZE, rte_socket_id());

if (mbuf_pool == NULL)
    rte_panic("%s\n", rte_strerror(rte_errno));
```

The rte_mempool is a generic structure used to handle pools of objects. In this case, it is necessary to create a pool that will be used by the driver.

The number of allocated pkt mbufs is MBUF_PER_POOL, with a data room size of MBUF_DATA_SIZE each. A per-lcore cache of 32 mbufs is kept. The memory is allocated in on the master lcore's socket, but it is possible to extend this code to allocate one mbuf pool per socket.

The rte_pktmbuf_pool_create() function uses the default mbuf pool and mbuf initializers, respectively rte_pktmbuf_pool_init() and rte_pktmbuf_init(). An advanced application may want to use the mempool API to create the mbuf pool with more control.

Ports Configuration and Pairing

Each port in the port mask is configured and a corresponding ring is created in the master lcore's array of rings. This ring is the first in the pipeline and will hold the packets directly coming from the port.

```
for (port_id = 0; port_id < RTE_MAX_ETHPORTS; port_id++)
    if (is_bit_set(port_id, portmask)) {
        configure_eth_port(port_id);
        init_ring(master_lcore_id, port_id);
    }

pair_ports();
```

The configure_eth_port() and init_ring() functions are used to configure a port and a ring respectively and are defined in init.c. They make use of the DPDK APIs defined in rte_eth.h and rte_ring.h.

pair_ports() builds the port_pairs[] array so that its key-value pairs are a mapping between reception and transmission ports. It is defined in init.c.

Logical Cores Assignment

The application uses the master logical core to poll all the ports for new packets and enqueue them on a ring associated with the port.

Each logical core except the last runs pipeline_stage() after a ring for each used port is initialized on that core. pipeline_stage() on core X dequeues packets from core X-1's rings and enqueue them on its own rings. See Fig. 3.20.

```
/* Start pipeline_stage() on all the available slave lcore but the last */

for (lcore_id = 0 ; lcore_id < last_lcore_id; lcore_id++) {
    if (rte_lcore_is_enabled(lcore_id) && lcore_id != master_lcore_id) {
        for (port_id = 0; port_id < RTE_MAX_ETHPORTS; port_id++)
            if (is_bit_set(port_id, portmask))
                init_ring(lcore_id, port_id);

            rte_eal_remote_launch(pipeline_stage, NULL, lcore_id);
    }
}
```

The last available logical core runs send_stage(), which is the last stage of the pipeline dequeuing packets from the last ring in the pipeline and sending them out on the destination port setup by pair_ports().

```
/* Start send_stage() on the last slave core */

rte_eal_remote_launch(send_stage, NULL, last_lcore_id);
```

Receive, Process and Transmit Packets

In the receive_stage() function running on the master logical core, the main task is to read ingress packets from the RX ports and enqueue them on the port's corresponding first ring in the pipeline. This is done using the following code:

```
lcore_id = rte_lcore_id();

/* Process each port round robin style */

for (port_id = 0; port_id < RTE_MAX_ETHPORTS; port_id++) {
        if (!is_bit_set(port_id, portmask))
                continue;

        ring = rings[lcore_id][port_id];

        if (ring_state[port_id] != RING_READY) {
                if (rte_ring_count(ring) > *low_watermark)
                        continue;
                else
                        ring_state[port_id] = RING_READY;
        }

        /* Enqueue received packets on the RX ring */
        nb_rx_pkts = rte_eth_rx_burst(port_id, 0, pkts,
                        (uint16_t) *quota);
        ret = rte_ring_enqueue_bulk(ring, (void *) pkts,
                        nb_rx_pkts, &free);
        if (RING_SIZE - free > *high_watermark) {
                ring_state[port_id] = RING_OVERLOADED;
                send_pause_frame(port_id, 1337);
        }

        if (ret == 0) {
```

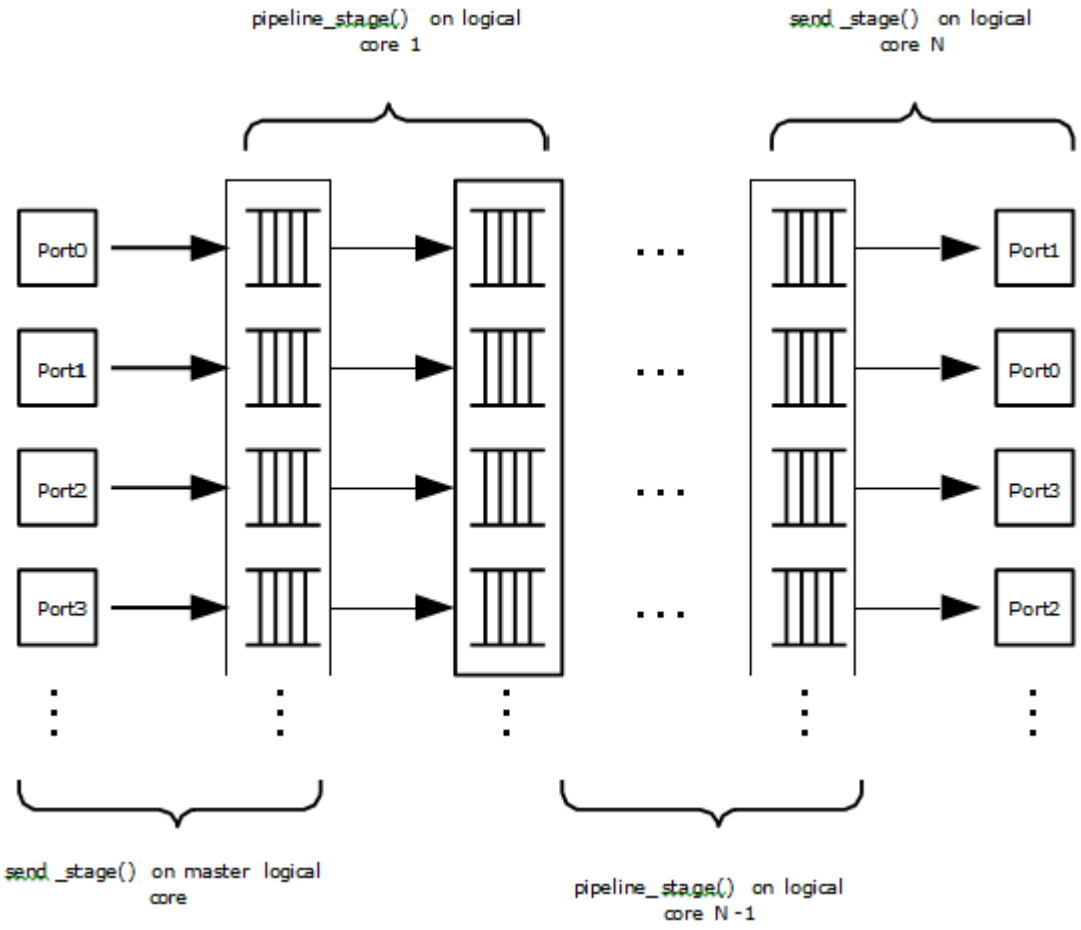


Fig. 3.20: Threads and Pipelines

```
                /*
                 * Return  mbufs to the pool,
                 * effectively dropping packets
                 */
                for (i = 0; i < nb_rx_pkts; i++)
                        rte_pktmbuf_free(pkts[i]);
        }
}
```

For each port in the port mask, the corresponding ring's pointer is fetched into ring and that ring's state is checked:

- If it is in the RING_READY state, *quota packets are grabbed from the port and put on the ring. Should this operation make the ring's usage cross its high watermark, the ring is marked as overloaded and an Ethernet flow control frame is sent to the source.
- If it is not in the RING_READY state, this port is ignored until the ring's usage crosses the *low_watermark value.

The pipeline_stage() function's task is to process and move packets from the preceding pipeline stage. This thread is running on most of the logical cores to create and arbitrarily long pipeline.

```
lcore_id = rte_lcore_id();

previous_lcore_id = get_previous_lcore_id(lcore_id);

for (port_id = 0; port_id < RTE_MAX_ETHPORTS; port_id++) {
        if (!is_bit_set(port_id, portmask))
                continue;

        tx = rings[lcore_id][port_id];
        rx = rings[previous_lcore_id][port_id];

        if (ring_state[port_id] != RING_READY) {
                if (rte_ring_count(tx) > *low_watermark)
                        continue;
                else
                        ring_state[port_id] = RING_READY;
        }

        /* Dequeue up to quota mbuf from rx */
        nb_dq_pkts = rte_ring_dequeue_burst(rx, pkts,
                        *quota, NULL);
        if (unlikely(nb_dq_pkts < 0))
                continue;

        /* Enqueue them on tx */
        ret = rte_ring_enqueue_bulk(tx, pkts,
                        nb_dq_pkts, &free);
        if (RING_SIZE - free > *high_watermark)
                ring_state[port_id] = RING_OVERLOADED;

        if (ret == 0) {

                /*
                 * Return  mbufs to the pool,
                 * effectively dropping packets
                 */
                for (i = 0; i < nb_dq_pkts; i++)
```

```
                        rte_pktmbuf_free(pkts[i]);
        }
}
```

The thread's logic works mostly like receive_stage(), except that packets are moved from ring to ring instead of port to ring.

In this example, no actual processing is done on the packets, but pipeline_stage() is an ideal place to perform any processing required by the application.

Finally, the send_stage() function's task is to read packets from the last ring in a pipeline and send them on the destination port defined in the port_pairs[] array. It is running on the last available logical core only.

```
lcore_id = rte_lcore_id();

previous_lcore_id = get_previous_lcore_id(lcore_id);

for (port_id = 0; port_id < RTE_MAX_ETHPORTS; port_id++) {
    if (!is_bit_set(port_id, portmask)) continue;

    dest_port_id = port_pairs[port_id];
    tx = rings[previous_lcore_id][port_id];

    if (rte_ring_empty(tx)) continue;

    /* Dequeue packets from tx and send them */

    nb_dq_pkts = rte_ring_dequeue_burst(tx, (void *) tx_pkts, *quota);
    nb_tx_pkts = rte_eth_tx_burst(dest_port_id, 0, tx_pkts, nb_dq_pkts);
}
```

For each port in the port mask, up to *quota packets are pulled from the last ring in its pipeline and sent on the destination port paired with the current port.

Control Application - qwctl

The qwctl application uses the rte_cmdline library to provide the user with an interactive command line that can be used to modify and inspect parameters in a running qw application. Those parameters are the global quota and low_watermark value as well as each ring's built-in high watermark.

Command Definitions

The available commands are defined in commands.c.

It is advised to use the cmdline sample application user guide as a reference for everything related to the rte_cmdline library.

Accessing Shared Variables

The setup_shared_variables() function retrieves the shared variables quota and low_watermark from the rte_memzone previously created by qw.

```
static void
setup_shared_variables(void)
```

```
{
    const struct rte_memzone *qw_memzone;

    qw_memzone = rte_memzone_lookup(QUOTA_WATERMARK_MEMZONE_NAME);
    if (qw_memzone == NULL)
        rte_exit(EXIT_FAILURE, "Couldn't find memzone\n");

    quota = qw_memzone->addr;

    low_watermark = (unsigned int *) qw_memzone->addr + 1;
    high_watermark = (unsigned int *) qw_memzone->addr + 2;
}
```

3.28 Timer Sample Application

The Timer sample application is a simple application that demonstrates the use of a timer in a DPDK application. This application prints some messages from different lcores regularly, demonstrating the use of timers.

3.28.1 Compiling the Application

1. Go to the example directory:

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}/examples/timer
```

2. Set the target (a default target is used if not specified). For example:

```
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

   See the *DPDK Getting Started Guide* for possible *RTE_TARGET* values.

3. Build the application:

```
make
```

3.28.2 Running the Application

To run the example in linuxapp environment:

```
$ ./build/timer -l 0-3 -n 4
```

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

3.28.3 Explanation

The following sections provide some explanation of the code.

Initialization and Main Loop

In addition to EAL initialization, the timer subsystem must be initialized, by calling the rte_timer_subsystem_init() function.

```
/* init EAL */

ret = rte_eal_init(argc, argv);
if (ret < 0)
    rte_panic("Cannot init EAL\n");

/* init RTE timer library */

rte_timer_subsystem_init();
```

After timer creation (see the next paragraph), the main loop is executed on each slave lcore using the well-known rte_eal_remote_launch() and also on the master.

```
/* call lcore_mainloop() on every slave lcore  */

RTE_LCORE_FOREACH_SLAVE(lcore_id) {
    rte_eal_remote_launch(lcore_mainloop, NULL, lcore_id);
}

/* call it on master lcore too */

(void) lcore_mainloop(NULL);
```

The main loop is very simple in this example:

```
while (1) {
    /*
     *   Call the timer handler on each core: as we don't
     *   need a very precise timer, so only call
     *   rte_timer_manage() every ~10ms (at 2 GHz). In a real
     *   application, this will enhance performances as
     *   reading the HPET timer is not efficient.
     */

    cur_tsc = rte_rdtsc();

    diff_tsc = cur_tsc - prev_tsc;

    if (diff_tsc > TIMER_RESOLUTION_CYCLES) {
        rte_timer_manage();
        prev_tsc = cur_tsc;
    }
}
```

As explained in the comment, it is better to use the TSC register (as it is a per-lcore register) to check if the rte_timer_manage() function must be called or not. In this example, the resolution of the timer is 10 milliseconds.

Managing Timers

In the main() function, the two timers are initialized. This call to rte_timer_init() is necessary before doing any other operation on the timer structure.

```
/* init timer structures */

rte_timer_init(&timer0);
rte_timer_init(&timer1);
```

Then, the two timers are configured:

- The first timer (timer0) is loaded on the master lcore and expires every second. Since the PERIODICAL flag is provided, the timer is reloaded automatically by the timer subsystem. The callback function is timer0_cb().
- The second timer (timer1) is loaded on the next available lcore every 333 ms. The SINGLE flag means that the timer expires only once and must be reloaded manually if required. The callback function is timer1_cb().

```
/* load timer0, every second, on master lcore, reloaded automatically */

hz = rte_get_hpet_hz();

lcore_id = rte_lcore_id();

rte_timer_reset(&timer0, hz, PERIODICAL, lcore_id, timer0_cb, NULL);

/* load timer1, every second/3, on next lcore, reloaded manually */

lcore_id = rte_get_next_lcore(lcore_id, 0, 1);

rte_timer_reset(&timer1, hz/3, SINGLE, lcore_id, timer1_cb, NULL);
```

The callback for the first timer (timer0) only displays a message until a global counter reaches 20 (after 20 seconds). In this case, the timer is stopped using the rte_timer_stop() function.

```
/* timer0 callback */

static void
timer0_cb( attribute ((unused)) struct rte_timer *tim, __attribute ((unused)) void␣
↪*arg)
{
    static unsigned counter = 0;

    unsigned lcore_id = rte_lcore_id();

    printf("%s() on lcore %u\n", FUNCTION , lcore_id);

    /* this timer is automatically reloaded until we decide to stop it, when counter␣
↪reaches 20. */

    if ((counter ++) == 20)
        rte_timer_stop(tim);
}
```

The callback for the second timer (timer1) displays a message and reloads the timer on the next lcore, using the rte_timer_reset() function:

```
/* timer1 callback */

static void
timer1_cb( attribute ((unused)) struct rte_timer *tim, _attribute ((unused)) void␣
↪*arg)
{
```

```
    unsigned lcore_id = rte_lcore_id();
    uint64_t hz;

    printf("%s() on lcore %u\\n", FUNCTION , lcore_id);

    /* reload it on another lcore */

    hz = rte_get_hpet_hz();

    lcore_id = rte_get_next_lcore(lcore_id, 0, 1);

    rte_timer_reset(&timer1, hz/3, SINGLE, lcore_id, timer1_cb, NULL);
}
```

3.29 Packet Ordering Application

The Packet Ordering sample app simply shows the impact of reordering a stream. It's meant to stress the library with different configurations for performance.

3.29.1 Overview

The application uses at least three CPU cores:

- RX core (maser core) receives traffic from the NIC ports and feeds Worker cores with traffic through SW queues.
- Worker core (slave core) basically do some light work on the packet. Currently it modifies the output port of the packet for configurations with more than one port enabled.
- TX Core (slave core) receives traffic from Worker cores through software queues, inserts out-of-order packets into reorder buffer, extracts ordered packets from the reorder buffer and sends them to the NIC ports for transmission.

3.29.2 Compiling the Application

1. Go to the example directory:

   ```
   export RTE_SDK=/path/to/rte_sdk
   cd ${RTE_SDK}/examples/helloworld
   ```

2. Set the target (a default target is used if not specified). For example:

   ```
   export RTE_TARGET=x86_64-native-linuxapp-gcc
   ```

   See the *DPDK Getting Started* Guide for possible RTE_TARGET values.

3. Build the application:

   ```
   make
   ```

3.29.3 Running the Application

Refer to *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

Application Command Line

The application execution command line is:

```
./test-pipeline [EAL options] -- -p PORTMASK [--disable-reorder]
```

The -c EAL CPU_COREMASK option has to contain at least 3 CPU cores. The first CPU core in the core mask is the master core and would be assigned to RX core, the last to TX core and the rest to Worker cores.

The PORTMASK parameter must contain either 1 or even enabled port numbers. When setting more than 1 port, traffic would be forwarded in pairs. For example, if we enable 4 ports, traffic from port 0 to 1 and from 1 to 0, then the other pair from 2 to 3 and from 3 to 2, having [0,1] and [2,3] pairs.

The disable-reorder long option does, as its name implies, disable the reordering of traffic, which should help evaluate reordering performance impact.

3.30 VMDQ and DCB Forwarding Sample Application

The VMDQ and DCB Forwarding sample application is a simple example of packet processing using the DPDK. The application performs L2 forwarding using VMDQ and DCB to divide the incoming traffic into queues. The traffic splitting is performed in hardware by the VMDQ and DCB features of the Intel® 82599 and X710/XL710 Ethernet Controllers.

3.30.1 Overview

This sample application can be used as a starting point for developing a new application that is based on the DPDK and uses VMDQ and DCB for traffic partitioning.

The VMDQ and DCB filters work on MAC and VLAN traffic to divide the traffic into input queues on the basis of the Destination MAC address, VLAN ID and VLAN user priority fields. VMDQ filters split the traffic into 16 or 32 groups based on the Destination MAC and VLAN ID. Then, DCB places each packet into one of queues within that group, based upon the VLAN user priority field.

All traffic is read from a single incoming port (port 0) and output on port 1, without any processing being performed. With Intel® 82599 NIC, for example, the traffic is split into 128 queues on input, where each thread of the application reads from multiple queues. When run with 8 threads, that is, with the -c FF option, each thread receives and forwards packets from 16 queues.

As supplied, the sample application configures the VMDQ feature to have 32 pools with 4 queues each as indicated in Fig. 3.21. The Intel® 82599 10 Gigabit Ethernet Controller NIC also supports the splitting of traffic into 16 pools of 8 queues. While the Intel® X710 or XL710 Ethernet Controller NICs support many configurations of VMDQ pools of 4 or 8 queues each. For simplicity, only 16 or 32 pools is supported in this sample. And queues numbers for each VMDQ pool can be changed by setting CONFIG_RTE_LIBRTE_I40E_QUEUE_NUM_PER_VM in config/common_* file. The nb-pools, nb-tcs and enable-rss parameters can be passed on the command line, after the EAL parameters:

```
./build/vmdq_dcb [EAL options] -- -p PORTMASK --nb-pools NP --nb-tcs TC --enable-rss
```

where, NP can be 16 or 32, TC can be 4 or 8, rss is disabled by default.

Fig. 3.21: Packet Flow Through the VMDQ and DCB Sample Application

In Linux* user space, the application can display statistics with the number of packets received on each queue. To have the application display the statistics, send a SIGHUP signal to the running application process.

The VMDQ and DCB Forwarding sample application is in many ways simpler than the L2 Forwarding application (see *L2 Forwarding Sample Application (in Real and Virtualized Environments)*) as it performs unidirectional L2 forwarding of packets from one port to a second port. No command-line options are taken by this application apart from the standard EAL command-line options.

Note: Since VMD queues are being used for VMM, this application works correctly when VTd is disabled in the BIOS or Linux* kernel (intel_iommu=off).

3.30.2 Compiling the Application

1. Go to the examples directory:

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}/examples/vmdq_dcb
```

2. Set the target (a default target is used if not specified). For example:

```
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

   See the *DPDK Getting Started Guide* for possible RTE_TARGET values.

3. Build the application:

```
make
```

3.30.3 Running the Application

To run the example in a linuxapp environment:

```
user@target:~$ ./build/vmdq_dcb -l 0-3 -n 4 -- -p 0x3 --nb-pools 32 --nb-tcs 4
```

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

3.30.4 Explanation

The following sections provide some explanation of the code.

Initialization

The EAL, driver and PCI configuration is performed largely as in the L2 Forwarding sample application, as is the creation of the mbuf pool. See *L2 Forwarding Sample Application (in Real and Virtualized Environments)*. Where this example application differs is in the configuration of the NIC port for RX.

The VMDQ and DCB hardware feature is configured at port initialization time by setting the appropriate values in the rte_eth_conf structure passed to the rte_eth_dev_configure() API. Initially in the application, a default structure is provided for VMDQ and DCB configuration to be filled in later by the application.

```
/* empty vmdq+dcb configuration structure. Filled in programmatically */
static const struct rte_eth_conf vmdq_dcb_conf_default = {
    .rxmode = {
        .mq_mode        = ETH_MQ_RX_VMDQ_DCB,
        .split_hdr_size = 0,
        .header_split   = 0, /**< Header Split disabled */
        .hw_ip_checksum = 0, /**< IP checksum offload disabled */
        .hw_vlan_filter = 0, /**< VLAN filtering disabled */
        .jumbo_frame    = 0, /**< Jumbo Frame Support disabled */
    },
    .txmode = {
        .mq_mode = ETH_MQ_TX_VMDQ_DCB,
    },
    /*
     * should be overridden separately in code with
     * appropriate values
     */
    .rx_adv_conf = {
        .vmdq_dcb_conf = {
            .nb_queue_pools = ETH_32_POOLS,
            .enable_default_pool = 0,
            .default_pool = 0,
            .nb_pool_maps = 0,
            .pool_map = {{0, 0},},
            .dcb_tc = {0},
        },
        .dcb_rx_conf = {
            .nb_tcs = ETH_4_TCS,
            /** Traffic class each UP mapped to. */
            .dcb_tc = {0},
        },
        .vmdq_rx_conf = {
            .nb_queue_pools = ETH_32_POOLS,
            .enable_default_pool = 0,
            .default_pool = 0,
            .nb_pool_maps = 0,
            .pool_map = {{0, 0},},
        },
    },
    .tx_adv_conf = {
        .vmdq_dcb_tx_conf = {
            .nb_queue_pools = ETH_32_POOLS,
            .dcb_tc = {0},
        },
    },
};
```

The get_eth_conf() function fills in an rte_eth_conf structure with the appropriate values, based on the global vlan_tags array, and dividing up the possible user priority values equally among the individual queues (also referred to as traffic classes) within each pool. With Intel® 82599 NIC, if the number of pools is 32, then the user priority fields are allocated 2 to a queue. If 16 pools are used, then each of the 8 user priority fields is allocated to its own queue within the pool. With Intel® X710/XL710 NICs, if number of tcs is 4, and number of queues in pool is 8, then the user priority fields are allocated 2 to one tc, and a tc has 2 queues mapping to it, then RSS will determine the destination queue in 2. For the VLAN IDs, each one can be allocated to possibly multiple pools of queues, so the pools parameter in the rte_eth_vmdq_dcb_conf structure is specified as a bitmask value. For destination MAC, each VMDQ pool will be assigned with a MAC address. In this sample, each VMDQ pool is assigned to the MAC like 52:54:00:12:<port_id>:<pool_id>, that is, the MAC of VMDQ pool 2 on port 1 is 52:54:00:12:01:02.

```
const uint16_t vlan_tags[] = {
    0, 1, 2, 3, 4, 5, 6, 7,
    8, 9, 10, 11, 12, 13, 14, 15,
    16, 17, 18, 19, 20, 21, 22, 23,
    24, 25, 26, 27, 28, 29, 30, 31
};

/* pool mac addr template, pool mac addr is like: 52 54 00 12 port# pool# */
static struct ether_addr pool_addr_template = {
    .addr_bytes = {0x52, 0x54, 0x00, 0x12, 0x00, 0x00}
};

/* Builds up the correct configuration for vmdq+dcb based on the vlan tags array
 * given above, and the number of traffic classes available for use. */
static inline int
get_eth_conf(struct rte_eth_conf *eth_conf)
{
    struct rte_eth_vmdq_dcb_conf conf;
    struct rte_eth_vmdq_rx_conf  vmdq_conf;
    struct rte_eth_dcb_rx_conf   dcb_conf;
    struct rte_eth_vmdq_dcb_tx_conf tx_conf;
    uint8_t i;

    conf.nb_queue_pools = (enum rte_eth_nb_pools)num_pools;
    vmdq_conf.nb_queue_pools = (enum rte_eth_nb_pools)num_pools;
    tx_conf.nb_queue_pools = (enum rte_eth_nb_pools)num_pools;
    conf.nb_pool_maps = num_pools;
    vmdq_conf.nb_pool_maps = num_pools;
    conf.enable_default_pool = 0;
    vmdq_conf.enable_default_pool = 0;
    conf.default_pool = 0; /* set explicit value, even if not used */
    vmdq_conf.default_pool = 0;

    for (i = 0; i < conf.nb_pool_maps; i++) {
        conf.pool_map[i].vlan_id = vlan_tags[i];
        vmdq_conf.pool_map[i].vlan_id = vlan_tags[i];
        conf.pool_map[i].pools = 1UL << i ;
        vmdq_conf.pool_map[i].pools = 1UL << i;
    }
    for (i = 0; i < ETH_DCB_NUM_USER_PRIORITIES; i++){
        conf.dcb_tc[i] = i % num_tcs;
        dcb_conf.dcb_tc[i] = i % num_tcs;
        tx_conf.dcb_tc[i] = i % num_tcs;
    }
    dcb_conf.nb_tcs = (enum rte_eth_nb_tcs)num_tcs;
    (void)(rte_memcpy(eth_conf, &vmdq_dcb_conf_default, sizeof(*eth_conf)));
    (void)(rte_memcpy(&eth_conf->rx_adv_conf.vmdq_dcb_conf, &conf,
               sizeof(conf)));
    (void)(rte_memcpy(&eth_conf->rx_adv_conf.dcb_rx_conf, &dcb_conf,
               sizeof(dcb_conf)));
    (void)(rte_memcpy(&eth_conf->rx_adv_conf.vmdq_rx_conf, &vmdq_conf,
               sizeof(vmdq_conf)));
    (void)(rte_memcpy(&eth_conf->tx_adv_conf.vmdq_dcb_tx_conf, &tx_conf,
               sizeof(tx_conf)));
    if (rss_enable) {
        eth_conf->rxmode.mq_mode= ETH_MQ_RX_VMDQ_DCB_RSS;
        eth_conf->rx_adv_conf.rss_conf.rss_hf = ETH_RSS_IP |
                            ETH_RSS_UDP |
```

```
                            ETH_RSS_TCP |
                            ETH_RSS_SCTP;
    }
    return 0;
}

......

/* Set mac for each pool.*/
for (q = 0; q < num_pools; q++) {
    struct ether_addr mac;
    mac = pool_addr_template;
    mac.addr_bytes[4] = port;
    mac.addr_bytes[5] = q;
    printf("Port %u vmdq pool %u set mac %02x:%02x:%02x:%02x:%02x:%02x\n",
        port, q,
        mac.addr_bytes[0], mac.addr_bytes[1],
        mac.addr_bytes[2], mac.addr_bytes[3],
        mac.addr_bytes[4], mac.addr_bytes[5]);
    retval = rte_eth_dev_mac_addr_add(port, &mac,
            q + vmdq_pool_base);
    if (retval) {
        printf("mac addr add failed at pool %d\n", q);
        return retval;
    }
}
```

Once the network port has been initialized using the correct VMDQ and DCB values, the initialization of the port's RX and TX hardware rings is performed similarly to that in the L2 Forwarding sample application. See *L2 Forwarding Sample Application (in Real and Virtualized Environments)* for more information.

Statistics Display

When run in a linuxapp environment, the VMDQ and DCB Forwarding sample application can display statistics showing the number of packets read from each RX queue. This is provided by way of a signal handler for the SIGHUP signal, which simply prints to standard output the packet counts in grid form. Each row of the output is a single pool with the columns being the queue number within that pool.

To generate the statistics output, use the following command:

```
user@host$ sudo killall -HUP vmdq_dcb_app
```

Please note that the statistics output will appear on the terminal where the vmdq_dcb_app is running, rather than the terminal from which the HUP signal was sent.

3.31 Vhost Sample Application

The vhost sample application demonstrates integration of the Data Plane Development Kit (DPDK) with the Linux* KVM hypervisor by implementing the vhost-net offload API. The sample application performs simple packet switching between virtual machines based on Media Access Control (MAC) address or Virtual Local Area Network (VLAN) tag. The splitting of Ethernet traffic from an external switch is performed in hardware by the Virtual Machine Device Queues (VMDQ) and Data Center Bridging (DCB) features of the Intel® 82599 10 Gigabit Ethernet Controller.

3.31.1 Testing steps

This section shows the steps how to test a typical PVP case with this vhost-switch sample, whereas packets are received from the physical NIC port first and enqueued to the VM's Rx queue. Through the guest testpmd's default forwarding mode (io forward), those packets will be put into the Tx queue. The vhost-switch example, in turn, gets the packets and puts back to the same physical NIC port.

Build

Follow the *Getting Started Guide for Linux* on generic info about environment setup and building DPDK from source.

In this example, you need build DPDK both on the host and inside guest. Also, you need build this example.

```
export RTE_SDK=/path/to/dpdk_source
export RTE_TARGET=x86_64-native-linuxapp-gcc

cd ${RTE_SDK}/examples/vhost
make
```

Start the vswitch example

```
./vhost-switch -l 0-3 -n 4 --socket-mem 1024  \
     -- --socket-file /tmp/sock0 --client \
     ...
```

Check the *Parameters* section for the explanations on what do those parameters mean.

Start the VM

```
qemu-system-x86_64 -machine accel=kvm -cpu host \
    -m $mem -object memory-backend-file,id=mem,size=$mem,mem-path=/dev/hugepages,
↪share=on \
            -mem-prealloc -numa node,memdev=mem \
    \
    -chardev socket,id=char1,path=/tmp/sock0,server \
    -netdev type=vhost-user,id=hostnet1,chardev=char1  \
    -device virtio-net-pci,netdev=hostnet1,id=net1,mac=52:54:00:00:00:14 \
    ...
```

Note: For basic vhost-user support, QEMU 2.2 (or above) is required. For some specific features, a higher version might be need. Such as QEMU 2.7 (or above) for the reconnect feature.

Run testpmd inside guest

Make sure you have DPDK built inside the guest. Also make sure the corresponding virtio-net PCI device is bond to a uio driver, which could be done by:

```
modprobe uio_pci_generic
$RTE_SDK/usertools/dpdk-devbind.py -b=uio_pci_generic 0000:00:04.0
```

Then start testpmd for packet forwarding testing.

```
./x86_64-native-gcc/app/testpmd -l 0-1 -- -i
> start tx_first
```

3.31.2 Inject packets

While a virtio-net is connected to vhost-switch, a VLAN tag starts with 1000 is assigned to it. So make sure configure your packet generator with the right MAC and VLAN tag, you should be able to see following log from the vhost-switch console. It means you get it work:

```
VHOST_DATA: (0) mac 52:54:00:00:00:14 and vlan 1000 registered
```

3.31.3 Parameters

–socket-file path Specifies the vhost-user socket file path.

–client DPDK vhost-user will act as the client mode when such option is given. In the client mode, QEMU will create the socket file. Otherwise, DPDK will create it. Put simply, it's the server to create the socket file.

–vm2vm mode The vm2vm parameter sets the mode of packet switching between guests in the host.

- 0 disables vm2vm, impling that VM's packets will always go to the NIC port.
- 1 means a normal mac lookup packet routing.
- 2 means hardware mode packet forwarding between guests, it allows packets go to the NIC port, hardware L2 switch will determine which guest the packet should forward to or need send to external, which bases on the packet destination MAC address and VLAN tag.

–mergeable 0|1 Set 0/1 to disable/enable the mergeable Rx feature. It's disabled by default.

–stats interval The stats parameter controls the printing of virtio-net device statistics. The parameter specifies an interval (in unit of seconds) to print statistics, with an interval of 0 seconds disabling statistics.

–rx-retry 0|1 The rx-retry option enables/disables enqueue retries when the guests Rx queue is full. This feature resolves a packet loss that is observed at high data rates, by allowing it to delay and retry in the receive path. This option is enabled by default.

–rx-retry-num num The rx-retry-num option specifies the number of retries on an Rx burst, it takes effect only when rx retry is enabled. The default value is 4.

–rx-retry-delay msec The rx-retry-delay option specifies the timeout (in micro seconds) between retries on an RX burst, it takes effect only when rx retry is enabled. The default value is 15.

–dequeue-zero-copy Dequeue zero copy will be enabled when this option is given.

–vlan-strip 0|1 VLAN strip option is removed, because different NICs have different behaviors when disabling VLAN strip. Such feature, which heavily depends on hardware, should be removed from this example to reduce confusion. Now, VLAN strip is enabled and cannot be disabled.

3.31.4 Common Issues

- QEMU fails to allocate memory on hugetlbfs, with an error like the following:

```
file_ram_alloc: can't mmap RAM pages: Cannot allocate memory
```

When running QEMU the above error indicates that it has failed to allocate memory for the Virtual Machine on the hugetlbfs. This is typically due to insufficient hugepages being free to support the allocation request. The number of free hugepages can be checked as follows:

```
cat /sys/kernel/mm/hugepages/hugepages-<pagesize>/nr_hugepages
```

The command above indicates how many hugepages are free to support QEMU's allocation request.

- vhost-user will not work with QEMU without the `-mem-prealloc` option

  The current implementation works properly only when the guest memory is pre-allocated.

- vhost-user will not work with a QEMU version without shared memory mapping:

  Make sure `share=on` QEMU option is given.

- Failed to build DPDK in VM

  Make sure "-cpu host" QEMU option is given.

3.32 Netmap Compatibility Sample Application

3.32.1 Introduction

The Netmap compatibility library provides a minimal set of APIs to give programs written against the Netmap APIs the ability to be run, with minimal changes to their source code, using the DPDK to perform the actual packet I/O.

Since Netmap applications use regular system calls, like `open()`, `ioctl()` and `mmap()` to communicate with the Netmap kernel module performing the packet I/O, the `compat_netmap` library provides a set of similar APIs to use in place of those system calls, effectively turning a Netmap application into a DPDK application.

The provided library is currently minimal and doesn't support all the features that Netmap supports, but is enough to run simple applications, such as the bridge example detailed below.

Knowledge of Netmap is required to understand the rest of this section. Please refer to the Netmap distribution for details about Netmap.

3.32.2 Available APIs

The library provides the following drop-in replacements for system calls usually used in Netmap applications:

- `rte_netmap_close()`
- `rte_netmap_ioctl()`
- `rte_netmap_open()`
- `rte_netmap_mmap()`
- `rte_netmap_poll()`

They use the same signature as their libc counterparts, and can be used as drop-in replacements in most cases.

3.32.3 Caveats

Given the difference between the way Netmap and the DPDK approach packet I/O, there are caveats and limitations to be aware of when trying to use the `compat_netmap` library, the most important of these are listed below. These may change as the library is updated:

- Any system call that can potentially affect file descriptors cannot be used with a descriptor returned by the `rte_netmap_open()` function.

Note that:

- The `rte_netmap_mmap()` function merely returns the address of a DPDK memzone. The address, length, flags, offset, and other arguments are ignored.
- The `rte_netmap_poll()` function only supports infinite (negative) or zero time outs. It effectively turns calls to the `poll()` system call made in a Netmap application into polling of the DPDK ports, changing the semantics of the usual POSIX defined poll.
- Not all of Netmap's features are supported: host rings, slot flags and so on are not supported or are simply not relevant in the DPDK model.
- The Netmap manual page states that "*a device obtained through /dev/netmap also supports the ioctl supported by network devices*". This is not the case with this compatibility layer.
- The Netmap kernel module exposes a sysfs interface to change some internal parameters, such as the size of the shared memory region. This interface is not available when using this compatibility layer.

3.32.4 Porting Netmap Applications

Porting Netmap applications typically involves two major steps:

- Changing the system calls to use their `compat_netmap` library counterparts.
- Adding further DPDK initialization code.

Since the `compat_netmap` functions have the same signature as the usual libc calls, the change is trivial in most cases.

The usual DPDK initialization code involving `rte_eal_init()` and `rte_eal_pci_probe()` has to be added to the Netmap application in the same way it is used in all other DPDK sample applications. Please refer to the *DPDK Programmer's Guide* and example source code for details about initialization.

In addition of the regular DPDK initialization code, the ported application needs to call initialization functions for the `compat_netmap` library, namely `rte_netmap_init()` and `rte_netmap_init_port()`.

These two initialization functions take `compat_netmap` specific data structures as parameters: `struct rte_netmap_conf` and `struct rte_netmap_port_conf`. The structures' fields are Netmap related and are self-explanatory for developers familiar with Netmap. They are defined in `$RTE_SDK/examples/netmap_compat/lib/compat_netmap.h`.

The bridge application is an example largely based on the bridge example shipped with the Netmap distribution. It shows how a minimal Netmap application with minimal and straightforward source code changes can be run on top of the DPDK. Please refer to `$RTE_SDK/examples/netmap_compat/bridge/bridge.c` for an example of a ported application.

3.32.5 Compiling the "bridge" Sample Application

1. Go to the example directory:

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}/examples/netmap_compat
```

2. Set the target (a default target is used if not specified). For example:

```
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

See the *DPDK Getting Started Guide for Linux* for possible `RTE_TARGET` values.

3. Build the application:

```
make
```

3.32.6 Running the "bridge" Sample Application

The application requires a single command line option:

```
./build/bridge [EAL options] -- -i INTERFACE_A [-i INTERFACE_B]
```

where,

- `-i INTERFACE`: Interface (DPDK port number) to use.

  If a single `-i` parameter is given, the interface will send back all the traffic it receives. If two `-i` parameters are given, the two interfaces form a bridge, where traffic received on one interface is replicated and sent to the other interface.

For example, to run the application in a linuxapp environment using port 0 and 2:

```
./build/bridge [EAL options] -- -i 0 -i 2
```

Refer to the *DPDK Getting Started Guide for Linux* for general information on running applications and the Environment Abstraction Layer (EAL) options.

Note that unlike a traditional bridge or the `l2fwd` sample application, no MAC address changes are done on the frames. Do not forget to take this into account when configuring a traffic generators and testing this sample application.

3.33 Internet Protocol (IP) Pipeline Application

3.33.1 Application overview

The *Internet Protocol (IP) Pipeline* application is intended to be a vehicle for rapid development of packet processing applications running on multi-core CPUs.

The application provides a library of reusable functional blocks called pipelines. These pipelines can be seen as prefabricated blocks that can be instantiated and inter-connected through packet queues to create complete applications (super-pipelines).

Pipelines are created and inter-connected through the application configuration file. By using different configuration files, different applications are effectively created, therefore this application can be seen as an application generator. The configuration of each pipeline can be updated at run-time through the application Command Line Interface (CLI).

Main application components are:

A Library of reusable pipelines

- Each pipeline represents a functional block, e.g. flow classification, firewall, routing, master, etc.
- Each pipeline type can be instantiated several times in the same application, which each instance configured separately and mapped to a single CPU core. Each CPU core can run one or several pipeline instances, which can be of same or different type.

- Pipeline instances are inter-connected through packet queues (for packet processing) and message queues (for run-time configuration).
- Pipelines are implemented using DPDK Packet Framework.
- More pipeline types can always be built and added to the existing pipeline types.

The Configuration file

- The configuration file defines the application structure. By using different configuration files, different applications are created.
- All the application resources are created and configured through the application configuration file: pipeline instances, buffer pools, links (i.e. network interfaces), hardware device RX/TX queues, software queues, traffic manager devices, EAL startup arguments, etc.
- The configuration file syntax is "define by reference", meaning that resources are defined as they are referenced. First time a resource name is detected, it is registered with default parameters. Optionally, the resource parameters can be further refined through a configuration file section dedicated to that resource.
- Command Line Interface (CLI)

Global CLI commands: link configuration, etc.

- Common pipeline CLI commands: ping (keep-alive), statistics, etc.
- Pipeline type specific CLI commands: used to configure instances of specific pipeline type. These commands are registered with the application when the pipeline type is registered. For example, the commands for routing pipeline instances include: route add, route delete, route list, etc.
- CLI commands can be grouped into scripts that can be invoked at initialization and at runtime.

3.33.2 Design goals

Rapid development

This application enables rapid development through quick connectivity of standard components called pipelines. These components are built using DPDK Packet Framework and encapsulate packet processing features at different levels: ports, tables, actions, pipelines and complete applications.

Pipeline instances are instantiated, configured and inter-connected through low complexity configuration files loaded during application initialization. Each pipeline instance is mapped to a single CPU core, with each CPU core able to run one or multiple pipeline instances of same or different types. By loading a different configuration file, a different application is effectively started.

Flexibility

Each packet processing application is typically represented as a chain of functional stages which is often called the functional pipeline of the application. These stages are mapped to CPU cores to create chains of CPU cores (pipeline model), clusters of CPU cores (run-to-completion model) or chains of clusters of CPU cores (hybrid model).

This application allows all the above programming models. By applying changes to the configuration file, the application provides the flexibility to reshuffle its building blocks in different ways until the configuration providing the best performance is identified.

Move pipelines around

The mapping of pipeline instances to CPU cores can be reshuffled through the configuration file. One or several pipeline instances can be mapped to the same CPU core.

Fig. 3.22: Example of moving pipeline instances across different CPU cores

Move tables around

There is some degree of flexibility for moving tables from one pipeline instance to another. Based on the configuration arguments passed to each pipeline instance in the configuration file, specific tables can be enabled or disabled. This way, a specific table can be "moved" from pipeline instance A to pipeline instance B by simply disabling its associated functionality for pipeline instance A while enabling it for pipeline instance B.

Due to requirement to have simple syntax for the configuration file, moving tables across different pipeline instances is not as flexible as the mapping of pipeline instances to CPU cores, or mapping actions to pipeline tables. Complete flexibility in moving tables from one pipeline to another could be achieved through a complex pipeline description language that would detail the structural elements of the pipeline (ports, tables and actions) and their connectivity, resulting in complex syntax for the configuration file, which is not acceptable. Good configuration file readability through simple syntax is preferred.

Example: the IP routing pipeline can run the routing function only (with ARP function run by a different pipeline instance), or it can run both the routing and ARP functions as part of the same pipeline instance.

Fig. 3.23: Example of moving tables across different pipeline instances

Move actions around

When it makes sense, packet processing actions can be moved from one pipeline instance to another. Based on the configuration arguments passed to each pipeline instance in the configuration file, specific actions can be enabled or disabled. This way, a specific action can be "moved" from pipeline instance A to pipeline instance B by simply disabling its associated functionality for pipeline instance A while enabling it for pipeline instance B.

Example: The flow actions of accounting, traffic metering, application identification, NAT, etc can be run as part of the flow classification pipeline instance or split across several flow actions pipeline instances, depending on the number of flow instances and their compute requirements.

Fig. 3.24: Example of moving actions across different tables and pipeline instances

Performance

Performance of the application is the highest priority requirement. Flexibility is not provided at the expense of performance.

The purpose of flexibility is to provide an incremental development methodology that allows monitoring the performance evolution:

- Apply incremental changes in the configuration (e.g. mapping on pipeline instances to CPU cores) in order to identify the configuration providing the best performance for a given application;
- Add more processing incrementally (e.g. by enabling more actions for specific pipeline instances) until the application is feature complete while checking the performance impact at each step.

Debug capabilities

The application provides a significant set of debug capabilities:

- Command Line Interface (CLI) support for statistics polling: pipeline instance ping (keep-alive checks), pipeline instance statistics per input port/output port/table, link statistics, etc;
- Logging: Turn on/off application log messages based on priority level;

3.33.3 Running the application

The application startup command line is:

```
ip_pipeline [-f CONFIG_FILE] [-s SCRIPT_FILE] -p PORT_MASK [-l LOG_LEVEL]
```

The application startup arguments are:

`-f CONFIG_FILE`

- Optional: Yes
- Default: `./config/ip_pipeline.cfg`
- Argument: Path to the configuration file to be loaded by the application. Please refer to the *Configuration file syntax* for details on how to write the configuration file.

`-s SCRIPT_FILE`

- Optional: Yes
- Default: Not present
- Argument: Path to the CLI script file to be run by the master pipeline at application startup. No CLI script file will be run at startup of this argument is not present.

`-p PORT_MASK`

- Optional: No
- Default: N/A
- Argument: Hexadecimal mask of NIC port IDs to be used by the application. First port enabled in this mask will be referenced as LINK0 as part of the application configuration file, next port as LINK1, etc.

`-l LOG_LEVEL`

- Optional: Yes
- Default: 1 (High priority)
- Argument: Log level to determine which application messages are to be printed to standard output. Available log levels are: 0 (None), 1 (High priority), 2 (Low priority). Only application messages whose priority is higher than or equal to the application log level will be printed.

3.33.4 Application stages

Configuration

During this stage, the application configuration file is parsed and its content is loaded into the application data structures. In case of any configuration file parse error, an error message is displayed and the application is terminated. Please refer to the *Configuration file syntax* for a description of the application configuration file format.

Configuration checking

In the absence of any parse errors, the loaded content of application data structures is checked for overall consistency. In case of any configuration check error, an error message is displayed and the application is terminated.

Initialization

During this stage, the application resources are initialized and the handles to access them are saved into the application data structures. In case of any initialization error, an error message is displayed and the application is terminated.

The typical resources to be initialized are: pipeline instances, buffer pools, links (i.e. network interfaces), hardware device RX/TX queues, software queues, traffic management devices, etc.

Run-time

Each CPU core runs the pipeline instances assigned to it in time sharing mode and in round robin order:

1. *Packet processing task*: The pipeline run-time code is typically a packet *processing* task built on top of DPDK Packet Framework rte_pipeline library, which reads bursts of packets from the pipeline input ports, performs table lookups and executes the identified actions for all tables in the pipeline, with packet eventually written to pipeline output ports or dropped.
2. *Message handling task*: Each CPU core will also periodically execute the *message handling* code of each of the pipelines mapped to it. The pipeline message handling code is processing the messages that are pending in the pipeline input message queues, which are typically sent by the master CPU core for the on-the-fly pipeline configuration: check that pipeline is still alive (ping), add/delete entries in the pipeline tables, get statistics, etc. The frequency of executing the message handling code is usually much smaller than the frequency of executing the packet processing work.

Please refer to the *PIPELINE section* for more details about the application pipeline module encapsulation.

3.33.5 Configuration file syntax

Syntax overview

The syntax of the configuration file is designed to be simple, which favors readability. The configuration file is parsed using the DPDK library librte_cfgfile, which supports simple INI file format for configuration files.

As result, the configuration file is split into several sections, with each section containing one or more entries. The scope of each entry is its section, and each entry specifies a variable that is assigned a specific value. Any text after the `;` character is considered a comment and is therefore ignored.

The following are application specific: number of sections, name of each section, number of entries of each section, name of the variables used for each section entry, the value format (e.g. signed/unsigned integer, string, etc) and range of each section entry variable.

Generic example of configuration file section:

```
[<section_name>]

<variable_name_1> = <value_1>

; ...

<variable_name_N> = <value_N>
```

Application resources present in the configuration file

Table 3.3: Application resource names in the configuration file

Resource type	Format	Examples
Pipeline	PIPELINE<ID>	PIPELINE0, PIPELINE1
Mempool	MEMPOOL<ID>	MEMPOOL0, MEMPOOL1
Link (network interface)	LINK<ID>	LINK0, LINK1
Link RX queue	RXQ<LINK_ID>.<QUEUE_ID>	RXQ0.0, RXQ1.5
Link TX queue	TXQ<LINK_ID>.<QUEUE_ID>	TXQ0.0, TXQ1.5
Software queue	SWQ<ID>	SWQ0, SWQ1
Traffic Manager	TM<LINK_ID>	TM0, TM1
KNI (kernel NIC interface)	KNI<LINK_ID>	KNI0, KNI1
Source	SOURCE<ID>	SOURCE0, SOURCE1
Sink	SINK<ID>	SINK0, SINK1
Message queue	MSGQ<ID> MSGQ-REQ-PIPELINE<ID> MSGQ-RSP-PIPELINE<ID> MSGQ-REQ-CORE-<CORE_ID> MSGQ-RSP-CORE-<CORE_ID>	MSGQ0, MSGQ1, MSGQ-REQ-PIPELINE2, MSGQ-RSP-PIPELINE2, MSGQ-REQ-CORE-s0c1, MSGQ-RSP-CORE-s0c1

LINK instances are created implicitly based on the PORT_MASK application startup argument. LINK0 is the first port enabled in the PORT_MASK, port 1 is the next one, etc. The LINK ID is different than the DPDK PMD-level NIC port ID, which is the actual position in the bitmask mentioned above. For example, if bit 5 is the first bit set in the bitmask, then LINK0 is having the PMD ID of 5. This mechanism creates a contiguous LINK ID space and isolates the configuration file against changes in the board PCIe slots where NICs are plugged in.

RXQ, TXQ, TM and KNI instances have the LINK ID as part of their name. For example, RXQ2.1, TXQ2.1 and TM2 are all associated with LINK2.

Rules to parse the configuration file

The main rules used to parse the configuration file are:

1. Application resource name determines the type of resource based on the name prefix.

   Example: all software queues need to start with SWQ prefix, so SWQ0 and SWQ5 are valid software queue names.

2. An application resource is defined by creating a configuration file section with its name. The configuration file section allows fine tuning on any of the resource parameters. Some resource parameters are mandatory, in which case it is required to have them specified as part of the section, while some others are optional, in which case they get assigned their default value when not present.

   Example: section SWQ0 defines a software queue named SWQ0, whose parameters are detailed as part of this section.

3. An application resource can also be defined by referencing it. Referencing a resource takes place by simply using its name as part of the value assigned to a variable in any configuration file section. In this case, the resource is registered with all its parameters having their default values. Optionally, a section with the resource name can be added to the configuration file to fine tune some or all of the resource parameters.

   Example: in section `PIPELINE3`, variable `pktq_in` includes `SWQ5` as part of its list, which results in defining a software queue named `SWQ5`; when there is no `SWQ5` section present in the configuration file, `SWQ5` gets registered with default parameters.

PIPELINE section

Table 3.4: Configuration file PIPELINE section (1/2)

Section	Description	Optional	Range	Default value
type	Pipeline type. Defines the functionality to be executed.	NO	See "List of pipeline types"	N/A
core	CPU core to run the current pipeline.	YES	See "CPU Core notation"	CPU socket 0, core 0, hyper-thread 0
pktq_in	Packet queues to serve as input ports for the current pipeline instance. The acceptable packet queue types are: `RXQ`, `SWQ`, `TM` and `SOURCE`. First device in this list is used as pipeline input port 0, second as pipeline input port 1, etc.	YES	List of input packet queue IDs	Empty list
pktq_out	Packet queues to serve as output ports for the current pipeline instance. The acceptable packet queue types are: `TXQ`, `SWQ`, `TM` and `SINK`. First device in this list is used as pipeline output port 0, second as pipeline output port 1, etc.	YES	List of output packet queue IDs.	Empty list

Table 3.5: Configuration file PIPELINE section (2/2)

Section	Description	Optional	Range	Default value
msgq_in	Input message queues. These queues contain request messages that need to be handled by the current pipeline instance. The type and format of request messages is defined by the pipeline type. For each pipeline instance, there is an input message queue defined implicitly, whose name is: `MSGQ-REQ-<PIPELINE_ID>`. This message queue should not be mentioned as part of msgq_in list.	YES	List of message queue IDs	Empty list
msgq_out	Output message queues. These queues are used by the current pipeline instance to write response messages as result of request messages being handled. The type and format of response messages is defined by the pipeline type. For each pipeline instance, there is an output message queue defined implicitly, whose name is: `MSGQ-RSP-<PIPELINE_ID>`. This message queue should not be mentioned as part of msgq_out list.	YES	List of message queue IDs	Empty list
timer_period	Time period, measured in milliseconds, for handling the input message queues.	YES	milliseconds	1 ms
<any other>	Arguments to be passed to the current pipeline instance. Format of the arguments, their type, whether each argument is optional or mandatory and its default value (when optional) are defined by the pipeline type. The value of the arguments is applicable to the current pipeline instance only.	Depends on pipeline type	Depends on pipeline type	Depends on pipeline type

CPU core notation

The CPU Core notation is:

```
<CPU core> ::= [s|S<CPU socket ID>][c|C]<CPU core ID>[h|H]
```

For example:

```
CPU socket 0, core 0, hyper-thread 0: 0, c0, s0c0

CPU socket 0, core 0, hyper-thread 1: 0h, c0h, s0c0h

CPU socket 3, core 9, hyper-thread 1: s3c9h
```

MEMPOOL section

Table 3.6: Configuration file MEMPOOL section

Section	Description	Optional	Type	Default value
buffer_size	Buffer size (in bytes) for the current buffer pool.	YES	uint32_t	2048 + sizeof(struct rte_mbuf) + HEADROOM
pool_size	Number of buffers in the current buffer pool.	YES	uint32_t	32K
cache_size	Per CPU thread cache size (in number of buffers) for the current buffer pool.	YES	uint32_t	256
cpu	CPU socket ID where to allocate memory for the current buffer pool.	YES	uint32_t	0

LINK section

Table 3.7: Configuration file LINK section

Section entry	Description	Optional	Type	Default value
arp_q	NIC RX queue where ARP packets should be filtered.	YES	0 .. 127	0 (default queue)
tcp_syn_local_q	NIC RX queue where TCP packets with SYN flag should be filtered.	YES	0 .. 127	0 (default queue)
ip_local_q	NIC RX queue where IP packets with local destination should be filtered. When TCP, UDP and SCTP local queues are defined, they take higher priority than this queue.	YES	0 .. 127	0 (default queue)
tcp_local_q	NIC RX queue where TCP packets with local destination should be filtered.	YES	0 .. 127	0 (default queue)
udp_local_q	NIC RX queue where TCP packets with local destination should be filtered.	YES	0 .. 127	0 (default queue)
sctp_local_q	NIC RX queue where TCP packets with local destination should be filtered.	YES	0 .. 127	0 (default queue)
promisc	Indicates whether current link should be started in promiscuous mode.	YES	YES/NO	YES

RXQ section

Table 3.8: Configuration file RXQ section

Section	Description	Optional	Type	Default value
mempool	Mempool to use for buffer allocation for current NIC RX queue. The mempool ID has to be associated with a valid instance defined in the mempool entry of the global section.	YES	uint32_t	MEMPOOL0
Size	NIC RX queue size (number of descriptors)	YES	uint32_t	128
burst	Read burst size (number of descriptors)	YES	uint32_t	32

TXQ section

Table 3.9: Configuration file TXQ section

Section	Description	Optional	Type	Default value
size	NIC TX queue size (number of descriptors)	YES	uint32_t power of 2 > 0	512
burst	Write burst size (number of descriptors)	YES	uint32_t power of 2 0 < burst < size	32
dropless	When dropless is set to NO, packets can be dropped if not enough free slots are currently available in the queue, so the write operation to the queue is non- blocking. When dropless is set to YES, packets cannot be dropped if not enough free slots are currently available in the queue, so the write operation to the queue is blocking, as the write operation is retried until enough free slots become available and all the packets are successfully written to the queue.	YES	YES/NO	NO
n_retries	Number of retries. Valid only when dropless is set to YES. When set to 0, it indicates unlimited number of retries.	YES	uint32_t	0

SWQ section

Table 3.10: Configuration file SWQ section

Section	Description	Optional	Type	Default value
size	Queue size (number of packets)	YES	uint32_t power of 2	256
burst_read	Read burst size (number of packets)	YES	uint32_t power of 2 0 < burst < size	32
burst_write	Write burst size (number of packets)	YES	uint32_t power of 2 0 < burst < size	32
dropless	When dropless is set to NO, packets can be dropped if not enough free slots are currently available in the queue, so the write operation to the queue is non- blocking. When dropless is set to YES, packets cannot be dropped if not enough free slots are currently available in the queue, so the write operation to the queue is blocking, as the write operation is retried until enough free slots become available and all the packets are successfully written to the queue.	YES	YES/NO	NO
n_retries	Number of retries. Valid only when dropless is set to YES. When set to 0, it indicates unlimited number of retries.	YES	uint32_t	0
cpu	CPU socket ID where to allocate memory for this SWQ.	YES	uint32_t	0

TM section

Table 3.11: Configuration file TM section

Section	Description	Optional	Type	Default value
Cfg	File name to parse for the TM configuration to be applied. The syntax of this file is described in the examples/qos_sched DPDK application documentation.	YES	string	tm_profile
burst_read	Read burst size (number of packets)	YES	uint32_t	64
burst_write	Write burst size (number of packets)	YES	uint32_t	32

KNI section

Table 3.12: Configuration file KNI section

Section	Description	Optional	Type	Default value
core	CPU core to run the KNI kernel thread. When core config is set, the KNI kernel thread will be bound to the particular core. When core config is not set, the KNI kernel thread will be scheduled by the OS.	YES	See "CPU Core notation"	Not set
mempool	Mempool to use for buffer allocation for current KNI port. The mempool ID has to be associated with a valid instance defined in the mempool entry of the global section.	YES	uint32_t	MEMPOOL0
burst_read	Read burst size (number of packets)	YES	uint32_t power of 2 0 < burst < size	32
burst_write	Write burst size (number of packets)	YES	uint32_t power of 2 0 < burst < size	32
dropless	When dropless is set to NO, packets can be dropped if not enough free slots are currently available in the queue, so the write operation to the queue is non- blocking. When dropless is set to YES, packets cannot be dropped if not enough free slots are currently available in the queue, so the write operation to the queue is blocking, as the write operation is retried until enough free slots become available and all the packets are successfully written to the queue.	YES	YES/NO	NO
n_retries	Number of retries. Valid only when dropless is set to YES. When set to 0, it indicates unlimited number of retries.	YES	uint64_t	0

SOURCE section

Table 3.13: Configuration file SOURCE section

Section	Description	Optional	Type	Default value
Mempool	Mempool to use for buffer allocation.	YES	uint32_t	MEMPOOL0
Burst	Read burst size (number of packets)		uint32_t	32

SINK section

Currently, there are no parameters to be passed to a sink device, so SINK section is not allowed.

MSGQ section

Table 3.14: Configuration file MSGQ section

Section	Description	Optional	Type	Default value
size	Queue size (number of packets)	YES	uint32_t != 0 power of 2	64
cpu	CPU socket ID where to allocate memory for the current queue.	YES	uint32_t	0

EAL section

The application generates the EAL parameters rather than reading them from the command line.

The CPU core mask parameter is generated based on the core entry of all PIPELINE sections. All the other EAL parameters can be set from this section of the application configuration file.

3.33.6 Library of pipeline types

Pipeline module

A pipeline is a self-contained module that implements a packet processing function and is typically implemented on top of the DPDK Packet Framework *librte_pipeline* library. The application provides a run-time mechanism to register different pipeline types.

Depending on the required configuration, each registered pipeline type (pipeline class) is instantiated one or several times, with each pipeline instance (pipeline object) assigned to one of the available CPU cores. Each CPU core can run one or more pipeline instances, which might be of same or different types. For more information of the CPU core threading model, please refer to the *Run-time* section.

Pipeline type

Each pipeline type is made up of a back-end and a front-end. The back-end represents the packet processing engine of the pipeline, typically implemented using the DPDK Packet Framework libraries, which reads packets from the input packet queues, handles them and eventually writes them to the output packet queues or drops them. The front-end represents the run-time configuration interface of the pipeline, which is exposed as CLI commands. The front-end communicates with the back-end through message queues.

Table 3.15: Pipeline back-end

Field name	Field type	Description
f_init	Function pointer	Function to initialize the back-end of the current pipeline instance. Typical work implemented by this function for the current pipeline instance: Memory allocation; Parse the pipeline type specific arguments; Initialize the pipeline input ports, output ports and tables, interconnect input ports to tables; Set the message handlers.
f_free	Function pointer	Function to free the resources allocated by the back-end of the current pipeline instance.
f_run	Function pointer	Set to NULL for pipelines implemented using the DPDK library librte_pipeline (typical case), and to non-NULL otherwise. This mechanism is made available to support quick integration of legacy code. This function is expected to provide the packet processing related code to be called as part of the CPU thread dispatch loop, so this function is not allowed to contain an infinite loop.
f_timer	Function pointer	Function to read the pipeline input message queues, handle the request messages, create response messages and write the response queues. The format of request and response messages is defined by each pipeline type, with the exception of some requests which are mandatory for all pipelines (e.g. ping, statistics).
f_track	Function pointer	See section Tracking pipeline output port to physical link

Table 3.16: Pipeline front-end

Field name	Field type	Description
f_init	Function pointer	Function to initialize the front-end of the current pipeline instance.
f_free	Function pointer	Function to free the resources allocated by the front-end of the current pipeline instance.
cmds	Array of CLI commands	Array of CLI commands to be registered to the application CLI for the current pipeline type. Even though the CLI is executed by a different pipeline (typically, this is the master pipeline), from modularity perspective is more efficient to keep the message client side (part of the front-end) together with the message server side (part of the back-end).

Tracking pipeline output port to physical link

Each pipeline instance is a standalone block that does not have visibility into the other pipeline instances or the application-level pipeline inter-connectivity. In some cases, it is useful for a pipeline instance to get application level information related to pipeline connectivity, such as to identify the output link (e.g. physical NIC port) where one of its output ports connected, either directly or indirectly by traversing other pipeline instances.

Tracking can be successful or unsuccessful. Typically, tracking for a specific pipeline instance is successful when each one of its input ports can be mapped to a single output port, meaning that all packets read from the current input port can only go out on a single output port. Depending on the pipeline type, some exceptions may be allowed: a small portion of the packets, considered exception packets, are sent out on an output port that is pre-configured for this purpose.

For pass-through pipeline type, the tracking is always successful. For pipeline types as flow classification, firewall or routing, the tracking is only successful when the number of output ports for the current pipeline instance is 1.

This feature is used by the IP routing pipeline for adding/removing implicit routes every time a link is brought up/down.

Table copies

Fast table copy: pipeline table used by pipeline for the packet processing task, updated through messages, table data structures are optimized for lookup operation.

Slow table copy: used by the configuration layer, typically updated through CLI commands, kept in sync with the fast copy (its update triggers the fast copy update). Required for executing advanced table queries without impacting the packet processing task, therefore the slow copy is typically organized using different criteria than the fast copy.

Examples:

- Flow classification: Search through current set of flows (e.g. list all flows with a specific source IP address);
- Firewall: List rules in descending order of priority;
- Routing table: List routes sorted by prefix depth and their type (local, remote, default);
- ARP: List entries sorted per output interface.

Packet meta-data

Packet meta-data field offsets provided as argument to pipeline instances are essentially defining the data structure for the packet meta-data used by the current application use-case. It is very useful to put it in the configuration file as a comment in order to facilitate the readability of the configuration file.

The reason to use field offsets for defining the data structure for the packet meta-data is due to the C language limitation of not being able to define data structures at run-time. Feature to consider: have the configuration file parser automatically generate and print the data structure defining the packet meta-data for the current application use-case.

Packet meta-data typically contains:

1. Pure meta-data: intermediate data per packet that is computed internally, passed between different tables of the same pipeline instance (e.g. lookup key for the ARP table is obtained from the routing table), or between different pipeline instances (e.g. flow ID, traffic metering color, etc);
2. Packet fields: typically, packet header fields that are read directly from the packet, or read from the packet and saved (duplicated) as a working copy at a different location within the packet meta-data (e.g. Diffserv 5-tuple, IP destination address, etc).

Several strategies are used to design the packet meta-data, as described in the next subsections.

Store packet meta-data in a different cache line as the packet headers

This approach is able to support protocols with variable header length, like MPLS, where the offset of IP header from the start of the packet (and, implicitly, the offset of the IP header in the packet buffer) is not fixed. Since the pipelines typically require the specification of a fixed offset to the packet fields (e.g. Diffserv 5-tuple, used by the flow classification pipeline, or the IP destination address, used by the IP routing pipeline), the workaround is to have the packet RX pipeline copy these fields at fixed offsets within the packet meta-data.

As this approach duplicates some of the packet fields, it requires accessing more cache lines per packet for filling in selected packet meta-data fields (on RX), as well as flushing selected packet meta-data fields into the packet (on TX).

Example:

```
; struct app_pkt_metadata {
;   uint32_t ip_da;
;      uint32_t hash;
;      uint32_t flow_id;
```

```
;      uint32_t color;
; } __attribute__((__packed__));
;

[PIPELINE1]
; Packet meta-data offsets
ip_da_offset = 0;   Used by: routing
hash_offset = 4;    Used by: RX, flow classification
flow_id_offset = 8; Used by: flow classification, flow actions
color_offset = 12;  Used by: flow actions, routing
```

Overlay the packet meta-data in the same cache line with the packet headers

This approach is minimizing the number of cache line accessed per packet by storing the packet metadata in the same cache line with the packet headers. To enable this strategy, either some headroom is reserved for meta-data at the beginning of the packet headers cache line (e.g. if 16 bytes are needed for meta-data, then the packet headroom can be set to 128+16 bytes, so that NIC writes the first byte of the packet at offset 16 from the start of the first packet cache line), or meta-data is reusing the space of some packet headers that are discarded from the packet (e.g. input Ethernet header).

Example:

```
; struct app_pkt_metadata {
;     uint8_t headroom[RTE_PKTMBUF_HEADROOM]; /* 128 bytes (default) */
;     union {
;         struct {
;             struct ether_hdr ether; /* 14 bytes */
;             struct qinq_hdr qinq; /* 8 bytes */
;         };
;         struct {
;             uint32_t hash;
;             uint32_t flow_id;
;             uint32_t color;
;         };
;     };
;     struct ipv4_hdr ip; /* 20 bytes */
; } __attribute__((__packed__));
;
[PIPELINE2]
; Packet meta-data offsets
qinq_offset = 142;    Used by: RX, flow classification
ip_da_offset = 166;   Used by: routing
hash_offset = 128;    Used by: RX, flow classification
flow_id_offset = 132; Used by: flow classification, flow actions
color_offset = 136;   Used by: flow actions, routing
```

List of pipeline types

Table 3.17: List of pipeline types provided with the application

Name	Table(s)	Actions	Messages
Pass-through Note: depending on port type, can be used for RX, TX, IP fragmentation, IP reassembly or Traffic Management	Passthrough	1. Pkt metadata build 2. Flow hash 3. Pkt checks 4. Load balancing	1. Ping 2. Stats
Flow classification	Exact match • Key = byte array (source: pkt metadata) • Data = action dependent	1. Flow ID 2. Flow stats 3. Metering 4. Network Address 5. Translation (NAT)	1. Ping 2. Stats 3. Flow stats 4. Action stats 5. Flow add/ update/ delete 6. Default flow add/ update/ delete 7. Action update
Flow actions	Array • Key = Flow ID (source: pkt metadata) • Data = action dependent	1. Flow stats 2. Metering 3. Network Address 4. Translation (NAT)	1. Ping 2. Stats 3. Action stats 4. Action update
Firewall	ACL • Key = n-tuple (source: pkt headers) • Data = none	1. Allow/Drop	1. Ping 2. Stats 3. Rule add/ update/ delete 4. Default rule add/ update/ delete
IP routing	LPM (IPv4 or IPv6, depending on pipeline type) • Key = IP destination (source: pkt metadata) • Data = Dependent on actions and next hop type Hash table (for ARP, only when ARP is enabled) • Key = (Port ID, next hop IP address) (source: pkt metadata) • Data: MAC address	1. TTL decrement and 2. IPv4 checksum 3. update 4. Header 5. encapsulation 6. (based on next hop 7. type)	1. Ping 2. Stats 3. Route add/ update/ delete 4. Default route add/ update/ delete 5. ARP entry add/ update/ delete 6. Default ARP entry add/ update/ delete

3.33.7 Command Line Interface (CLI)

Global CLI commands

Table 3.18: Global CLI commands

Command	Description	Syntax
run	Run CLI commands script file.	run <file> <file> = path to file with CLI commands to execute
quit	Gracefully terminate the application.	quit

CLI commands for link configuration

Table 3.19: List of run-time configuration commands for link configuration

Command	Description	Syntax
link config	Link configuration	link <link ID> config <IP address> <depth>
link up	Link up	link <link ID> up
link down	Link down	link <link ID> down
link ls	Link list	link ls

CLI commands common for all pipeline types

Table 3.20: CLI commands mandatory for all pipelines

Command	Description	Syntax
ping	Check whether specific pipeline instance is alive. The master pipeline sends a ping request message to given pipeline instance and waits for a response message back. Timeout message is displayed when the response message is not received before the timer expires.	p <pipeline ID> ping
stats	Display statistics for specific pipeline input port, output port or table.	p <pipeline ID> stats port in <port in ID> p <pipeline ID> stats port out <port out ID> p <pipeline ID> stats table <table ID>
input port enable	Enable given input port for specific pipeline instance.	p <pipeline ID> port in <port ID> enable
input port disable	Disable given input port for specific pipeline instance.	p <pipeline ID> port in <port ID> disable

Pipeline type specific CLI commands

The pipeline specific CLI commands are part of the pipeline type front-end.

3.34 Test Pipeline Application

The Test Pipeline application illustrates the use of the DPDK Packet Framework tool suite. Its purpose is to demonstrate the performance of single-table DPDK pipelines.

3.34.1 Overview

The application uses three CPU cores:

- Core A ("RX core") receives traffic from the NIC ports and feeds core B with traffic through SW queues.
- Core B ("Pipeline core") implements a single-table DPDK pipeline whose type is selectable through specific command line parameter. Core B receives traffic from core A through software queues, processes it according to the actions configured in the table entries that are hit by the input packets and feeds it to core C through another set of software queues.
- Core C ("TX core") receives traffic from core B through software queues and sends it to the NIC ports for transmission.

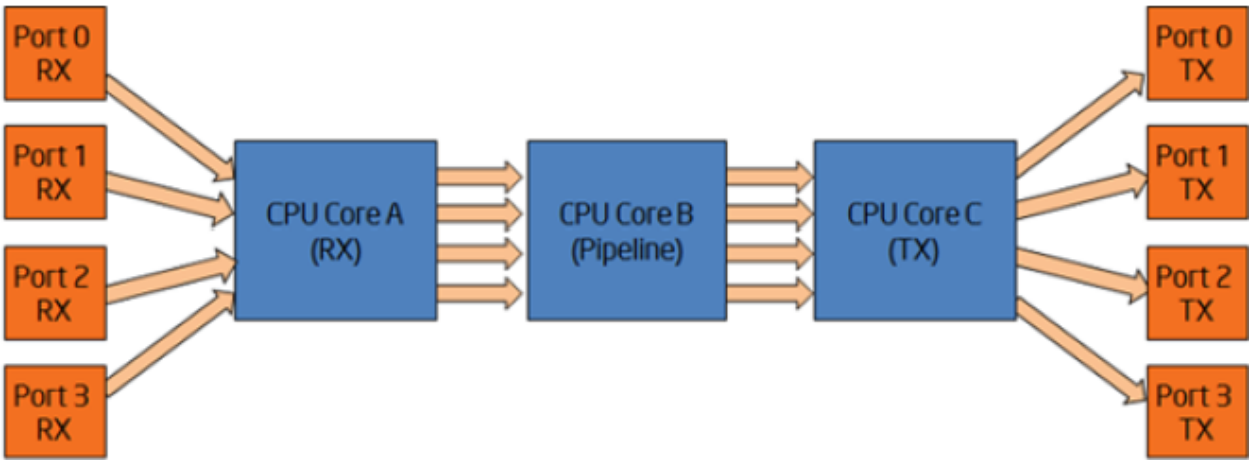


Fig. 3.25: Test Pipeline Application

3.34.2 Compiling the Application

1. Go to the app/test directory:

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}/app/test/test-pipeline
```

2. Set the target (a default target is used if not specified):

```
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

3. Build the application:

```
make
```

3.34.3 Running the Application

Application Command Line

The application execution command line is:

```
./test-pipeline [EAL options] -- -p PORTMASK --TABLE_TYPE
```

The -c or -l EAL CPU coremask/corelist option has to contain exactly 3 CPU cores. The first CPU core in the core mask is assigned for core A, the second for core B and the third for core C.

The PORTMASK parameter must contain 2 or 4 ports.

Table Types and Behavior

Table 3.21 describes the table types used and how they are populated.

The hash tables are pre-populated with 16 million keys. For hash tables, the following parameters can be selected:

- **Configurable key size implementation or fixed (specialized) key size implementation (e.g. hash-8-ext or hash-spec-8-ext).** The key size specialized implementations are expected to provide better performance for 8-byte and 16-byte key sizes, while the key-size-non-specialized implementation is expected to provide better performance for larger key sizes;
- **Key size (e.g. hash-spec-8-ext or hash-spec-16-ext).** The available options are 8, 16 and 32 bytes;
- **Table type (e.g. hash-spec-16-ext or hash-spec-16-lru).** The available options are ext (extendable bucket) or lru (least recently used).

Table 3.21: Table Types

#	TABLE_TYPE	Description of Core B Table	Pre-added Table Entries
1	none	Core B is not implementing a DPDK pipeline. Core B is implementing a pass-through from its input set of software queues to its output set of software queues.	N/A
2	stub	Stub table. Core B is implementing the same pass-through functionality as described for the "none" option by using the DPDK Packet Framework by using one stub table for each input NIC port.	N/A
3	hash-[spec]-8-lru	LRU hash table with 8-byte key size and 16 million entries.	16 million entries are successfully added to the hash table with the following key format: [4-byte index, 4 bytes of 0] The action configured for all table entries is "Sendto output port", with the output port index uniformly distributed for the range of output ports. The default table rule (used in the case of a lookup miss) is to drop the packet. At run time, core A is creating the following lookup key and storing it into the packet meta data for core B to use for table lookup: [destination IPv4 address, 4 bytes of 0]
4	hash-[spec]-8-ext	Extendable bucket hash table with 8-byte key size and 16 million entries.	Same as hash-[spec]-8-lru table entries, above.
5	hash-[spec]-16-lru	LRU hash table with 16-byte key size and 16 million entries.	16 million entries are successfully added to the hash table with the following key format: [4-byte index, 12 bytes of 0] The action configured for all table entries is "Send to output port", with the output port index uniformly distributed for the range of output ports. The default table rule (used in the case of a lookup miss) is to drop the

Input Traffic

Regardless of the table type used for the core B pipeline, the same input traffic can be used to hit all table entries with uniform distribution, which results in uniform distribution of packets sent out on the set of output NIC ports. The profile for input traffic is TCP/IPv4 packets with:

- destination IP address as A.B.C.D with A fixed to 0 and B, C,D random
- source IP address fixed to 0.0.0.0
- destination TCP port fixed to 0
- source TCP port fixed to 0

3.35 Distributor Sample Application

The distributor sample application is a simple example of packet distribution to cores using the Data Plane Development Kit (DPDK).

3.35.1 Overview

The distributor application performs the distribution of packets that are received on an RX_PORT to different cores. When processed by the cores, the destination port of a packet is the port from the enabled port mask adjacent to the one on which the packet was received, that is, if the first four ports are enabled (port mask 0xf), ports 0 and 1 RX/TX into each other, and ports 2 and 3 RX/TX into each other.

This application can be used to benchmark performance using the traffic generator as shown in the figure below.

Fig. 3.26: Performance Benchmarking Setup (Basic Environment)

3.35.2 Compiling the Application

1. Go to the sample application directory:

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}/examples/distributor
```

2. Set the target (a default target is used if not specified). For example:

```
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

   See the DPDK Getting Started Guide for possible RTE_TARGET values.

3. Build the application:

```
make
```

3.35.3 Running the Application

1. The application has a number of command line options:

```
./build/distributor_app [EAL options] -- -p PORTMASK
```

where,

- -p PORTMASK: Hexadecimal bitmask of ports to configure

2. To run the application in linuxapp environment with 10 lcores, 4 ports, issue the command:

```
$ ./build/distributor_app -l 1-9,22 -n 4 -- -p f
```

3. Refer to the DPDK Getting Started Guide for general information on running applications and the Environment Abstraction Layer (EAL) options.

3.35.4 Explanation

The distributor application consists of four types of threads: a receive thread (`lcore_rx()`), a distributor thread (`lcore_dist()`), a set of worker threads (`lcore_worker()`), and a transmit thread(`lcore_tx()`). How these threads work together is shown in Fig. 3.27 below. The `main()` function launches threads of these four types. Each thread has a while loop which will be doing processing and which is terminated only upon SIGINT or ctrl+C.

The receive thread receives the packets using `rte_eth_rx_burst()` and will enqueue them to an rte_ring. The distributor thread will dequeue the packets from the ring and assign them to workers (using `rte_distributor_process()` API). This assignment is based on the tag (or flow ID) of the packet - indicated by the hash field in the mbuf. For IP traffic, this field is automatically filled by the NIC with the "usr" hash value for the packet, which works as a per-flow tag. The distributor thread communicates with the worker threads using a cache-line swapping mechanism, passing up to 8 mbuf pointers at a time (one cache line) to each worker.

More than one worker thread can exist as part of the application, and these worker threads do simple packet processing by requesting packets from the distributor, doing a simple XOR operation on the input port mbuf field (to indicate the output port which will be used later for packet transmission) and then finally returning the packets back to the distributor thread.

The distributor thread will then call the distributor api `rte_distributor_returned_pkts()` to get the processed packets, and will enqueue them to another rte_ring for transfer to the TX thread for transmission on the output port. The transmit thread will dequeue the packets from the ring and transmit them on the output port specified in packet mbuf.

Users who wish to terminate the running of the application have to press ctrl+C (or send SIGINT to the app). Upon this signal, a signal handler provided in the application will terminate all running threads gracefully and print final statistics to the user.

Fig. 3.27: Distributor Sample Application Layout

3.35.5 Debug Logging Support

Debug logging is provided as part of the application; the user needs to uncomment the line "#define DEBUG" defined in start of the application in main.c to enable debug logs.

3.35.6 Statistics

The main function will print statistics on the console every second. These statistics include the number of packets enqueued and dequeued at each stage in the application, and also key statistics per worker, including how many packets of each burst size (1-8) were sent to each worker thread.

3.35.7 Application Initialization

Command line parsing is done in the same way as it is done in the L2 Forwarding Sample Application. See *Command Line Arguments*.

Mbuf pool initialization is done in the same way as it is done in the L2 Forwarding Sample Application. See *Mbuf Pool Initialization*.

Driver Initialization is done in same way as it is done in the L2 Forwarding Sample Application. See *Driver Initialization*.

RX queue initialization is done in the same way as it is done in the L2 Forwarding Sample Application. See *RX Queue Initialization*.

TX queue initialization is done in the same way as it is done in the L2 Forwarding Sample Application. See *TX Queue Initialization*.

3.36 VM Power Management Application

3.36.1 Introduction

Applications running in Virtual Environments have an abstract view of the underlying hardware on the Host, in particular applications cannot see the binding of virtual to physical hardware. When looking at CPU resourcing, the pinning of Virtual CPUs(vCPUs) to Host Physical CPUs(pCPUS) is not apparent to an application and this pinning may change over time. Furthermore, Operating Systems on virtual machines do not have the ability to govern their own power policy; the Machine Specific Registers (MSRs) for enabling P-State transitions are not exposed to Operating Systems running on Virtual Machines(VMs).

The Virtual Machine Power Management solution shows an example of how a DPDK application can indicate its processing requirements using VM local only information(vCPU/lcore) to a Host based Monitor which is responsible for accepting requests for frequency changes for a vCPU, translating the vCPU to a pCPU via libvirt and affecting the change in frequency.

The solution is comprised of two high-level components:

1. Example Host Application

   Using a Command Line Interface(CLI) for VM->Host communication channel management allows adding channels to the Monitor, setting and querying the vCPU to pCPU pinning, inspecting and manually changing the frequency for each CPU. The CLI runs on a single lcore while the thread responsible for managing VM requests runs on a second lcore.

   VM requests arriving on a channel for frequency changes are passed to the librte_power ACPI cpufreq sysfs based library. The Host Application relies on both qemu-kvm and libvirt to function.

2. librte_power for Virtual Machines

   Using an alternate implementation for the librte_power API, requests for frequency changes are forwarded to the host monitor rather than the APCI cpufreq sysfs interface used on the host.

   The l3fwd-power application will use this implementation when deployed on a VM (see *L3 Forwarding with Power Management Sample Application*).

Fig. 3.28: Highlevel Solution

3.36.2 Overview

VM Power Management employs qemu-kvm to provide communications channels between the host and VMs in the form of Virtio-Serial which appears as a paravirtualized serial device on a VM and can be configured to use various backends on the host. For this example each Virtio-Serial endpoint on the host is configured as AF_UNIX file socket, supporting poll/select and epoll for event notification. In this example each channel endpoint on the host is monitored via epoll for EPOLLIN events. Each channel is specified as qemu-kvm arguments or as libvirt XML for each VM, where each VM can have a number of channels up to a maximum of 64 per VM, in this example each DPDK lcore on a VM has exclusive access to a channel.

To enable frequency changes from within a VM, a request via the librte_power interface is forwarded via Virtio-Serial to the host, each request contains the vCPU and power command(scale up/down/min/max). The API for host and guest librte_power is consistent across environments, with the selection of VM or Host Implementation determined at automatically at runtime based on the environment.

Upon receiving a request, the host translates the vCPU to a pCPU via the libvirt API before forwarding to the host librte_power.

Fig. 3.29: VM request to scale frequency

Performance Considerations

While Haswell Microarchitecture allows for independent power control for each core, earlier Microarchtectures do not offer such fine grained control. When deployed on pre-Haswell platforms greater care must be taken in selecting which cores are assigned to a VM, for instance a core will not scale down until its sibling is similarly scaled.

3.36.3 Configuration

BIOS

Enhanced Intel SpeedStep® Technology must be enabled in the platform BIOS if the power management feature of DPDK is to be used. Otherwise, the sys file folder /sys/devices/system/cpu/cpu0/cpufreq will not exist, and the CPU frequency-based power management cannot be used. Consult the relevant BIOS documentation to determine how these settings can be accessed.

Host Operating System

The Host OS must also have the *apci_cpufreq* module installed, in some cases the *intel_pstate* driver may be the default Power Management environment. To enable *acpi_cpufreq* and disable *intel_pstate*, add the following to the grub Linux command line:

```
intel_pstate=disable
```

Upon rebooting, load the *acpi_cpufreq* module:

```
modprobe acpi_cpufreq
```

Hypervisor Channel Configuration

Virtio-Serial channels are configured via libvirt XML:

```
<name>{vm_name}</name>
<controller type='virtio-serial' index='0'>
  <address type='pci' domain='0x0000' bus='0x00' slot='0x06' function='0x0'/>
</controller>
<channel type='unix'>
  <source mode='bind' path='/tmp/powermonitor/{vm_name}.{channel_num}'/>
  <target type='virtio' name='virtio.serial.port.poweragent.{vm_channel_num}'/>
  <address type='virtio-serial' controller='0' bus='0' port='{N}'/>
</channel>
```

Where a single controller of type *virtio-serial* is created and up to 32 channels can be associated with a single controller and multiple controllers can be specified. The convention is to use the name of the VM in the host path *{vm_name}* and to increment *{channel_num}* for each channel, likewise the port value *{N}* must be incremented for each channel.

Each channel on the host will appear in *path*, the directory */tmp/powermonitor/* must first be created and given qemu permissions

```
mkdir /tmp/powermonitor/
chown qemu:qemu /tmp/powermonitor
```

Note that files and directories within /tmp are generally removed upon rebooting the host and the above steps may need to be carried out after each reboot.

The serial device as it appears on a VM is configured with the *target* element attribute *name* and must be in the form of *virtio.serial.port.poweragent.{vm_channel_num}*, where *vm_channel_num* is typically the lcore channel to be used in DPDK VM applications.

Each channel on a VM will be present at */dev/virtio-ports/virtio.serial.port.poweragent.{vm_channel_num}*

3.36.4 Compiling and Running the Host Application

Compiling

1. export RTE_SDK=/path/to/rte_sdk
2. cd ${RTE_SDK}/examples/vm_power_manager
3. make

Running

The application does not have any specific command line options other than *EAL*:

```
./build/vm_power_mgr [EAL options]
```

The application requires exactly two cores to run, one core is dedicated to the CLI, while the other is dedicated to the channel endpoint monitor, for example to run on cores 0 & 1 on a system with 4 memory channels:

```
./build/vm_power_mgr -l 0-1 -n 4
```

After successful initialization the user is presented with VM Power Manager CLI:

```
vm_power>
```

Virtual Machines can now be added to the VM Power Manager:

```
vm_power> add_vm {vm_name}
```

When a {vm_name} is specified with the *add_vm* command a lookup is performed with libvirt to ensure that the VM exists, {vm_name} is used as an unique identifier to associate channels with a particular VM and for executing operations on a VM within the CLI. VMs do not have to be running in order to add them.

A number of commands can be issued via the CLI in relation to VMs:

Remove a Virtual Machine identified by {vm_name} from the VM Power Manager.

```
rm_vm {vm_name}
```

Add communication channels for the specified VM, the virtio channels must be enabled in the VM configuration(qemu/libvirt) and the associated VM must be active. {list} is a comma-separated list of channel numbers to add, using the keyword 'all' will attempt to add all channels for the VM:

```
add_channels {vm_name} {list}|all
```

Enable or disable the communication channels in {list}(comma-separated) for the specified VM, alternatively list can be replaced with keyword 'all'. Disabled channels will still receive packets on the host, however the commands they specify will be ignored. Set status to 'enabled' to begin processing requests again:

```
set_channel_status {vm_name} {list}|all enabled|disabled
```

Print to the CLI the information on the specified VM, the information lists the number of vCPUS, the pinning to pCPU(s) as a bit mask, along with any communication channels associated with each VM, along with the status of each channel:

```
show_vm {vm_name}
```

Set the binding of Virtual CPU on VM with name {vm_name} to the Physical CPU mask:

```
set_pcpu_mask {vm_name} {vcpu} {pcpu}
```

Set the binding of Virtual CPU on VM to the Physical CPU:

```
set_pcpu {vm_name} {vcpu} {pcpu}
```

Manual control and inspection can also be carried in relation CPU frequency scaling:

Get the current frequency for each core specified in the mask:

```
show_cpu_freq_mask {mask}
```

Set the current frequency for the cores specified in {core_mask} by scaling each up/down/min/max:

```
set_cpu_freq {core_mask} up|down|min|max
```

Get the current frequency for the specified core:

```
show_cpu_freq {core_num}
```

Set the current frequency for the specified core by scaling up/down/min/max:

```
set_cpu_freq {core_num} up|down|min|max
```

3.36.5 Compiling and Running the Guest Applications

For compiling and running l3fwd-power, see *L3 Forwarding with Power Management Sample Application*.

A guest CLI is also provided for validating the setup.

For both l3fwd-power and guest CLI, the channels for the VM must be monitored by the host application using the *add_channels* command on the host.

Compiling

1. export RTE_SDK=/path/to/rte_sdk
2. cd ${RTE_SDK}/examples/vm_power_manager/guest_cli
3. make

Running

The application does not have any specific command line options other than *EAL*:

```
./build/vm_power_mgr [EAL options]
```

The application for example purposes uses a channel for each lcore enabled, for example to run on cores 0,1,2,3 on a system with 4 memory channels:

```
./build/guest_vm_power_mgr -l 0-3 -n 4
```

After successful initialization the user is presented with VM Power Manager Guest CLI:

```
vm_power(guest)>
```

To change the frequency of a lcore, use the set_cpu_freq command. Where {core_num} is the lcore and channel to change frequency by scaling up/down/min/max.

```
set_cpu_freq {core_num} up|down|min|max
```

3.37 TEP termination Sample Application

The TEP (Tunnel End point) termination sample application simulates a VXLAN Tunnel Endpoint (VTEP) termination in DPDK, which is used to demonstrate the offload and filtering capabilities of Intel® XL710 10/40 Gigabit Ethernet Controller for VXLAN packet. This sample uses the basic virtio devices management mechanism from vhost example, and also uses the us-vHost interface and tunnel filtering mechanism to direct a specified traffic to a specific VM. In addition, this sample is also designed to show how tunneling protocols can be handled.

3.37.1 Background

With virtualization, overlay networks allow a network structure to be built or imposed across physical nodes which is abstracted away from the actual underlining physical network connections. This allows network isolation, QOS, etc to be provided on a per client basis.

Fig. 3.30: Overlay Networking.

In a typical setup, the network overlay tunnel is terminated at the Virtual/Tunnel End Point (VEP/TEP). The TEP is normally located at the physical host level ideally in the software switch. Due to processing constraints and the inevitable bottleneck that the switch becomes, the ability to offload overlay support features becomes an important requirement. Intel® XL710 10/40 Gigabit Ethernet network card provides hardware filtering and offload capabilities to support overlay networks implementations such as MAC in UDP and MAC in GRE.

3.37.2 Sample Code Overview

The DPDK TEP termination sample code demonstrates the offload and filtering capabilities of Intel® XL710 10/40 Gigabit Ethernet Controller for VXLAN packet.

The sample code is based on vhost library. The vhost library is developed for user space Ethernet switch to easily integrate with vhost functionality.

The sample will support the followings:

- Tunneling packet recognition.
- The port of UDP tunneling is configurable
- Directing incoming traffic to the correct queue based on the tunnel filter type. The supported filter type are listed below.
  - Inner MAC and VLAN and tenant ID
  - Inner MAC and tenant ID, and Outer MAC
  - Inner MAC and tenant ID

  The tenant ID will be assigned from a static internal table based on the us-vhost device ID. Each device will receive a unique device ID. The inner MAC will be learned by the first packet transmitted from a device.
- Decapsulation of RX VXLAN traffic. This is a software only operation.
- Encapsulation of TX VXLAN traffic. This is a software only operation.
- Inner IP and inner L4 checksum offload.
- TSO offload support for tunneling packet.

The following figure shows the framework of the TEP termination sample application based on DPDK vhost lib.

Fig. 3.31: TEP termination Framework Overview

3.37.3 Supported Distributions

The example in this section have been validated with the following distributions:

- Fedora* 18
- Fedora* 19
- Fedora* 20

3.37.4 Compiling the Sample Code

1. Compile vhost lib:

   To enable vhost, turn on vhost library in the configure file config/common_linuxapp.

```
CONFIG_RTE_LIBRTE_VHOST=y
```

2. Go to the examples directory:

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}/examples/tep_termination
```

3. Set the target (a default target is used if not specified). For example:

```
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

   See the DPDK Getting Started Guide for possible RTE_TARGET values.

4. Build the application:

```
cd ${RTE_SDK}
make config ${RTE_TARGET}
make install ${RTE_TARGET}
cd ${RTE_SDK}/examples/tep_termination
make
```

3.37.5 Running the Sample Code

1. Go to the examples directory:

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}/examples/tep_termination
```

2. Run the tep_termination sample code:

```
user@target:~$ ./build/app/tep_termination -l 0-3 -n 4 --huge-dir /mnt/huge --
                -p 0x1 --dev-basename tep-termination --nb-devices 4
                --udp-port 4789 --filter-type 1
```

Note: Please note the huge-dir parameter instructs the DPDK to allocate its memory from the 2 MB page hugetlbfs.

Parameters

The same parameters with the vhost sample.

Refer to *Parameters* for detailed explanation.

Number of Devices.

The nb-devices option specifies the number of virtIO device. The default value is 2.

```
user@target:~$ ./build/app/tep_termination -l 0-3 -n 4 --huge-dir /mnt/huge --
               --nb-devices 2
```

Tunneling UDP port.

The udp-port option is used to specify the destination UDP number for UDP tunneling packet. The default value is 4789.

```
user@target:~$ ./build/app/tep_termination -l 0-3 -n 4 --huge-dir /mnt/huge --
               --nb-devices 2 --udp-port 4789
```

Filter Type.

The filter-type option is used to specify which filter type is used to filter UDP tunneling packet to a specified queue. The default value is 1, which means the filter type of inner MAC and tenant ID is used.

```
user@target:~$ ./build/app/tep_termination -l 0-3 -n 4 --huge-dir /mnt/huge --
            --nb-devices 2 --udp-port 4789 --filter-type 1
```

TX Checksum.

The tx-checksum option is used to enable or disable the inner header checksum offload. The default value is 0, which means the checksum offload is disabled.

```
user@target:~$ ./build/app/tep_termination -l 0-3 -n 4 --huge-dir /mnt/huge --
            --nb-devices 2 --tx-checksum
```

TCP segment size.

The tso-segsz option specifies the TCP segment size for TSO offload for tunneling packet. The default value is 0, which means TSO offload is disabled.

```
user@target:~$ ./build/app/tep_termination -l 0-3 -n 4 --huge-dir /mnt/huge --
            --tx-checksum --tso-segsz 800
```

Decapsulation option.

The decap option is used to enable or disable decapsulation operation for received VXLAN packet. The default value is 1.

```
user@target:~$ ./build/app/tep_termination -l 0-3 -n 4 --huge-dir /mnt/huge --
            --nb-devices 4 --udp-port 4789 --decap 1
```

Encapsulation option.

The encap option is used to enable or disable encapsulation operation for transmitted packet. The default value is 1.

```
user@target:~$ ./build/app/tep_termination -l 0-3 -n 4 --huge-dir /mnt/huge --
            --nb-devices 4 --udp-port 4789 --encap 1
```

3.37.6 Running the Virtual Machine (QEMU)

Refer to *Start the VM*.

3.37.7 Running DPDK in the Virtual Machine

Refer to *Run testpmd inside guest*.

3.37.8 Passing Traffic to the Virtual Machine Device

For a virtio-net device to receive traffic, the traffic's Layer 2 header must include both the virtio-net device's MAC address. The DPDK sample code behaves in a similar manner to a learning switch in that it learns the MAC address of the virtio-net devices from the first transmitted packet. On learning the MAC address, the DPDK vhost sample code prints a message with the MAC address and tenant ID virtio-net device. For example:

```
DATA: (0) MAC_ADDRESS cc:bb:bb:bb:bb:bb and VNI 1000 registered
```

The above message indicates that device 0 has been registered with MAC address cc:bb:bb:bb:bb:bb and VNI 1000. Any packets received on the NIC with these values are placed on the devices receive queue.

3.38 PTP Client Sample Application

The PTP (Precision Time Protocol) client sample application is a simple example of using the DPDK IEEE1588 API to communicate with a PTP master clock to synchronize the time on the NIC and, optionally, on the Linux system.

Note, PTP is a time syncing protocol and cannot be used within DPDK as a time-stamping mechanism. See the following for an explanation of the protocol: Precision Time Protocol.

3.38.1 Limitations

The PTP sample application is intended as a simple reference implementation of a PTP client using the DPDK IEEE1588 API. In order to keep the application simple the following assumptions are made:

- The first discovered master is the master for the session.
- Only L2 PTP packets are supported.
- Only the PTP v2 protocol is supported.
- Only the slave clock is implemented.

3.38.2 How the Application Works

Fig. 3.32: PTP Synchronization Protocol

The PTP synchronization in the sample application works as follows:

- Master sends *Sync* message - the slave saves it as T2.
- Master sends *Follow Up* message and sends time of T1.
- Slave sends *Delay Request* frame to PTP Master and stores T3.
- Master sends *Delay Response* T4 time which is time of received T3.

The adjustment for slave can be represented as:

adj = -[(T2-T1)-(T4 - T3)]/2

If the command line parameter `-T 1` is used the application also synchronizes the PTP PHC clock with the Linux kernel clock.

3.38.3 Compiling the Application

To compile the application, export the path to the DPDK source tree and edit the `config/common_linuxapp` configuration file to enable IEEE1588:

```
export RTE_SDK=/path/to/rte_sdk

# Edit  common_linuxapp and set the following options:
CONFIG_RTE_LIBRTE_IEEE1588=y
```

Set the target, for example:

```
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

See the *DPDK Getting Started* Guide for possible `RTE_TARGET` values.

Build the application as follows:

```
# Recompile DPDK.
make install T=$RTE_TARGET

# Compile the application.
cd ${RTE_SDK}/examples/ptpclient
make
```

3.38.4 Running the Application

To run the example in a `linuxapp` environment:

```
./build/ptpclient -l 1 -n 4 -- -p 0x1 -T 0
```

Refer to *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

- `-p portmask`: Hexadecimal portmask.
- `-T 0`: Update only the PTP slave clock.
- `-T 1`: Update the PTP slave clock and synchronize the Linux Kernel to the PTP clock.

3.38.5 Code Explanation

The following sections provide an explanation of the main components of the code.

All DPDK library functions used in the sample code are prefixed with `rte_` and are explained in detail in the *DPDK API Documentation*.

The Main Function

The `main()` function performs the initialization and calls the execution threads for each lcore.

The first task is to initialize the Environment Abstraction Layer (EAL). The `argc` and `argv` arguments are provided to the `rte_eal_init()` function. The value returned is the number of parsed arguments:

```
int ret = rte_eal_init(argc, argv);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Error with EAL initialization\n");
```

And than we parse application specific arguments

```
argc -= ret;
argv += ret;

ret = ptp_parse_args(argc, argv);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Error with PTP initialization\n");
```

The `main()` also allocates a mempool to hold the mbufs (Message Buffers) used by the application:

```
mbuf_pool = rte_pktmbuf_pool_create("MBUF_POOL", NUM_MBUFS * nb_ports,
       MBUF_CACHE_SIZE, 0, RTE_MBUF_DEFAULT_BUF_SIZE, rte_socket_id());
```

Mbufs are the packet buffer structure used by DPDK. They are explained in detail in the "Mbuf Library" section of the *DPDK Programmer's Guide*.

The `main()` function also initializes all the ports using the user defined `port_init()` function with portmask provided by user:

```
for (portid = 0; portid < nb_ports; portid++)
    if ((ptp_enabled_port_mask & (1 << portid)) != 0) {

        if (port_init(portid, mbuf_pool) == 0) {
            ptp_enabled_ports[ptp_enabled_port_nb] = portid;
            ptp_enabled_port_nb++;
        } else {
            rte_exit(EXIT_FAILURE, "Cannot init port %"PRIu8 "\n",
                    portid);
        }
    }
```

Once the initialization is complete, the application is ready to launch a function on an lcore. In this example `lcore_main()` is called on a single lcore.

```
lcore_main();
```

The `lcore_main()` function is explained below.

The Lcores Main

As we saw above the `main()` function calls an application function on the available lcores.

The main work of the application is done within the loop:

```
for (portid = 0; portid < ptp_enabled_port_nb; portid++) {

    portid = ptp_enabled_ports[portid];
    nb_rx = rte_eth_rx_burst(portid, 0, &m, 1);

    if (likely(nb_rx == 0))
```

```
        continue;

    if (m->ol_flags & PKT_RX_IEEE1588_PTP)
        parse_ptp_frames(portid, m);

    rte_pktmbuf_free(m);
}
```

Packets are received one by one on the RX ports and, if required, PTP response packets are transmitted on the TX ports.

If the offload flags in the mbuf indicate that the packet is a PTP packet then the packet is parsed to determine which type:

```
if (m->ol_flags & PKT_RX_IEEE1588_PTP)
     parse_ptp_frames(portid, m);
```

All packets are freed explicitly using `rte_pktmbuf_free()`.

The forwarding loop can be interrupted and the application closed using `Ctrl-C`.

PTP parsing

The `parse_ptp_frames()` function processes PTP packets, implementing slave PTP IEEE1588 L2 functionality.

```
void
parse_ptp_frames(uint8_t portid, struct rte_mbuf *m) {
    struct ptp_header *ptp_hdr;
    struct ether_hdr *eth_hdr;
    uint16_t eth_type;

    eth_hdr = rte_pktmbuf_mtod(m, struct ether_hdr *);
    eth_type = rte_be_to_cpu_16(eth_hdr->ether_type);

    if (eth_type == PTP_PROTOCOL) {
        ptp_data.m = m;
        ptp_data.portid = portid;
        ptp_hdr = (struct ptp_header *)(rte_pktmbuf_mtod(m, char *)
                    + sizeof(struct ether_hdr));

        switch (ptp_hdr->msgtype) {
        case SYNC:
            parse_sync(&ptp_data);
            break;
        case FOLLOW_UP:
            parse_fup(&ptp_data);
            break;
        case DELAY_RESP:
            parse_drsp(&ptp_data);
            print_clock_info(&ptp_data);
            break;
        default:
            break;
        }
    }
}
```

There are 3 types of packets on the RX path which we must parse to create a minimal implementation of the PTP slave client:

- SYNC packet.
- FOLLOW UP packet
- DELAY RESPONSE packet.

When we parse the *FOLLOW UP* packet we also create and send a *DELAY_REQUEST* packet. Also when we parse the *DELAY RESPONSE* packet, and all conditions are met we adjust the PTP slave clock.

3.39 Performance Thread Sample Application

The performance thread sample application is a derivative of the standard L3 forwarding application that demonstrates different threading models.

3.39.1 Overview

For a general description of the L3 forwarding applications capabilities please refer to the documentation of the standard application in *L3 Forwarding Sample Application*.

The performance thread sample application differs from the standard L3 forwarding example in that it divides the TX and RX processing between different threads, and makes it possible to assign individual threads to different cores.

Three threading models are considered:

1. When there is one EAL thread per physical core.
2. When there are multiple EAL threads per physical core.
3. When there are multiple lightweight threads per EAL thread.

Since DPDK release 2.0 it is possible to launch applications using the `--lcores` EAL parameter, specifying cpu-sets for a physical core. With the performance thread sample application its is now also possible to assign individual RX and TX functions to different cores.

As an alternative to dividing the L3 forwarding work between different EAL threads the performance thread sample introduces the possibility to run the application threads as lightweight threads (L-threads) within one or more EAL threads.

In order to facilitate this threading model the example includes a primitive cooperative scheduler (L-thread) subsystem. More details of the L-thread subsystem can be found in *The L-thread subsystem*.

Note: Whilst theoretically possible it is not anticipated that multiple L-thread schedulers would be run on the same physical core, this mode of operation should not be expected to yield useful performance and is considered invalid.

3.39.2 Compiling the Application

The application is located in the sample application folder in the `performance-thread` folder.

1. Go to the example applications folder

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}/examples/performance-thread/l3fwd-thread
```

2. Set the target (a default target is used if not specified). For example:

```
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

See the *DPDK Linux Getting Started Guide* for possible RTE_TARGET values.

3. Build the application:

   make

3.39.3 Running the Application

The application has a number of command line options:

```
./build/l3fwd-thread [EAL options] --
    -p PORTMASK [-P]
    --rx(port,queue,lcore,thread)[,(port,queue,lcore,thread)]
    --tx(lcore,thread)[,(lcore,thread)]
    [--enable-jumbo] [--max-pkt-len PKTLEN]]  [--no-numa]
    [--hash-entry-num] [--ipv6] [--no-lthreads] [--stat-lcore lcore]
    [--parse-ptype]
```

Where:

- `-p PORTMASK`: Hexadecimal bitmask of ports to configure.
- `-P`: optional, sets all ports to promiscuous mode so that packets are accepted regardless of the packet's Ethernet MAC destination address. Without this option, only packets with the Ethernet MAC destination address set to the Ethernet address of the port are accepted.
- `--rx (port,queue,lcore,thread)[,(port,queue,lcore,thread)]`: the list of NIC RX ports and queues handled by the RX lcores and threads. The parameters are explained below.
- `--tx (lcore,thread)[,(lcore,thread)]`: the list of TX threads identifying the lcore the thread runs on, and the id of RX thread with which it is associated. The parameters are explained below.
- `--enable-jumbo`: optional, enables jumbo frames.
- `--max-pkt-len`: optional, maximum packet length in decimal (64-9600).
- `--no-numa`: optional, disables numa awareness.
- `--hash-entry-num`: optional, specifies the hash entry number in hex to be setup.
- `--ipv6`: optional, set it if running ipv6 packets.
- `--no-lthreads`: optional, disables l-thread model and uses EAL threading model. See below.
- `--stat-lcore`: optional, run CPU load stats collector on the specified lcore.
- `--parse-ptype`: optional, set to use software to analyze packet type. Without this option, hardware will check the packet type.

The parameters of the `--rx` and `--tx` options are:

- `--rx` parameters

port	RX port
queue	RX queue that will be read on the specified RX port
lcore	Core to use for the thread
thread	Thread id (continuously from 0 to N)

- `--tx` parameters

lcore	Core to use for L3 route match and transmit
thread	Id of RX thread to be associated with this TX thread

The `l3fwd-thread` application allows you to start packet processing in two threading models: L-Threads (default) and EAL Threads (when the `--no-lthreads` parameter is used). For consistency all parameters are used in the same way for both models.

Running with L-threads

When the L-thread model is used (default option), lcore and thread parameters in `--rx/--tx` are used to affinitize threads to the selected scheduler.

For example, the following places every l-thread on different lcores:

```
l3fwd-thread -l 0-7 -n 2 -- -P -p 3 \
             --rx="(0,0,0,0)(1,0,1,1)" \
             --tx="(2,0)(3,1)"
```

The following places RX l-threads on lcore 0 and TX l-threads on lcore 1 and 2 and so on:

```
l3fwd-thread -l 0-7 -n 2 -- -P -p 3 \
             --rx="(0,0,0,0)(1,0,0,1)" \
             --tx="(1,0)(2,1)"
```

Running with EAL threads

When the `--no-lthreads` parameter is used, the L-threading model is turned off and EAL threads are used for all processing. EAL threads are enumerated in the same way as L-threads, but the `--lcores` EAL parameter is used to affinitize threads to the selected cpu-set (scheduler). Thus it is possible to place every RX and TX thread on different lcores.

For example, the following places every EAL thread on different lcores:

```
l3fwd-thread -l 0-7 -n 2 -- -P -p 3 \
             --rx="(0,0,0,0)(1,0,1,1)" \
             --tx="(2,0)(3,1)" \
             --no-lthreads
```

To affinitize two or more EAL threads to one cpu-set, the EAL `--lcores` parameter is used.

The following places RX EAL threads on lcore 0 and TX EAL threads on lcore 1 and 2 and so on:

```
l3fwd-thread -l 0-7 -n 2 --lcores="(0,1)@0,(2,3)@1" -- -P -p 3 \
             --rx="(0,0,0,0)(1,0,1,1)" \
             --tx="(2,0)(3,1)" \
             --no-lthreads
```

Examples

For selected scenarios the command line configuration of the application for L-threads and its corresponding EAL threads command line can be realized as follows:

1. Start every thread on different scheduler (1:1):

```
l3fwd-thread -l 0-7 -n 2 -- -P -p 3 \
             --rx="(0,0,0,0)(1,0,1,1)" \
             --tx="(2,0)(3,1)"
```

EAL thread equivalent:

```
l3fwd-thread -l 0-7 -n 2 -- -P -p 3 \
             --rx="(0,0,0,0)(1,0,1,1)" \
             --tx="(2,0)(3,1)" \
             --no-lthreads
```

2. Start all threads on one core (N:1).

   Start 4 L-threads on lcore 0:

```
l3fwd-thread -l 0-7 -n 2 -- -P -p 3 \
             --rx="(0,0,0,0)(1,0,0,1)" \
             --tx="(0,0)(0,1)"
```

   Start 4 EAL threads on cpu-set 0:

```
l3fwd-thread -l 0-7 -n 2 --lcores="(0-3)@0" -- -P -p 3 \
             --rx="(0,0,0,0)(1,0,0,1)" \
             --tx="(2,0)(3,1)" \
             --no-lthreads
```

3. Start threads on different cores (N:M).

   Start 2 L-threads for RX on lcore 0, and 2 L-threads for TX on lcore 1:

```
l3fwd-thread -l 0-7 -n 2 -- -P -p 3 \
             --rx="(0,0,0,0)(1,0,0,1)" \
             --tx="(1,0)(1,1)"
```

   Start 2 EAL threads for RX on cpu-set 0, and 2 EAL threads for TX on cpu-set 1:

```
l3fwd-thread -l 0-7 -n 2 --lcores="(0-1)@0,(2-3)@1" -- -P -p 3 \
             --rx="(0,0,0,0)(1,0,1,1)" \
             --tx="(2,0)(3,1)" \
             --no-lthreads
```

3.39.4 Explanation

To a great extent the sample application differs little from the standard L3 forwarding application, and readers are advised to familiarize themselves with the material covered in the *L3 Forwarding Sample Application* documentation before proceeding.

The following explanation is focused on the way threading is handled in the performance thread example.

Mode of operation with EAL threads

The performance thread sample application has split the RX and TX functionality into two different threads, and the RX and TX threads are interconnected via software rings. With respect to these rings the RX threads are producers and the TX threads are consumers.

On initialization the TX and RX threads are started according to the command line parameters.

The RX threads poll the network interface queues and post received packets to a TX thread via a corresponding software ring.

The TX threads poll software rings, perform the L3 forwarding hash/LPM match, and assemble packet bursts before performing burst transmit on the network interface.

As with the standard L3 forward application, burst draining of residual packets is performed periodically with the period calculated from elapsed time using the timestamps counter.

The diagram below illustrates a case with two RX threads and three TX threads.

Mode of operation with L-threads

Like the EAL thread configuration the application has split the RX and TX functionality into different threads, and the pairs of RX and TX threads are interconnected via software rings.

On initialization an L-thread scheduler is started on every EAL thread. On all but the master EAL thread only a a dummy L-thread is initially started. The L-thread started on the master EAL thread then spawns other L-threads on different L-thread schedulers according the the command line parameters.

The RX threads poll the network interface queues and post received packets to a TX thread via the corresponding software ring.

The ring interface is augmented by means of an L-thread condition variable that enables the TX thread to be suspended when the TX ring is empty. The RX thread signals the condition whenever it posts to the TX ring, causing the TX thread to be resumed.

Additionally the TX L-thread spawns a worker L-thread to take care of polling the software rings, whilst it handles burst draining of the transmit buffer.

The worker threads poll the software rings, perform L3 route lookup and assemble packet bursts. If the TX ring is empty the worker thread suspends itself by waiting on the condition variable associated with the ring.

Burst draining of residual packets, less than the burst size, is performed by the TX thread which sleeps (using an L-thread sleep function) and resumes periodically to flush the TX buffer.

This design means that L-threads that have no work, can yield the CPU to other L-threads and avoid having to constantly poll the software rings.

The diagram below illustrates a case with two RX threads and three TX functions (each comprising a thread that processes forwarding and a thread that periodically drains the output buffer of residual packets).

CPU load statistics

It is possible to display statistics showing estimated CPU load on each core. The statistics indicate the percentage of CPU time spent: processing received packets (forwarding), polling queues/rings (waiting for work), and doing any other processing (context switch and other overhead).

When enabled statistics are gathered by having the application threads set and clear flags when they enter and exit pertinent code sections. The flags are then sampled in real time by a statistics collector thread running on another core. This thread displays the data in real time on the console.

This feature is enabled by designating a statistics collector core, using the `--stat-lcore` parameter.

3.39.5 The L-thread subsystem

The L-thread subsystem resides in the examples/performance-thread/common directory and is built and linked automatically when building the `l3fwd-thread` example.

The subsystem provides a simple cooperative scheduler to enable arbitrary functions to run as cooperative threads within a single EAL thread. The subsystem provides a pthread like API that is intended to assist in reuse of legacy code written for POSIX pthreads.

The following sections provide some detail on the features, constraints, performance and porting considerations when using L-threads.

Comparison between L-threads and POSIX pthreads

The fundamental difference between the L-thread and pthread models is the way in which threads are scheduled. The simplest way to think about this is to consider the case of a processor with a single CPU. To run multiple threads on a single CPU, the scheduler must frequently switch between the threads, in order that each thread is able to make timely progress. This is the basis of any multitasking operating system.

This section explores the differences between the pthread model and the L-thread model as implemented in the provided L-thread subsystem. If needed a theoretical discussion of preemptive vs cooperative multi-threading can be found in any good text on operating system design.

Scheduling and context switching

The POSIX pthread library provides an application programming interface to create and synchronize threads. Scheduling policy is determined by the host OS, and may be configurable. The OS may use sophisticated rules to determine which thread should be run next, threads may suspend themselves or make other threads ready, and the scheduler may employ a time slice giving each thread a maximum time quantum after which it will be preempted in favor of another thread that is ready to run. To complicate matters further threads may be assigned different scheduling priorities.

By contrast the L-thread subsystem is considerably simpler. Logically the L-thread scheduler performs the same multiplexing function for L-threads within a single pthread as the OS scheduler does for pthreads within an application process. The L-thread scheduler is simply the main loop of a pthread, and in so far as the host OS is concerned it is a regular pthread just like any other. The host OS is oblivious about the existence of and not at all involved in the scheduling of L-threads.

The other and most significant difference between the two models is that L-threads are scheduled cooperatively. L-threads cannot not preempt each other, nor can the L-thread scheduler preempt a running L-thread (i.e. there is no time slicing). The consequence is that programs implemented with L-threads must possess frequent rescheduling points, meaning that they must explicitly and of their own volition return to the scheduler at frequent intervals, in order to allow other L-threads an opportunity to proceed.

In both models switching between threads requires that the current CPU context is saved and a new context (belonging to the next thread ready to run) is restored. With pthreads this context switching is handled transparently and the set of CPU registers that must be preserved between context switches is as per an interrupt handler.

An L-thread context switch is achieved by the thread itself making a function call to the L-thread scheduler. Thus it is only necessary to preserve the callee registers. The caller is responsible to save and restore any other registers it is using before a function call, and restore them on return, and this is handled by the compiler. For `X86_64` on both Linux and BSD the System V calling convention is used, this defines registers RSP, RBP, and R12-R15 as callee-save registers (for more detailed discussion a good reference is X86 Calling Conventions).

Taking advantage of this, and due to the absence of preemption, an L-thread context switch is achieved with less than 20 load/store instructions.

The scheduling policy for L-threads is fixed, there is no prioritization of L-threads, all L-threads are equal and scheduling is based on a FIFO ready queue.

An L-thread is a struct containing the CPU context of the thread (saved on context switch) and other useful items. The ready queue contains pointers to threads that are ready to run. The L-thread scheduler is a simple loop that polls the ready queue, reads from it the next thread ready to run, which it resumes by saving the current context (the current position in the scheduler loop) and restoring the context of the next thread from its thread struct. Thus an L-thread is always resumed at the last place it yielded.

A well behaved L-thread will call the context switch regularly (at least once in its main loop) thus returning to the scheduler's own main loop. Yielding inserts the current thread at the back of the ready queue, and the process of servicing the ready queue is repeated, thus the system runs by flipping back and forth the between L-threads and scheduler loop.

In the case of pthreads, the preemptive scheduling, time slicing, and support for thread prioritization means that progress is normally possible for any thread that is ready to run. This comes at the price of a relatively heavier context switch and scheduling overhead.

With L-threads the progress of any particular thread is determined by the frequency of rescheduling opportunities in the other L-threads. This means that an errant L-thread monopolizing the CPU might cause scheduling of other threads to be stalled. Due to the lower cost of context switching, however, voluntary rescheduling to ensure progress of other threads, if managed sensibly, is not a prohibitive overhead, and overall performance can exceed that of an application using pthreads.

Mutual exclusion

With pthreads preemption means that threads that share data must observe some form of mutual exclusion protocol.

The fact that L-threads cannot preempt each other means that in many cases mutual exclusion devices can be completely avoided.

Locking to protect shared data can be a significant bottleneck in multi-threaded applications so a carefully designed cooperatively scheduled program can enjoy significant performance advantages.

So far we have considered only the simplistic case of a single core CPU, when multiple CPUs are considered things are somewhat more complex.

First of all it is inevitable that there must be multiple L-thread schedulers, one running on each EAL thread. So long as these schedulers remain isolated from each other the above assertions about the potential advantages of cooperative scheduling hold true.

A configuration with isolated cooperative schedulers is less flexible than the pthread model where threads can be affinitized to run on any CPU. With isolated schedulers scaling of applications to utilize fewer or more CPUs according to system demand is very difficult to achieve.

The L-thread subsystem makes it possible for L-threads to migrate between schedulers running on different CPUs. Needless to say if the migration means that threads that share data end up running on different CPUs then this will introduce the need for some kind of mutual exclusion system.

Of course `rte_ring` software rings can always be used to interconnect threads running on different cores, however to protect other kinds of shared data structures, lock free constructs or else explicit locking will be required. This is a consideration for the application design.

In support of this extended functionality, the L-thread subsystem implements thread safe mutexes and condition variables.

The cost of affinitizing and of condition variable signaling is significantly lower than the equivalent pthread operations, and so applications using these features will see a performance benefit.

Thread local storage

As with applications written for pthreads an application written for L-threads can take advantage of thread local storage, in this case local to an L-thread. An application may save and retrieve a single pointer to application data in the L-thread struct.

For legacy and backward compatibility reasons two alternative methods are also offered, the first is modelled directly on the pthread get/set specific APIs, the second approach is modelled on the `RTE_PER_LCORE` macros, whereby `PER_LTHREAD` macros are introduced, in both cases the storage is local to the L-thread.

Constraints and performance implications when using L-threads

API compatibility

The L-thread subsystem provides a set of functions that are logically equivalent to the corresponding functions offered by the POSIX pthread library, however not all pthread functions have a corresponding L-thread equivalent, and not all features available to pthreads are implemented for L-threads.

The pthread library offers considerable flexibility via programmable attributes that can be associated with threads, mutexes, and condition variables.

By contrast the L-thread subsystem has fixed functionality, the scheduler policy cannot be varied, and L-threads cannot be prioritized. There are no variable attributes associated with any L-thread objects. L-threads, mutexes and conditional variables, all have fixed functionality. (Note: reserved parameters are included in the APIs to facilitate possible future support for attributes).

The table below lists the pthread and equivalent L-thread APIs with notes on differences and/or constraints. Where there is no L-thread entry in the table, then the L-thread subsystem provides no equivalent function.

Table 3.22: Pthread and equivalent L-thread APIs.

Pthread function	L-thread function	Notes
pthread_barrier_destroy		
pthread_barrier_init		
pthread_barrier_wait		
pthread_cond_broadcast	lthread_cond_broadcast	See note 1
pthread_cond_destroy	lthread_cond_destroy	
pthread_cond_init	lthread_cond_init	
pthread_cond_signal	lthread_cond_signal	See note 1
pthread_cond_timedwait		
pthread_cond_wait	lthread_cond_wait	See note 5
pthread_create	lthread_create	See notes 2, 3
pthread_detach	lthread_detach	See note 4
pthread_equal		
pthread_exit	lthread_exit	
pthread_getspecific	lthread_getspecific	
pthread_getcpuclockid		
pthread_join	lthread_join	
pthread_key_create	lthread_key_create	
pthread_key_delete	lthread_key_delete	
pthread_mutex_destroy	lthread_mutex_destroy	
pthread_mutex_init	lthread_mutex_init	
pthread_mutex_lock	lthread_mutex_lock	See note 6

Continued on next page

Table 3.22 – continued from previous page

Pthread function	L-thread function	Notes
pthread_mutex_trylock	lthread_mutex_trylock	See note 6
pthread_mutex_timedlock		
pthread_mutex_unlock	lthread_mutex_unlock	
pthread_once		
pthread_rwlock_destroy		
pthread_rwlock_init		
pthread_rwlock_rdlock		
pthread_rwlock_timedrdlock		
pthread_rwlock_timedwrlock		
pthread_rwlock_tryrdlock		
pthread_rwlock_trywrlock		
pthread_rwlock_unlock		
pthread_rwlock_wrlock		
pthread_self	lthread_current	
pthread_setspecific	lthread_setspecific	
pthread_spin_init		See note 10
pthread_spin_destroy		See note 10
pthread_spin_lock		See note 10
pthread_spin_trylock		See note 10
pthread_spin_unlock		See note 10
pthread_cancel	lthread_cancel	
pthread_setcancelstate		
pthread_setcanceltype		
pthread_testcancel		
pthread_getschedparam		
pthread_setschedparam		
pthread_yield	lthread_yield	See note 7
pthread_setaffinity_np	lthread_set_affinity	See notes 2, 3, 8
	lthread_sleep	See note 9
	lthread_sleep_clks	See note 9

Note 1:

Neither lthread signal nor broadcast may be called concurrently by L-threads running on different schedulers, although multiple L-threads running in the same scheduler may freely perform signal or broadcast operations. L-threads running on the same or different schedulers may always safely wait on a condition variable.

Note 2:

Pthread attributes may be used to affinitize a pthread with a cpu-set. The L-thread subsystem does not support a cpu-set. An L-thread may be affinitized only with a single CPU at any time.

Note 3:

If an L-thread is intended to run on a different NUMA node than the node that creates the thread then, when calling `lthread_create()` it is advantageous to specify the destination core as a parameter of `lthread_create()`. See *Memory allocation and NUMA awareness* for details.

Note 4:

An L-thread can only detach itself, and cannot detach other L-threads.

Note 5:

A wait operation on a pthread condition variable is always associated with and protected by a mutex which must be owned by the thread at the time it invokes `pthread_wait()`. By contrast L-thread condition variables are thread safe (for waiters) and do not use an associated mutex. Multiple L-threads (including L-threads running on other schedulers) can safely wait on a L-thread condition variable. As a consequence the performance of an L-thread condition variables is typically an order of magnitude faster than its pthread counterpart.

Note 6:

Recursive locking is not supported with L-threads, attempts to take a lock recursively will be detected and rejected.

Note 7:

`lthread_yield()` will save the current context, insert the current thread to the back of the ready queue, and resume the next ready thread. Yielding increases ready queue backlog, see *Ready queue backlog* for more details about the implications of this.

N.B. The context switch time as measured from immediately before the call to `lthread_yield()` to the point at which the next ready thread is resumed, can be an order of magnitude faster that the same measurement for pthread_yield.

Note 8:

`lthread_set_affinity()` is similar to a yield apart from the fact that the yielding thread is inserted into a peer ready queue of another scheduler. The peer ready queue is actually a separate thread safe queue, which means that threads appearing in the peer ready queue can jump any backlog in the local ready queue on the destination scheduler.

The context switch time as measured from the time just before the call to `lthread_set_affinity()` to just after the same thread is resumed on the new scheduler can be orders of magnitude faster than the same measurement for `pthread_setaffinity_np()`.

Note 9:

Although there is no `pthread_sleep()` function, `lthread_sleep()` and `lthread_sleep_clks()` can be used wherever `sleep()`, `usleep()` or `nanosleep()` might ordinarily be used. The L-thread sleep functions suspend the current thread, start an `rte_timer` and resume the thread when the timer matures. The `rte_timer_manage()` entry point is called on every pass of the scheduler loop. This means that the worst case jitter on timer expiry is determined by the longest period between context switches of any running L-threads.

In a synthetic test with many threads sleeping and resuming then the measured jitter is typically orders of magnitude lower than the same measurement made for `nanosleep()`.

Note 10:

Spin locks are not provided because they are problematical in a cooperative environment, see *Locks and spinlocks* for a more detailed discussion on how to avoid spin locks.

Thread local storage

Of the three L-thread local storage options the simplest and most efficient is storing a single application data pointer in the L-thread struct.

The `PER_LTHREAD` macros involve a run time computation to obtain the address of the variable being saved/retrieved and also require that the accesses are de-referenced via a pointer. This means that code that has used `RTE_PER_LCORE` macros being ported to L-threads might need some slight adjustment (see *Thread local storage* for hints about porting code that makes use of thread local storage).

The get/set specific APIs are consistent with their pthread counterparts both in use and in performance.

Memory allocation and NUMA awareness

All memory allocation is from DPDK huge pages, and is NUMA aware. Each scheduler maintains its own caches of objects: lthreads, their stacks, TLS, mutexes and condition variables. These caches are implemented as unbounded lock free MPSC queues. When objects are created they are always allocated from the caches on the local core (current EAL thread).

If an L-thread has been affinitized to a different scheduler, then it can always safely free resources to the caches from which they originated (because the caches are MPSC queues).

If the L-thread has been affinitized to a different NUMA node then the memory resources associated with it may incur longer access latency.

The commonly used pattern of setting affinity on entry to a thread after it has started, means that memory allocation for both the stack and TLS will have been made from caches on the NUMA node on which the threads creator is running. This has the side effect that access latency will be sub-optimal after affinitizing.

This side effect can be mitigated to some extent (although not completely) by specifying the destination CPU as a parameter of `lthread_create()` this causes the L-thread's stack and TLS to be allocated when it is first scheduled on the destination scheduler, if the destination is a on another NUMA node it results in a more optimal memory allocation.

Note that the lthread struct itself remains allocated from memory on the creating node, this is unavoidable because an L-thread is known everywhere by the address of this struct.

Object cache sizing

The per lcore object caches pre-allocate objects in bulk whenever a request to allocate an object finds a cache empty. By default 100 objects are pre-allocated, this is defined by `LTHREAD_PREALLOC` in the public API header file lthread_api.h. This means that the caches constantly grow to meet system demand.

In the present implementation there is no mechanism to reduce the cache sizes if system demand reduces. Thus the caches will remain at their maximum extent indefinitely.

A consequence of the bulk pre-allocation of objects is that every 100 (default value) additional new object create operations results in a call to `rte_malloc()`. For creation of objects such as L-threads, which trigger the allocation of even more objects (i.e. their stacks and TLS) then this can cause outliers in scheduling performance.

If this is a problem the simplest mitigation strategy is to dimension the system, by setting the bulk object pre-allocation size to some large number that you do not expect to be exceeded. This means the caches will be populated once only, the very first time a thread is created.

Ready queue backlog

One of the more subtle performance considerations is managing the ready queue backlog. The fewer threads that are waiting in the ready queue then the faster any particular thread will get serviced.

In a naive L-thread application with N L-threads simply looping and yielding, this backlog will always be equal to the number of L-threads, thus the cost of a yield to a particular L-thread will be N times the context switch time.

This side effect can be mitigated by arranging for threads to be suspended and wait to be resumed, rather than polling for work by constantly yielding. Blocking on a mutex or condition variable or even more obviously having a thread sleep if it has a low frequency workload are all mechanisms by which a thread can be excluded from the ready queue until it really does need to be run. This can have a significant positive impact on performance.

Initialization, shutdown and dependencies

The L-thread subsystem depends on DPDK for huge page allocation and depends on the `rte_timer subsystem`. The DPDK EAL initialization and `rte_timer_subsystem_init()` **MUST** be completed before the L-thread sub system can be used.

Thereafter initialization of the L-thread subsystem is largely transparent to the application. Constructor functions ensure that global variables are properly initialized. Other than global variables each scheduler is initialized independently the first time that an L-thread is created by a particular EAL thread.

If the schedulers are to be run as isolated and independent schedulers, with no intention that L-threads running on different schedulers will migrate between schedulers or synchronize with L-threads running on other schedulers, then initialization consists simply of creating an L-thread, and then running the L-thread scheduler.

If there will be interaction between L-threads running on different schedulers, then it is important that the starting of schedulers on different EAL threads is synchronized.

To achieve this an additional initialization step is necessary, this is simply to set the number of schedulers by calling the API function `lthread_num_schedulers_set(n)`, where `n` is the number of EAL threads that will run L-thread schedulers. Setting the number of schedulers to a number greater than 0 will cause all schedulers to wait until the others have started before beginning to schedule L-threads.

The L-thread scheduler is started by calling the function `lthread_run()` and should be called from the EAL thread and thus become the main loop of the EAL thread.

The function `lthread_run()`, will not return until all threads running on the scheduler have exited, and the scheduler has been explicitly stopped by calling `lthread_scheduler_shutdown(lcore)` or `lthread_scheduler_shutdown_all()`.

All these function do is tell the scheduler that it can exit when there are no longer any running L-threads, neither function forces any running L-thread to terminate. Any desired application shutdown behavior must be designed and built into the application to ensure that L-threads complete in a timely manner.

Important Note: It is assumed when the scheduler exits that the application is terminating for good, the scheduler does not free resources before exiting and running the scheduler a subsequent time will result in undefined behavior.

Porting legacy code to run on L-threads

Legacy code originally written for a pthread environment may be ported to L-threads if the considerations about differences in scheduling policy, and constraints discussed in the previous sections can be accommodated.

This section looks in more detail at some of the issues that may have to be resolved when porting code.

pthread API compatibility

The first step is to establish exactly which pthread APIs the legacy application uses, and to understand the requirements of those APIs. If there are corresponding L-lthread APIs, and where the default pthread functionality is used by the application then, notwithstanding the other issues discussed here, it should be feasible to run the application with L-threads. If the legacy code modifies the default behavior using attributes then if may be necessary to make some adjustments to eliminate those requirements.

Blocking system API calls

It is important to understand what other system services the application may be using, bearing in mind that in a cooperatively scheduled environment a thread cannot block without stalling the scheduler and with it all other cooperative

threads. Any kind of blocking system call, for example file or socket IO, is a potential problem, a good tool to analyze the application for this purpose is the `strace` utility.

There are many strategies to resolve these kind of issues, each with it merits. Possible solutions include:

- Adopting a polled mode of the system API concerned (if available).
- Arranging for another core to perform the function and synchronizing with that core via constructs that will not block the L-thread.
- Affinitizing the thread to another scheduler devoted (as a matter of policy) to handling threads wishing to make blocking calls, and then back again when finished.

Locks and spinlocks

Locks and spinlocks are another source of blocking behavior that for the same reasons as system calls will need to be addressed.

If the application design ensures that the contending L-threads will always run on the same scheduler then it its probably safe to remove locks and spin locks completely.

The only exception to the above rule is if for some reason the code performs any kind of context switch whilst holding the lock (e.g. yield, sleep, or block on a different lock, or on a condition variable). This will need to determined before deciding to eliminate a lock.

If a lock cannot be eliminated then an L-thread mutex can be substituted for either kind of lock.

An L-thread blocking on an L-thread mutex will be suspended and will cause another ready L-thread to be resumed, thus not blocking the scheduler. When default behavior is required, it can be used as a direct replacement for a pthread mutex lock.

Spin locks are typically used when lock contention is likely to be rare and where the period during which the lock may be held is relatively short. When the contending L-threads are running on the same scheduler then an L-thread blocking on a spin lock will enter an infinite loop stopping the scheduler completely (see *Infinite loops* below).

If the application design ensures that contending L-threads will always run on different schedulers then it might be reasonable to leave a short spin lock that rarely experiences contention in place.

If after all considerations it appears that a spin lock can neither be eliminated completely, replaced with an L-thread mutex, or left in place as is, then an alternative is to loop on a flag, with a call to `lthread_yield()` inside the loop (n.b. if the contending L-threads might ever run on different schedulers the flag will need to be manipulated atomically).

Spinning and yielding is the least preferred solution since it introduces ready queue backlog (see also *Ready queue backlog*).

Sleeps and delays

Yet another kind of blocking behavior (albeit momentary) are delay functions like `sleep()`, `usleep()`, `nanosleep()` etc. All will have the consequence of stalling the L-thread scheduler and unless the delay is very short (e.g. a very short nanosleep) calls to these functions will need to be eliminated.

The simplest mitigation strategy is to use the L-thread sleep API functions, of which two variants exist, `lthread_sleep()` and `lthread_sleep_clks()`. These functions start an rte_timer against the L-thread, suspend the L-thread and cause another ready L-thread to be resumed. The suspended L-thread is resumed when the rte_timer matures.

Infinite loops

Some applications have threads with loops that contain no inherent rescheduling opportunity, and rely solely on the OS time slicing to share the CPU. In a cooperative environment this will stop everything dead. These kind of loops are not hard to identify, in a debug session you will find the debugger is always stopping in the same loop.

The simplest solution to this kind of problem is to insert an explicit `lthread_yield()` or `lthread_sleep()` into the loop. Another solution might be to include the function performed by the loop into the execution path of some other loop that does in fact yield, if this is possible.

Thread local storage

If the application uses thread local storage, the use case should be studied carefully.

In a legacy pthread application either or both the `__thread` prefix, or the pthread set/get specific APIs may have been used to define storage local to a pthread.

In some applications it may be a reasonable assumption that the data could or in fact most likely should be placed in L-thread local storage.

If the application (like many DPDK applications) has assumed a certain relationship between a pthread and the CPU to which it is affinitized, there is a risk that thread local storage may have been used to save some data items that are correctly logically associated with the CPU, and others items which relate to application context for the thread. Only a good understanding of the application will reveal such cases.

If the application requires an that an L-thread is to be able to move between schedulers then care should be taken to separate these kinds of data, into per lcore, and per L-thread storage. In this way a migrating thread will bring with it the local data it needs, and pick up the new logical core specific values from pthread local storage at its new home.

Pthread shim

A convenient way to get something working with legacy code can be to use a shim that adapts pthread API calls to the corresponding L-thread ones. This approach will not mitigate any of the porting considerations mentioned in the previous sections, but it will reduce the amount of code churn that would otherwise been involved. It is a reasonable approach to evaluate L-threads, before investing effort in porting to the native L-thread APIs.

Overview

The L-thread subsystem includes an example pthread shim. This is a partial implementation but does contain the API stubs needed to get basic applications running. There is a simple “hello world” application that demonstrates the use of the pthread shim.

A subtlety of working with a shim is that the application will still need to make use of the genuine pthread library functions, at the very least in order to create the EAL threads in which the L-thread schedulers will run. This is the case with DPDK initialization, and exit.

To deal with the initialization and shutdown scenarios, the shim is capable of switching on or off its adaptor functionality, an application can control this behavior by the calling the function `pt_override_set()`. The default state is disabled.

The pthread shim uses the dynamic linker loader and saves the loaded addresses of the genuine pthread API functions in an internal table, when the shim functionality is enabled it performs the adaptor function, when disabled it invokes the genuine pthread function.

The function pthread_exit() has additional special handling. The standard system header file pthread.h declares pthread_exit() with __attribute__((noreturn)) this is an optimization that is possible because the pthread is terminating and this enables the compiler to omit the normal handling of stack and protection of registers since the function is not expected to return, and in fact the thread is being destroyed. These optimizations are applied in both the callee and the caller of the pthread_exit() function.

In our cooperative scheduling environment this behavior is inadmissible. The pthread is the L-thread scheduler thread, and, although an L-thread is terminating, there must be a return to the scheduler in order that the system can continue to run. Further, returning from a function with attribute noreturn is invalid and may result in undefined behavior.

The solution is to redefine the pthread_exit function with a macro, causing it to be mapped to a stub function in the shim that does not have the noreturn attribute. This macro is defined in the file pthread_shim.h. The stub function is otherwise no different than any of the other stub functions in the shim, and will switch between the real pthread_exit() function or the lthread_exit() function as required. The only difference is that the mapping to the stub by macro substitution.

A consequence of this is that the file pthread_shim.h must be included in legacy code wishing to make use of the shim. It also means that dynamic linkage of a pre-compiled binary that did not include pthread_shim.h is not be supported.

Given the requirements for porting legacy code outlined in *Porting legacy code to run on L-threads* most applications will require at least some minimal adjustment and recompilation to run on L-threads so pre-compiled binaries are unlikely to be met in practice.

In summary the shim approach adds some overhead but can be a useful tool to help establish the feasibility of a code reuse project. It is also a fairly straightforward task to extend the shim if necessary.

Note: Bearing in mind the preceding discussions about the impact of making blocking calls then switching the shim in and out on the fly to invoke any pthread API this might block is something that should typically be avoided.

Building and running the pthread shim

The shim example application is located in the sample application in the performance-thread folder

To build and run the pthread shim example

1. Go to the example applications folder

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}/examples/performance-thread/pthread_shim
```

2. Set the target (a default target is used if not specified). For example:

```
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

   See the DPDK Getting Started Guide for possible RTE_TARGET values.

3. Build the application:

```
make
```

4. To run the pthread_shim example

```
lthread-pthread-shim -c core_mask -n number_of_channels
```

L-thread Diagnostics

When debugging you must take account of the fact that the L-threads are run in a single pthread. The current scheduler is defined by `RTE_PER_LCORE(this_sched)`, and the current lthread is stored at `RTE_PER_LCORE(this_sched)->current_lthread`. Thus on a breakpoint in a GDB session the current lthread can be obtained by displaying the pthread local variable `per_lcore_this_sched->current_lthread`.

Another useful diagnostic feature is the possibility to trace significant events in the life of an L-thread, this feature is enabled by changing the value of LTHREAD_DIAG from 0 to 1 in the file `lthread_diag_api.h`.

Tracing of events can be individually masked, and the mask may be programmed at run time. An unmasked event results in a callback that provides information about the event. The default callback simply prints trace information. The default mask is 0 (all events off) the mask can be modified by calling the function `lthread_diagniostic_set_mask()`.

It is possible register a user callback function to implement more sophisticated diagnostic functions. Object creation events (lthread, mutex, and condition variable) accept, and store in the created object, a user supplied reference value returned by the callback function.

The lthread reference value is passed back in all subsequent event callbacks, the mutex and APIs are provided to retrieve the reference value from mutexes and condition variables. This enables a user to monitor, count, or filter for specific events, on specific objects, for example to monitor for a specific thread signaling a specific condition variable, or to monitor on all timer events, the possibilities and combinations are endless.

The callback function can be set by calling the function `lthread_diagnostic_enable()` supplying a callback function pointer and an event mask.

Setting `LTHREAD_DIAG` also enables counting of statistics about cache and queue usage, and these statistics can be displayed by calling the function `lthread_diag_stats_display()`. This function also performs a consistency check on the caches and queues. The function should only be called from the master EAL thread after all slave threads have stopped and returned to the C main program, otherwise the consistency check will fail.

3.40 IPsec Security Gateway Sample Application

The IPsec Security Gateway application is an example of a "real world" application using DPDK cryptodev framework.

3.40.1 Overview

The application demonstrates the implementation of a Security Gateway (not IPsec compliant, see the Constraints section below) using DPDK based on RFC4301, RFC4303, RFC3602 and RFC2404.

Internet Key Exchange (IKE) is not implemented, so only manual setting of Security Policies and Security Associations is supported.

The Security Policies (SP) are implemented as ACL rules, the Security Associations (SA) are stored in a table and the routing is implemented using LPM.

The application classifies the ports as *Protected* and *Unprotected*. Thus, traffic received on an Unprotected or Protected port is consider Inbound or Outbound respectively.

The Path for IPsec Inbound traffic is:

- Read packets from the port.
- Classify packets between IPv4 and ESP.
- Perform Inbound SA lookup for ESP packets based on their SPI.

- Perform Verification/Decryption.
- Remove ESP and outer IP header
- Inbound SP check using ACL of decrypted packets and any other IPv4 packets.
- Routing.
- Write packet to port.

The Path for the IPsec Outbound traffic is:

- Read packets from the port.
- Perform Outbound SP check using ACL of all IPv4 traffic.
- Perform Outbound SA lookup for packets that need IPsec protection.
- Add ESP and outer IP header.
- Perform Encryption/Digest.
- Routing.
- Write packet to port.

3.40.2 Constraints

- No IPv6 options headers.
- No AH mode.
- Supported algorithms: AES-CBC, AES-CTR, AES-GCM, HMAC-SHA1 and NULL.
- Each SA must be handle by a unique lcore (*1 RX queue per port*).
- No chained mbufs.

3.40.3 Compiling the Application

To compile the application:

1. Go to the sample application directory:

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}/examples/ipsec-secgw
```

2. Set the target (a default target is used if not specified). For example:

```
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

   See the *DPDK Getting Started Guide* for possible RTE_TARGET values.

3. Build the application:

```
make
```

4. [Optional] Build the application for debugging: This option adds some extra flags, disables compiler optimizations and is verbose:

```
make DEBUG=1
```

3.40.4 Running the Application

The application has a number of command line options:

```
./build/ipsec-secgw [EAL options] --
                     -p PORTMASK -P -u PORTMASK
                     --config (port,queue,lcore)[,(port,queue,lcore]
                     --single-sa SAIDX
                     -f CONFIG_FILE_PATH
```

Where:

- `-p PORTMASK`: Hexadecimal bitmask of ports to configure.
- `-P`: *optional*. Sets all ports to promiscuous mode so that packets are accepted regardless of the packet's Ethernet MAC destination address. Without this option, only packets with the Ethernet MAC destination address set to the Ethernet address of the port are accepted (default is enabled).
- `-u PORTMASK`: hexadecimal bitmask of unprotected ports
- `--config (port,queue,lcore)[,(port,queue,lcore)]`: determines which queues from which ports are mapped to which cores.
- `--single-sa SAIDX`: use a single SA for outbound traffic, bypassing the SP on both Inbound and Outbound. This option is meant for debugging/performance purposes.
- `-f CONFIG_FILE_PATH`: the full path of text-based file containing all configuration items for running the application (See Configuration file syntax section below). `-f CONFIG_FILE_PATH` **must** be specified. **ONLY** the UNIX format configuration file is accepted.

The mapping of lcores to port/queues is similar to other l3fwd applications.

For example, given the following command line:

```
./build/ipsec-secgw -l 20,21 -n 4 --socket-mem 0,2048       \
       --vdev "cryptodev_null_pmd" -- -p 0xf -P -u 0x3      \
       --config="(0,0,20),(1,0,20),(2,0,21),(3,0,21)"       \
       -f /path/to/config_file                              \
```

where each options means:

- The `-l` option enables cores 20 and 21.
- The `-n` option sets memory 4 channels.
- The `--socket-mem` to use 2GB on socket 1.
- The `--vdev "cryptodev_null_pmd"` option creates virtual NULL cryptodev PMD.
- The `-p` option enables ports (detected) 0, 1, 2 and 3.
- The `-P` option enables promiscuous mode.
- The `-u` option sets ports 1 and 2 as unprotected, leaving 2 and 3 as protected.
- The `--config` option enables one queue per port with the following mapping:

Port	Queue	lcore	Description
0	0	20	Map queue 0 from port 0 to lcore 20.
1	0	20	Map queue 0 from port 1 to lcore 20.
2	0	21	Map queue 0 from port 2 to lcore 21.
3	0	21	Map queue 0 from port 3 to lcore 21.

- The `-f /path/to/config_file` option enables the application read and parse the configuration file specified, and configures the application with a given set of SP, SA and Routing entries accordingly. The syntax of the configuration file will be explained below in more detail. Please **note** the parser only accepts UNIX format text file. Other formats such as DOS/MAC format will cause a parse error.

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

The application would do a best effort to "map" crypto devices to cores, with hardware devices having priority. Basically, hardware devices if present would be assigned to a core before software ones. This means that if the application is using a single core and both hardware and software crypto devices are detected, hardware devices will be used.

A way to achieve the case where you want to force the use of virtual crypto devices is to whitelist the Ethernet devices needed and therefore implicitly blacklisting all hardware crypto devices.

For example, something like the following command line:

```
./build/ipsec-secgw -l 20,21 -n 4 --socket-mem 0,2048 \
        -w 81:00.0 -w 81:00.1 -w 81:00.2 -w 81:00.3 \
        --vdev "cryptodev_aesni_mb_pmd" --vdev "cryptodev_null_pmd" \
        -- \
        -p 0xf -P -u 0x3 --config="(0,0,20),(1,0,20),(2,0,21),(3,0,21)" \
        -f sample.cfg
```

3.40.5 Configurations

The following sections provide the syntax of configurations to initialize your SP, SA and Routing tables. Configurations shall be specified in the configuration file to be passed to the application. The file is then parsed by the application. The successful parsing will result in the appropriate rules being applied to the tables accordingly.

Configuration File Syntax

As mention in the overview, the Security Policies are ACL rules. The application parsers the rules specified in the configuration file and passes them to the ACL table, and replicates them per socket in use.

Following are the configuration file syntax.

General rule syntax

The parse treats one line in the configuration file as one configuration item (unless the line concatenation symbol exists). Every configuration item shall follow the syntax of either SP, SA, or Routing rules specified below.

The configuration parser supports the following special symbols:

- Comment symbol **#**. Any character from this symbol to the end of line is treated as comment and will not be parsed.
- Line concatenation symbol ****. This symbol shall be placed in the end of the line to be concatenated to the line below. Multiple lines' concatenation is supported.

SP rule syntax

The SP rule syntax is shown as follows:

```
sp <ip_ver> <dir> esp <action> <priority> <src_ip> <dst_ip>
<proto> <sport> <dport>
```

where each options means:

`<ip_ver>`

- IP protocol version
- Optional: No
- Available options:
  - *ipv4*: IP protocol version 4
  - *ipv6*: IP protocol version 6

`<dir>`

- The traffic direction
- Optional: No
- Available options:
  - *in*: inbound traffic
  - *out*: outbound traffic

`<action>`

- IPsec action
- Optional: No
- Available options:
  - *protect* <*SA_idx*>: the specified traffic is protected by SA rule with id SA_idx
  - *bypass*: the specified traffic traffic is bypassed
  - *discard*: the specified traffic is discarded

`<priority>`

- Rule priority
- Optional: Yes, default priority 0 will be used
- Syntax: *pri* <*id*>

`<src_ip>`

- The source IP address and mask
- Optional: Yes, default address 0.0.0.0 and mask of 0 will be used
- Syntax:
  - *src X.X.X.X/Y* for IPv4
  - *src XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XXXX/Y* for IPv6

`<dst_ip>`

- The destination IP address and mask
- Optional: Yes, default address 0.0.0.0 and mask of 0 will be used
- Syntax:

  - *dst X.X.X.X/Y* for IPv4
  - *dst XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XXXX/Y* for IPv6

`<proto>`

- The protocol start and end range
- Optional: yes, default range of 0 to 0 will be used
- Syntax: *proto X:Y*

`<sport>`

- The source port start and end range
- Optional: yes, default range of 0 to 0 will be used
- Syntax: *sport X:Y*

`<dport>`

- The destination port start and end range
- Optional: yes, default range of 0 to 0 will be used
- Syntax: *dport X:Y*

Example SP rules:

```
sp ipv4 out esp protect 105 pri 1 dst 192.168.115.0/24 sport 0:65535 \
dport 0:65535

sp ipv6 in esp bypass pri 1 dst 0000:0000:0000:0000:5555:5555:\
0000:0000/96 sport 0:65535 dport 0:65535
```

SA rule syntax

The successfully parsed SA rules will be stored in an array table.

The SA rule syntax is shown as follows:

```
sa <dir> <spi> <cipher_algo> <cipher_key> <auth_algo> <auth_key>
<mode> <src_ip> <dst_ip>
```

where each options means:

`<dir>`

- The traffic direction
- Optional: No
- Available options:
  - *in*: inbound traffic
  - *out*: outbound traffic

`<spi>`

- The SPI number
- Optional: No
- Syntax: unsigned integer number

<cipher_algo>

- Cipher algorithm
- Optional: No
- Available options:
  - *null*: NULL algorithm
  - *aes-128-cbc*: AES-CBC 128-bit algorithm
  - *aes-128-ctr*: AES-CTR 128-bit algorithm
  - *aes-128-gcm*: AES-GCM 128-bit algorithm
- Syntax: *cipher_algo <your algorithm>*

<cipher_key>

- Cipher key, NOT available when 'null' algorithm is used
- Optional: No, must followed by <cipher_algo> option
- Syntax: Hexadecimal bytes (0x0-0xFF) concatenate by colon symbol ':'. The number of bytes should be as same as the specified cipher algorithm key size.

  For example: *cipher_key A1:B2:C3:D4:A1:B2:C3:D4:A1:B2:C3:D4: A1:B2:C3:D4*

<auth_algo>

- Authentication algorithm
- Optional: No
- Available options:
  - *null*: NULL algorithm
  - *sha1-hmac*: HMAC SHA1 algorithm
  - *aes-128-gcm*: AES-GCM 128-bit algorithm

<auth_key>

- Authentication key, NOT available when 'null' or 'aes-128-gcm' algorithm is used.
- Optional: No, must followed by <auth_algo> option
- Syntax: Hexadecimal bytes (0x0-0xFF) concatenate by colon symbol ':'. The number of bytes should be as same as the specified authentication algorithm key size.

  For example: *auth_key A1:B2:C3:D4:A1:B2:C3:D4:A1:B2:C3:D4:A1:B2:C3:D4: A1:B2:C3:D4*

<mode>

- The operation mode
- Optional: No
- Available options:
  - *ipv4-tunnel*: Tunnel mode for IPv4 packets
  - *ipv6-tunnel*: Tunnel mode for IPv6 packets
  - *transport*: transport mode
- Syntax: mode XXX

<src_ip>

- The source IP address. This option is not available when transport mode is used
- Optional: Yes, default address 0.0.0.0 will be used
- Syntax:
  - *src X.X.X.X* for IPv4
  - *src XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XXXX* for IPv6

`<dst_ip>`

- The destination IP address. This option is not available when transport mode is used
- Optional: Yes, default address 0.0.0.0 will be used
- Syntax:
  - *dst X.X.X.X* for IPv4
  - *dst XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XXXX* for IPv6

Example SA rules:

```
sa out 5 cipher_algo null auth_algo null mode ipv4-tunnel \
src 172.16.1.5 dst 172.16.2.5

sa out 25 cipher_algo aes-128-cbc \
cipher_key c3:c3:c3:c3:c3:c3:c3:c3:c3:c3:c3:c3:c3:c3:c3:c3 \
auth_algo sha1-hmac \
auth_key c3:c3:c3:c3:c3:c3:c3:c3:c3:c3:c3:c3:c3:c3:c3:c3:c3:c3:c3:c3 \
mode ipv6-tunnel \
src 1111:1111:1111:1111:1111:1111:1111:5555 \
dst 2222:2222:2222:2222:2222:2222:2222:5555

sa in 105 cipher_algo aes-128-gcm \
cipher_key de:ad:be:ef:de:ad:be:ef:de:ad:be:ef:de:ad:be:ef \
auth_algo aes-128-gcm \
mode ipv4-tunnel src 172.16.2.5 dst 172.16.1.5
```

Routing rule syntax

The Routing rule syntax is shown as follows:

```
rt <ip_ver> <src_ip> <dst_ip> <port>
```

where each options means:

`<ip_ver>`

- IP protocol version
- Optional: No
- Available options:
  - *ipv4*: IP protocol version 4
  - *ipv6*: IP protocol version 6

`<src_ip>`

- The source IP address and mask

- Optional: Yes, default address 0.0.0.0 and mask of 0 will be used
- Syntax:
  - *src X.X.X.X/Y* for IPv4
  - *src XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XXXX/Y* for IPv6

`<dst_ip>`

- The destination IP address and mask
- Optional: Yes, default address 0.0.0.0 and mask of 0 will be used
- Syntax:
  - *dst X.X.X.X/Y* for IPv4
  - *dst XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XXXX/Y* for IPv6

`<port>`

- The traffic output port id
- Optional: yes, default output port 0 will be used
- Syntax: *port X*

Example SP rules:

```
rt ipv4 dst 172.16.1.5/32 port 0

rt ipv6 dst 1111:1111:1111:1111:1111:1111:1111:5555/116 port 0
```

Figures

Fig. 3.1 *Packet Flow*

Fig. 3.2 *Kernel NIC Application Packet Flow*

Fig. 3.4 *Performance Benchmark Setup (Basic Environment)*

Fig. 3.5 *Performance Benchmark Setup (Virtualized Environment)*

Fig. 3.6 *Performance Benchmark Setup (Basic Environment)*

Fig. 3.7 *Performance Benchmark Setup (Virtualized Environment)*

Fig. 3.3 *Encryption flow Through the L2 Forwarding with Crypto Application*

Fig. 3.9 *A typical IPv4 ACL rule*

Fig. 3.10 *Rules example*

Fig. 3.11 *Load Balancer Application Architecture*

Fig. 3.13 *Example Data Flow in a Symmetric Multi-process Application*

Fig. 3.14 *Example Data Flow in a Client-Server Symmetric Multi-process Application*

Fig. 3.15 *Master-slave Process Workflow*

Fig. 3.16 *Slave Process Recovery Process Flow*

Fig. 3.17 *QoS Scheduler Application Architecture*

Fig. 3.18 *Pipeline Overview*

Fig. 3.19 *Ring-based Processing Pipeline Performance Setup*

Fig. 3.20 *Threads and Pipelines*

Fig. 3.21 *Packet Flow Through the VMDQ and DCB Sample Application*

Fig. 3.25 *Test Pipeline Application*

Fig. 3.26 *Performance Benchmarking Setup (Basic Environment)*

Fig. 3.27 *Distributor Sample Application Layout*

Fig. 3.28 *Highlevel Solution*

Fig. 3.29 *VM request to scale frequency* Fig. 3.30 *Overlay Networking.* Fig. 3.31 *TEP termination Framework Overview*

Fig. 3.32 *PTP Synchronization Protocol*

Fig. 3.12 *Using EFD as a Flow-Level Load Balancer*

Tables

Table 3.1 *Output Traffic Marking*

Table 3.2 *Entity Types*

Table 3.21 *Table Types*

CHAPTER 4

编程指南

4.1 简介

本文档提供软件架构信息，开发环境及优化指南。

有关编程示例及如何编译运行这些示例，请参阅 *DPDK*示例用户指南 。

有关编译和运行应用程序的一般信息，请参阅 *DPDK*入门指南 。

4.1.1 文档地图

以下是一份建议顺序阅读的DPDK参考文档列表：

- **发布说明**：提供特性发行版本的信息，包括支持的功能，限制，修复的问题，已知的问题等等。此外，还以FAQ方式提供了常见问题及解答。
- **入门指南**：介绍如何安装和配置DPDK；旨在帮助用户快速上手。
- **FreeBSD* 入门指南**： DPDK1.6.0版本之后添加了FreeBSD*平台上DPDK入门指南。有关如何在FreeBSD*上安装配置DPDK，请参阅这个文档。
- **编程指南** (本文档): 描述如下内容：
  - 软件架构及如何使用（实例介绍），特别是在Linux环境中的用法
  - DPDK的主要内容，系统构建（包括可以在DPDK根目录Makefile中来构建工具包和应用程序的命令）及应用移植细则。
  - 软件中使用的，以及新开发中需要考虑的一些优化。

  还提供了文档使用的术语表。
- **API参考**： 提供有关DPDK功能、数据结构和其他编程结构的详细信息。
- **示例程序用户指南**： 描述了一组例程。 每个章节描述了一个用例，展示了具体的功能，并提供了有关如何编译、运行和使用的说明。

4.1.2 相关刊物

以下文档提供与使用DPDK开发应用程序相关的信息:

- Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide

第一部分：架构概述

4.2 概述

本章节给出了DPDK架构的一个全局的描述。

DPDK的主要目标就是要为数据面快速报文处理应用提供一个简洁但是完整的框架。 用户可以通过代码来理解其中使用的一些技术，并用来构建自己的应用原型或是添加自己的协议栈。 用户也可以替换DPDK提供的原生的选项。

通过创建环境抽象层EAL，DPDK框架为每个特殊的环境创建了运行库。 这个环境抽象层是对底层架构的抽象，通过make和配置文件，在Linux用户空间编译完成。 一旦EAL库编译完成，用户可以通过链接这些库来构建自己的app。 除开环境抽象层，还有一些其他库，包括哈希算法、最长前缀匹配、环形缓冲器。 DPDK提供了一些app用例用来指导如何使用这些特性来创建自己的应用程序。

DPDK实现了run-to-complete报文处理模型，数据面处理程序在调用之前必须预先分配好所有的资源，并作为执行单元运行与逻辑核心上。 这种模型并不支持调度，且所有的设备通过轮询方式访问。 不使用中断方式的主要原因就是中断处理增加了性能开销。

作为RTC模型的扩展，通过使用ring在不同core之间传递报文和消息，也可以实现报文处理的流水线模型（pipeline）。 流水线模型允许操作分阶段执行，在多核代码执行中可能更高效。

4.2.1 开发环境

DPDK项目创建要求Linux环境及相关的工具链，例如一个或多个编译工具、汇编程序、make工具、编辑器及DPDK组建和库用到的库。

当制定环境和架构的库编译出来，这些库就可以用于创建我们自己的数据面处理程序。

创建Linux用户空间app时，需要用到glibc库。 对于DPDP app，必须使用两个全局的环境变量（RTE_SDK & RTE_TARGET），这两个变量必须在编译app之间配置好:

```
export RTE_SDK=/home/user/DPDK
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

也可以参阅 *DPDK*入门指南 来获取更多搭建开发环境的信息。

4.2.2 环境适配层EAL

环境适配层(Environment Abstraction Layer)提供了通用的接口来隐藏了环境细节，使得上层app和库无需考虑这些细节。 EAL提供的服务有:

- DPDK的加载和启动
- 支持多线程和多进程执行方式
- CPU亲和性设置
- 系统内存分配和释放
- 原子操作

- 定时器引用
- PCI总线访问
- 跟踪和调试功能
- CPU特性编号
- 中断处理
- 警告操作
- 内存管理

EAL的更完整的描述请参阅 *Environment Abstraction Layer*.

4.2.3 核心组件

核心组件 指一系列的库，用于为高性能包处理程序提供所有必须的元素。核心组件及其之间的关系如下图所示:

Fig. 4.1: Core Components Architecture

环形缓冲区管理(librte_ring)

Ring数据结构提供了一个无锁的多生产者，多消费者的FIFO表处理接口。 他比无锁队列优异的地方在于它容易部署，适合大量的操作，而且更快。 Ring库在 *Memory Pool Manager (librte_mempool)* 中使用到，而且ring还用于不同核之间或是逻辑核上处理单元之间的通信。 Ring缓存机制及其使用可以参考 *Ring Library*。

内存池管理(librte_mempool)

内存池管理的主要职责就是在内存中分配指定数目对象的POOL。 每个POOL以名称来唯一标识，并且使用一个ring来存储空闲的对象节点。 它还提供了一些其他的服务如对象节点的每核备份缓存及自动对齐以保证元素能均衡的处于每核内存通道上。 内存池分配器具体行为参考 *Mempool Library*。

网络报文缓冲区管理(librte_mbuf)

报文缓存管理器提供了创建、释放报文缓存的能力，DPDK应用程序中可能使用这些报文缓存来存储消息。而消息通常在程序开始时通过DPDK的MEMPOOL库创建并存储。 BUFF库提供了报文申请释放的API，通常消息buff用于缓存普通消息，报文buff用于缓存网络报文。 报文缓存管理参考 *Mbuf Library*。

定时器管理(librte_timer)

这个库位DPDK执行单元提供了定时服务，为函数异步执行提供支持。 定时器可以设置周期调用或只调用一次。 使用EAL提供的接口获取高精度时钟，并且能在每个核上根据需要初始化。 具体参考 *Timer Library*。

4.2.4 以太网轮询驱动架构

DPDK的PMD驱动支持1G、10G、40G。 同时DPDK提供了虚拟的以太网控制器，被设计成非异步，基于中断的模式。 详细内容参考 *Poll Mode Driver*。

4.2.5 报文转发算法支持

DPDK提供了哈希（librte_hash）、最长前缀匹配的（librte_lpm）算法库用于支持包转发。 详细内容查看 *Hash Library* 和 *LPM Library* 。

4.2.6 网络协议库(librte_net)

这个库提供了IP协议的一些定义，以及一些常用的宏。 这些定义都基于FreeBSD IP协议栈的代码，并且包含相关的协议号，IP相关宏定义，IPV4和IPV6头部结构等等。

4.3 环境适配层EAL

环境抽象层为底层资源如硬件和内存空间的访问提供了接口。 这些通用的接口为APP和库隐藏了不同环境的特殊性。 EAL负责初始化及分配资源（内存、PCI设备、定时器、控制台等等）。

EAL提供的典型服务有：

- DPDK的加载和启动：DPDK和指定的程序链接成一个独立的进程，并以某种方式加载
- CPU亲和性和分配处理：DPDK提供机制将执行单元绑定到特定的核上，就像创建一个执行程序一样。
- 系统内存分配：EAL实现了不同区域内存的分配，例如为设备接口提供了物理内存。
- PCI地址抽象：EAL提供了对PCI地址空间的访问接口
- 跟踪调试功能：日志信息，堆栈打印、异常挂起等等。
- 公用功能：提供了标准libc不提供的自旋锁、原子计数器等。
- CPU特征辨识：用于决定CPU运行时的一些特殊功能，决定当前CPU支持的特性，以便编译对应的二进制文件。
- 中断处理：提供接口用于向中断注册/解注册回掉函数。
- 告警功能：提供接口用于设置/取消指定时间环境下运行的毁掉函数。

4.3.1 Linux环境下的EAL

在Linux用户空间环境，DPDK APP通过pthread库作为一个用户态程序运行。 设备的PCI信息和地址空间通过 /sys 内核接口及内核模块如uio_pci_generic或igb_uio来发现获取的。 linux内核文档中UIO描述，设备的UIO信息是在程序中用mmap重新映射的。

EAL通过对hugetlb使用mmap接口来实现物理内存的分配。这部分内存暴露给DPDK服务层，如 *Mempool Library*。

据此，DPDK服务层可以完成初始化，接着通过设置线程亲和性调用，每个执行单元将会分配给特定的逻辑核，以一个user-level等级的线程来运行。

定时器是通过CPU的时间戳计数器TSC或者通过mmap调用内核的HPET系统接口实现。

初始化和运行

初始化部分从glibc的开始函数就执行了。 检查也在初始化过程中被执行，用于保证配置文件所选择的架构宏定义是本CPU所支持的，然后才开始调用main函数。 Core的初始化和运行时在rte_eal_init()接

口上执行的（参考API文档）。它包括对pthread库的调用（更具体的说是 pthread_self(), pthread_create(), 和pthread_setaffinity_np()函数）。

Fig. 4.2: EAL在Linux APP环境中被初始化。

Note: 对象的初始化，例如内存区间、ring、内存池、lpm表或hash表等，必须作为整个程序初始化的一部分，在主逻辑核上完成。创建和初始化这些对象的函数不是多线程安全的，但是，一旦初始化完成，这些对象本身可以作为安全线程运行。

多进程支持

Linux下APP支持像多线程一样部署多进程运行模式，具体参考 *Multi-process Support* 。

内存映射和内存分配

大量连续的物理内存分配时通过hugetlbfs内核文件系统来实现的。EAL提供了相应的接口用于申请指定名字的连续内存空间。这个API同时会将这段连续空间的地址返回给用户程序。

Note: 内存申请时使用rte_malloc接口来做的，它也是hugetlbfs文件系统大页支持的。

Xen Dom0非大页运行支持

现存的内存管理是基于Linux内核的大页机制，然而，Xen Dom0并不支持大页，所以要将一个新的内核模块rte_dom0_mem加载上，以便避开这个限制。

EAL使用IOCTL接口用于通告Linux内核模块rte_mem_dom0去申请指定大小的内存块，并从该模块中获取内存段的信息。EAL使用MMAP接口来映射这段内存。对于申请到的内存段，在其内的物理地址都是连续的，但是实际上，硬件地址只在2M内连续。

PCI 访问

EAL使用Linux内核提供的文件系统 /sys/bus/pci 来扫描PCI总线上的内容。内核模块uio_pci_generic提供了/dev/uioX设备文件及/sys下对应的资源文件用于访问PCI设备。DPDK特有的igb_uio模块也提供了相同的功能用于PCI设备的访问。这两个驱动模块都用到了Linux内核提供的uio特性。

逻辑核及共享变量

Note: 逻辑核就是处理器的逻辑单元，有时候也称为硬件 线程。

共享变量是默认的做法。每逻辑核变量的实现则是通过线程局部存储技术TLS来实现的，它提供了每个线程本地存储的功能。

日志

EAL提供了日志信息接口。 默认的，在linux 应用程序中，日志信息被发送到syslog和concole中。 当然，用户可以通过使用不同的日志机制来代替DPDK的日志功能。

跟踪与调试功能

Glibc中提供了一些调试函数用于打印堆栈信息。 Rte_panic函数可以产生一个SIG_ABORT信号，这个信号可以触发产生core文件，可以通过gdb来加载调试。

CPU 特性识别

EAL可以在运行时查询CPU状态（使用rte_cpu_get_feature()接口），用于决定哪个CPU可用。

用户空间中断事件

- 主线程中的用户空间中断和警告处理

EAL创建一个主线程用于轮询UIO设备描述文件以检测中断。 可以通过EAL提供的函数为特定的中断事件注册/解注册回掉函数，回掉函数在主线程中被异步调用。 EAL同时也允许像NIC中断那样定时调用回掉函数。

Note: 在DPDK的PMD中，主线程只对连接状态改变的中断处理，例如网卡的打开和关闭。

- RX 中断事件

PMD提供的报文收发程序并不只限制于自身轮询下执行。 为了缓解小吞吐量下轮询模式对CPU资源的浪费，暂停轮询并等待唤醒事件发生时一种有效的手段。 收包中断是这种场景的一种很好的选择，但也不是唯一的。

EAL提供了事件驱动模式相关的API。以Linux APP为例，其实现依赖于epoll技术。 每个线程可以监控一个epoll实例，而在实例中可以添加所有需要的wake-up事件文件描述符。 事件文件描述符创建并根据UIO/VFIO的说明来映射到制定的中断向量上。 对于BSD APP，可以使用kqueue来代替，但是目前尚未实现。

EAL初始化中断向量和事件文件描述符之间的映射关系，同时每个设备初始化中断向量和队列之间的映射关系， 这样，EAL实际上并不知道在指定向量上发生的中断，由设备驱动负责执行后面的映射。

Note: 每个RX中断事件队列只支持VFIO模式，VFIO支持多个MSI-X向量。 在UIO中，收包中断和其他中断共享中断向量，因此，当RX中断和LSC（连接状态改变）中断同时发生时，只有前者生效。

RX中断由API（rte_eth_dev_rx_intr_*）来实现控制、使能、关闭。当PMD不支持时，这些API返回失败。Intr_conf.rxq标识用于打开每个设备的RX中断。

黑名单

EAL PCI设备的黑名单功能是用于标识制定NIC端口，以便DPDK忽略该端口。 可以使用PCIe设备地址描述符（Domain:Bus:Device:Function）将对应端口标记为黑名单。

Misc功能

包括锁和原子操作（i686和x86-64架构）。

4.3.2 内存段和内存区间

物理内存映射就是通过EAL的这个特性实现的。 物理内存块之间可能是不连续的，所有的内存通过一个内存描述符表进行管理，且每个描述符指向一块连续的物理内存。

基于此，内存区块分配器的作用就是保证分配到一块连续的物理内存。 这些区块被分配出来时会用一个唯一的名字来标识。

Rte_memzone描述符也在配置结构体中，可以通过rte_eal_get_configuration()接口来获取。 通过名字访问一个内存区块会返回对应内存区块的描述符。

内存分配可以从指定开始地址和对齐方式来分配（默认是cache line大小对齐），对齐一般是以2的次幂来的，并且不小于64字节对齐。 内存区可以是2M或是1G大小的内存页，这两者系统都支持。

4.3.3 多线程

DPDK通常制定在core上跑线程以避免任务在核上切换的开销。 这有利于性能的提升，但不总是有效的，并且缺乏灵活性。

电源管理通过限制CPU的运行频率来提升CPU的工作效率。 当然，我们也可以通过充分利用CPU的空闲周期来提升效率。

通过使用cgroup技术，CPU的使用量可以很方便的分配，这也提供了新的方法来提升CPU性能， 但是这里有个前提，DPDK必须处理每个核多线程的上下文切换。

想要更多的灵活性，就要设置线程的CPU亲和性是对CPU集合而不是CPU了。

EAL pthread and lcore Affinity

The term "lcore" refers to an EAL thread, which is really a Linux/FreeBSD pthread. "EAL pthreads" are created and managed by EAL and execute the tasks issued by *remote_launch*. In each EAL pthread, there is a TLS (Thread Local Storage) called *_lcore_id* for unique identification. As EAL pthreads usually bind 1:1 to the physical CPU, the *_lcore_id* is typically equal to the CPU ID.

When using multiple pthreads, however, the binding is no longer always 1:1 between an EAL pthread and a specified physical CPU. The EAL pthread may have affinity to a CPU set, and as such the *_lcore_id* will not be the same as the CPU ID. For this reason, there is an EAL long option '–lcores' defined to assign the CPU affinity of lcores. For a specified lcore ID or ID group, the option allows setting the CPU set for that EAL pthread.

The format pattern: –lcores='<lcore_set>[@cpu_set][,<lcore_set>[@cpu_set],...]'

'lcore_set' and 'cpu_set' can be a single number, range or a group.

A number is a "digit([0-9]+)"; a range is "<number>-<number>"; a group is "(<number|range>[,<number|range>,...])".

If a '@cpu_set' value is not supplied, the value of 'cpu_set' will default to the value of 'lcore_set'.

```
For example, "--lcores='1,2@(5-7),(3-5)@(0,2),(0,6),7-8'" which means start␣
↪9 EAL thread;
    lcore 0 runs on cpuset 0x41 (cpu 0,6);
    lcore 1 runs on cpuset 0x2 (cpu 1);
    lcore 2 runs on cpuset 0xe0 (cpu 5,6,7);
```

```
lcore 3,4,5 runs on cpuset 0x5 (cpu 0,2);
lcore 6 runs on cpuset 0x41 (cpu 0,6);
lcore 7 runs on cpuset 0x80 (cpu 7);
lcore 8 runs on cpuset 0x100 (cpu 8).
```

Using this option, for each given lcore ID, the associated CPUs can be assigned. It's also compatible with the pattern of corelist('-l') option.

非EAL的线程支持

可以在任何用户线程（non-EAL线程）上执行DPDK任务上下文。 在non-EAL线程中，*_lcore_id* 始终是LCORE_ID_ANY，它标识一个no-EAL线程的有效、唯一的 *_lcore_id*。 一些库可能会使用一个唯一的ID替代，一些库将不受影响，有些库虽然能工作，但是会受到限制（如定时器和内存池库）。

所有这些影响将在 已知问题 章节中提到。

公共线程API

DPDK为线程操作引入了两个公共API `rte_thread_set_affinity()` 和 `rte_pthread_get_affinity()`。 当他们在任何线程上下文中调用时，将获取或设置线程本地存储(TLS)。

这些TLS包括 *_cpuset* 和 *_socket_id*:

- *_cpuset* 存储了与线程相关联的CPU位图。
- *_socket_id* 存储了CPU set所在的NUMA节点。如果CPU set中的cpu属于不同的NUMA节点, *_socket_id* 将设置为SOCKET_ID_ANY。

已知问题

- rte_mempool

  rte_mempool在mempool中使用per-lcore缓存。 对于non-EAL线程，`rte_lcore_id()` 无法返回一个合法的值。 因此，当rte_mempool与non-EAL线程一起使用时，put/get操作将绕过默认的mempool缓存，这个旁路操作将造成性能损失。 结合 `rte_mempool_generic_put()` 和 `rte_mempool_generic_get()` 可以在non-EAL线程中使用用户拥有的外部缓存。

- rte_ring

  rte_ring支持多生产者入队和多消费者出队操作。 然而，这是非抢占的，这使得rte_mempool操作都是非抢占的。

  Note: "非抢占" 意味着:

  - 在给定的ring上做入队操作的pthread不能被另一个在同一个ring上做入队的pthread抢占
  - 在给定ring上做出对操作的pthread不能被另一个在同一ring上做出队的pthread抢占

  绕过此约束则可能造成第二个进程自旋等待，知道第一个进程再次被调度为止。 此外，如果第一个线程被优先级较高的上下文抢占，甚至可能造成死锁。

  这并不意味着不能使用它，简单讲，当同一个core上的多线程使用时，需要缩小这种情况。

  1. 它可以用于任一单一生产者或者单一消费者的情况。

2. 它可以由多生产者/多消费者使用，要求调度策略都是SCHED_OTHER(cfs)。用户需要预先了解性能损失。
3. 它不能被调度策略是SCHED_FIFO 或 SCHED_RR的多生产者/多消费者使用。

- rte_timer

  不允许在non-EAL线程上运行 `rte_timer_manager()`。但是，允许在non-EAL线程上重置/停止定时器。

- rte_log

  在non-EAL线程上，没有per thread loglevel和logtype，但是global loglevels可以使用。

- misc

  在non-EAL线程上不支持rte_ring, rte_mempool 和rte_timer的调试统计信息。

cgroup控制

以下是cgroup控件使用的简单示例，在同一个核心($CPU)上两个线程(t0 and t1)执行数据包I/O。 我们期望只有50%的CPU消耗在数据包IO操作上。

```
mkdir /sys/fs/cgroup/cpu/pkt_io
mkdir /sys/fs/cgroup/cpuset/pkt_io

echo $cpu > /sys/fs/cgroup/cpuset/cpuset.cpus

echo $t0 > /sys/fs/cgroup/cpu/pkt_io/tasks
echo $t0 > /sys/fs/cgroup/cpuset/pkt_io/tasks

echo $t1 > /sys/fs/cgroup/cpu/pkt_io/tasks
echo $t1 > /sys/fs/cgroup/cpuset/pkt_io/tasks

cd /sys/fs/cgroup/cpu/pkt_io
echo 100000 > pkt_io/cpu.cfs_period_us
echo  50000 > pkt_io/cpu.cfs_quota_us
```

4.3.4 内存申请

EAL提供了一个malloc API用于申请任意大小内存。

这个API的目的是提供类似malloc的功能，以允许从hugepage中分配内存并方便应用程序移植。 *DPDK API*参考手册 介绍了可用的功能。

通常，这些类型的分配不应该在数据面处理中进行，因为他们比基于池的分配慢，并且在分配和释放路径中使用了锁操作。 但是，他们可以在配置代码中使用。

更多信息请参阅 *DPDK API*参考手册 中rte_malloc()函数描述。

Cookies

当 CONFIG_RTE_MALLOC_DEBUG 开启时，分配的内存包括保护字段，这个字段用于帮助识别缓冲区溢出。

对齐和NUMA约束

接口rte_malloc()传入一个对齐参数，该参数用于请求在该值的倍数上对齐的内存区域(这个值必须是2的幂)。

在支持NUMA的系统上，对rte_malloc()接口调用将返回在调用函数的core所在的插槽上分配的内存。DPDK还提供了另一组API，以允许在NUMA插槽上直接显式分配内存，或者分配另一个NUAM插槽上的内存。

用例

这个API旨在由初始化时需要类似malloc功能的应用程序调用。

为了在运行时分配/释放数据，在应用程序的快速路径中，应该使用内存池库。

内部实现

数据结构

Malloc库中内部使用两种数据结构类型：

- struct malloc_heap - 用于在每个插槽上跟踪可用内存空间
- struct malloc_elem - 库内部分配和释放空间跟踪的基本要素

Structure: malloc_heap

数据结构malloc_heap用于管理每个插槽上的可用内存空间。在内部，每个NUMA节点有一个堆结构，这允许我们根据此线程运行的NUMA节点为线程分配内存。虽然这并不能保证在NUMA节点上使用内存，但是它并不比内存总是在固定或随机节点上的方案更糟。

堆结构及其关键字段和功能描述如下：

- lock - 需要锁来同步对堆的访问。假定使用链表来跟踪堆中的可用空间，我们需要一个锁来防止多个线程同时处理该链表。
- free_head - 指向这个malloc堆的空闲结点链表中的第一个元素

Note: 数据结构malloc_heap并不会跟踪使用的内存块，因为除了要再次释放他们之外，他们不会被接触，需要释放时，将指向块的指针作为参数传给fres函数。

Fig. 4.3: Malloc库中malloc heap 和 malloc elements。

Structure: malloc_elem

数据结构malloc_elem用作各种内存块的通用头结构。它以三种不同的方式使用，如上图所示：

1. 作为一个释放/申请内存的头部 – 正常使用
2. 作为内存块内部填充头
3. 作为内存结尾标记

结构中重要的字段和使用方法如下所述:

Note: 如果一个字段没有上述三个用法之一的用处，则可以假设对应字段在该情况下具有未定义的值。 例如，对于填充头，只有 “state” 和 “pad”字段具有有效的值。

- heap - 这个指针指向了该内存块从哪个堆申请。 它被用于正常的内存块，当他们被释放时，将新释放的块添加到堆的空闲列表中。
- prev - 这个指针用于指向紧跟这当前memseg的头元素。当释放一个内存块时，该指针用于引用上一个内存块，检查上一个块是否也是空闲。 如果空闲，则将两个空闲块合并成一个大块。
- next_free - 这个指针用于将空闲块列表连接在一起。 它用于正常的内存块，在 `malloc()` 接口中用于找到一个合适的空闲块申请出来，在 `free()` 函数中用于将内存块添加到空闲链表。
- state - 该字段可以有三个可能值：`FREE`, `BUSY` 或 `PAD`。 前两个是指示正常内存块的分配状态，后者用于指示元素结构是在块开始填充结束时的虚拟结构，即，由于对齐限制，块内的数据开始的地方不在块本身的开始处。

  在这种情况下，pad头用于定位块的实际malloc元素头。

  对于结尾的结构，这个字段总是 `BUSY` ，它确保没有元素在释放之后搜索超过 memseg的结尾以供其它块合并到更大的空闲块。
- pad - 这个字段为块开始处的填充长度。 在正常块头部情况下，它被添加到头结构的结尾，以给出数据区的开始地址，即在malloc上传回的地址。 在填充虚拟头部时，存储相同的值，并从虚拟头部的地址中减去实际块头部的地址。
- size - 数据块的大小，包括头部本身。 对于结尾结构，这个大小需要指定为0，虽然从未使用。 对于正在释放的正常内存块，使用此大小值替代 “next” 指针，以标识下一个块的存储位置，在 `FREE` 情况下，可以合并两个空闲块。

申请内存

On EAL initialization, all memsegs are setup as part of the malloc heap. This setup involves placing a dummy structure at the end with `BUSY` state, which may contain a sentinel value if `CONFIG_RTE_MALLOC_DEBUG` is enabled, and a proper *element header* with `FREE` at the start for each memseg. ‘‘FREE‘‘元素被添加到malloc堆的空闲列表中。

当应用程序调用类似malloc功能的函数时，malloc函数将首先为调用线程索引 `lcore_config` 结构，并确定该线程的NUMA节点。 NUMA节点将作为参数传给 `heap_alloc()``函数，用于索引``malloc_heap` 结构数组。参与索引参数还有大小、类型、对齐方式和边界参数。

函数 `heap_alloc()` 将扫描堆的空闲链表，尝试找到一个适用于所请求的大小、对齐方式和边界约束的内存块。

当已经识别出合适的空闲元素时，将计算要返回给用户的指针。 紧跟在该指针之前的内存的高速缓存行填充了一个malloc_elem头部。 由于对齐和边界约束，在元素的开头和结尾可能会有空闲的空间，这将导致已下行为:

1. 检查尾随空间。 如果尾部空间足够大，例如 > 128 字节，那么空闲元素将被分割。 否则，仅仅忽略它（浪费空间）。
2. 检查元素开始处的空间。 如果起始处的空间很小， <=128 字节，那么使用填充头，这部分空间被浪费。 但是，如果空间很大，那么空闲元素将被分割。

从现有元素的末尾分配内存的优点是不需要调整空闲链表， 空闲链表中现有元素仅调整大小指针，并且后面的元素使用 “prev” 指针重定向到新创建的元素位置。

释放内存

要释放内存，将指向数据区开始的指针传递给free函数。 从该指针中减去 `malloc_elem` 结构的大小，以获得内存块元素头部。 如果这个头部类型是 `PAD`，那么进一步减去pad长度，以获得整个块的正确元素头。

从这个元素头中，我们获得指向块所分配的堆的指针及必须被释放的位置，以及指向前一个元素的指针，并且通过size字段，可以计算下一个元素的指针。 这意味着我们永远不会有两个相邻的 `FREE` 内存块，因为他们总是会被合并成一个大的块。

4.4 Ring 库

环形缓冲区支持队列管理。rte_ring并不是具有无限大小的链表，它具有如下属性:

- 先进先出（FIFO）
- 最大大小固定，指针存储在表中
- 无锁实现
- 多消费者或单消费者出队操作
- 多生产者或单生产者入队操作
- 批量出队 - 如果成功，将指定数量的元素出队，否则什么也不做
- 批量入队 - 如果成功，将指定数量的元素入队，否则什么也不做
- 突发出队 - 如果指定的数目出队失败，则将最大可用数目对象出队
- 突发入队 - 如果指定的数目入队失败，则将最大可入队数目对象入队

相比于链表，这个数据结构的优点如下:

- 更快；只需要一个sizeof(void *)的Compare-And-Swap指令，而不是多个双重比较和交换指令
- 与完全无锁队列像是
- 适应批量入队/出队操作。 因为指针是存储在表中的，应i多个对象的出队将不会产生于链表队列中一样多的cache miss。 此外，批量出队成本并不比单个对象出队高。

缺点:

- 大小固定
- 大量ring相比于链表，消耗更多的内存，空ring至少包含n个指针。

数据结构中存储的生产者和消费者头部和尾部指针显示了一个简化版本的ring。

Fig. 4.4: Ring 结构

4.4.1 FreeBSD* 中 Ring 的实现参考

FreeBSD 8.0中添加了如下代码，并应用到了某些网络设备驱动程序中（至少Interl驱动中应用了）:

- bufring.h in FreeBSD
- bufring.c in FreeBSD

4.4.2 Linux* 中的无锁环形缓冲区

参考链接 Linux Lockless Ring Buffer Design.

4.4.3 附加功能

Name

每个ring都有唯一的名字。 用户不可能创建两个具有相同名称的ring（如果尝试调用rte_ring_create()这样做的话，将返回NULL）。

4.4.4 使用场景

Ring库的使用情况包括：

- DPDK app之间的交互
- 用于内存池申请

4.4.5 Ring Buffer解析

本节介绍ring buffer的运行方式。 Ring结构有两组头尾指针组成，一组被生产者调用，一组被消费者调用。以下将简单称为 prod_head、prod_tail、cons_head 及 cons_tail。

每个图代表了ring的简化状态，是一个循环缓冲器。 本地变量的内容在图上方表示，Ring结构的内容在图下方表示。

单生产者入队

本节介绍了一个生产者向队列添加对象的情况。 在本例中，只有生产者头和尾指针(prod_head and prod_tail)被修改，只有一个生产者。

初始状态是将prod_head 和 prod_tail 指向相同的位置。

入队第一步

首先， *ring->prod_head* 和 *ring->cons_tail**复制到本地变量中。 **prod_next* 本地变量指向下一个元素，或者，如果是批量入队的话，指向下几个元素。

如果ring中没有足够的空间存储元素的话(通过检查cons_tail来确定)，则返回错误。

Fig. 4.5: Enqueue first step

入队第二步

第二步是在环结构中修改 *ring->prod_head*，以指向与prod_next相同的位置。

指向待添加对象的指针被复制到ring中。

Fig. 4.6: Enqueue second step

入队最后一步

一旦将对象添加到ring中，ring结构中的 *ring->prod_tail* 将被修改，指向与 *ring->prod_head* 相同的位置。 入队操作完成。

Fig. 4.7: Enqueue last step

单消费者出队

本节介绍一个消费者从ring中取出对象的情况。 在本例中，只有消费者头尾指针(cons_head and cons_tail)被修改，只有一个消费者。

初始状态是将cons_head 和 cons_tail指向相同位置。

出队第一步

首先，将 *ring->cons_head* 和 *ring->prod_tail**复制到局部变量中。 **cons_next* 本地变量指向表的下一个元素，或者在批量出队的情况下指向下几个元素。

如果ring中没有足够的对象用于出队(通过检查prod_tail)，将返回错误。

Fig. 4.8: Dequeue last step

出队第二步

第二步是修改ring结构中 ring->cons_head，以指向cons_next相同的位置。

指向出队对象(obj1) 的指针被复制到用户指定的指针中。

出队最后一步

最后，ring中的ring->cons_tail被修改为指向ring->cons_head相同的位置。 出队操作完成。

多生产者入队

本节说明两个生产者同时向ring中添加对象的情况。 在本例中，仅修改生产者头尾指针(prod_head and prod_tail)。

初始状态是将prod_head 和 prod_tail 指向相同的位置。

Fig. 4.9: Dequeue second step

Fig. 4.10: Dequeue last step

多生产者入队第一步

在生产者的两个core上， *ring->prod_head* 及 *ring->cons_tail* 都被复制到局部变量。 局部变量prod_next指向下一个元素，或者在批量入队的情况下指向下几个元素。

如果ring中没有足够的空间用于入队(通过检查cons_tail)，将返回错误。

Fig. 4.11: Multiple producer enqueue first step

多生产者入队第二步

第二步是修改ring结构中 ring->prod_head ，来指向prod_next相同的位置。 此操作使用比较和交换(CAS)指令，该指令以原子操作的方式执行以下操作:

- 如果ring->prod_head 与本地变量prod_head不同，则CAS操作失败，代码将在第一步重新启动。
- 否则，ring->prod_head设置为本地变量prod_next，CAS操作成功并继续下一步处理。

在图中，core1执行成功，core2重新启动。

多生产者入队第三步

Core 2的CAS操作成功重试。

Core 1更新一个对象(obj4)到ring上。Core 2更新一个对象(obj5)到ring上

多生产者入队地四步

每个core现在都想更新 ring->prod_tail。 只有ring->prod_tail等于prod_head本地变量，core才能更新它。 当前只有core 1满足，操作在core 1上完成。

多生产者入队最后一步

一旦ring->prod_tail被core 1更新完，core 2也满足条件，允许更新。 Core 2上也完成了操作。

32-bit取模索引

在前面的途中，prod_head, prod_tail, cons_head 和 cons_tail索引由箭头表示。 但是，在实际实现中，这些值不会假定在0和 size(ring)-1 之间。 索引值在 0 ～ 2^32 -1之间，当我们访问ring本身时，我们屏蔽他们的值。32bit模数也意味着如果溢出32bit的范围，对索引的操作将自动执行2^32 模。

以下是两个例子，用于帮助解释索引值如何在ring中使用。

Fig. 4.12: Multiple producer enqueue second step

Fig. 4.13: Multiple producer enqueue third step

Note: 为了简化说明，使用模16bit操作，而不是32bit。 另外，四个索引被定义为16bit无符号整数，与实际情况下的32bit无符号数相反。

这个ring包含11000对象。

这个ring包含12536个对象。

Note: 为了便于理解，我们在上面的例子中使用模65536操作。 在实际执行情况下，这种低效操作是多余的，但是，当溢出时会自动执行。

代码始终保证生产者和消费者之间的距离在0 ～ size(ring)-1之间。 基于这个属性，我们可以对两个索引值做减法，而不用考虑溢出问题

任何情况下，ring中的对象和空闲对象都在 0 ～ size(ring)-1之间，即便第一个减法操作已经溢出：

```
uint32_t entries = (prod_tail - cons_head);
uint32_t free_entries = (mask + cons_tail -prod_head);
```

4.4.6 参考文档

- bufring.h in FreeBSD (version 8)
- bufring.c in FreeBSD (version 8)
- Linux Lockless Ring Buffer Design

4.5 Mempool 库

内存池是固定大小的对象分配器。 在DPDK中，它由名称唯一标识，并且使用mempool操作来存储空闲对象。 默认的mempool操作是基于ring的。它提供了一些可选的服务，如per-core缓存和对齐帮助，以确保对象被填充， 方便将他们均匀扩展到DRAM或DDR3通道上。

这个库由 *Mbuf Library* 使用。

4.5.1 Cookies

在调试模式(CONFIG_RTE_LIBRTE_MEMPOOL_DEBUG is enabled)中，将在块的开头和结尾处添加cookies。 分配的对象包含保护字段，以帮助调试缓冲区溢出。

Fig. 4.14: Multiple producer enqueue fourth step

Fig. 4.15: Multiple producer enqueue last step

Fig. 4.16: Modulo 32-bit indexes - Example 1

4.5.2 Stats

在调试模式(CONFIG_RTE_LIBRTE_MEMPOOL_DEBUG is enabled)中，从池中获取/释放的统计信息存放在mempool结构体中。 统计信息是per-lcore的，避免并发访问统计计数器。

4.5.3 内存对齐约束

根据硬件内存配置，可以通过在对象之间添加特定的填充来大大提高性能。 其目的是确保每个对象开始于不同的通道上，并在内存中排列，以便实现所有通道负载均衡。

特别是当进行L3转发或流分类时，报文缓冲对齐尤为重要。此时仅访问报文的前64B，因此可以通过在不同的信道之间扩展对象的起始地址来提升性能。

DIMM上的rank数目是可访问DIMM完整数据位宽的独立DIMM集合的数量。 由于他们共享相同的路径，因此rank不能被同事访问。 DIMM上的DRAM芯片的物理布局无需与rank数目相关。

当运行app时，EAL命令行选项提供了添加内存通道和rank数目的能力。

Note: 命令行必须始终指定处理器的内存通道数目。

不同DIMM架构的对齐示例如图所示 Fig. 4.18 及 Fig. 4.19 。

在这种情况下，假设吧平稳是64B块就不成立了。

Intel® 5520芯片组有三个通道，因此，在大多数情况下，对象之间不需要填充。(除了大小为n x 3 x 64B的块)

当创建一个新池时，用户可以指定使用此功能。

4.5.4 本地缓存

在CPU使用率方面，由于每个访问需要compare-and-set (CAS)操作，所以多核访问内存池的空闲缓冲区成本比较高。 为了避免对内存池ring的访问请求太多，内存池分配器可以维护per-core cache，并通过实际内存池中具有较少锁定的缓存对内存池ring执行批量请求。 通过这种方式，每个core都可以访问自己空闲对象的缓存（带锁）， 只有当缓存填充时，内核才需要将某些空闲对象重新放回到缓冲池ring，或者当缓存空时，从缓冲池中获取更多对象。

虽然这意味着一些buffer可能在某些core的缓存上处于空闲状态，但是core可以无锁访问其自己的缓存提供了性能上的提升。

缓存由一个小型的per-core表及其长度组成。可以在创建池时启用/禁用此缓存。

缓存大小的最大值是静态配置，并在编译时定义的(CONFIG_RTE_MEMPOOL_CACHE_MAX_SIZE)。

Fig. 4.20 显示了一个缓存操作。

Fig. 4.17: Modulo 32-bit indexes - Example 2

Fig. 4.18: Two Channels and Quad-ranked DIMM Example

Fig. 4.19: Three Channels and Two Dual-ranked DIMM Example

不同于per-lcore内部缓存，应用程序可以通过接口 `rte_mempool_cache_create()`，`rte_mempool_cache_free()` 和 `rte_mempool_cache_flush()` 创建和管理外部缓存。这些用户拥有的缓存可以被显式传递给 `rte_mempool_generic_put()` 和 `rte_mempool_generic_get()`。接口 `rte_mempool_default_cache()` 返回默认内部缓存。与默认缓存相反，用户拥有的高速缓存可以由非EAL线程使用。

4.5.5 Mempool 句柄

这允许外部存储子系统，如外部硬件存储管理系统和软件存储管理与DPDK一起使用。

mempool 操作包括两方面：

- 添加新的mempool操作代码。这是通过添加mempool ops代码，并使用 `MEMPOOL_REGISTER_OPS` 宏来实现的。
- 使用新的API调用 `rte_mempool_create_empty()` 及 `rte_mempool_set_ops_byname()` 用于创建新的mempool，并制定用户要使用的操作。

在同一个应用程序中可能会使用几个不同的mempool处理。可以使用 `rte_mempool_create_empty()` 创建一个新的mempool，然后用 `rte_mempool_set_ops_byname()` 将mempool指向相关的 mempool处理回调（ops）结构体。

传统的应用程序可能会继续使用旧的 `rte_mempool_create()` API调用，它默认使用基于ring的mempool处理。这些应用程序需要修改为新的mempool处理。

对于使用 `rte_pktmbuf_create()` 的应用程序，有一个配置设置(`RTE_MBUF_DEFAULT_MEMPOOL_OPS`)，允许应用程序使用另一个mempool处理。

4.5.6 用例

需要高性能的所有分配器应该使用内存池实现。以下是一些使用实例：

- *Mbuf Library*
- *Environment Abstraction Layer*
- 任何需要在程序中分配固定大小对象，并将被系统持续使用的应用程序

4.6 Mbuf 库

Mbuf库提供了申请和释放mbufs的功能，DPDK应用程序使用这些buffer存储消息缓冲。消息缓冲存储在mempool中，使用 *Mempool Library* 。

Fig. 4.20: A mempool in Memory with its Associated Ring

数据结构rte_mbuf可以承载网络数据包buffer或者通用控制消息buffer(由CTRL_MBUF_FLAG指示)。 也可以扩展到其他类型。 rte_mbuf头部结构尽可能小，目前只使用两个缓存行，最常用的字段位于第一个缓存行中。

4.6.1 Packet Buffer 设计

为了存储数据包数据(报价协议头部), 考虑了两种方法:

1. 在单个存储buffer中嵌入metadata，后面跟着数据包数据固定大小区域
2. 为metadata和报文数据分别使用独立的存储buffer。

第一种方法的优点是他只需要一个操作来分配/释放数据包的整个存储表示。 但是，第二种方法更加灵活，并允许将元数据的分配与报文数据缓冲区的分配完全分离。

DPDK选择了第一种方法。 Metadata包含诸如消息类型，长度，到数据开头的偏移量等控制信息，以及允许缓冲链接的附加mbuf结构指针。

用于承载网络数据包buffer的消息缓冲可以处理需要多个缓冲区来保存完整数据包的情况。 许多通过下一个字段链接在一起的mbuf组成的jumbo帧，就是这种情况。

对于新分配的mbuf，数据开始的区域是buffer之后 RTE_PKTMBUF_HEADROOM 字节的位置，这是缓存对齐的。 Message buffers可以在系统中的不同实体中携带控制信息，报文，事件等。 Message buffers也可以使用起buffer指针来指向其他消息缓冲的数据字段或其他数据结构。

Fig. 4.21 and Fig. 4.22 显示了其中个一些场景。

Fig. 4.21: An mbuf with One Segment

Fig. 4.22: An mbuf with Three Segments

Buffer Manager实现了一组相当标准的buffer访问操作来操纵网络数据包。

4.6.2 存储在Mempool中的Buffer

Buffer Manager 使用 *Mempool Library* 来申请buffer。 因此确保了数据包头部均衡分布到信道上并进行L3处理。 mbuf中包含一个字段，用于表示它从哪个池中申请出来。 当调用 rte_ctrlmbuf_free(m) 或 rte_pktmbuf_free(m)，mbuf被释放到原来的池中。

4.6.3 构造函数

Packet 及 control mbuf构造函数由API提供。 接口rte_pktmbuf_init() 及 rte_ctrlmbuf_init() 初始化mbuf结构中的某些字段，这些字段一旦创建将不会被用户修改（如mbuf类型、源池、缓冲区起始地址等）。 此函数在池创建时作为rte_mempool_create()函数的回掉函数给出。

4.6.4 申请及释放 mbufs

分配一个新mbuf需要用户指定从哪个池中申请。 对于任意新分配的mbuf，它包含一个段，长度为0。 缓冲区到数据的偏移量被初始化，以便使得buffer具有一些字节（RTE_PKTMBUF_HEADROOM）的headroom。

释放mbuf意味着将其返回到原始的mempool。 当mbuf的内容存储在一个池中（作为一个空闲的mbuf）时，mbuf的内容不会被修改。 由构造函数初始化的字段不需要在mbuf分配时重新初始化。

当释放包含多个段的数据包mbuf时，他们都被释放，并返回到原始mempool。

4.6.5 操作 mbufs

这个库提供了一些操作数据包mbuf中的数据的功能。 例如:

- 获取数据长度
- 获取指向数据开始位置的指针
- 数据前插入数据
- 数据之后添加数据
- 删除缓冲区开头的数据(rte_pktmbuf_adj())
- 删除缓冲区末尾的数据(rte_pktmbuf_trim()) 详细信息请参阅 *DPDK API Reference*

4.6.6 元数据信息

部分信息由网络驱动程序检索并存储在mbuf中使得处理更简单。 例如，VLAN、RSS哈希结果(参见 *Poll Mode Driver*)及校验和由硬件计算的标志等。

mbuf中还包含数据源端口和报文链中mbuf数目。 对于链接的mbuf，只有链的第一个mbuf存储这个元信息。

例如，对于IEEE1588数据包，RX侧就是这种情况，时间戳机制，VLAN标记和IP校验和计算。 在TX端，应用程序还可以将一些处理委托给硬件。 例如，PKT_TX_IP_CKSUM标志允许卸载IPv4校验和的计算。

以下示例说明如何在vxlan封装的tcp数据包上配置不同的TX卸载：`out_eth/out_ip/out_udp/vxlan/in_eth/in_ip/in_tcp/payload`

- 计算out_ip的校验和:

```
mb->l2_len = len(out_eth)
mb->l3_len = len(out_ip)
mb->ol_flags |= PKT_TX_IPV4 | PKT_TX_IP_CSUM
set out_ip checksum to 0 in the packet
```

  配置DEV_TX_OFFLOAD_IPV4_CKSUM支持在硬件计算。

- 计算out_ip 和 out_udp的校验和:

```
mb->l2_len = len(out_eth)
mb->l3_len = len(out_ip)
mb->ol_flags |= PKT_TX_IPV4 | PKT_TX_IP_CSUM | PKT_TX_UDP_CKSUM
set out_ip checksum to 0 in the packet
set out_udp checksum to pseudo header using rte_ipv4_phdr_cksum()
```

  配置DEV_TX_OFFLOAD_IPV4_CKSUM 和 DEV_TX_OFFLOAD_UDP_CKSUM支持在硬件上计算。

- 计算in_ip的校验和:

```
mb->l2_len = len(out_eth + out_ip + out_udp + vxlan + in_eth)
mb->l3_len = len(in_ip)
mb->ol_flags |= PKT_TX_IPV4 | PKT_TX_IP_CSUM
set in_ip checksum to 0 in the packet
```

这以情况1类似，但是l2_len不同。 配置DEV_TX_OFFLOAD_IPV4_CKSUM支持硬件计算。 注意，只有外部L4校验和为0时才可以工作。

- 计算in_ip 和 in_tcp的校验和:

```
mb->l2_len = len(out_eth + out_ip + out_udp + vxlan + in_eth)
mb->l3_len = len(in_ip)
mb->ol_flags |= PKT_TX_IPV4 | PKT_TX_IP_CSUM | PKT_TX_TCP_CKSUM
在报文中设置in_ip校验和为0
使用rte_ipv4_phdr_cksum()将in_tcp校验和设置为伪头
```

这与情况2类似，但是l2_len不同。 配置DEV_TX_OFFLOAD_IPV4_CKSUM 和 DEV_TX_OFFLOAD_TCP_CKSUM支持硬件实现。 注意，只有外部L4校验和为0才能工作。

- segment inner TCP:

```
mb->l2_len = len(out_eth + out_ip + out_udp + vxlan + in_eth)
mb->l3_len = len(in_ip)
mb->l4_len = len(in_tcp)
mb->ol_flags |= PKT_TX_IPV4 | PKT_TX_IP_CKSUM | PKT_TX_TCP_CKSUM | PKT_TX_TCP_SEG;
在报文中设置in_ip校验和为0
将in_tcp校验和设置为伪头部，而不使用IP载荷长度
```

配置DEV_TX_OFFLOAD_TCP_TSO支持硬件实现。 注意，只有L4校验和为0时才能工作。

- 计算out_ip, in_ip, in_tcp的校验和:

```
mb->outer_l2_len = len(out_eth)
mb->outer_l3_len = len(out_ip)
mb->l2_len = len(out_udp + vxlan + in_eth)
mb->l3_len = len(in_ip)
mb->ol_flags |= PKT_TX_OUTER_IPV4 | PKT_TX_OUTER_IP_CKSUM  | PKT_TX_IP_CKSUM | ␣
↪PKT_TX_TCP_CKSUM;
设置 out_ip 校验和为0
设置 in_ip 校验和为0
使用rte_ipv4_phdr_cksum()设置in_tcp校验和为伪头部
```

配置DEV_TX_OFFLOAD_IPV4_CKSUM, DEV_TX_OFFLOAD_UDP_CKSUM 和 DEV_TX_OFFLOAD_OUTER_IPV4_CKSUM支持硬件实现。

Flage标记的意义在mbuf API文档(rte_mbuf.h)中有详细描述。 更多详细信息还可以参阅testpmd 源码(特别是csumonly.c)。

4.6.7 直接及间接 Buffers

直接缓冲区是指缓冲区完全独立。 间接缓冲区的行为类似于直接缓冲区，但缓冲区的指针和数据便宜量指的是另一个直接缓冲区的数据。 这在数据包需要复制或分段的情况下是很有用的，因为间接缓冲区提供跨越多个缓冲区重用相同数据包数据的手段。

当使用接口 `rte_pktmbuf_attach()` 函数将缓冲区附加到直接缓冲区时，该缓冲区变成间接缓冲区。 每个缓冲区有一个引用计数器字段，每当直接缓冲区附加一个间接缓冲区时，直接缓冲区上的应用计数器递增。 类似的，每当间接缓冲区被分裂时，直接缓冲区上的引用计数器递减。 如果生成的引用计数器为0，则直接缓冲区将被释放，因为它不再使用。

处理间接缓冲区时需要注意几件事情。 首先，间接缓冲区从不附加到另一个间接缓冲区。 尝试将缓冲区A附加到间接缓冲区B（且B附加到C上了），将使得rte_pktmbuf_attach() 自动将A附加到C上。 其次，为了使缓冲区变成间接缓冲区，其引用计数必须等于1，也就是说它不能被另一个间接缓冲区引用。 最后，不可能将间接缓冲区重新链接到直接缓冲区（除非它已经被分离了）。

虽然可以使用推荐的rte_pktmbuf_attach（）和rte_pktmbuf_detach（）函数直接调用附加/分离操作， 但建议使用更高级的rte_pktmbuf_clone（）函数，该函数负责间接缓冲区的正确初始化，并可以克隆具有多个段的缓冲区。

由于间接缓冲区不应该实际保存任何数据，间接缓冲区的内存池应配置为指示减少的内存消耗。 可以在几个示例应用程序中找到用于间接缓冲区的内存池（以及间接缓冲区的用例示例）的初始化示例，例如IPv4组播示例应用程序。

4.6.8 调试

在调试模式 (CONFIG_RTE_MBUF_DEBUG使能)下，mbuf库的功能在任何操作之前执行完整性检查(如缓冲区检查、类型错误等)。

4.6.9 用例

所有网络应用程序都应该使用mbufs来传输网络数据包。

4.7 轮询模式驱动

DPDK包括Gigabit、10Gigabit 及 40Gigabit 和半虚拟化IO的轮询模式驱动程序。

轮询模式驱动程序(PMD)由通过在用户空间中运行的BSD驱动提供的API组成，以配置设备及各自的队列。

此外，PMD直接访问 RX 和 TX 描述符，且不会有任何中断（链路状态更改中断除外）产生，这可以保证在用户空间应用程序中快速接收，处理和传送数据包。 本节介绍PMD的要求、设计原则和高级架构，并介绍了以太网PMD的对外通用API。

4.7.1 要求及假设条件

DPDK环境支持两种模式的数据包处理，RTC和pipeline：

- 在 *run-to-completion* 模式中，通过调用API来轮询指定端口的RX描述符以获取报文。 紧接着，在同一个core上处理报文，并通过API调用将报文放到接口的TX描述符中以发送报文。
- 在 *pipe-line* 模式中，一个core轮询一个或多个接口的RX描述符以获取报文。然后报文经由ring被其他core处理。 其他core可以继续处理报文，最终报文被放到TX描述符中以发送出去。

在同步 run-to-completion 模式中，每个逻辑和处理数据包的流程包括以下步骤：

- 通过PMD报文接收API来获取报文
- 一次性处理每个数据报文，直到转发阶段
- 通过PMD发包API将报文发送出去

相反地，在异步的pipline模式中，一些逻辑核可能专门用于接收报文，其他逻辑核用于处理前面收到的报文。 收到的数据包通过报文ring在逻辑核之间交换。 数据包收包过程包括以下步骤：

- 通过PMD收包API获取报文
- 通过数据包队列想逻辑核提供接收到的数据包

数据包处理过程包括以下步骤：

- 从数据包队列中获取数据包
- 处理接收到的数据包，直到重新发送出去

为了避免任何不必要的中断处理开销，执行环境不得使用任何异步通知机制。即便有需要，也应该尽量使用ring来引入通知信息。

在多核环境中避免锁竞争是一个关键问题。 为了解决这个问题，PMD旨在尽可能地使用每个core的私有资源。 例如，PMD每个端口维护每个core单独的传输队列。 同样的，端口的每个接收队列都被分配给单个逻辑核并由其轮询。

为了适用NUMA架构，内存管理旨在为每个逻辑核分配本地（相同插槽）中的专用缓冲池，以最大限度地减少远程内存访问。 数据包缓冲池的配置应该考虑到DIMMs、channels 和 ranks等底层物理内存架构。 应用程序必须确保在内存池创建时给出合适的参数。具体内容参阅 *Mempool Library* 。

4.7.2 设计原则

Ethernet* PMDs的API和架构设计遵考虑到以下原则。

PMDs 必须能够帮助上层的应用实现全局的策略。 反之，不能阻止或妨碍上层应用的实施。

例如，PMD的发送和接收函数都有大量的报文或描述符需要轮询。 这运行RTC处理协议栈通过不同的全局循环策略静态修护或动态调整其行为如:

- 立即接收，处理并以零碎的方式一次传送数据包。
- 尽可能所的接收数据包，然后处理所有数据包，再发送。
- 接收给定的最大量的数据包，处理接收的数据包，累加，最后将累加的数据包发送出去。

为了实现最优性能，需要考考整体软件设计选择和纯软件优化技术，并与可用的低层次硬件优化功能（如CPU缓存属性、总线速度、NIC PCI带宽等）进行考虑和平衡。 报文传输的情况就是突发性网络报文处理是软硬件权衡问题的一个例子。 在初始情况下，PMD只能导出一个 rte_eth_tx_one 函数，以便在给定的队列上一次传输一个数据包。 最重要的是，可以轻松构建一个 rte_eth_tx_burst 函数，循环调用 rte_eth_tx_one 函数以便一次传输多个数据包。 然而，PMD有效地实现了 rte_eth_tx_burst 函数，以通过以下优化来最小化每个数据包的驱动级传输开销:

- 在多个数据包之间共享调用 rte_eth_tx_one 函数的非摊销成本。
- 启用 rte_eth_tx_burst 函数以利用burst-oriented 硬件特性（缓存数据预取、使用NIC头/尾寄存器）以最小化每个数据包的CPU周期数， 例如，通过避免对环形缓传输描述符的不必要的读取寄存器访问，或通过系统地使用精确匹配告诉缓存行边界大小的指针数组。
- 使用burst-oriented软件优化技术来移除失败的操作结果，如ring索引的回滚。

还通过API引入了Burst-oriented函数，这些函数在PMD服务中密集使用。 这些函数特别适用于NIC ring的缓冲区分配器，他们提供一次分配/释放多个缓冲区的功能。 例如，一个 mbuf_multiple_alloc 函数返回一个指向 rte_mbuf 缓冲区的指针数组，它可以在向ring添加多个描述符来加速PMD的接收轮询功能。

4.7.3 逻辑核、内存及网卡队列的联系

当处理器的逻辑核和接口利用其本地存储时，DPDK提供NUMA支持，以提供更好的性能。 因此，与本地PCIE接口相关的mbuf分配应从本地内存中创建的内存池中申请。 如果可能，缓冲区应该保留在本地处理器上以获取最佳性能，并且应使用从本地内存中分配的mempool中申请的mbuf来填充RX和TX缓冲区描述符。

如果数据包或数据操作在本地内存中，而不是在远程处理器内存上，则RTC模型也会运行得更好。 只要所有使用的逻辑核位于同一个处理器上，pipeline模型也将获得更好的性能。

所个逻辑核不应共享接口的接收或发送队列，因为这将需要全局上锁保护，而导致性能下降。

4.7.4 设备标识及配置

设备标识

每个NIC端口（总线/桥、设备、功能）由其PCI标识符唯一指定。该PCI标识符在DPDK初始化时执行的PCI探测/枚举功能分配。根据PCI标识符，NIC端口被分配了两个其他的表示：

- 一个端口索引，用于在PMD API导出的所有函数中指定NIC端口
- 端口名称，用于在控制消息中指定端口，主要用于管理和调试目的。为了便于使用，端口名称包括端口索引。

设备配置

每个NIC端口的配置包括以下步骤：

- 分配 PCI 资源
- 将硬件复位为公知的默认状态
- 设置PHY和链路
- 初始化统计计数器

PMD API还必须导出函数用于启动/终止端口的全部组播功能，并且可以在混杂模式下设置/取消设置端口。

某些硬件卸载功能必须通过特定的配置参数在端口初始化时单独配置。例如，接收侧缩放（RSS）和数据中心桥接（DCB）功能就是这种情况。

即时配置

所有可以“即时”启动或停止的设备功能（即不停止设备），无需PMD API来导出。

所需要的是设备PCI寄存器的映射地址，以在驱动程序之外使用特殊的函数来配置实现这些功能。

为此，PMD API导出一个函数提供可用于在驱动程序外部设置给定设备功能的设备相关联的所有信息。这些信息包括PCI供应商标识符，PCI设备标识符，PCI设备寄存器的映射地址以及驱动程序的名称。

这种方法的主要优点是可以自由地选择API来启动、配置、停止这些设备功能。

例如，testpmd应用程序中的英特尔®82576千兆以太网控制器和英特尔®82599万兆以太网控制器控制器的IEEE1588功能配置。

可以以相同的方式配置端口的L3 / L4 5-Tuple包过滤功能等其他功能。以太网流控（暂停帧）可以在单个端口上进行配置。有关详细信息，请参阅testpmd源代码。此外，只要数据包mbuf设置正确，就可以为单个数据包启用网卡的L4（UDP / TCP / SCTP）校验和卸载。相关详细信息，请参阅 **‘Hardware Offload‘_** 。

传输队列配置

每个传输队列都独立配置了以下信息：

- 发送环上的描述符数目
- NUMA架构中，用于标识从哪个socket的DMA存储区分配传输环的标识
- 传输队列的 Prefetch, Host 及 Write-Back 阈值寄存器的值
- 传输报文释放的最小阈值。当用于传输数据包的描述符数量超过此阈值时，应检查网络适配器以查看是否有回写描述符。在TX队列配置期间可以传递值0，以指示应使用默认值。tx_free_thresh的默认值为32。这使得PMD不会去检索完成的描述符，直到NIC已经为此队列处理了32个报文。

- RS位最小阈值。在发送描述符中设置报告状态（RS）位之前要使用的最小发送描述符数。请注意，此参数仅适用于Intel 10 GbE网络适配器。 如果从最后一个RS位设置开始使用的描述符数量（直到用于发送数据包的第一个描述符）超过发送RS位阈值（tx_rs_thresh），则RS位被设置在用于发送数据包的最后一个描述符上。简而言之，此参数控制网络适配器将哪些传输描述符写回主机内存。在TX队列配置期间可以传递值为0，以指示应使用默认值。 tx_rs_thresh的默认值为32。这确保在网络适配器回写最近使用的描述符之前至少使用32个描述符。这样可以节省TX描述符回写所产生的上游PCIe *带宽。重要的是注意，当tx_rs_thresh大于1时，应将TX写回阈值（TX wthresh）设置为0。有关更多详细信息，请参阅英特尔®82599万兆以太网控制器数据手册。

对于tx_free_thresh和tx_rs_thresh，必须满足以下约束：

- tx_rs_thresh必须大于0。
- tx_rs_thresh必须小于环的大小减去2。
- tx_rs_thresh必须小于或等于tx_free_thresh。
- tx_free_thresh必须大于0。
- tx_free_thresh必须小于环的大小减去3。
- 为了获得最佳性能，当tx_rs_thresh大于1时，TX wthresh应设置为0。

TX环中的一个描述符用作哨兵以避免硬件竞争条件，因此是最大阈值限制。

Note: 当配置DCB操作时，在端口初始化时，发送队列数和接收队列数必须设置为128。

释放 Tx 缓存

许多驱动程序并没有在数据包传输后立即将mbuf释放回到mempool或本地缓存中。 相反，他们将mbuf留在Tx环中，当需要在Tx环中插入，或者 `tx_rs_thresh` 已经超过时，执行批量释放。

应用程序请求驱动程通过接口 `rte_eth_tx_done_cleanup()` 释放使用的mbuf。、 该API请求驱动程序释放不再使用的mbufs，而不管‘‘tx_rs_thresh‘‘ 是否已被超过。 有两种情况会使得应用程序可能想要立即释放mbuf：

- 当给定的数据包需要发送到多个目标接口（对于第2层洪泛或第3层多播）。 一种方法是复制数据包或者复制需要操作的数据包头部。 另一种方法是发送数据包，然后轮询 `rte_eth_tx_done_cleanup()` 接口直到报文引用递减。 接下来，这个报文就可以发送到下一个目的接口。 该应用程序仍然负责管理不同目标接口之间所需的任何数据包操作，但可以避免数据复制。 该API独立于数据包是传输还是丢弃，只是mbuf不再被接口使用。
- 一些应用程序被设计为进行多次运行，如数据包生成器。 为了运行的性能原因和一致性，应用程序可能希望在每个运行之间重新设置为初始状态，其中所有mbufs都返回到mempool。 在这种情况下，它可以为其已使用的每个目标接口调 `rte_eth_tx_done_cleanup()` API 以请求它释放所有使用的mbuf。

要确定驱动程序是否支持该API，请检查 *Network Interface Controller Drivers* 文档中的 * Free Tx mbuf on demand * 功能。

硬件卸载

根据 `rte_eth_dev_info_get()` 提供的驱动程序功能，PMD可能支持硬件卸载功能，如校验和TCP分段或VLAN插入。

这些卸载功能的支持意味着将专用状态位和值字段添加到rte_mbuf数据结构中，以及由每个PMD导出的接收/发送功能的适当处理。 标记列表及其精确含义在mbuf API文档及 *Mbuf Library* 中 “Meta Information”章节。

4.7.5 PMD API

概要

默认情况下，PMD提供的所有外部函数都是无锁函数，这些函数假定在同一目标设备上不会再不同的逻辑core上并行调用。 例如，PMD接收函数不能再两个逻辑核上并行调用，以轮询相同端口的相同RX队列。当然，这个函数可以由不同的RX队列上的不同逻辑核并行调用。 上级应用程序应该保证强制执行这条规则。

如果需要，多个逻辑核到并行队列的并行访问可以通过专门的在线加锁来显式保护，这些加锁函数是建立在相应的无锁API之上的。

通用报文表示

数据包由数据结构 rte_mbuf 表示，这是一个包含所有必要信息的通用元数据结构。 这些信息包括与硬件特征相对应的字段和状态位，如IP头部和VLAN标签的校验和。

数据结构 rte_mbuf 包括以通用方式表示网络控制器提供的硬件功能对应的字段。 对于输入数据包，rte_mbuf的大部分字段都由PMD来填充，包括接收描述符中的信息。 相反，对于输出数据包，rte_mbuf的大部分字段由PMD发送函数用于初始化发送描述符。

数据结构 mbuf 的更全面的描述，请参阅 *Mbuf Library* 章节。

以太网设备 API

以太网PMD驱动导出的以太网设备API请参阅 *DPDK API Reference* 描述。

扩展的统计 API

扩展的统计API允许每个独立的PMD导出一组唯一的统计信息。 应用程序通过以下两个操作来访问这些统计信息:

- `rte_eth_xstats_get`: 使用扩展统计信息填充 `struct rte_eth_xstat` 数组。
- `rte_eth_xstats_get_names`: 使用扩展统计名称查找信息填充 `struct rte_eth_xstat_name` 数组。

每个 `struct rte_eth_xstat` 包含一个键-值对，每个 `struct rte_eth_xstat_name` 包含一个字符串。 `struct rte_eth_xstat` 查找数组中的每个标识符必须在 `struct rte_eth_xstat_name` 查找数组中有一个对应条目。

在后者中，条目的索引是字符串关联的标识符。 这些标识符以及暴露的扩展统计技术在运行是必须保持不变。请注意，扩展统计信息标识符是驱动程序特定的，因此，对于不同的端口可能不一样。

对于暴露给API的客户端的字符串，存在一个命名方案。 这是为了允许API获取感兴趣的信息。 命名方案使用下划线分割的字符串来表示，如下:

- direction
- detail 1
- detail 2

- detail n
- unit

常规统计示例字符串如下，符合上面的方案：

- `rx_bytes`
- `rx_crc_errors`
- `tx_multicast_packets`

该方案虽然简单，但可以灵活地显示和读取统计字符串中的信息。 以下示例说明了命名方案 `rx_packets` 的使用。 在这个例子中，字符串被分成两个组件。 第一个 `rx` 表示统计信息与NIC的接收端相关联。 第二个 `packets` 表示测量单位是数据包。

一个更为复杂的例子是 `tx_size_128_to_255_packets` 。 在这个例子中， `tx` 表示传输， `size` 是第一个细节， `128` 等表示更多的细节， `packets` 表示这是一个数据包计数器。

元数据中的一些方案补充如下：

- 如果第一部分不符合 `rx` 或 `tx` ，统计计数器与传送或接收不相关。
- 如果第二部分的第一个字母是 `q` 且这个 `q` 后跟一个数字，则这个统计数据是特定队列的一部分。

使用队列号的示例如下： `tx_q7_bytes` 表示此统计信息适用于队列号7，并表示该队列上传输的字节数。

4.8 通用流 API (rte_flow)

4.8.1 概述

此API提供了一种通用的方式来配置硬件以匹配特定的 Ingress 或 Egress 流量，根据用户的任何配置规则更改其操作及查询相关计数器。

所有API带有前缀 *rte_flow* ，在文件 `rte_flow.h` 中定义。

- 可以对报文数据(如协议头部，载荷)及报文属性(如关联的物理端口，虚拟设备ID等)执行匹配。
- 可能的操作包括丢弃流量，将流量转移到特定队列、虚拟/物理设备或端口，执行隧道解封、添加标记等操作。

它比涵盖其功能的传统过滤框架层次更高，以便明确的行为暴露对所有P轮询模式驱动程序来讲相同的单个操作接口。

迁移现有应用程序的几种方法在 *API migration* 中有描述。

4.8.2 流规则

描述

流规则是具有匹配模式的属性和动作列表的组合。 流规则构成了此API的基础。

一个流规则可以具有几个不同的动作(如在将数据重定向到特定队列之前执行计数，封装，解封装等操作)，而不是依靠几个规则来实现这些动作，应用程序操作具体的硬件实现细节来顺序执行。

API提供了基于规则的不同优先级支持，例如，当报文匹配两个规则时，强制先执行特定规则。 然而，对于支持多个优先级的硬件，这一条不能保证。 当支持时，可用优先级的数量通常较低，这也是为什么还可以通过PMDs在软件中实现(如通过重新排序规则可以模拟缺失的优先级)。

为了尽可能保持与硬件无关，默认情况下所有规则都被认为具有相同的优先级，这意味着重叠规则（当数据包被多个过滤器匹配时）之间的顺序是未定义的。

PMD可以在可以被检测到的情况下（例如，如果模式匹配现有过滤器）拒绝在给定优先级下创建重叠规则。

因此，对于给定的优先级，可预测的结果只能通过非重叠规则来实现，在所有协议层上使用完全匹配。

流规则也可以分组，流规则优先级特定于它们所属的组。因此，给定组中的所有流规则在另一组之前或之后进行处理。

根据规则支持多个操作可以在非默认硬件优先级之前内部实现，因此两个功能可能不能同时应用于应用程序。

考虑到允许的模式/动作组合不能提前知道，并且将导致不切实际地大量的暴露能力，提供了从当前设备配置状态验证给定规则的方法。

这样，在启动数据路径之前，应用程序可以检查在初始化时是否支持所需的规则类型。该方法可以随时使用，其唯一要求是应该存在规则所需的资源（例如，应首先配置目标RX队列）。

每个定义的规则与由PMD管理的不透明句柄相关联，应用程序负责维护它。这些句柄可用于查询和规则管理，例如检索计数器或其他数据并销毁它们。

为了避免PMD方面的资源泄漏，在释放相关资源（如队列和端口）之前，应用程序必须显式地销毁句柄。

以下小节覆盖如下内容：

- **属性** (由 `struct rte_flow_attr` 表示): 流规则的属性，例如其方向（Ingress或Egress）和优先级。
- **模式条目** (由 `struct rte_flow_item` 表示): 匹配模式的一部分，匹配特定的数据包数据或流量属性。也可以描述模式本身属性，如反向匹配。
- **匹配条目**: 查找的属性，组合任意的模式。
- **动作** (由 `struct rte_flow_action` 表示): 每当数据包被模式匹配时执行的操作。

属性

属性: 组

流规则可以通过为其分配一个公共的组号来分组。较低的值具有较高的优先级。组0具有最高优先级。

虽然是可选的，但是建议应用程序尽可能将类似的规则分组，以充分利用硬件功能（例如，优化的匹配）并解决限制（例如，给定组中可能允许的单个模式类型）。

请注意，并不保证支持多个组。

属性: 优先级

可以将优先级分配给流规则。像Group一样，较低的值表示较高的优先级，0为最大值。

具有优先级0的Group 8流规则，总是在Group 0优先级8的优先级之后才匹配。

组和优先级是任意的，取决于应用程序，它们不需要是连续的，也不需要从0开始，但是最大数量因设备而异，并且可能受到现有流规则的影响。

如果某个报文在给定的优先级和Group中被几个规则匹配，那么结果是未定义的。 它可以采取任何路径，可能重复，甚至导致不可恢复的错误。

请注意，不保证能支持超过一个优先级。

属性: 流量方向

流量规则可以应用于入站和/或出站流量（Ingress/Egress）。

多个模式条目和操作都是有效的，可以在个方向中使用。但是必须至少指定一个方向。

不推荐对给定规则一次指定两个方向，但在少数情况下可能是有效的（例如共享计数器）。

模式条目

模式条目分成两类：

- 匹配协议头部及报文数据（ANY，RAW，ETH，VLAN，IPV4，IPV6，ICMP，UDP，TCP，SCTP,，VXLAN，MPLS，等）。
- 匹配元数据或影响模式处理（END，VOID，INVERT，PF，VF，PORT等等）。

条目规范结构用于匹配协议字段（或项目属性）中的特定值。文档描述每个条目是否与一个条目及其类型名称相关联。

可以为给定的条目最多设置三个相同类型的结构：

- `spec`: 要匹配的数值（如IPv4地址）。
- `last`: 规格中的相应字段的范围上限。
- `mask`: 应用于spec和last的位掩码（如匹配IPv4地址的前缀）。

使用限制和期望行为：

- 没有 `spec` 就设置 `mask` 或 `last` 是错误的。
- 错误的 `last` 值如0或者等于 `spec` 将被忽略，他们不能产生范围。不支持低于 `spec` 的非0值。
- 设置 `spce` 和可选的 `last` ，而不设置 `mask` 会导致PMD使用该条目定义的默认‘‘ mask‘‘（定义为 `rte_flow_item_{name}_mask` 常量）。不设置他们相当于提供空掩码匹配。
- 不设置他们相当于提供空掩码匹配。
- 掩码是用于 `spec` 和 `last` 的简单位掩码，如果不小心使用，可能会产生意想不到的结果。例如，对于IPv4地址字段，spec提供10.1.2.3，last提供10.3.4.5，掩码为255.255.0.0，有效范围为10.1.0.0～10.3.255.255。

匹配以太网头部的条目示例：

Table 4.1: Ethernet item

Field	Subfield	Value
`spec`	`src`	`00:01:02:03:04`
	`dst`	`00:2a:66:00:01`
	`type`	`0x22aa`
`last`	unspecified	
`mask`	`src`	`00:ff:ff:ff:00`
	`dst`	`00:00:00:00:ff`
	`type`	`0x0000`

无掩码的位可以匹配任意的值(显示为 `?`), 以太头部具有如下的属性匹配信息：

- `src`: `??:01:02:03:??`
- `dst`: `??:??:??:??:01`

- `type: 0x????`

匹配模式

模式是指通过堆叠从底层协议开始匹配条目。这种堆叠限制不适用于可以放在任意位置而不影响其结果的元条目。

模式由最后的条目终结。

例子:

Table 4.2: TCPv4 as L4

Index	Item
0	Ethernet
1	IPv4
2	TCP
3	END

Table 4.3: TCPv6 in VXLAN

Index	Item
0	Ethernet
1	IPv4
2	UDP
3	VXLAN
4	Ethernet
5	IPv6
6	TCP
7	END

Table 4.4: TCPv4 as L4 with meta items

Index	Item
0	VOID
1	Ethernet
2	VOID
3	IPv4
4	TCP
5	VOID
6	VOID
7	END

上面的例子显示了一个元条目，如何实现不影响报文数据匹配结果，只要他们保持堆叠正确。结果匹配与“TCPv4 as L4”条目相同。

Table 4.5: UDPv6 anywhere

Index	Item
0	IPv6
1	UDP
2	END

如果PMD支持，如上述示例（缺少以太网规范），忽略堆栈底部的一个或多个协议层，可以查找数据包中的任何位置。

无论支持的封装（例如VXLAN有效载荷）是否通过模式匹配， It is unspecified whether the payload of supported encapsulations (e.g. VXLAN payload) is matched by such a pattern, which may apply to inner, outer or both packets.

Table 4.6: Invalid, missing L3

Index	Item
0	Ethernet
1	UDP
2	END

The above pattern is invalid due to a missing L3 specification between L2 (Ethernet) and L4 (UDP). Doing so is only allowed at the bottom and at the top of the stack.

Meta item types

They match meta-data or affect pattern processing instead of matching packet data directly, most of them do not need a specification structure. This particularity allows them to be specified anywhere in the stack without causing any side effect.

Item: END

End marker for item lists. Prevents further processing of items, thereby ending the pattern.

- Its numeric value is 0 for convenience.
- PMD support is mandatory.
- `spec`, `last` and `mask` are ignored.

Table 4.7: END

Field	Value
`spec`	ignored
`last`	ignored
`mask`	ignored

Item: VOID

Used as a placeholder for convenience. It is ignored and simply discarded by PMDs.

- PMD support is mandatory.
- spec, last and mask are ignored.

Table 4.8: VOID

Field	Value
spec	ignored
last	ignored
mask	ignored

One usage example for this type is generating rules that share a common prefix quickly without reallocating memory, only by updating item types:

Table 4.9: TCP, UDP or ICMP as L4

Index	Item		
0	Ethernet		
1	IPv4		
2	UDP	VOID	VOID
3	VOID	TCP	VOID
4	VOID	VOID	ICMP
5	END		

Item: INVERT

Inverted matching, i.e. process packets that do not match the pattern.

- spec, last and mask are ignored.

Table 4.10: INVERT

Field	Value
spec	ignored
last	ignored
mask	ignored

Usage example, matching non-TCPv4 packets only:

Table 4.11: Anything but TCPv4

Index	Item
0	INVERT
1	Ethernet
2	IPv4
3	TCP
4	END

Item: PF

Matches packets addressed to the physical function of the device.

If the underlying device function differs from the one that would normally receive the matched traffic, specifying this item prevents it from reaching that device unless the flow rule contains a *Action: PF*. Packets are not duplicated between device instances by default.

- Likely to return an error or never match any traffic if applied to a VF device.
- Can be combined with any number of *Item: VF* to match both PF and VF traffic.
- `spec`, `last` and `mask` must not be set.

Table 4.12: PF

Field	Value
`spec`	unset
`last`	unset
`mask`	unset

Item: VF

Matches packets addressed to a virtual function ID of the device.

If the underlying device function differs from the one that would normally receive the matched traffic, specifying this item prevents it from reaching that device unless the flow rule contains a *Action: VF*. Packets are not duplicated between device instances by default.

- Likely to return an error or never match any traffic if this causes a VF device to match traffic addressed to a different VF.
- Can be specified multiple times to match traffic addressed to several VF IDs.
- Can be combined with a PF item to match both PF and VF traffic.
- Default `mask` matches any VF ID.

Table 4.13: VF

Field	Subfield	Value
`spec`	`id`	destination VF ID
`last`	`id`	upper range value
`mask`	`id`	zeroed to match any VF ID

Item: PORT

Matches packets coming from the specified physical port of the underlying device.

The first PORT item overrides the physical port normally associated with the specified DPDK input port (port_id). This item can be provided several times to match additional physical ports.

Note that physical ports are not necessarily tied to DPDK input ports (port_id) when those are not under DPDK control. Possible values are specific to each device, they are not necessarily indexed from zero and may not be contiguous.

As a device property, the list of allowed values as well as the value associated with a port_id should be retrieved by other means.

- Default `mask` matches any port index.

Table 4.14: PORT

Field	Subfield	Value
`spec`	`index`	physical port index
`last`	`index`	upper range value
`mask`	`index`	zeroed to match any port index

Data matching item types

Most of these are basically protocol header definitions with associated bit-masks. They must be specified (stacked) from lowest to highest protocol layer to form a matching pattern.

The following list is not exhaustive, new protocols will be added in the future.

Item: `ANY`

Matches any protocol in place of the current layer, a single ANY may also stand for several protocol layers.

This is usually specified as the first pattern item when looking for a protocol anywhere in a packet.

- Default `mask` stands for any number of layers.

Table 4.15: ANY

Field	Subfield	Value
`spec`	`num`	number of layers covered
`last`	`num`	upper range value
`mask`	`num`	zeroed to cover any number of layers

Example for VXLAN TCP payload matching regardless of outer L3 (IPv4 or IPv6) and L4 (UDP) both matched by the first ANY specification, and inner L3 (IPv4 or IPv6) matched by the second ANY specification:

Table 4.16: TCP in VXLAN with wildcards

Index	Item	Field	Subfield	Value
0	Ethernet			
1	ANY	`spec`	`num`	2
2	VXLAN			
3	Ethernet			
4	ANY	`spec`	`num`	1
5	TCP			
6	END			

Item: `RAW`

Matches a byte string of a given length at a given offset.

Offset is either absolute (using the start of the packet) or relative to the end of the previous matched item in the stack, in which case negative values are allowed.

If search is enabled, offset is used as the starting point. The search area can be delimited by setting limit to a nonzero value, which is the maximum number of bytes after offset where the pattern may start.

Matching a zero-length pattern is allowed, doing so resets the relative offset for subsequent items.

- This type does not support ranges (`last` field).
- Default `mask` matches all fields exactly.

Table 4.17: RAW

Field	Subfield	Value
`spec`	`relative`	look for pattern after the previous item
	`search`	search pattern from offset (see also `limit`)
	`reserved`	reserved, must be set to zero
	`offset`	absolute or relative offset for `pattern`
	`limit`	search area limit for start of `pattern`
	`length`	`pattern` length
	`pattern`	byte string to look for
`last`	if specified, either all 0 or with the same values as `spec`	
`mask`	bit-mask applied to `spec` values with usual behavior	

Example pattern looking for several strings at various offsets of a UDP payload, using combined RAW items:

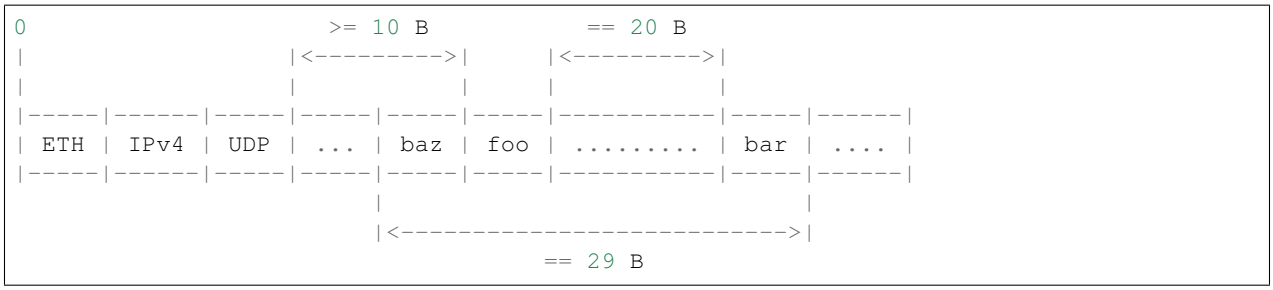
Table 4.18: UDP payload matching

Index	Item	Field	Subfield	Value
0	Ethernet			
1	IPv4			
2	UDP			
3	RAW	`spec`	`relative`	1
			`search`	1
			`offset`	10
			`limit`	0
			`length`	3
			`pattern`	“foo”
4	RAW	`spec`	`relative`	1
			`search`	0
			`offset`	20
			`limit`	0
			`length`	3
			`pattern`	“bar”
5	RAW	`spec`	`relative`	1
			`search`	0
			`offset`	-29
			`limit`	0
			`length`	3
			`pattern`	“baz”
6	END			

This translates to:

- Locate “foo” at least 10 bytes deep inside UDP payload.
- Locate “bar” after “foo” plus 20 bytes.
- Locate “baz” after “bar” minus 29 bytes.

Such a packet may be represented as follows (not to scale):



Note that matching subsequent pattern items would resume after "baz", not "bar" since matching is always performed after the previous item of the stack.

Item: `ETH`

Matches an Ethernet header.

- `dst`: destination MAC.
- `src`: source MAC.
- `type`: EtherType.
- Default `mask` matches destination and source addresses only.

Item: `VLAN`

Matches an 802.1Q/ad VLAN tag.

- `tpid`: tag protocol identifier.
- `tci`: tag control information.
- Default `mask` matches TCI only.

Item: `IPV4`

Matches an IPv4 header.

Note: IPv4 options are handled by dedicated pattern items.

- `hdr`: IPv4 header definition (`rte_ip.h`).
- Default `mask` matches source and destination addresses only.

Item: `IPV6`

Matches an IPv6 header.

Note: IPv6 options are handled by dedicated pattern items.

- `hdr`: IPv6 header definition (`rte_ip.h`).
- Default `mask` matches source and destination addresses only.

Item: ICMP

Matches an ICMP header.

- hdr: ICMP header definition (rte_icmp.h).
- Default mask matches ICMP type and code only.

Item: UDP

Matches a UDP header.

- hdr: UDP header definition (rte_udp.h).
- Default mask matches source and destination ports only.

Item: TCP

Matches a TCP header.

- hdr: TCP header definition (rte_tcp.h).
- Default mask matches source and destination ports only.

Item: SCTP

Matches a SCTP header.

- hdr: SCTP header definition (rte_sctp.h).
- Default mask matches source and destination ports only.

Item: VXLAN

Matches a VXLAN header (RFC 7348).

- flags: normally 0x08 (I flag).
- rsvd0: reserved, normally 0x000000.
- vni: VXLAN network identifier.
- rsvd1: reserved, normally 0x00.
- Default mask matches VNI only.

Item: MPLS

Matches a MPLS header.

- label_tc_s_ttl: label, TC, Bottom of Stack and TTL.
- Default mask matches label only.

Item: GRE

Matches a GRE header.

- `c_rsvd0_ver`: checksum, reserved 0 and version.
- `protocol`: protocol type.
- Default `mask` matches protocol only.

Actions

Each possible action is represented by a type. Some have associated configuration structures. Several actions combined in a list can be affected to a flow rule. That list is not ordered.

They fall in three categories:

- Terminating actions (such as QUEUE, DROP, RSS, PF, VF) that prevent processing matched packets by subsequent flow rules, unless overridden with PASSTHRU.
- Non-terminating actions (PASSTHRU, DUP) that leave matched packets up for additional processing by subsequent flow rules.
- Other non-terminating meta actions that do not affect the fate of packets (END, VOID, MARK, FLAG, COUNT).

When several actions are combined in a flow rule, they should all have different types (e.g. dropping a packet twice is not possible).

Only the last action of a given type is taken into account. PMDs still perform error checking on the entire list.

Like matching patterns, action lists are terminated by END items.

Note that PASSTHRU is the only action able to override a terminating rule.

Example of action that redirects packets to queue index 10:

Table 4.19: Queue action

Field	Value
`index`	10

Action lists examples, their order is not significant, applications must consider all actions to be performed simultaneously:

Table 4.20: Count and drop

Index	Action
0	COUNT
1	DROP
2	END

Table 4.21: Mark, count and redirect

Index	Action	Field	Value
0	MARK	`mark`	0x2a
1	COUNT		
2	QUEUE	`queue`	10
3	END		

Table 4.22: Redirect to queue 5

Index	Action	Field	Value
0	DROP		
1	QUEUE	`queue`	5
2	END		

In the above example, considering both actions are performed simultaneously, the end result is that only QUEUE has any effect.

Table 4.23: Redirect to queue 3

Index	Action	Field	Value
0	QUEUE	`queue`	5
1	VOID		
2	QUEUE	`queue`	3
3	END		

As previously described, only the last action of a given type found in the list is taken into account. The above example also shows that VOID is ignored.

Action types

Common action types are described in this section. Like pattern item types, this list is not exhaustive as new actions will be added in the future.

Action: `END`

End marker for action lists. Prevents further processing of actions, thereby ending the list.

- Its numeric value is 0 for convenience.
- PMD support is mandatory.
- No configurable properties.

Table 4.24: END

Field
no properties

Action: VOID

Used as a placeholder for convenience. It is ignored and simply discarded by PMDs.

- PMD support is mandatory.
- No configurable properties.

Table 4.25: VOID

Field
no properties

Action: PASSTHRU

Leaves packets up for additional processing by subsequent flow rules. This is the default when a rule does not contain a terminating action, but can be specified to force a rule to become non-terminating.

- No configurable properties.

Table 4.26: PASSTHRU

Field
no properties

Example to copy a packet to a queue and continue processing by subsequent flow rules:

Table 4.27: Copy to queue 8

Index	Action	Field	Value
0	PASSTHRU		
1	QUEUE	`queue`	8
2	END		

Action: MARK

Attaches an integer value to packets and sets `PKT_RX_FDIR` and `PKT_RX_FDIR_ID` mbuf flags.

This value is arbitrary and application-defined. Maximum allowed value depends on the underlying implementation. It is returned in the `hash.fdir.hi` mbuf field.

Table 4.28: MARK

Field	Value
`id`	integer value to return with packets

Action: FLAG

Flags packets. Similar to *Action: MARK* without a specific value; only sets the `PKT_RX_FDIR` mbuf flag.

- No configurable properties.

Table 4.29: FLAG

Field
no properties

Action: QUEUE

Assigns packets to a given queue index.

- Terminating by default.

Table 4.30: QUEUE

Field	Value
index	queue index to use

Action: DROP

Drop packets.

- No configurable properties.
- Terminating by default.
- PASSTHRU overrides this action if both are specified.

Table 4.31: DROP

Field
no properties

Action: COUNT

Enables counters for this rule.

These counters can be retrieved and reset through `rte_flow_query()`, see `struct rte_flow_query_count`.

- Counters can be retrieved with `rte_flow_query()`.
- No configurable properties.

Table 4.32: COUNT

Field
no properties

Query structure to retrieve and reset flow rule counters:

Table 4.33: COUNT query

Field	I/O	Value
`reset`	in	reset counter after query
`hits_set`	out	`hits` field is set
`bytes_set`	out	`bytes` field is set
`hits`	out	number of hits for this rule
`bytes`	out	number of bytes through this rule

Action: `DUP`

Duplicates packets to a given queue index.

This is normally combined with QUEUE, however when used alone, it is actually similar to QUEUE + PASSTHRU.

- Non-terminating by default.

Table 4.34: DUP

Field	Value
`index`	queue index to duplicate packet to

Action: `RSS`

Similar to QUEUE, except RSS is additionally performed on packets to spread them among several queues according to the provided parameters.

Note: RSS hash result is stored in the `hash.rss` mbuf field which overlaps `hash.fdir.lo`. Since *Action: MARK* sets the `hash.fdir.hi` field only, both can be requested simultaneously.

- Terminating by default.

Table 4.35: RSS

Field	Value
`rss_conf`	RSS parameters
`num`	number of entries in queue[]
`queue[]`	queue indices to use

Action: `PF`

Redirects packets to the physical function (PF) of the current device.

- No configurable properties.
- Terminating by default.

Table 4.36: PF

Field
no properties

Action: VF

Redirects packets to a virtual function (VF) of the current device.

Packets matched by a VF pattern item can be redirected to their original VF ID instead of the specified one. This parameter may not be available and is not guaranteed to work properly if the VF part is matched by a prior flow rule or if packets are not addressed to a VF in the first place.

- Terminating by default.

Table 4.37: VF

Field	Value
`original`	use original VF ID if possible
`vf`	VF ID to redirect packets to

Negative types

All specified pattern items (`enum rte_flow_item_type`) and actions (`enum rte_flow_action_type`) use positive identifiers.

The negative space is reserved for dynamic types generated by PMDs during run-time. PMDs may encounter them as a result but must not accept negative identifiers they are not aware of.

A method to generate them remains to be defined.

Planned types

Pattern item types will be added as new protocols are implemented.

Variable headers support through dedicated pattern items, for example in order to match specific IPv4 options and IPv6 extension headers would be stacked after IPv4/IPv6 items.

Other action types are planned but are not defined yet. These include the ability to alter packet data in several ways, such as performing encapsulation/decapsulation of tunnel headers.

4.8.3 Rules management

A rather simple API with few functions is provided to fully manage flow rules.

Each created flow rule is associated with an opaque, PMD-specific handle pointer. The application is responsible for keeping it until the rule is destroyed.

Flows rules are represented by `struct rte_flow` objects.

Validation

Given that expressing a definite set of device capabilities is not practical, a dedicated function is provided to check if a flow rule is supported and can be created.

```
int
rte_flow_validate(uint8_t port_id,
                  const struct rte_flow_attr *attr,
                  const struct rte_flow_item pattern[],
```

```
                const struct rte_flow_action actions[],
                struct rte_flow_error *error);
```

While this function has no effect on the target device, the flow rule is validated against its current configuration state and the returned value should be considered valid by the caller for that state only.

The returned value is guaranteed to remain valid only as long as no successful calls to `rte_flow_create()` or `rte_flow_destroy()` are made in the meantime and no device parameter affecting flow rules in any way are modified, due to possible collisions or resource limitations (although in such cases `EINVAL` should not be returned).

Arguments:

- `port_id`: port identifier of Ethernet device.
- `attr`: flow rule attributes.
- `pattern`: pattern specification (list terminated by the END pattern item).
- `actions`: associated actions (list terminated by the END action).
- `error`: perform verbose error reporting if not NULL. PMDs initialize this structure in case of error only.

Return values:

- 0 if flow rule is valid and can be created. A negative errno value otherwise (`rte_errno` is also set), the following errors are defined.
- `-ENOSYS`: underlying device does not support this functionality.
- `-EINVAL`: unknown or invalid rule specification.
- `-ENOTSUP`: valid but unsupported rule specification (e.g. partial bit-masks are unsupported).
- `-EEXIST`: collision with an existing rule.
- `-ENOMEM`: not enough resources.
- `-EBUSY`: action cannot be performed due to busy device resources, may succeed if the affected queues or even the entire port are in a stopped state (see `rte_eth_dev_rx_queue_stop()` and `rte_eth_dev_stop()`).

Creation

Creating a flow rule is similar to validating one, except the rule is actually created and a handle returned.

```
struct rte_flow *
rte_flow_create(uint8_t port_id,
                const struct rte_flow_attr *attr,
                const struct rte_flow_item pattern[],
                const struct rte_flow_action *actions[],
                struct rte_flow_error *error);
```

Arguments:

- `port_id`: port identifier of Ethernet device.
- `attr`: flow rule attributes.
- `pattern`: pattern specification (list terminated by the END pattern item).
- `actions`: associated actions (list terminated by the END action).
- `error`: perform verbose error reporting if not NULL. PMDs initialize this structure in case of error only.

Return values:

A valid handle in case of success, NULL otherwise and `rte_errno` is set to the positive version of one of the error codes defined for `rte_flow_validate()`.

Destruction

Flow rules destruction is not automatic, and a queue or a port should not be released if any are still attached to them. Applications must take care of performing this step before releasing resources.

```
int
rte_flow_destroy(uint8_t port_id,
                 struct rte_flow *flow,
                 struct rte_flow_error *error);
```

Failure to destroy a flow rule handle may occur when other flow rules depend on it, and destroying it would result in an inconsistent state.

This function is only guaranteed to succeed if handles are destroyed in reverse order of their creation.

Arguments:

- `port_id`: port identifier of Ethernet device.
- `flow`: flow rule handle to destroy.
- `error`: perform verbose error reporting if not NULL. PMDs initialize this structure in case of error only.

Return values:

- 0 on success, a negative errno value otherwise and `rte_errno` is set.

Flush

Convenience function to destroy all flow rule handles associated with a port. They are released as with successive calls to `rte_flow_destroy()`.

```
int
rte_flow_flush(uint8_t port_id,
               struct rte_flow_error *error);
```

In the unlikely event of failure, handles are still considered destroyed and no longer valid but the port must be assumed to be in an inconsistent state.

Arguments:

- `port_id`: port identifier of Ethernet device.
- `error`: perform verbose error reporting if not NULL. PMDs initialize this structure in case of error only.

Return values:

- 0 on success, a negative errno value otherwise and `rte_errno` is set.

Query

Query an existing flow rule.

This function allows retrieving flow-specific data such as counters. Data is gathered by special actions which must be present in the flow rule definition.

```
int
rte_flow_query(uint8_t port_id,
               struct rte_flow *flow,
               enum rte_flow_action_type action,
               void *data,
               struct rte_flow_error *error);
```

Arguments:

- `port_id`: port identifier of Ethernet device.
- `flow`: flow rule handle to query.
- `action`: action type to query.
- `data`: pointer to storage for the associated query data type.
- `error`: perform verbose error reporting if not NULL. PMDs initialize this structure in case of error only.

Return values:

- 0 on success, a negative errno value otherwise and `rte_errno` is set.

4.8.4 Verbose error reporting

The defined *errno* values may not be accurate enough for users or application developers who want to investigate issues related to flow rules management. A dedicated error object is defined for this purpose:

```
enum rte_flow_error_type {
    RTE_FLOW_ERROR_TYPE_NONE, /**< No error. */
    RTE_FLOW_ERROR_TYPE_UNSPECIFIED, /**< Cause unspecified. */
    RTE_FLOW_ERROR_TYPE_HANDLE, /**< Flow rule (handle). */
    RTE_FLOW_ERROR_TYPE_ATTR_GROUP, /**< Group field. */
    RTE_FLOW_ERROR_TYPE_ATTR_PRIORITY, /**< Priority field. */
    RTE_FLOW_ERROR_TYPE_ATTR_INGRESS, /**< Ingress field. */
    RTE_FLOW_ERROR_TYPE_ATTR_EGRESS, /**< Egress field. */
    RTE_FLOW_ERROR_TYPE_ATTR, /**< Attributes structure. */
    RTE_FLOW_ERROR_TYPE_ITEM_NUM, /**< Pattern length. */
    RTE_FLOW_ERROR_TYPE_ITEM, /**< Specific pattern item. */
    RTE_FLOW_ERROR_TYPE_ACTION_NUM, /**< Number of actions. */
    RTE_FLOW_ERROR_TYPE_ACTION, /**< Specific action. */
};

struct rte_flow_error {
    enum rte_flow_error_type type; /**< Cause field and error types. */
    const void *cause; /**< Object responsible for the error. */
    const char *message; /**< Human-readable error message. */
};
```

Error type `RTE_FLOW_ERROR_TYPE_NONE` stands for no error, in which case remaining fields can be ignored. Other error types describe the type of the object pointed by `cause`.

If non-NULL, `cause` points to the object responsible for the error. For a flow rule, this may be a pattern item or an individual action.

If non-NULL, `message` provides a human-readable error message.

This object is normally allocated by applications and set by PMDs in case of error, the message points to a constant string which does not need to be freed by the application, however its pointer can be considered valid only as long as its associated DPDK port remains configured. Closing the underlying device or unloading the PMD invalidates it.

4.8.5 Caveats

- DPDK does not keep track of flow rules definitions or flow rule objects automatically. Applications may keep track of the former and must keep track of the latter. PMDs may also do it for internal needs, however this must not be relied on by applications.
- Flow rules are not maintained between successive port initializations. An application exiting without releasing them and restarting must re-create them from scratch.
- API operations are synchronous and blocking (`EAGAIN` cannot be returned).
- There is no provision for reentrancy/multi-thread safety, although nothing should prevent different devices from being configured at the same time. PMDs may protect their control path functions accordingly.
- Stopping the data path (TX/RX) should not be necessary when managing flow rules. If this cannot be achieved naturally or with workarounds (such as temporarily replacing the burst function pointers), an appropriate error code must be returned (`EBUSY`).
- PMDs, not applications, are responsible for maintaining flow rules configuration when stopping and restarting a port or performing other actions which may affect them. They can only be destroyed explicitly by applications.

For devices exposing multiple ports sharing global settings affected by flow rules:

- All ports under DPDK control must behave consistently, PMDs are responsible for making sure that existing flow rules on a port are not affected by other ports.
- Ports not under DPDK control (unaffected or handled by other applications) are user's responsibility. They may affect existing flow rules and cause undefined behavior. PMDs aware of this may prevent flow rules creation altogether in such cases.

4.8.6 PMD interface

The PMD interface is defined in `rte_flow_driver.h`. It is not subject to API/ABI versioning constraints as it is not exposed to applications and may evolve independently.

It is currently implemented on top of the legacy filtering framework through filter type *RTE_ETH_FILTER_GENERIC* that accepts the single operation *RTE_ETH_FILTER_GET* to return PMD-specific *rte_flow* callbacks wrapped inside `struct rte_flow_ops`.

This overhead is temporarily necessary in order to keep compatibility with the legacy filtering framework, which should eventually disappear.

- PMD callbacks implement exactly the interface described in *Rules management*, except for the port ID argument which has already been converted to a pointer to the underlying `struct rte_eth_dev`.
- Public API functions do not process flow rules definitions at all before calling PMD functions (no basic error checking, no validation whatsoever). They only make sure these callbacks are non-NULL or return the `ENOSYS` (function not supported) error.

This interface additionally defines the following helper functions:

- `rte_flow_ops_get()`: get generic flow operations structure from a port.
- `rte_flow_error_set()`: initialize generic flow error structure.

More will be added over time.

4.8.7 Device compatibility

No known implementation supports all the described features.

Unsupported features or combinations are not expected to be fully emulated in software by PMDs for performance reasons. Partially supported features may be completed in software as long as hardware performs most of the work (such as queue redirection and packet recognition).

However PMDs are expected to do their best to satisfy application requests by working around hardware limitations as long as doing so does not affect the behavior of existing flow rules.

The following sections provide a few examples of such cases and describe how PMDs should handle them, they are based on limitations built into the previous APIs.

Global bit-masks

Each flow rule comes with its own, per-layer bit-masks, while hardware may support only a single, device-wide bit-mask for a given layer type, so that two IPv4 rules cannot use different bit-masks.

The expected behavior in this case is that PMDs automatically configure global bit-masks according to the needs of the first flow rule created.

Subsequent rules are allowed only if their bit-masks match those, the `EEXIST` error code should be returned otherwise.

Unsupported layer types

Many protocols can be simulated by crafting patterns with the *Item: RAW* type.

PMDs can rely on this capability to simulate support for protocols with headers not directly recognized by hardware.

`ANY` pattern item

This pattern item stands for anything, which can be difficult to translate to something hardware would understand, particularly if followed by more specific types.

Consider the following pattern:

Table 4.38: Pattern with ANY as L3

Index	Item		
0	ETHER		
1	ANY	`num`	`1`
2	TCP		
3	END		

Knowing that TCP does not make sense with something other than IPv4 and IPv6 as L3, such a pattern may be translated to two flow rules instead:

Table 4.39: ANY replaced with IPV4

Index	Item
0	ETHER
1	IPV4 (zeroed mask)
2	TCP
3	END

Table 4.40: ANY replaced with IPV6

Index	Item
0	ETHER
1	IPV6 (zeroed mask)
2	TCP
3	END

Note that as soon as a ANY rule covers several layers, this approach may yield a large number of hidden flow rules. It is thus suggested to only support the most common scenarios (anything as L2 and/or L3).

Unsupported actions

- When combined with *Action: QUEUE*, packet counting (*Action: COUNT*) and tagging (*Action: MARK* or *Action: FLAG*) may be implemented in software as long as the target queue is used by a single rule.
- A rule specifying both *Action: DUP* + *Action: QUEUE* may be translated to two hidden rules combining *Action: QUEUE* and *Action: PASSTHRU*.
- When a single target queue is provided, *Action: RSS* can also be implemented through *Action: QUEUE*.

Flow rules priority

While it would naturally make sense, flow rules cannot be assumed to be processed by hardware in the same order as their creation for several reasons:

- They may be managed internally as a tree or a hash table instead of a list.
- Removing a flow rule before adding another one can either put the new rule at the end of the list or reuse a freed entry.
- Duplication may occur when packets are matched by several rules.

For overlapping rules (particularly in order to use *Action: PASSTHRU*) predictable behavior is only guaranteed by using different priority levels.

Priority levels are not necessarily implemented in hardware, or may be severely limited (e.g. a single priority bit).

For these reasons, priority levels may be implemented purely in software by PMDs.

- For devices expecting flow rules to be added in the correct order, PMDs may destroy and re-create existing rules after adding a new one with a higher priority.
- A configurable number of dummy or empty rules can be created at initialization time to save high priority slots for later.
- In order to save priority levels, PMDs may evaluate whether rules are likely to collide and adjust their priority accordingly.

4.8.8 Future evolutions

- A device profile selection function which could be used to force a permanent profile instead of relying on its automatic configuration based on existing flow rules.
- A method to optimize *rte_flow* rules with specific pattern items and action types generated on the fly by PMDs. DPDK should assign negative numbers to these in order to not collide with the existing types. See *Negative types*.

- Adding specific egress pattern items and actions as described in **'Attribute: Traffic direction'_**.
- Optional software fallback when PMDs are unable to handle requested flow rules so applications do not have to implement their own.

4.8.9 API migration

Exhaustive list of deprecated filter types (normally prefixed with *RTE_ETH_FILTER_*) found in `rte_eth_ctrl.h` and methods to convert them to *rte_flow* rules.

MACVLAN to ETH → VF, PF

MACVLAN can be translated to a basic *Item: ETH* flow rule with a terminating *Action: VF* or *Action: PF*.

Table 4.41: MACVLAN conversion

Pattern				Actions
0	ETH	`spec`	any	VF, PF
		`last`	N/A	
		`mask`	any	
1	END			END

ETHERTYPE to ETH → QUEUE, DROP

ETHERTYPE is basically an *Item: ETH* flow rule with a terminating *Action: QUEUE* or *Action: DROP*.

Table 4.42: ETHERTYPE conversion

Pattern				Actions
0	ETH	`spec`	any	QUEUE, DROP
		`last`	N/A	
		`mask`	any	
1	END			END

FLEXIBLE to RAW → QUEUE

FLEXIBLE can be translated to one *Item: RAW* pattern with a terminating *Action: QUEUE* and a defined priority level.

Table 4.43: FLEXIBLE conversion

Pattern				Actions
0	RAW	`spec`	any	QUEUE
		`last`	N/A	
		`mask`	any	
1	END			END

SYN to TCP → QUEUE

SYN is a *Item: TCP* rule with only the `syn` bit enabled and masked, and a terminating *Action: QUEUE*.

Priority level can be set to simulate the high priority bit.

Table 4.44: SYN conversion

Pattern					Actions
0	ETH	`spec`	unset		QUEUE
		`last`	unset		
		`mask`	unset		
1	IPV4	`spec`	unset		END
		`mask`	unset		
		`mask`	unset		
2	TCP	`spec`	`syn`	1	
		`mask`	`syn`	1	
3	END				

`NTUPLE` to `IPV4`, `TCP`, `UDP` → `QUEUE`

NTUPLE is similar to specifying an empty L2, *Item: IPV4* as L3 with *Item: TCP* or *Item: UDP* as L4 and a terminating *Action: QUEUE*.

A priority level can be specified as well.

Table 4.45: NTUPLE conversion

Pattern				Actions
0	ETH	`spec`	unset	QUEUE
		`last`	unset	
		`mask`	unset	
1	IPV4	`spec`	any	END
		`last`	unset	
		`mask`	any	
2	TCP, UDP	`spec`	any	
		`last`	unset	
		`mask`	any	
3	END			

`TUNNEL` to `ETH`, `IPV4`, `IPV6`, `VXLAN` (or other) → `QUEUE`

TUNNEL matches common IPv4 and IPv6 L3/L4-based tunnel types.

In the following table, *Item: ANY* is used to cover the optional L4.

Table 4.46: TUNNEL conversion

Pattern				Actions
0	ETH	spec	any	QUEUE
		last	unset	
		mask	any	
1	IPV4, IPV6	spec	any	END
		last	unset	
		mask	any	
2	ANY	spec	any	
		last	unset	
		mask	num 0	
3	VXLAN, GENEVE, TEREDO, NVGRE, GRE, ...	spec	any	
		last	unset	
		mask	any	
4	END			

FDIR to most item types → QUEUE, DROP, PASSTHRU

FDIR is more complex than any other type, there are several methods to emulate its functionality. It is summarized for the most part in the table below.

A few features are intentionally not supported:

- The ability to configure the matching input set and masks for the entire device, PMDs should take care of it automatically according to the requested flow rules.

  For example if a device supports only one bit-mask per protocol type, source/address IPv4 bit-masks can be made immutable by the first created rule. Subsequent IPv4 or TCPv4 rules can only be created if they are compatible.

  Note that only protocol bit-masks affected by existing flow rules are immutable, others can be changed later. They become mutable again after the related flow rules are destroyed.

- Returning four or eight bytes of matched data when using flex bytes filtering. Although a specific action could implement it, it conflicts with the much more useful 32 bits tagging on devices that support it.
- Side effects on RSS processing of the entire device. Flow rules that conflict with the current device configuration should not be allowed. Similarly, device configuration should not be allowed when it affects existing flow rules.
- Device modes of operation. "none" is unsupported since filtering cannot be disabled as long as a flow rule is present.
- "MAC VLAN" or "tunnel" perfect matching modes should be automatically set according to the created flow rules.
- Signature mode of operation is not defined but could be handled through a specific item type if needed.

Table 4.47: FDIR conversion

Pattern				Actions
0	ETH, RAW	`spec`	any	QUEUE, DROP, PASSTHRU
		`last`	N/A	
		`mask`	any	
1	IPV4, IPv6	`spec`	any	MARK
		`last`	N/A	
		`mask`	any	
2	TCP, UDP, SCTP	`spec`	any	END
		`last`	N/A	
		`mask`	any	
3	VF, PF (optional)	`spec`	any	
		`last`	N/A	
		`mask`	any	
4	END			

`HASH`

There is no counterpart to this filter type because it translates to a global device setting instead of a pattern item. Device settings are automatically set according to the created flow rules.

`L2_TUNNEL` to `VOID` → `VXLAN` (or others)

All packets are matched. This type alters incoming packets to encapsulate them in a chosen tunnel type, optionally redirect them to a VF as well.

The destination pool for tag based forwarding can be emulated with other flow rules using *Action: DUP*.

Table 4.48: L2_TUNNEL conversion

Pattern				Actions
0	VOID	`spec`	N/A	VXLAN, GENEVE, ...
		`last`	N/A	
		`mask`	N/A	
1	END			VF (optional)
2				END

4.9 Cryptography Device Library

The cryptodev library provides a Crypto device framework for management and provisioning of hardware and software Crypto poll mode drivers, defining generic APIs which support a number of different Crypto operations. The framework currently only supports cipher, authentication, chained cipher/authentication and AEAD symmetric Crypto operations.

4.9.1 Design Principles

The cryptodev library follows the same basic principles as those used in DPDKs Ethernet Device framework. The Crypto framework provides a generic Crypto device framework which supports both physical (hardware) and virtual

(software) Crypto devices as well as a generic Crypto API which allows Crypto devices to be managed and configured and supports Crypto operations to be provisioned on Crypto poll mode driver.

4.9.2 Device Management

Device Creation

Physical Crypto devices are discovered during the PCI probe/enumeration of the EAL function which is executed at DPDK initialization, based on their PCI device identifier, each unique PCI BDF (bus/bridge, device, function). Specific physical Crypto devices, like other physical devices in DPDK can be white-listed or black-listed using the EAL command line options.

Virtual devices can be created by two mechanisms, either using the EAL command line options or from within the application using an EAL API directly.

From the command line using the –vdev EAL option

```
--vdev  'cryptodev_aesni_mb_pmd0,max_nb_queue_pairs=2,max_nb_sessions=1024,socket_id=0
↪'
```

Our using the rte_eal_vdev_init API within the application code.

```
rte_eal_vdev_init("cryptodev_aesni_mb_pmd",
                  "max_nb_queue_pairs=2,max_nb_sessions=1024,socket_id=0")
```

All virtual Crypto devices support the following initialization parameters:

- `max_nb_queue_pairs` - maximum number of queue pairs supported by the device.
- `max_nb_sessions` - maximum number of sessions supported by the device
- `socket_id` - socket on which to allocate the device resources on.

Device Identification

Each device, whether virtual or physical is uniquely designated by two identifiers:

- A unique device index used to designate the Crypto device in all functions exported by the cryptodev API.
- A device name used to designate the Crypto device in console messages, for administration or debugging purposes. For ease of use, the port name includes the port index.

Device Configuration

The configuration of each Crypto device includes the following operations:

- Allocation of resources, including hardware resources if a physical device.
- Resetting the device into a well-known default state.
- Initialization of statistics counters.

The rte_cryptodev_configure API is used to configure a Crypto device.

```
int rte_cryptodev_configure(uint8_t dev_id,
                            struct rte_cryptodev_config *config)
```

The `rte_cryptodev_config` structure is used to pass the configuration parameters. In contains parameter for socket selection, number of queue pairs and the session mempool configuration.

```
struct rte_cryptodev_config {
    int socket_id;
    /**< Socket to allocate resources on */
    uint16_t nb_queue_pairs;
    /**< Number of queue pairs to configure on device */

    struct {
        uint32_t nb_objs;
        uint32_t cache_size;
    } session_mp;
    /**< Session mempool configuration */
};
```

Configuration of Queue Pairs

Each Crypto devices queue pair is individually configured through the `rte_cryptodev_queue_pair_setup` API. Each queue pairs resources may be allocated on a specified socket.

```
int rte_cryptodev_queue_pair_setup(uint8_t dev_id, uint16_t queue_pair_id,
            const struct rte_cryptodev_qp_conf *qp_conf,
            int socket_id)

struct rte_cryptodev_qp_conf {
    uint32_t nb_descriptors; /**< Number of descriptors per queue pair */
};
```

Logical Cores, Memory and Queues Pair Relationships

The Crypto device Library as the Poll Mode Driver library support NUMA for when a processor's logical cores and interfaces utilize its local memory. Therefore Crypto operations, and in the case of symmetric Crypto operations, the session and the mbuf being operated on, should be allocated from memory pools created in the local memory. The buffers should, if possible, remain on the local processor to obtain the best performance results and buffer descriptors should be populated with mbufs allocated from a mempool allocated from local memory.

The run-to-completion model also performs better, especially in the case of virtual Crypto devices, if the Crypto operation and session and data buffer is in local memory instead of a remote processor's memory. This is also true for the pipe-line model provided all logical cores used are located on the same processor.

Multiple logical cores should never share the same queue pair for enqueuing operations or dequeuing operations on the same Crypto device since this would require global locks and hinder performance. It is however possible to use a different logical core to dequeue an operation on a queue pair from the logical core which it was enqueued on. This means that a crypto burst enqueue/dequeue APIs are a logical place to transition from one logical core to another in a packet processing pipeline.

4.9.3 Device Features and Capabilities

Crypto devices define their functionality through two mechanisms, global device features and algorithm capabilities. Global devices features identify device wide level features which are applicable to the whole device such as the device having hardware acceleration or supporting symmetric Crypto operations,

The capabilities mechanism defines the individual algorithms/functions which the device supports, such as a specific symmetric Crypto cipher or authentication operation.

Device Features

Currently the following Crypto device features are defined:

- Symmetric Crypto operations
- Asymmetric Crypto operations
- Chaining of symmetric Crypto operations
- SSE accelerated SIMD vector operations
- AVX accelerated SIMD vector operations
- AVX2 accelerated SIMD vector operations
- AESNI accelerated instructions
- Hardware off-load processing

Device Operation Capabilities

Crypto capabilities which identify particular algorithm which the Crypto PMD supports are defined by the operation type, the operation transform, the transform identifier and then the particulars of the transform. For the full scope of the Crypto capability see the definition of the structure in the *DPDK API Reference*.

```
struct rte_cryptodev_capabilities;
```

Each Crypto poll mode driver defines its own private array of capabilities for the operations it supports. Below is an example of the capabilities for a PMD which supports the authentication algorithm SHA1_HMAC and the cipher algorithm AES_CBC.

```
static const struct rte_cryptodev_capabilities pmd_capabilities[] = {
    {    /* SHA1 HMAC */
        .op = RTE_CRYPTO_OP_TYPE_SYMMETRIC,
        .sym = {
            .xform_type = RTE_CRYPTO_SYM_XFORM_AUTH,
            .auth = {
                .algo = RTE_CRYPTO_AUTH_SHA1_HMAC,
                .block_size = 64,
                .key_size = {
                    .min = 64,
                    .max = 64,
                    .increment = 0
                },
                .digest_size = {
                    .min = 12,
                    .max = 12,
                    .increment = 0
                },
                .aad_size = { 0 }
            }
        }
    },
    {    /* AES CBC */
        .op = RTE_CRYPTO_OP_TYPE_SYMMETRIC,
```

```
        .sym = {
            .xform_type = RTE_CRYPTO_SYM_XFORM_CIPHER,
            .cipher = {
                .algo = RTE_CRYPTO_CIPHER_AES_CBC,
                .block_size = 16,
                .key_size = {
                    .min = 16,
                    .max = 32,
                    .increment = 8
                },
                .iv_size = {
                    .min = 16,
                    .max = 16,
                    .increment = 0
                }
            }
        }
    }
}
```

Capabilities Discovery

Discovering the features and capabilities of a Crypto device poll mode driver is achieved through the `rte_cryptodev_info_get` function.

```
void rte_cryptodev_info_get(uint8_t dev_id,
                            struct rte_cryptodev_info *dev_info);
```

This allows the user to query a specific Crypto PMD and get all the device features and capabilities. The `rte_cryptodev_info` structure contains all the relevant information for the device.

```
struct rte_cryptodev_info {
    const char *driver_name;
    enum rte_cryptodev_type dev_type;
    struct rte_pci_device *pci_dev;

    uint64_t feature_flags;

    const struct rte_cryptodev_capabilities *capabilities;

    unsigned max_nb_queue_pairs;

    struct {
        unsigned max_nb_sessions;
    } sym;
};
```

4.9.4 Operation Processing

Scheduling of Crypto operations on DPDK's application data path is performed using a burst oriented asynchronous API set. A queue pair on a Crypto device accepts a burst of Crypto operations using enqueue burst API. On physical Crypto devices the enqueue burst API will place the operations to be processed on the devices hardware input queue, for virtual devices the processing of the Crypto operations is usually completed during the enqueue call to the Crypto device. The dequeue burst API will retrieve any processed operations available from the queue pair on the Crypto

device, from physical devices this is usually directly from the devices processed queue, and for virtual device's from a rte_ring where processed operations are place after being processed on the enqueue call.

Enqueue / Dequeue Burst APIs

The burst enqueue API uses a Crypto device identifier and a queue pair identifier to specify the Crypto device queue pair to schedule the processing on. The nb_ops parameter is the number of operations to process which are supplied in the ops array of rte_crypto_op structures. The enqueue function returns the number of operations it actually enqueued for processing, a return value equal to nb_ops means that all packets have been enqueued.

```
uint16_t rte_cryptodev_enqueue_burst(uint8_t dev_id, uint16_t qp_id,
                                     struct rte_crypto_op **ops, uint16_t nb_ops)
```

The dequeue API uses the same format as the enqueue API of processed but the nb_ops and ops parameters are now used to specify the max processed operations the user wishes to retrieve and the location in which to store them. The API call returns the actual number of processed operations returned, this can never be larger than nb_ops.

```
uint16_t rte_cryptodev_dequeue_burst(uint8_t dev_id, uint16_t qp_id,
                                     struct rte_crypto_op **ops, uint16_t nb_ops)
```

Operation Representation

An Crypto operation is represented by an rte_crypto_op structure, which is a generic metadata container for all necessary information required for the Crypto operation to be processed on a particular Crypto device poll mode driver.

The operation structure includes the operation type and the operation status, a reference to the operation specific data, which can vary in size and content depending on the operation being provisioned. It also contains the source mempool for the operation, if it allocate from a mempool. Finally an opaque pointer for user specific data is provided.

If Crypto operations are allocated from a Crypto operation mempool, see next section, there is also the ability to allocate private memory with the operation for applications purposes.

Application software is responsible for specifying all the operation specific fields in the rte_crypto_op structure which are then used by the Crypto PMD to process the requested operation.

Operation Management and Allocation

The cryptodev library provides an API set for managing Crypto operations which utilize the Mempool Library to allocate operation buffers. Therefore, it ensures that the crytpo operation is interleaved optimally across the channels and ranks for optimal processing. A rte_crypto_op contains a field indicating the pool that it originated from. When calling rte_crypto_op_free(op), the operation returns to its original pool.

```
extern struct rte_mempool *
rte_crypto_op_pool_create(const char *name, enum rte_crypto_op_type type,
                          unsigned nb_elts, unsigned cache_size, uint16_t priv_size,
                          int socket_id);
```

During pool creation rte_crypto_op_init() is called as a constructor to initialize each Crypto operation which subsequently calls __rte_crypto_op_reset() to configure any operation type specific fields based on the type parameter.

`rte_crypto_op_alloc()` and `rte_crypto_op_bulk_alloc()` are used to allocate Crypto operations of a specific type from a given Crypto operation mempool. `__rte_crypto_op_reset()` is called on each operation before being returned to allocate to a user so the operation is always in a good known state before use by the application.

```
struct rte_crypto_op *rte_crypto_op_alloc(struct rte_mempool *mempool,
                                          enum rte_crypto_op_type type)

unsigned rte_crypto_op_bulk_alloc(struct rte_mempool *mempool,
                                  enum rte_crypto_op_type type,
                                  struct rte_crypto_op **ops, uint16_t nb_ops)
```

`rte_crypto_op_free()` is called by the application to return an operation to its allocating pool.

```
void rte_crypto_op_free(struct rte_crypto_op *op)
```

4.9.5 Symmetric Cryptography Support

The cryptodev library currently provides support for the following symmetric Crypto operations; cipher, authentication, including chaining of these operations, as well as also supporting AEAD operations.

Session and Session Management

Session are used in symmetric cryptographic processing to store the immutable data defined in a cryptographic transform which is used in the operation processing of a packet flow. Sessions are used to manage information such as expand cipher keys and HMAC IPADs and OPADs, which need to be calculated for a particular Crypto operation, but are immutable on a packet to packet basis for a flow. Crypto sessions cache this immutable data in a optimal way for the underlying PMD and this allows further acceleration of the offload of Crypto workloads.

The Crypto device framework provides a set of session pool management APIs for the creation and freeing of the sessions, utilizing the Mempool Library.

The framework also provides hooks so the PMDs can pass the amount of memory required for that PMDs private session parameters, as well as initialization functions for the configuration of the session parameters and freeing function so the PMD can managed the memory on destruction of a session.

Note: Sessions created on a particular device can only be used on Crypto devices of the same type, and if you try to use a session on a device different to that on which it was created then the Crypto operation will fail.

`rte_cryptodev_sym_session_create()` is used to create a symmetric session on Crypto device. A symmetric transform chain is used to specify the particular operation and its parameters. See the section below for details on transforms.

```
struct rte_cryptodev_sym_session * rte_cryptodev_sym_session_create(
       uint8_t dev_id, struct rte_crypto_sym_xform *xform);
```

Note: For AEAD operations the algorithm selected for authentication and ciphering must aligned, eg AES_GCM.

Transforms and Transform Chaining

Symmetric Crypto transforms (`rte_crypto_sym_xform`) are the mechanism used to specify the details of the Crypto operation. For chaining of symmetric operations such as cipher encrypt and authentication generate, the next pointer allows transform to be chained together. Crypto devices which support chaining must publish the chaining of symmetric Crypto operations feature flag.

Currently there are two transforms types cipher and authentication, to specify an AEAD operation it is required to chain a cipher and an authentication transform together. Also it is important to note that the order in which the transforms are passed indicates the order of the chaining.

```
struct rte_crypto_sym_xform {
    struct rte_crypto_sym_xform *next;
    /**< next xform in chain */
    enum rte_crypto_sym_xform_type type;
    /**< xform type */
    union {
        struct rte_crypto_auth_xform auth;
        /**< Authentication / hash xform */
        struct rte_crypto_cipher_xform cipher;
        /**< Cipher xform */
    };
};
```

The API does not place a limit on the number of transforms that can be chained together but this will be limited by the underlying Crypto device poll mode driver which is processing the operation.

Symmetric Operations

The symmetric Crypto operation structure contains all the mutable data relating to performing symmetric cryptographic processing on a referenced mbuf data buffer. It is used for either cipher, authentication, AEAD and chained operations.

As a minimum the symmetric operation must have a source data buffer (`m_src`), the session type (session-based/less), a valid session (or transform chain if in session-less mode) and the minimum authentication/ cipher parameters required depending on the type of operation specified in the session or the transform chain.

```
struct rte_crypto_sym_op {
    struct rte_mbuf *m_src;
    struct rte_mbuf *m_dst;

    enum rte_crypto_sym_op_sess_type type;

    union {
        struct rte_cryptodev_sym_session *session;
        /**< Handle for the initialised session context */
        struct rte_crypto_sym_xform *xform;
        /**< Session-less API Crypto operation parameters */
    };

    struct {
        struct {
            uint32_t offset;
            uint32_t length;
        } data;   /**< Data offsets and length for ciphering */

        struct {
            uint8_t *data;
            phys_addr_t phys_addr;
            uint16_t length;
        } iv;     /**< Initialisation vector parameters */
    } cipher;
```

```
    struct {
        struct {
            uint32_t offset;
            uint32_t length;
        } data;    /**< Data offsets and length for authentication */

        struct {
            uint8_t *data;
            phys_addr_t phys_addr;
            uint16_t length;
        } digest; /**< Digest parameters */

        struct {
            uint8_t *data;
            phys_addr_t phys_addr;
            uint16_t length;
        } aad;     /**< Additional authentication parameters */
    } auth;
}
```

4.9.6 Asymmetric Cryptography

Asymmetric functionality is currently not supported by the cryptodev API.

Crypto Device API

The cryptodev Library API is described in the *DPDK API Reference* document.

4.10 链路绑定PMD

除了用于物理和虚拟硬件的轮询模式驱动程序（PMD）之外，DPDK还包括一个纯软件库，可将多个物理PMD绑定在一起以创建单个逻辑PMD。

Fig. 4.23: Bonded PMDs

Link Bonding PMD库（librte_pmd_bond）支持绑定相同速度和双工的rte_eth_dev端口组，以提供类似于Linux绑定驱动程序中的功能，以允许将多个（从属）NIC聚合到服务器和交换机中的单个逻辑接口。然后，新的聚合的PMD将根据指定的操作模式处理这些接口，以支持冗余链路，容错和/或负载均衡等功能。

librte_pmd_bond库导出一个C语言API，包括用于创建绑定设备的API，以及配置和管理绑定设备及其从属设备的API。

Note: 链路绑定PMD库默认情况下在构建配置文件中启用，可以通过设置CONFIG_RTE_LIBRTE_PMD_BOND = n并重新编译DPDK来禁用该库。

4.10.1 链路绑定模式概述

目前，Link Bonding PMD库支持以下网卡绑定模式:

- **轮询（模式0）:**

Fig. 4.24: Round-Robin (Mode 0)

轮询模式通过从第一个可用从设备到最后一个的顺序来传输数据包，以提供负载平衡和容错。数据包是从设备批量出队，然后以循环方式提供服务。这种模式不能保证接收到数据包仍然有序，下行流需要能够处理乱序数据包。

- **主动备份（模式1）:**

Fig. 4.25: Active Backup (Mode 1)

在此模式下，在任何时间只有一个从设备处于活动状态，当且仅当当前活跃从设备发生故障时，不同的从设备才会激活，从而为故障设备提供容错。单个逻辑绑定接口的MAC地址只能在一个NIC（端口）上外部可见，以避免网络交换混淆。 to avoid confusing the network switch.

- **平衡策略:**

Fig. 4.26: Balance XOR (Mode 2)

此模式提供传输负载均衡（基于所选传输策略）和容错。默认策略（layer2）使用基于报文流的源和目标MAC地址的简单计算以及绑定设备可用活动从设备的数量，将数据包分类到特定从设备进行传输。额外支持的备用传输策略是L2 + L3，这将IP源和目标地址用于传输从端口的计算，最终需要支持的策略是L3 + L4层，这使用IP源和目标地址以及TCP / UDP源和目的端口进行计算。

Note: 报文的着色差异用于识别由所选择的传输策略计算的不同流分类

- **广播策略:**
- **链路聚合802.3AD:**
- **传输负载均衡策略:**

4.10.2 实现细节

librte_pmd_bond绑定设备与DPDK API参考中描述的以太网PMD导出的以太网设备API兼容。

链路绑定库支持在EAL初始化期间的应用程序启动时使用 –vdev 选项以及通过C语言 API接口rte_eth_bond_create函数以编程方式创建绑定的设备。

绑定设备支持使用接口rte_eth_bond_slave_add / rte_eth_bond_slave_remove实现动态添加和移除。

在将从设备添加到绑定设备后，从设备使用rte_eth_dev_stop停止，然后使用rte_eth_dev_configure进行重新配置，也可以使用rte_eth_tx_queue_setup / rte_eth_rx_queue_setup重新配置RX和TX队列，并配置用于配置绑定设备的参数。如果启用绑定设备的RSS，则此模式也将在新从站上启用并进行配置。 设置用于将设备绑定到RSS的多队列模式，使其完全具有RSS功能，因此所有从设备都与其配置同步。此模式旨在提供用于客户端应用程序实现的从站上的RSS配置。

绑定设备存储其自己的RSS设置版本，即RETA，RSS散列函数和RSS密钥，用于设置其从设备。 这就是为了将绑定装置的RSS配置的含义定义为整个绑定（作为一个单元）的所需配置，而不指向任何从属内部。需要确保一致性并使其更具错误性。

Fig. 4.27: Broadcast (Mode 3)

这种模式通过在所有从设备端口上传输数据来实现容错。

Fig. 4.28: Link Aggregation 802.3AD (Mode 4)

此模式根据802.3ad规范提供了动态链路聚合。它使用所选择的均衡传输策略来协商和监视共享相同速度和双工设置的聚合组，以平衡出口流量。

这种模式的DPDK实现对应用程序提供了一些额外的要求。

1. 需要调用rte_eth_tx_burst和rte_eth_rx_burst，间隔时间小于100ms。
2. 对rte_eth_tx_burst的调用必须至少具有2xN的缓冲区大小，其中N是从设备数。这是LACP帧所需的空间。另外LACP数据包也包含在统计信息中，但不会返回给应用程序。

用于绑定设备的RSS散列函数集，是所有绑定从站支持的RSS哈希函数的最大集合。RETA大小是其所有RETA大小的GCD，因此即使从属RETA的大小不同，它也可以轻松地用作提供预期行为的模式。如果没有为绑定设备设置RSS键，则在从站上不更改，并且使用设备的默认密钥。

所有设置都通过绑定端口API进行管理，并始终沿一个方向传播（从绑定到从站）。

链路状态改变中断与轮询

链路绑定设备支持链路状态更改回调的注册，使用rte_eth_dev_callback_register接口，当绑定设备的状态发生更改时，将调用此函数进行处理。例如，在具有3个从设备的绑定设备中，当所有从设备变为不活跃时，链路状态变为DOWN，当一个从设备变为活动状态时，链路状态将变为UP。当单个从设备更改状态并且不满足先前的条件时，没有回调通知。如果用户希望监视单个从设备，则它们必须直接向该从设备注册回调。

链路绑定库还支持不实现链路状态改变中断处理的设备，这是通过使用接口rte_eth_bond_link_monitoring_set设置的周期轮询设备链路状态来实现的，默认轮询间隔为10ms。当设备作为从设备添加到绑定设备时，使用RTE_PCI_DRV_INTR_LSC标志确定设备是支持中断还是通过轮询来监视链路状态。

要求与限制

目前的实现只支持相同速度和双工的设备作为从设备提供给同一个绑定设备。绑定设备从添加到绑定设备的第一个活动从设备上继承这些属性，然后添加到绑定设备的所有其他从设备必须支持这些参数。

绑定设备本身启动之前，必须至少一个从设备。

为了有效地使用绑定设备动态RSS配置功能，还需要所有的从设备都应该是具有RSS能力和支持的，至少有一个通用的散列函数可用于它们。只有当所有从设备支持相同的密钥大小时才可以更改RSS密钥。

为了防止从设备对于如何处理数据包产生矛盾，一旦将设备添加到绑定设备，RSS配置应通过绑定设备API进行管理，而不是直接在从设备上进行管理。

像所有其他PMD一样，PMD导出的所有功能都是无锁功能，假定不会在不同逻辑核心上并行调用以操作同一目标对象。

Fig. 4.29: Transmit Load Balancing (Mode 5)

此模式提供自适应传输负载均衡。它根据计算的负载动态地更改发送从设备。以100ms的间隔收集统计数据，每10ms调度一次。

还应该注意的是，PMD接收功能在它们已经到达绑定设备之后不应该直接在从设备上被调用，因为直接从从设备读取的数据包将不再可用于绑定设备读取。

配置

链路绑定设备使用rte_eth_bond_create API创建，该API需要传入唯一的设备名称，绑定模式和套接字ID来分配绑定设备的资源。绑定设备的其他可配置参数是其从设备，主从，用户定义的MAC地址，如果设备处于平衡XOR模式还需要定义要使用的传输策略。

从设备

绑定设备支持相同速度和双工的设备，最大数目为RTE_MAX_ETHPORTS。每个以太网设备可以作为从设备添加到最多一个绑定设备上。从设备在被加入绑定设备时被重新配置为绑定设备的配置。

绑定还保证将从设备的MAC地址返回到其原始值。

主从

主从关系用于定义绑定设备处于主动备份模式（模式1）时使用的默认端口。当且仅当当前主端口关闭时，才会使用不同的端口。如果用户没有指定主端口，则默认为添加到绑定设备的第一个端口。

MAC地址

绑定设备可以配置用户指定的MAC地址，该地址将由某些或所有从设备根据操作模式继承。如果设备处于主动备份模式，则只有主设备具有用户指定的MAC，所有其他从设备将保留其原始MAC地址。在模式0,2,3,4中，所有从站设备都配置了绑定设备的MAC地址。

如果未定义用户定义的MAC地址，则绑定设备将默认使用主从站MAC地址。

均衡XOR模式传输策略

对于在均衡XOR模式下运行的绑定设备，有3种支持的传输策略。层2，层2 + 3，层3 + 4。

- **Layer 2:** 默认的传输策略是以太网基于MAC地址的均衡策略。它对包的源MAC地址和目的MAC地址使用简单的XOR计算，然后计算该值的模数，以计算需要输出数据包的从设备。
- **Layer 2 + 3:** 以太网MAC地址和基于IP地址的均衡策略使用源/目的MAC地址和数据包的源/目的IP地址组合来决定数据包将被传输的从设备端口。
- **Layer 3 + 4:** IP地址和UDP基于端口的均衡策略使用源/目的IP地址和数据包的数据包的源/目的UDP端口的组合来决定数据包将被传输的从设备端口。

所有这些策略都支持802.1Q VLAN以太网报文，还支持IPv4，IPv6和UDP协议进行负载分担。

4.10.3 使用链路绑定设备

librte_pmd_bond库支持两种设备创建模式，库导出完整的C API或使用EAL命令行在应用程序启动时静态配置链路绑定设备。使用EAL选项，可以透明地使用链接绑定功能，而不需要库API的具体知识，例如，可以使用这种功能来将绑定功能（如主动备份）添加到不了解链接的现有应用程序上。

程序中使用轮询模式驱动

使用librte_pmd_bond库API，可以在任何应用程序内动态创建和管理链路绑定设备。链路绑定设备使用rte_eth_bond_create API创建，该API需要唯一的设备名称，用于初始化设备的链路绑定模式，以及最后将要分配设备资源的套接字ID。在成功创建绑定设备之后，必须使用通用的以太网设备配置API rte_eth_dev_configure来配置，然后使用rte_eth_tx_queue_setup/rte_eth_rx_queue_setup将要使用的RX和TX队列进行设置。

可以使用rte_eth_bond_slave_add/rte_eth_bond_slave_remove API对链路绑定设备动态添加和删除从设备，但在使用rte_eth_dev_start启动链路绑定设备之前，必须至少添加一个从设备。

绑定设备的链路状态由其从设备的链路状态决定，如果所有从设备链路状态都关闭，或者所有从设备都从链路绑定设备中删除，则绑定设备的链路状态为DOWN。

还可以使用提供的rte_eth_bond_mode_set/get，rte_eth_bond_primary_set/get，rte_eth_bond_mac_set/reset和rte_eth_bond_xmit_po
配置/查询绑定设备的控制参数的配置。

在EAL命令行中使用链路绑定设备

链路绑定设备可以在应用程序启动时使用–vdev EAL命令行选项创建。 设备名称必须以net_bonding前缀开头，后跟数字或字母。每个设备的名称必须是唯一的。每个设备可以有多个选项，以逗号分隔列表排列。可以多次调用–vdev选项来安排多个设备定义。

设备名称和绑定选项必须用逗号分隔，如下所示：

```
$RTE_TARGET/app/testpmd -l 0-3 -n 4 --vdev 'net_bond0,bond_opt0=..,bond opt1=..'--
↪vdev 'net_bond1,bond _opt0=..,bond_opt1=..'
```

链路绑定EAL选项

只要遵守以下两个规则，可以对多种定义方式组合使用：

- 提供了一种独特的设备名称，格式为net_bondingX，其中X可以是数字和/或字母的任意组合，名称不大于32个字符。
- 每个绑定设备定义提供至少一个从设备。
- 提供了所创建的绑定设备的操作模式。

不同的选项包括：

- 模式：定义设备的绑定模式的整数值。目前支持模式0,1,2,3,4,5（循环，主动备份，平衡，广播，链路聚合，传输负载均衡）。

```
mode=2
```

- 从设备：定义将作为从设备添加到绑定设备的PMD设备。可以多次选择此选项，每个设备要作为从设备添加。物理设备应使用其PCI地址指定，格式为 domain:bus:devid.function。

```
slave=0000:0a:00.0,slave=0000:0a:00.1
```

- 主设备：定义主从端口的可选参数用于主动备份模式，以便在数据TX / RX可用时选择主从机。 当主端口未被用户定义时，主端口也用于选择要使用的MAC地址。如果未指定该设备，则默认为添加到设备的第一个从设备。主设备必须是绑定设备的从设备。

```
primary=0000:0a:00.0
```

- Socket_id：可选参数，用于选择NUMA设备上将分配绑定设备资源的哪个套接字。

```
socket_id=0
```

- Mac：可选参数，选择链路绑定设备的MAC地址，这将覆盖主设备的值。

```
mac=00:1e:67:1d:fd:1d
```

- xmit_policy：绑定设备处于均衡模式时定义传输策略的可选参数。如果没有用户指定，则默认为l2（第2层）转发，其他可用的传输策略为l23（第2层+3层）和l34层（3 + 4层）。

```
xmit_policy=l23
```

- lsc_poll_period_ms：可选参数，用于定义不支持lsc中断的设备以毫秒为单位的轮询间隔，检查设备链路状态的变化。

```
lsc_poll_period_ms=100
```

- ups delay：可选参数，增加了设备链路状态传播的延迟（以毫秒为单位），默认情况下该参数为零。

```
up_delay=10
```

- down_delay：可选参数，以毫秒为单位，将设备链路状态DOWN的传播延迟，默认情况下，该参数为零。

```
down_delay=50
```

使用实例

以轮询模式创建一个绑定设备，两个从设备由其PCI地址指定：

```
$RTE_TARGET/app/testpmd -l 0-3 -n 4 --vdev 'net_bond0,mode=0, slave=0000:00a:00.01,
↪slave=0000:004:00.00' -- --port-topology=chained
```

以轮询模式创建一个绑定设备，其中两个从站由其PCI地址和覆盖MAC地址指定：

```
$RTE_TARGET/app/testpmd -l 0-3 -n 4 --vdev 'net_bond0,mode=0, slave=0000:00a:00.01,
↪slave=0000:004:00.00,mac=00:1e:67:1d:fd:1d' -- --port-topology=chained
```

Create a bonded device in active backup mode with two slaves specified, and a primary slave specified by their PCI addresses:

```
$RTE_TARGET/app/testpmd -l 0-3 -n 4 --vdev 'net_bond0,mode=1, slave=0000:00a:00.01,
↪slave=0000:004:00.00,primary=0000:00a:00.01' -- --port-topology=chained
```

在平衡模式下创建一个绑定设备，其中两个从站由其PCI地址指定，第3 + 4层传输策略：

```
$RTE_TARGET/app/testpmd -l 0-3 -n 4 --vdev 'net_bond0,mode=2, slave=0000:00a:00.01,
↪slave=0000:004:00.00,xmit_policy=l34' -- --port-topology=chained
```

4.11 定时器库

定时器库为DPDK执行单元提供定时器服务，使得执行单元可以为异步操作执行回调函数。定时器库的特性如下：

- 定时器可以周期执行，也可以执行一次。
- 定时器可以在一个核心加载并在另一个核心执行。但是必须在调用rte_timer_reset()中指定它。
- 定时器提供高精度（取决于检查本地核心的定时器到期的rte_timer_manage()的调用频率）。
- 如果应用程序不需要，可以在编译时禁用定时器，并且程序中不调用rte_timer_manage()来提高性能。

定时器库使用rte_get_timer_cycles()获取高精度事件定时器（HPET）或CPU时间戳计数器（TSC）提供的可靠时间参考。

该库提供了添加，删除和重新启动定时器的接口。API基于BSD callout()，可能会有一些差异。详细请参考 callout manual.

4.11.1 实现细节

定时器以每个逻辑核为基础进行跟踪，一个逻辑核上要维护的所有挂起的定时器，按照定时器到期顺序插入到跳跃表数据结构。

所使用的跳跃表有十个层，表中的每个条目都以1/4的概率显示在每个层上。这意味着所有条目都存在于第0层中，每4个条目中的1个条目存在于第一层，每16个中1个条目存在于第2层，等等。同时，这意味着从逻辑核的定时器列表中添加和删除条目可以在log(n)时间内完成，最多4 ^ 10个条目，即每个逻辑核约有1,000,000个定时器。

定时器结构包含一个称为状态的特殊字段，它是定时器状态（stopped，pending，running，config）及其所有者（lcore id）的联合体。根据定时器状态，我们可以知道定时器当前是否存在于列表中：

- STOPPED：没有所有者，不再链表中。
- CONFIG: 由一个逻辑核持有，其他逻辑核不能修改，是否存在于跳表中取决于以前的状态。
- PENDING: 由一个逻辑核持有，其他逻辑核不能修改，是否存在于跳表中取决于以前的状态。
- RUNNING: 由一个逻辑核持有，其他逻辑核不能修改，是否存在于跳表中取决于以前的状态。

不允许在定时器处于CONFIG或RUNNING状态时复位或停止定时器。当修改定时器的状态时，应使用CAP指令来保证状态修改操作（状态+所有者）是原子操作。

在rte_timer_manage()函数里面，跳跃表作为常规的链表，通过沿着包含所有计时器条目的第0层链表迭代，直到遇到尚未到期的条目为止。当列表中有条目，但是没有任何条目定时器到期时，为了提高性能，第一个定时器条目的到期时间保存在每个逻辑和计时器列表结构本身内部。在64位平台上，可以直接检查该值，而无需对整个结构进行锁定。（由于到期时间维持为64位值，所以在32位平台上无法直接对该值进行检查，而不使用（CAS）指令或使用锁机制，因此，一旦数据结构被上锁，此额外的检查将被跳过。）在64位和32位平台上，在调用逻辑核的计时器列表为空的情况下，对rte_timer_manage（）的调用将直接返回而不进行锁定。=

4.11.2 用例

定时器库用于定期调用，如垃圾收集器或某些状态机（ARP，桥接等）。

4.11.3 参考

- callout manual - 唤醒功能，提供定时器到期执行的功能。
- HPET - 有关高精度事件定时器（HPET）的信息。

4.12 哈希库

DPDK提供了一个用于创建哈希表的哈希库，哈希表可以用于快速查找。哈希表是针对一组条目进行搜索而优化的数据结构，每个条目由唯一Key标识。为了提高性能，DPDK哈希要求所有的Key值具有与哈希创建时指定的相同字节数。

4.12.1 哈希API概述

哈希的主要配置参数包括：

- 哈希条目总数（哈希容量）。
- Key的字节数。

哈希还允许配置一些低级实现相关的参数：

- 将Key转换为哈希桶索引值的哈希函数

哈希库导出的主要方法包括：

- 使用Key值添加条目：Key值作为输入参数。如果新条目成功添加到指定Key的哈希中，或者已经有指定Key的条目，则返回该条目的位置。如果操作不成功，例如由于在哈希中缺少空闲条目，则返回负值;
- 使用Key删除条目：Key值作为输入参数。如果在哈希中找到具有指定Key的条目，则会从哈希中删除该条目，并返回该条目在哈希中找到的位置。如果哈希中没有指定Key的条目存在，则返回一个负值;
- 使用Key查找条目：Key值作为输入参数。如果在哈希（查找命中）中找到具有指定Key的条目，则返回条目的位置，否则返回（查询未命中）一个负值。

除了这些方法，API还为用户提供了三个选项：

- 使用Key和precomputed hash来查找/添加/删除条目：Key及其precomputed hash都作为输入。这允许用户更快地执行操作，因为已经预先计算了散列。
- 使用Key和数据来查找/添加条目：提供Key-value作为输入。这允许用户不仅存储Key，还可以存储8byte的整形或是一个指向外部数据的指针（数据超过8byte）。
- 上述两个选项的组合：用户可以提供Key、precomputed hash或是data。

此外，API包含一种方法，允许用户在突发中查找条目，实现比查找单个条目更高的性能，因为该函数在与第一个条目操作时预取下一个条目，显著降低了必要的内存访问。请注意，此方法使用8个条目（4个阶段2条目）的流水线，因此强烈建议每个突发使用至少8个条目。

与每个Key相关联的实际数据可以由用户使用单独的表格进行管理，该表格根据哈希条目数量和每个条目的位置来反映哈希表，如以下部分中描述的流分类用例所示，当然，也可以直接存储在哈希表本身。

L2 / L3转发示例应用程序中的哈希表根据由五元组查找标识的数据包流定义将数据包转发到哪个端口。然而，该表还可以用于更复杂的特征，并提供可以在分组和流上执行的许多其他功能和动作。

4.12.2 多进程支持

哈希库可以在多进程环境中使用，只需查找线程安全即可。只能在单进程模式下使用的唯一函数是rte_hash_set_cmp_func()，它设置一个自定义的比较功能，分配给一个函数指针（因此在多进程模式下不支持）。

4.12.3 实现细节

哈希表有两个主表：

- 第一个表是一组条目，进一步分为桶，每个桶中具有相同数量的连续数组条目。每个条目包含计算的给定Key的主要和次要散列（如下所述）和第二个表的索引。
- 第二个表是存储在哈希表中的所有Key的数组及其与每个Key相关联的数据。

哈希库使用Cuckoo hash（布谷鸟散列）方法来解决冲突。对于任何输入Key，有两个可能的桶（主要和次要/替代位置），其中该Key可以存储在散列中，因此只有当查询Key时才需要检查桶中的条目。与通过线性扫描阵列中的所有条目的基本方法相反，通过将散列条目的总数减少到两个哈希桶中的条目数来减少要扫描的条目数以提升查找速度。哈希使用散列函数（可配置）将输入Key转换为4字节Key签名。桶索引值是将哈希Key签名对哈希桶数取模数的值。

一旦识别出桶，哈希添加，删除和查找操作的范围就减少到这些存储区中的条目（很可能条目在主存储桶中）。

为了加快桶内的搜索逻辑，每个散列条目将4字节Key签名与每个哈希条目的完整Key一起存储。对于大的Key，将输入Key与来自存储桶的Key进行比较比将输入Key的4字节签名与来自存储桶的Key签名进行比较要花费更多的时间。因此，首先完成签名比较，仅在签名匹配时才完成Key比较。完全Key比较仍然是必要的，因为来自相同存储桶的两个输入Key仍然可能具有相同的4字节签名，尽管对于该组输入密钥提供良好的均匀分布的散列函数，该事件相对较少。

查找实例：

首先，主桶被识别，条目可能存储在那里。如果签名存储在那里，我们将其Key与提供的Key进行比较，并返回其存储位置和/或与该密钥相关联的数据（如果有匹配）。如果签名不在主桶中，则查找辅助桶，在那里执行相同的过程。如果没有匹配，Key对应条目被认为不在表中。

添加实例：

像查找操作一样，Key标识主和二级桶。如果主桶中有一个空槽，则主签名和辅助签名存储在该槽中，Key和数据（如果有的话）被添加到第二个表中，并且第二个表中的位置的索引被存储在第一张表上。如果主桶中没有空槽，则该桶中的一个条目将被推送到其替代位置，并将要添加的Key插入第一个条目的位置上。要知道驱逐条目（第一个条目）的替代桶的哪个位置，则查找器辅助签名，并从上面的模数中计算备用桶索引。如果替代桶中有空间，则将被驱入的条目存储在其中。如果没有，则重复相同的过程（其中一个条目被推送），直到找到非完整的数据桶。请注意，尽管所有的条目移动都在第一张表中，第二张表没有被触动，这也将在性能上受到很大影响。

在非常不太可能的事件中，该表进入循环，其中相同的条目被无限期地驱逐，则认为Key不能被存储。使用随机Key，该方法允许用户获取约90%的表利用率，而不必放弃任何存储的条目（LRU）或分配更多内存（扩展桶）。

4.12.4 哈希表中的条目分发

如上所述，如果有一个新的条目要被添加到哪个主桶，而当前已经有数据在里面时，则将数据推送到他们的替代位置，Cuckoo哈希实现了将元素推出他们的存储区。

因此，当用户向哈希表添加更多条目时，桶中散列值的分布将发生变化，其中大部分位于主要位置，并且其次要位置会随之增加，随后表将增加。

这些信息是非常有用的，因为随着更多条目逐出其次要位置，性能可能会降低。

下表显示了表利用率增加时的示例条目分布。

Table 4.49: Entry distribution measured with an example table with 1024 random entries using jhash algorithm

% Table used	% In Primary location	% In Secondary location
25	100	0
50	96.1	3.9
75	88.2	11.8
80	86.3	13.7
85	83.1	16.9
90	77.3	22.7
95.8	64.5	35.5

Table 4.50: Entry distribution measured with an example table with 1 million random entries using jhash algorithm

% Table used	% In Primary location	% In Secondary location
50	96	4
75	86.9	13.1
80	83.9	16.1
85	80.1	19.9
90	74.8	25.2
94.5	67.4	32.6

Note: 上表上的最后值是具有随机密钥和使用Jenkins散列函数的平均最大表利用率。

4.12.5 用例：流分类

流分类用于将每个输入数据包映射到它所属的连接/流。这种操作是必需的，因为每个输入分组的处理通常在其连接的上下文中进行，因此相同的操作集合被应用于来自相同流的所有分组。

使用流分类的应用通常具有要管理的流表，每个单独的流具有与该表相关联的条目。流表条目的大小是特定于应用程序的，典型值为4,16,32或64字节。

使用流分类的每个应用通常具有被定义为从输入报文中读取一个或多个字段来构成Key，用于标识流。一个例子是使用由IP和传输层数据包头的以下字段组成的DiffServ 5元组：源IP地址，目标IP地址，协议，源端口，目标端口。

DPDK哈希提供了一种通用的方法来实现应用程序指定的流分类机制。给定一个用数组实现的流表，应用程序应该创建与流表相同数量的条目的哈希对象，并将哈希密钥大小设置为所选流Key中的字节数。

应用侧的流程表操作如下：

- Add flow: 将流Key添加到哈希。如果返回的位置有效，则使用它来访问流表中用于添加新流或更新与现有流相关联的信息的流条目。否则，流添加失败，例如由于缺少用于存储新流的空闲条目。
- Delete flow: 从哈希中删除流Key。如果返回的位置有效，则使用它来访问流表中的流条目以使与流相关联的信息无效。
- Lookup flow: 在哈希中查找流Key。如果返回的位置有效（流查找命中），则使用返回的位置来访问流表中的流条目。否则（流查找未命中）表示当前数据包没有注册流。

4.12.6 参考

- Donald E. Knuth, The Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition), 1998, Addison-Wesley Professional

4.13 Elastic Flow Distributor Library

4.13.1 Introduction

In Data Centers today, clustering and scheduling of distributed workloads is a very common task. Many workloads require a deterministic partitioning of a flat key space among a cluster of machines. When a packet enters the cluster, the ingress node will direct the packet to its handling node. For example, data-centers with disaggregated storage use storage metadata tables to forward I/O requests to the correct back end storage cluster, stateful packet inspection will use match incoming flows to signatures in flow tables to send incoming packets to their intended deep packet inspection (DPI) devices, and so on.

EFD is a distributor library that uses perfect hashing to determine a target/value for a given incoming flow key. It has the following advantages: first, because it uses perfect hashing it does not store the key itself and hence lookup performance is not dependent on the key size. Second, the target/value can be any arbitrary value hence the system designer and/or operator can better optimize service rates and inter-cluster network traffic locating. Third, since the storage requirement is much smaller than a hash-based flow table (i.e. better fit for CPU cache), EFD can scale to millions of flow keys. Finally, with the current optimized library implementation, performance is fully scalable with any number of CPU cores.

4.13.2 Flow Based Distribution

Computation Based Schemes

Flow distribution and/or load balancing can be simply done using a stateless computation, for instance using round-robin or a simple computation based on the flow key as an input. For example, a hash function can be used to direct a certain flow to a target based on the flow key (e.g. `h(key) mod n`) where h(key) is the hash value of the flow key and n is the number of possible targets.

Fig. 4.30: Load Balancing Using Front End Node

In this scheme (Fig. 4.30), the front end server/distributor/load balancer extracts the flow key from the input packet and applies a computation to determine where this flow should be directed. Intuitively, this scheme is very simple and requires no state to be kept at the front end node, and hence, storage requirements are minimum.

Fig. 4.31: Consistent Hashing

A widely used flow distributor that belongs to the same category of computation-based schemes is `consistent hashing`, shown in Fig. 4.31. Target destinations (shown in red) are hashed into the same space as the flow keys (shown in blue), and keys are mapped to the nearest target in a clockwise fashion. Dynamically adding and removing targets with consistent hashing requires only K/n keys to be remapped on average, where K is the number of keys, and n is the number of targets. In contrast, in a traditional hash-based scheme, a change in the number of targets causes nearly all keys to be remapped.

Although computation-based schemes are simple and need very little storage requirement, they suffer from the drawback that the system designer/operator can't fully control the target to assign a specific key, as this is dictated by the hash function. Deterministically co-locating of keys together (for example, to minimize inter-server traffic or to optimize for network traffic conditions, target load, etc.) is simply not possible.

Flow-Table Based Schemes

When using a Flow-Table based scheme to handle flow distribution/load balancing, in contrast with computation-based schemes, the system designer has the flexibility of assigning a given flow to any given target. The flow table (e.g. DPDK RTE Hash Library) will simply store both the flow key and the target value.

Fig. 4.32: Table Based Flow Distribution

As shown in Fig. 4.32, when doing a lookup, the flow-table is indexed with the hash of the flow key and the keys (more than one is possible, because of hash collision) stored in this index and corresponding values are retrieved. The retrieved key(s) is matched with the input flow key and if there is a match the value (target id) is returned.

The drawback of using a hash table for flow distribution/load balancing is the storage requirement, since the flow table need to store keys, signatures and target values. This doesn't allow this scheme to scale to millions of flow keys. Large tables will usually not fit in the CPU cache, and hence, the lookup performance is degraded because of the latency to access the main memory.

EFD Based Scheme

EFD combines the advantages of both flow-table based and computation-based schemes. It doesn't require the large storage necessary for flow-table based schemes (because EFD doesn't store the key as explained below), and it supports any arbitrary value for any given key.

Fig. 4.33: Searching for Perfect Hash Function

The basic idea of EFD is when a given key is to be inserted, a family of hash functions is searched until the correct hash function that maps the input key to the correct value is found, as shown in Fig. 4.33. However, rather than explicitly storing all keys and their associated values, EFD stores only indices of hash functions that map keys to values, and thereby consumes much less space than conventional flow-based tables. The lookup operation is very simple, similar to a computational-based scheme: given an input key the lookup operation is reduced to hashing that key with the correct hash function.

Fig. 4.34: Divide and Conquer for Millions of Keys

Intuitively, finding a hash function that maps each of a large number (millions) of input keys to the correct output value is effectively impossible, as a result EFD, as shown in Fig. 4.34, breaks the problem into smaller pieces (divide and conquer). EFD divides the entire input key set into many small groups. Each group consists of approximately 20-28 keys (a configurable parameter for the library), then, for each small group, a brute force search to find a hash function that produces the correct outputs for each key in the group.

It should be mentioned that, since the online lookup table for EFD doesn't store the key itself, the size of the EFD table is independent of the key size and hence EFD lookup performance which is almost constant irrespective of the length of the key which is a highly desirable feature especially for longer keys.

In summary, EFD is a set separation data structure that supports millions of keys. It is used to distribute a given key to an intended target. By itself EFD is not a FIB data structure with an exact match the input flow key.

4.13.3 Example of EFD Library Usage

EFD can be used along the data path of many network functions and middleboxes. As previously mentioned, it can used as an index table for <key,value> pairs, meta-data for objects, a flow-level load balancer, etc. Fig. 4.35 shows an example of using EFD as a flow-level load balancer, where flows are received at a front end server before being forwarded to the target back end server for processing. The system designer would deterministically co-locate flows together in order to minimize cross-server interaction. (For example, flows requesting certain webpage objects are co-located together, to minimize forwarding of common objects across servers).

Fig. 4.35: EFD as a Flow-Level Load Balancer

As shown in Fig. 4.35, the front end server will have an EFD table that stores for each group what is the perfect hash index that satisfies the correct output. Because the table size is small and fits in cache (since keys are not stored), it sustains a large number of flows (N*X, where N is the maximum number of flows served by each back end server of the X possible targets).

With an input flow key, the group id is computed (for example, using last few bits of CRC hash) and then the EFD table is indexed with the group id to retrieve the corresponding hash index to use. Once the index is retrieved the key is hashed using this hash function and the result will be the intended correct target where this flow is supposed to be processed.

It should be noted that as a result of EFD not matching the exact key but rather distributing the flows to a target back end node based on the perfect hash index, a key that has not been inserted before will be distributed to a valid target. Hence, a local table which stores the flows served at each node is used and is exact matched with the input key to rule out new never seen before flows.

4.13.4 Library API Overview

The EFD library API is created with a very similar semantics of a hash-index or a flow table. The application creates an EFD table for a given maximum number of flows, a function is called to insert a flow key with a specific target value, and another function is used to retrieve target values for a given individual flow key or a bulk of keys.

EFD Table Create

The function `rte_efd_create()` is used to create and return a pointer to an EFD table that is sized to hold up to num_flows key. The online version of the EFD table (the one that does not store the keys and is used for lookups) will be allocated and created in the last level cache (LLC) of the socket defined by the online_socket_bitmask, while the offline EFD table (the one that stores the keys and is used for key inserts and for computing the perfect hashing) is allocated and created in the LLC of the socket defined by offline_socket_bitmask. It should be noted, that for highest performance the socket id should match that where the thread is running, i.e. the online EFD lookup table should be created on the same socket as where the lookup thread is running.

EFD Insert and Update

The EFD function to insert a key or update a key to a new value is `rte_efd_update()`. This function will update an existing key to a new value (target) if the key has already been inserted before, or will insert the <key,value> pair if this key has not been inserted before. It will return 0 upon success. It will return `EFD_UPDATE_WARN_GROUP_FULL (1)` if the operation is insert, and the last available space in the key's group was just used. It will return `EFD_UPDATE_FAILED (2)` when the insertion or update has failed (either it failed to find a suitable perfect hash or the group was full). The function will return `EFD_UPDATE_NO_CHANGE (3)` if there is no change to the EFD table (i.e, same value already exists).

Note: This function is not multi-thread safe and should only be called from one thread.

EFD Lookup

To lookup a certain key in an EFD table, the function `rte_efd_lookup()` is used to return the value associated with single key. As previously mentioned, if the key has been inserted, the correct value inserted is returned, if the key has not been inserted before, a 'random' value (based on hashing of the key) is returned. For better performance and to decrease the overhead of function calls per key, it is always recommended to use a bulk lookup function (simultaneous lookup of multiple keys) instead of a single key lookup function. `rte_efd_lookup_bulk()` is the bulk lookup function, that looks up num_keys simultaneously stored in the key_list and the corresponding return values will be returned in the value_list.

Note: This function is multi-thread safe, but there should not be other threads writing in the EFD table, unless locks are used.

EFD Delete

To delete a certain key in an EFD table, the function `rte_efd_delete()` can be used. The function returns zero upon success when the key has been found and deleted. Socket_id is the parameter to use to lookup the existing value, which is ideally the caller's socket id. The previous value associated with this key will be returned in the prev_value argument.

Note: This function is not multi-thread safe and should only be called from one thread.

4.13.5 Library Internals

This section provides the brief high-level idea and an overview of the library internals to accompany the RFC. The intent of this section is to explain to readers the high-level implementation of insert, lookup and group rebalancing in the EFD library.

Insert Function Internals

As previously mentioned the EFD divides the whole set of keys into groups of a manageable size (e.g. 28 keys) and then searches for the perfect hash that satisfies the intended target value for each key. EFD stores two version of the <key,value> table:

- Offline Version (in memory): Only used for the insertion/update operation, which is less frequent than the lookup operation. In the offline version the exact keys for each group is stored. When a new key is added, the hash function is updated that will satisfy the value for the new key together with the all old keys already inserted in this group.
- Online Version (in cache): Used for the frequent lookup operation. In the online version, as previously mentioned, the keys are not stored but rather only the hash index for each group.

Fig. 4.36: Group Assignment

Fig. 4.36 depicts the group assignment for 7 flow keys as an example. Given a flow key, a hash function (in our implementation CRC hash) is used to get the group id. As shown in the figure, the groups can be unbalanced. (We highlight group rebalancing further below).

Fig. 4.37: Perfect Hash Search - Assigned Keys & Target Value

Focusing on one group that has four keys, Fig. 4.37 depicts the search algorithm to find the perfect hash function. Assuming that the target value bit for the keys is as shown in the figure, then the online EFD table will store a 16 bit hash index and 16 bit lookup table per group per value bit.

Fig. 4.38: Perfect Hash Search - Satisfy Target Values

For a given keyX, a hash function `(h(keyX, seed1) + index * h(keyX, seed2))` is used to point to certain bit index in the 16bit lookup_table value, as shown in Fig. 4.38. The insert function will brute force search for all possible values for the hash index until a non conflicting lookup_table is found.

Fig. 4.39: Finding Hash Index for Conflict Free lookup_table

For example, since both key3 and key7 have a target bit value of 1, it is okay if the hash function of both keys point to the same bit in the lookup table. A conflict will occur if a hash index is used that maps both Key4 and Key7 to the same index in the lookup_table, as shown in Fig. 4.39, since their target value bit are not the same. Once a hash index is found that produces a lookup_table with no contradictions, this index is stored for this group. This procedure is repeated for each bit of target value.

Lookup Function Internals

The design principle of EFD is that lookups are much more frequent than inserts, and hence, EFD's design optimizes for the lookups which are faster and much simpler than the slower insert procedure (inserts are slow, because of perfect hash search as previously discussed).

Fig. 4.40 depicts the lookup operation for EFD. Given an input key, the group id is computed (using CRC hash) and then the hash index for this group is retrieved from the EFD table. Using the retrieved hash index, the hash function `h(key, seed1) + index *h(key, seed2)` is used which will result in an index in the lookup_table, the bit corresponding to this index will be the target value bit. This procedure is repeated for each bit of the target value.

Group Rebalancing Function Internals

When discussing EFD inserts and lookups, the discussion is simplified by assuming that a group id is simply a result of hash function. However, since hashing in general is not perfect and will not always produce a uniform output, this simplified assumption will lead to unbalanced groups, i.e., some group will have more keys than other groups. Typically, and to minimize insert time with an increasing number of keys, it is preferable that all groups will have a balanced number of keys, so the brute force search for the perfect hash terminates with a valid hash index. In order to achieve this target, groups are rebalanced during runtime inserts, and keys are moved around from a busy group to a less crowded group as the more keys are inserted.

Fig. 4.41 depicts the high level idea of group rebalancing, given an input key the hash result is split into two parts a chunk id and 8-bit bin id. A chunk contains 64 different groups and 256 bins (i.e. for any given bin it can map to 4 distinct groups). When a key is inserted, the bin id is computed, for example in Fig. 4.41 bin_id=2, and since each bin can be mapped to one of four different groups (2 bit storage), the four possible mappings are evaluated and the one that will result in a balanced key distribution across these four is selected the mapping result is stored in these two bits.

Fig. 4.40: EFD Lookup Operation

Fig. 4.41: Runtime Group Rebalancing

4.13.6 References

1- EFD is based on collaborative research work between Intel and Carnegie Mellon University (CMU), interested readers can refer to the paper "Scaling Up Clustered Network Appliances with ScaleBricks;" Dong Zhou et al. at SIGCOMM 2015 (*http://conferences.sigcomm.org/sigcomm/2015/pdf/papers/p241.pdf*) for more information.

4.14 LPM库

DPDK LPM库组件实现了32位Key的最长前缀匹配（LPM）表搜索方法，该方法通常用于在IP转发应用程序中找到最佳路由。

4.14.1 LPM API概述

LPM组件实例的主要配置参数是要支持的最大数量的规则。LPM前缀由一对参数（32位Key，深度）表示，深度范围为1到32。LPM规则由LPM前缀和与前缀相关联的一些用户数据表示。该前缀作为LPM规则的唯一标识符。在该实现中，用户数据为1字节长，被称为下一跳，与其在路由表条目中存储下一跳的ID的主要用途相关。

LPM组件导出的主要方法有：

- 添加LPM规则：LPM规则作为输入参数。如果表中没有存在相同前缀的规则，则将新规则添加到LPM表中。如果表中已经存在具有相同前缀的规则，则会更新规则的下一跳。当没有可用的规则空间时，返回错误。
- 删除LPM规则：LPM规则的前缀作为输入参数。如果具有指定前缀的规则存在于LPM表中，则会被删除。
- LPM规则查找：32位Key作为输入参数。该算法用于选择给定Key的最佳匹配的LPM规则，并返回该规则的下一跳。在LPM表中具有多个相同32位Key的规则的情况下，算法将最高深度的规则选为最佳匹配规则（最长前缀匹配），这意味着该规则Key和输入的Key之间具有最高有效位的匹配。

4.14.2 实现细节

目前的实现使用DIR-24-8算法的变体，可以改善内存使用量，以提高LPM查找速度。该算法允许以典型的单个存储器读访问来执行查找操作。在统计上看，即便是不常出现的情况，当即最佳匹配规则的深度大于24时，查找操作也仅需要两次内存读取访问。因此，特定存储器位置是否存在于处理器高速缓存中将很大程度上影响LPM查找操作的性能。

主要数据结构使用以下元素构建：

- 一个2^24个条目的表。
- 多个表（RTE_LPM_TBL8_NUM_GROUPS），每个表有2 ^ 8个条目。

第一个表，称为tbl24，使用要查找的IP地址的前24位进行索引；而第二个表，称为tbl8使用IP地址的最后8位进行索引。这意味着根据输入数据包的IP地址与存储在tbl24中的规则进行匹配的结果，我们可能需要在第二级继续查找过程。

由于tbl24的每个条目都可以指向tbl8，理想情况下，我们将具有2 ^ 24 tbl8，这与具有2 ^ 32个条目的单个表占用空间相同。因为资源限制，这显然是不可行的。相反，这种组织方法就是利用了超过24位的规则是非常罕见的这一特定。通过将这个过程分为两个不同的表/级别并限制tbl8的数量，我们可以大大降低内存消耗，同时保持非常好的查找速度（大部分时间仅一个内存访问）。

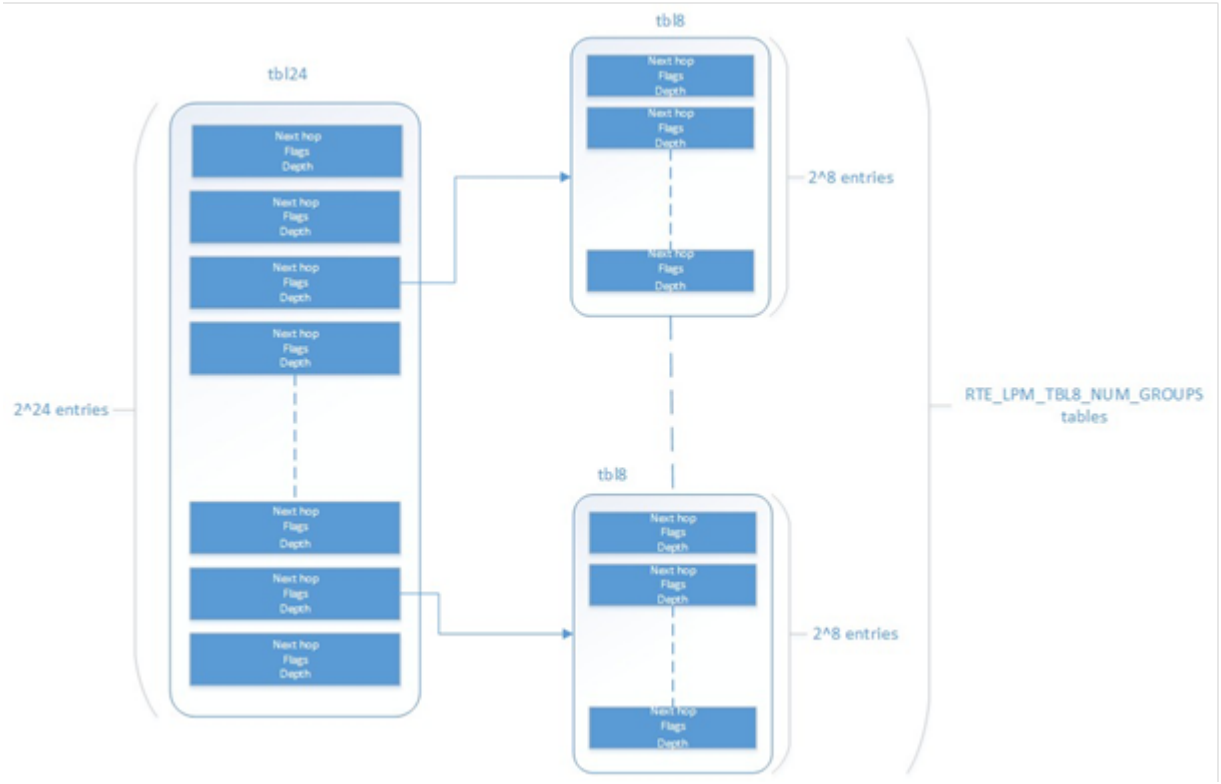


Fig. 4.42: Table split into different levels

tbl24中的条目包含以下字段：

- 下一跳，或者下一级查找表tbl8的索引值。
- 有效标志。
- 外部条目标志。
- 规则深度。

第一个字段可以包含指示查找过程应该继续的tbl8的数字，或者如果已经找到最长的前缀匹配，则可以包含下一跳本身。两个标志字段用于确定条目是否有效，以及搜索过程是否分别完成。规则的深度或长度是存储在特定条目中的规则的位数。

tbl8中的条目包含以下字段：

- 下一跳。
- 有效标志。
- 有效组。
- 深度。

下一跳和深度包含与tbl24中相同的信息。两个标志字段显示条目和表分别是否有效。

其他主要数据结构是包含有关规则（IP和下一跳）的主要信息的表。这是一个更高级别的表，用于不同的东西：

- 在添加或删除之前，检查规则是否已经存在，而无需实际执行查找。
- 删除时，检查是否存在包含要删除的规则。这很重要，因为主数据结构必须相应更新。

添加

添加规则时，存在不同的可能性。如果规则的深度恰好是24位，那么：

- 使用规则（IP地址）作为tbl24的索引。
- 如果条目无效（即它不包含规则），则将其下一跳设置为其值，将有效标志设置为1（表示此条目正在使用中），并将外部条目标志设置为0（表示查找 此过程结束，因为这是匹配的最长的前缀）。

如果规则的深度正好是32位，那么：

- 使用规则的前24位作为tbl24的索引。
- 如果条目无效（即它不包含规则），则查找一个空闲的tbl8，将该值的tbl8的索引设置为该值，将有效标志设置为1（表示此条目正在使用中），并将外部 条目标志为1（意味着查找过程必须继续，因为规则尚未被完全探测）。

如果规则的深度是任何其他值，则必须执行前缀扩展。这意味着规则被复制到所有下一级条目（只要它们不被使用），这也将导致匹配。

作为一个简单的例子，我们假设深度是20位。这意味着有可能导致匹配的IP地址的前24位的2 ^（24 - 20）= 16种不同的组合。因此，在这种情况下，我们将完全相同的条目复制到由这些组合索引的每个位置。

通过这样做，我们确保在查找过程中，如果存在与IP地址匹配的规则，则可以在一个或两个内存访问中找到，具体取决于是否需要移动到下一个表。前缀扩展是该算法的关键之一，因为它通过添加冗余来显着提高速度。

查询

查找过程要简单得多，速度更快。在这种情况下：

- 使用IP地址的前24位作为tbl24的索引。如果该条目未被使用，那么这意味着我们没有匹配此IP的规则。如果它有效并且外部条目标志设置为0，则返回下一跳。
- 如果它是有效的并且外部条目标志被设置为1，那么我们使用tbl8索引来找出要检查的tbl8，并且将该IP地址的最后8位作为该表的索引。类似地，如果条目未被使用，那么我们没有与该IP地址匹配的规则。如果它有效，则返回下一跳。

规则数目的限制

规则数量受到诸多不同因素的限制。第一个是规则的最大数量，这是通过API传递的参数。一旦达到这个数字，就不可能再添加任何更多的规则到路由表，除非有一个或多个删除。

第二个因素是算法的内在限制。如前所述，为了避免高内存消耗，tbl8的数量在编译时间有限（此值默认为256）。如果我们耗尽tbl8，我们将无法再添加任何规则。特定路由表中需要多少路由表是很难提前确定的。

只要我们有一个深度大于24的新规则，并且该规则的前24位与先前添加的规则的前24位不同，就会消耗tbl8。如果相同，那么新规则将与前一个规则共享相同的tbl8，因为两个规则之间的唯一区别是在最后一个字节内。

默认值为256情况下，我们最多可以有256个规则，长度超过24位，且前三个字节都不同。由于长度超过24位的路由不太可能，因此在大多数设置中不应该是一个问题。即便如此，tbl8的数量也可以通过设置更改。

用例：IPv4转发

LPM算法用于实现IPv4转发的路由器所使用的无类别域间路由（CIDR）策略。

References

- RFC1519 Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy http://www.ietf.org/rfc/rfc1519
- Pankaj Gupta, Algorithms for Routing Lookups and Packet Classification, PhD Thesis, Stanford University, 2000 (http://klamath.stanford.edu/~pankaj/thesis/ thesis_1sided.pdf)

4.15 LPM6库

LPM6（LPM for IPv6）库组件实现了128位Key的最长前缀匹配表查找方法，该方法通常用于在IPv6转发应用程序中找到最佳匹配路由。

4.15.1 LPM6 API概述

LPM6库主要配置参数有：

- 支持的LPM规则最大数目：这定义了保存规则的表的大小，也就是最多可以添加的规则数目。
- tbl8的数量：tbl8是trie的一个节点，是LPM6算法的基础。

tbl8与您可以拥有的规则数量相关，但无法准确预测持有特定数量规则所需的空间，因为它强烈依赖于每个规则的深度和IP地址。一个tbl8消耗1 kb的内存。作为推荐，65536个tbl8应该足以存储数千个IPv6规则，但可能因情况而异。

LPM前缀由一对参数（128位Key，深度）表示，深度范围为1到128。LPM规则由LPM前缀和与前缀相关联的一些用户数据表示。该前缀作为LPM规则的唯一标识符。在当前实现中，用户数据为21位长，称为“下一跳”，对应于其主要用途，用于存储路由表条目中下一跳的ID。

为LPM组件导出的主要方法有：

- 添加LPM规则：LPM规则作为输入参数。如果表中没有存在相同前缀的规则，则将新规则添加到LPM表中。如果表中已经存在具有相同前缀的规则，则会更新规则的下一跳。当没有可用空间时返回错误。
- 删除LPM规则：LPM前缀作为输入参数。如果具有指定前缀的规则存在于LPM表中，则会被删除。
- 查找LPM规则：128位Key作为输入参数。该算法选择代表给定Key的最佳匹配的规则，并返回该规则的下一跳。在LPM表中存在多个具有相同128位Key值的规则的情况下，算法选择最高深度的规则作为最佳匹配规则，这意味着该规则在输入键和规则Key之间具有最高有效位数匹配。

实现细节

这个实现是用IPv4的算法做的修改（参见IPv4 LPM实现细节）。在这种情况下，不是使用两级表，而是使用一级的tbl24和14级的tbl8。

该实现可以看作是一个Multi-bit trie，在每个级别上检查的步长或位数根据级别有所不同。具体来说，在根节点检查24位，剩下的104位以8位的组进行检查。这意味根据添加到表中的规则，该trie最多具有14个级。

该算法允许用户直接通过存储器访问操作来执行规则查找，存储器访问次数直接取决于规则长度，以及在数据结构中是否存在具有较大深度的其他规则和相同的Key。它可以在1到14次访存操作之间变化，IPv6中最常用的长度的平均值为5次访问操作。 主要数据结构使用以下元素构建：

- 一个有224个条目的表
- 具有28个条目的表，表的数目由API配置

第一个表称为tbl24，使用要查找的IP地址的前24位进行索引，其余表称为tbl8，使用IP地址的其余字节进行索引，大小为8位。这意味着尝试将输入数据包的IP地址与存储在tbl24或后续tbl8中的规则进行匹配的结果，我们可能需要在较深级别的树中继续查找过程。

类似于IPv4算法中的限制，为了存储所有可能的IPv6规则，我们需要一个具有2 ^ 128个条目的表。 由于资源限制，这显然是不可行的。

通过将查找过程分成不同的表/级别并限制tbl8的数量，我们可以大大减少内存消耗，同时保持非常好的查找速度（每级一个内存访问）。

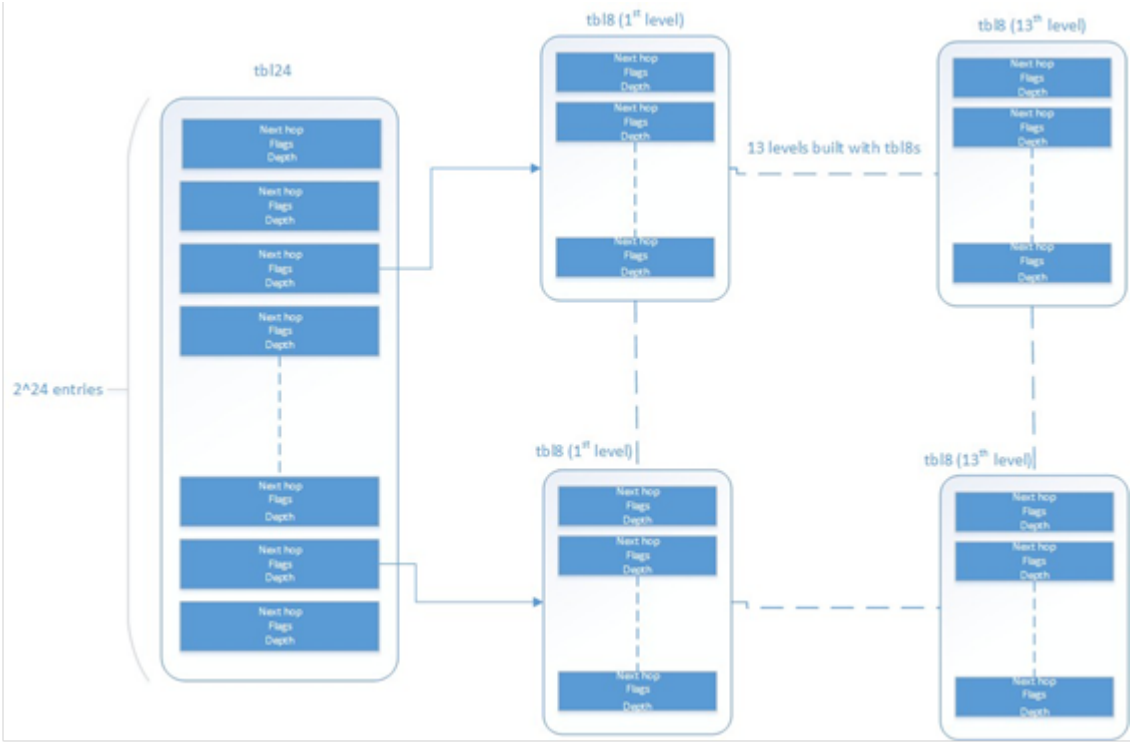


Fig. 4.43: Table split into different levels

表中的条目包含以下字段：

- 下一跳信息或者tbl8索引
- 规则深度

- 有效标志
- 有效组标志
- 外部条目标志

第一个字段可以包含指示查找过程应该继续的tbl8的索引，或者如果已经找到最长的前缀匹配，则可以包含下一跳本身。规则的深度或长度是存储在特定条目中的规则的位数。标志位用于确定条目/表是否有效以及搜索过程是否分别完成。 两种类型的表共享相同的结构。

另一个主要数据结构是一个包含规则（IP，下一跳和深度）的主要信息的表。这是一个更高级别的表，用于不同的目的：

- 在添加或删除之前，检查规则是否已经存在，而无需实际执行查找。

删除时，检查是否存在包含要删除的规则是很重要的，因为主数据结构必须相应更新。

添加

添加规则时存在不同的可能性。如果规则的深度恰好是24位，那么：

- 添加规则时存在不同的可能性。如果规则的深度恰好是24位，那么：
- 如果条目无效（即表中原来不包含规则），则将其下一跳设置为其值，将有效标志设置为1（表示此条目正在使用中），并将外部条目标志设置为0（表示查找过程结束，因为这是匹配的最长的前缀）。

如果规则的深度大于24位，但倍数为8，则：

- 使用规则（IP地址）作为tbl24的索引。
- 如果条目无效（即它不包含规则），则查找一个空闲的tbl8，将该值的tbl8的索引设置为该值，将有效标志设置为1（表示此条目正在使用中），并将外部条目标志为1（意味着查找过程必须继续，因为规则尚未被完全探测）。
- 使用规则的下8位作为下一个tbl8的索引。
- 重复该过程，直到达到正确级别的tbl8（取决于深度），并将其填充到下一跳，将下一个条目标志设置为0。

如果规则的深度是其他值，则必须执行前缀扩展。这意味着规则被复制到所有条目（尽管它们不被使用）以实现致匹配。

举一个简单的例子，我们假设深度是20位。这意味着有可能导致匹配的IP地址的前24位的2 ^（24-20）= 16种不同的组合。因此，在这种情况下，我们将完全相同的条目复制到由这些组合索引的每个位置。

通过这样做，我们确保在查找过程中，如果存在与IP地址匹配的规则，则最多可以在14个内存访问中找到，具体取决于需要移动到下一个表的次数。前缀扩展是该算法的关键之一，因为它通过添加冗余显著提高速度。

前缀扩展可以在任何级别执行。因此，例如，深度是34位，它将在第三级（第二个基于tbl8的级别）执行。

查询

查找过程要简单得多，速度更快。在这种情况下：

- 使用IP地址的前24位作为tbl24的索引。如果该条目未被使用，那么这意味着我们没有匹配此IP的规则。如果它有效并且外部条目标志设置为0，则返回下一跳。
- 如果它有效并且外部条目标志被设置为1，那么我们使用tbl8索引来找出要检查的tbl8，并且将该IP地址的下一个8位作为该表的索引。类似地，如果条目未被使用，那么我们没有与该IP地址匹配的规则。如果它是有效的，那么检查外部条目标志以检查新的tbl8。

- 重复该过程，直到找到无效条目（查找未命中）或外部条目标志设置为0的有效条目。在后一种情况下返回下一跳。

规则数目限制

有不同的因素限制可以添加的规则数量。第一个是规则的最大数量，这是通过API传递的参数。一旦达到这个数字，就不可能再添加任何更多的规则到路由表，除非有一个或多个删除。

第二个限制是可用的tbl8数量。如果我们耗尽tbl8s，我们将无法再添加任何规则。很难提前确定其中有多少是特定的路由表所必需的。

在该算法中，单个规则可以消耗的tbl8的最大数量为13，这是级别数减1，因为前三个字节在tbl24中被解析。然而：

- 通常，在IPv6上，路由不超过48位，这意味着规则通常需要3个tbl8。

如在LPM for IPv4算法中所解释的，根据它们的第一个字节是多少，很可能会有几个规则共享一个或多个tbl8。如果它们共享相同的前24位，例如，第二级的tbl8将被共享。这可能会在更深的级别再次发生，所以有效的是，如果两个48位长的规则在最后一个字节中唯一的区别就可能使用相同的三个tbl8。

由于其对内存消耗的影响以及可以添加到LPM表中的数量或规则，tbl8的数量是在该版本的算法中通过API暴露给用户的参数。一个tbl8消耗1KB的内存。

4.15.2 用例：IPv6转发

LPM算法用于实现实现IP转发的路由器所使用的无类别域间路由（CIDR）策略。

4.16 报文分发库

DPDK报文分发器是一种库，用于在一次 操作中获取单个数据包，以支持流量的动态负载均衡。当使用这个库时，需要考虑两种角色的逻辑核：首先是负责负载均衡及分发数据包的分发逻辑核，另一个是一组工作逻辑核，负责接收来自分发逻辑核的数据包并对其进行操作。

操作模式如下图所示：

在报文分发器库中有两种API操作模式：一种是使用32bit的flow_id，一次向一个worker发送一个报文；另一种优化模式是一次性最多发送8个数据包给worker，使用15bit的flow_id。该模式由 `rte_distributor_create()` 函数中的类型字段指定。

4.16.1 分发逻辑核操作

分发逻辑核执行了大部分的处理以确保数据包在worker之间公平分发。分发逻辑核的运作情况如下：

1. 分发逻辑核的lcore线程通过调用 `rte_distributor_process()` 来获取数据包。
2. 所有的worker lcore与distributor lcore共享一条缓存线行，以便在worker和distributor之间传递消息和数据包。执行API调用将轮询所有worker的缓存行，以查看哪些worker正在请求数据包。
3. 当有worker请求数据包时，distributor从第一步中传过来的一组数据包中取出数据包，并将其分发给worker。它检查每个数据包中存储在mbuf中RSS哈希字段中的“tag”，并记录每个worker正在处理的tag。
4. 如果输入报文集中的下一个数据包有一个已经被worker处理的tag，则该数据包将排队等待worker的处理，并在下一个worker请求数据包时，优先考虑其他的数据包。这可以确保不会并发处理具有相同tag的两个报文，并且，具有相同tag的两个报文按输入顺序被处理。

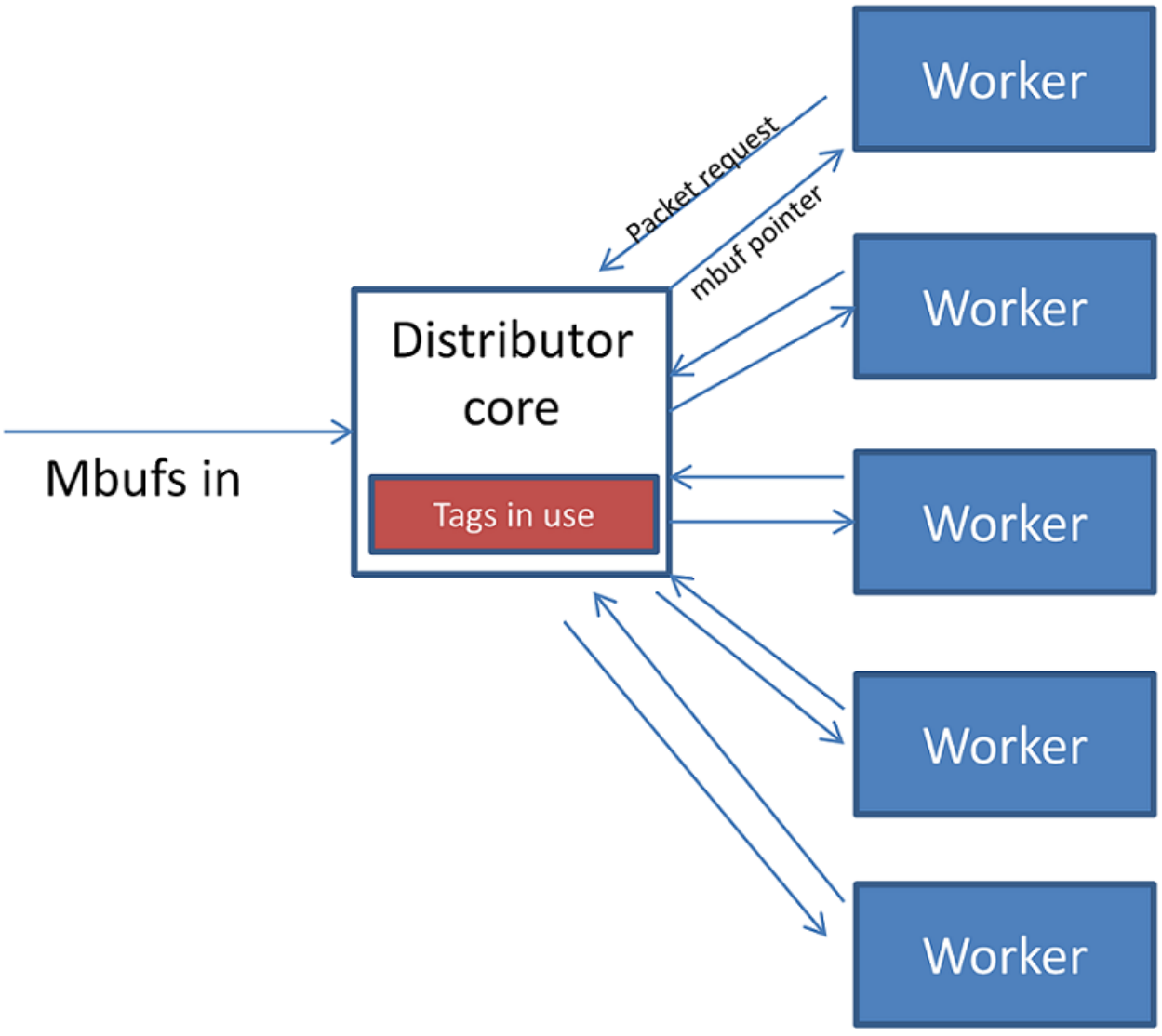


Fig. 4.44: Packet Distributor mode of operation

5. 一旦传递给执行API的所有报文已经分发给worker，或者已经排队等待给定tag的worker处理，则执行API返回给调用者。

Distributor lcore可以使用的其他功能有：

- rte_distributor_returned_pkts()
- rte_distributor_flush()
- rte_distributor_clear_returns()

其中最重要的API调用是 `rte_distributor_returned_pkts()` ，它只能在调用进程API的lcore上调用。它将所有worker core完成处理的所有数据包返回给调用者。在这组返回的数据包中，共享相同标签的所有数据包将按原始顺序返回。

NOTE: 如果worker lcore在内部缓存数据包进行批量传输，则共享tag的数据包可能会出现故障。一旦一个worker lcore请求一个新的数据包，distributor就会假定它已经完成了先前的数据包，因此具有相同tag的附加数据包可以安全地分配给其他worker，然后他们可能会更早地刷新缓冲的数据包，使数据包发生故障。

NOTE: 对于不共享公共数据包tag的数据包，不提供数据包排序保证。

使用上述执行过程及returned_pkts API，可以使用以下应用程序工作流，同时允许维护由tag识别的数据包流中的数据包顺序。

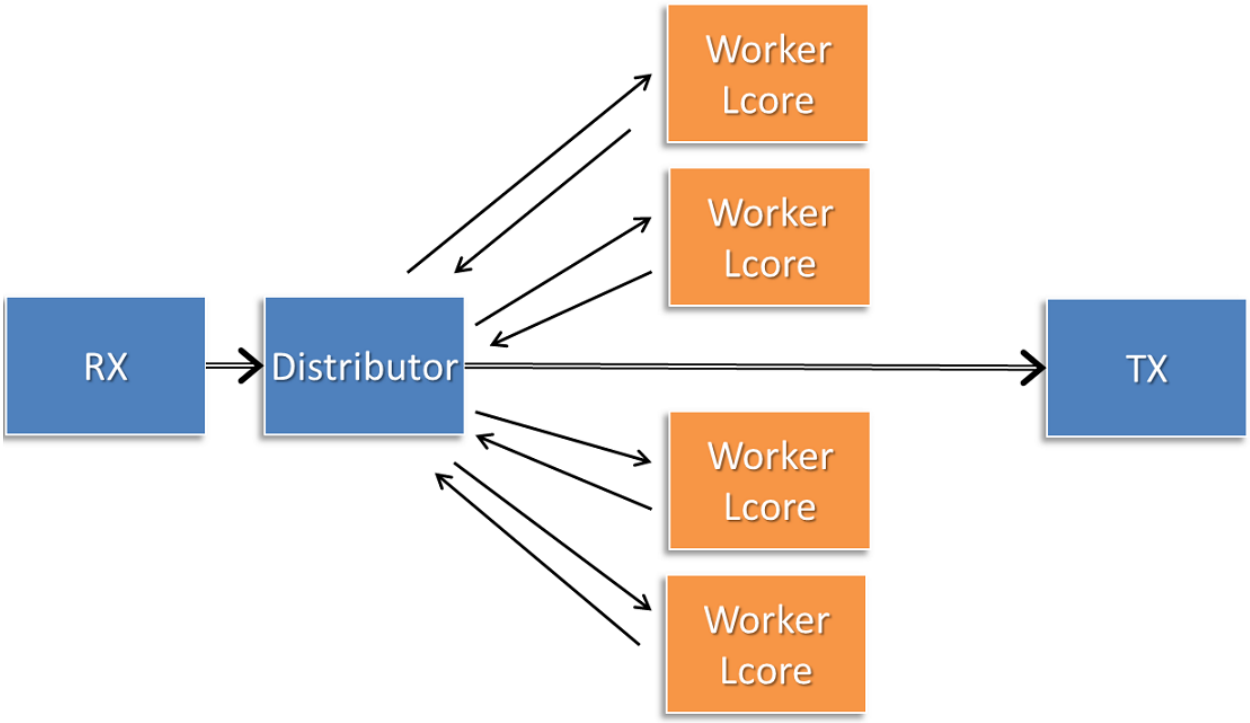


Fig. 4.45: Application workflow

之前提到的flush和clear_returns API调用可能不太用于进程和returned_pkts APIS，并且主要用于帮助对库进行单元测试。可以在DPDK API参考文档中找到这些功能及其用途的描述。

4.16.2 Worker Operation

Worker lcore是对distributor分发的数据包进行实际操作的lcore。 Worker调用rte_distributor_get_pkt() API在完成处理前一个数据包时请求一个新的数据包。前一个数据包应通过将其作为最终参数传递给该API调用而返

回给分发器组件。

有时候可能需要改变worker lcore的数量，这取决于业务负载，即在较轻的负载时节省功率，可以worker通过调用rte_distributor_return_pkt()接口停止处理报文，以指示在完成当前数据包处理后，不需要新的数据包。

4.17 排序器库

重新排序库提供了一种根据序列号重新排序mbufs的机制。

4.17.1 操作

重新排序库本质上是一个重新排列mbufs的缓冲区。用户将乱序的mbufs插入到重新排序缓冲区中，并从中输出顺序mbufs。

在给定的时间，重新排序缓冲区包含序列号在序列窗口内的mbufs。顺序窗口由缓冲区配置的可以维护的最小序列号和条目数决定。例如，给定具有200个条目并且最小序列号为350的重排序缓冲器，序列窗口分别具有350和550的低和高限制。

当插入mbufs时，重新排序库根据插入的mbuf的序列号区分valid，late和early的mbuf:

- valid: 序列号在有序窗口的限制内。
- late: 序列号在窗口限制外，小于下限。
- early: 序列号在窗口限制外，大于上限。

重新排序缓冲区直接返回late mbufs，并尝试适应early mbufs。

4.17.2 实现细节

重新排序库被实现为一对缓冲区，称为Order buffer和Ready buffer。

在插入调用时，valid mbufs将直接插入到Order buffer中，late mbufs将直接返回给用户错误。

对于early buffer的情况，重排序缓冲区将尝试移动窗口（递增最小序列号），以使mbuf成为有效的一个。为此，Order buffer中的mbufs被移动到就Ready buffer中。任何尚未到达的mbufs都被忽略，且将变成late mbufs。这意味着只要Ready buffer中有空间，窗口将被移动以适应early mbufs，否则将在重新排序窗口之外。

例如，假设我们有一个具有350个最小序列号的200个条目的缓冲区，并且我们需要插入一个具有565序列号的early mbuf。这意味着我们需要移动窗口至少15个位置来容纳mbuf。只要在Ready buffer中有空间，重新排序缓冲区将尝试将至少在Order buffer中的下一个15个槽中的mbufs移动到Ready buffer区。在这一点上的顺序缓冲区中的任何间隙将被跳过，并且这些数据包在报文到达时将被报告为late buffer的数据包。将数据包移动到Ready buffer的过程继续超出所需的最小值，直到遇到了缓冲区中的间隙，即缺少mbuf。

排出mbufs时，重新排序缓冲区首先返回Ready buffer中的mbufs，然后从Order buffer返回到尚未到达的mbufs。

4.17.3 用例：报文分发

使用DPDK数据包分发器的应用程序可以利用重新排序库以与它们相同的顺序传送数据包。

基本的报文分配器用例将由具有多个worker cors的分配器组成。worker对数据包的处理不能保证按顺序进行，因此可以使用重排序缓冲区来尽可能多地重排数据包。

在这种情况下，distributor将序列号分配给mbufs，然后再将其发送给工作人员。随着worker完成处理数据包，distributor将这些mbufs插入重排序缓冲区，最后传输排出的mbufs。

4.18 IP分片及重组库

IP分段和重组库实现IPv4和IPv6报文的分片和重组。

4.18.1 报文分片

报文分段例程将输入报文划分成多个分片。rte_ipv4_fragment_packet()和rte_ipv6_fragment_packet()函数都假定输入mbuf数据指向报文的IP报头的开始（即L2报头已经被剥离）。为了避免复制实际数据包的数据，使用零拷贝技术（rte_pktmbuf_attach）。对于每个片段，将创建两个新的mbuf：

- Direct mbuf – mbuf将包含新片段的L3头部。
- Indirect mbuf – 源数据包附加到mbuf。数据字段指向原始数据包数据的附加数据偏移量开始处。

然后将L3头部从原始mbuf复制到“direct”mbuf并更新以反映新的碎片状态。 请注意，对于IPv4，不会重新计算头校验和，其值设置为零。

最后，通过mbuf的下next字段将每个片段的“direxT”和“indirect”mbuf链接在一起，以构成新片段的数据包。

调用者可以明确指定哪些mempools应用于从中分配“direct”和“indirect”mbufs。

有关direct和indirect mbufs的信息，请参阅 直接及间接 *Buffers* 。

4.18.2 报文重组

IP分片表

报文分片表中维护已经接收到的数据包片段的信息。

每个IP数据包由三个字段：<源IP地址>，<目标IP地址>，唯一标识。

请注意，报文分片表上的所有更新/查找操作都不是线程安全的。因此，如果不同的执行上下文（线程/进程）要同时访问同一个表，那么必须提供一些外部同步机制。

每个表项可以保存最多RTE_LIBRTE_IP_FRAG_MAX（默认值为4）片段的数据包的信息。

代码示例，演示了创建新的片段表：

```
frag_cycles = (rte_get_tsc_hz() + MS_PER_S - 1) / MS_PER_S * max_flow_ttl;
bucket_num = max_flow_num + max_flow_num / 4;
frag_tbl = rte_ip_frag_table_create(max_flow_num, bucket_entries, max_flow_num, frag_
↪cycles, socket_id);
```

内部片段表是一个简单的哈希表。 基本思想是使用两个哈希函数和 关联性。 这为每个Key在散列表中提供了2 可能的位置。当发生冲突并且所有2 * 都被占用时，ip_frag_tbl_add()只是返回失败，而不是将现有的Key重新插入到另外的位置。

此外，驻留在表中的条目如果比更长，被认为是无效的，可以被新的条目删除/替换。

请注意，重新组合需要分配很多mbuf。在任何给定时间（2 bucket_entries RTE_LIBRTE_IP_FRAG_MAX * <每个数据包的最大mbufs数>>）可以存储在等待剩余片段的Fragment Table中。

报文重组

报文分组处理和重组由rte_ipv4_frag_reassemble_packet()/rte_ipv6_frag_reassemble_packet()完成。它们返回一个指向有效mbuf的指针，它包含重新组合的数据包，或者返回NULL（如果数据包由于某种原因而无法重新组合）。

这些功能包括：

1. 搜索片段表，输入数据包的。
2. 如果找到该条目，则检查该条目是否已经超时。如果是，则释放所有以前收到的碎片，并从条目中删除有关它们的信息。
3. 如果没有找到这样的Key的条目，那么尝试通过以下两种方法之一创建一个新的：
   (a) 用作空条目。
   (b) 删除一个超时条目，与它mbufs关联的空闲mbufs，并在其中存储一个带有指定键的新条目。
4. 使用新的片段信息更新条目，并检查是否可以重新组合数据包（数据包的条目包含所有片段）。
   (a) 如果是，则重新组装数据包，将表的条目标记为空，并将重新组装的mbuf返回给调用者。
   (b) 如果否，则向调用者返回一个NULL。

如果在分组处理的任何阶段遇到错误（例如：不能将新条目插入片段表或无效/超时片段），则该函数将释放所有与分组片段相关联的标记表条目 作为无效并将NULL返回给调用者。

调试日志及统计收集

RTE_LIBRTE_IP_FRAG_TBL_STAT配置宏用于控制片段表的统计信息收集。默认情况下未启用。

RTE_LIBRTE_IP_FRAG_DEBUG控制IP片段处理和重组的调试日志记录。默认情况下禁用。请注意，在日志记录包含大量详细信息时，会减慢数据包处理速度，并可能导致丢失大量数据包。

4.19 Librte_pdump库

`librte_pdump` 库为DPDK中的数据包捕获提供了一个框架。该库将Rx和Tx mbufs的完整复制到新的mempool，因此会降低应用程序的性能，故建议只使用该库进行调试。

该库提供以下API来初始化数据包捕获框架，启用或禁用数据包捕获，或者对其进行反初始化：

- `rte_pdump_init()`: 初始化数据包捕获框架。
- `rte_pdump_enable()`: 在给定的端口和队列上进行数据包捕获。注意：API中的过滤器选项是用于未来增强功能的占位符。
- `rte_pdump_enable_by_deviceid()`: 启用在给定设备ID（vdev名称或pci地址）和队列上的数据包捕获。 注意：API中的过滤器选项是用于未来增强功能的占位符。
- `rte_pdump_disable()`: 禁用给定端口和队列上的数据包捕获。
- `rte_pdump_disable_by_deviceid()`: 禁用给定设备ID（vdev名称或pci地址）和队列上的数据包捕获。
- `rte_pdump_uninit()`: 反初始化数据包捕获框架。
- `rte_pdump_set_socket_dir()`: 设置服务器和客户端套接字路径。注意：此API不是线程安全的。

4.19.1 操作

librte_pdump库适用于客户端/服务器型号。服务器负责启用或禁用数据包捕获，客户端负责请求启用或禁用数据包捕获。

数据包捕获框架作为程序初始化的一部分，在pthread中创建pthread和服务器套接字。调用框架初始化的应用程序将创建服务器套接字，可能是在应用程序传入的路径，也可能是默认路径（root用户的/var/run/.dpdk，非root用户～/.dpdk）下创建。

请求启用或禁用数据包捕获的应用程序将在应用程序传入的路径下或默认路径（root用户的/var/run/.dpdk，非root用户～/.dpdk）下创建客户机套接字，用户将请求发送到服务器。服务器套接字将监听用于启用或禁用数据包捕获的客户端请求。

4.19.2 实现细节

库API rte_pdump_init()通过创建pthread和服务器套接字来初始化数据包捕获框架。pthread上下文中的服务器套接字将监听客户端请求以启用或禁用数据包捕获。

库API rte_pdump_enable()和rte_pdump_enable_by_deviceid()启用数据包捕获。每次调用这些API时，库创建一个单独的客户端套接字，生成“pdump enable”请求，并将请求发送到服务器。在套接字上监听的服务器将通过对给定的端口或设备ID和队列组合的以太网Rx/TX注册回调函数来接收请求并启用数据包捕获。然后，服务器将镜像数据包到新的mempool并将它们入队到客户端传递给这些API的rte_ring。服务器还将响应发送回客户端，以了解处理过的请求的状态。从服务器收到响应后，客户端套接字关闭。

库API rte_pdump_disable()和rte_pdump_disable_by_deviceid()禁用数据包捕获。每次调用这些API时，库会创建一个单独的客户端套接字，生成“pdump disable”请求，并将请求发送到服务器。正在监听套接字的服务器将通过对给定端口或设备ID和队列组合的以太网RX和TX删除回调函数来执行请求并禁用数据包捕获。服务器还将响应发送回客户端，以了解处理过的请求的状态。从服务器收到响应后，客户端套接字关闭。

库API rte_pdump_uninit()通过关闭pthread和服务器套接字来初始化数据包捕获框架。

库API rte_pdump_set_socket_dir()根据API的类型参数将给定路径设置为服务器套接字路径或客户端套接字路径。如果给定路径为NULL，则将选择默认路径（即root用户的/var/run/.dpdk或非root用户的～/.dpdk）。如果服务器套接字路径与默认路径不同，客户端还需要调用此API来设置其服务器套接字路径。

4.19.3 用例:抓包

DPDK应用程序/pdump工具是基于此库开发的，用于捕获DPDK中的数据包。用户可以用它来开发自己的数据包捕获工具。

4.20 多进程支持

在DPDK中，多进程支持旨在允许一组DPDK进程以简单的透明方式协同工作，以执行数据包处理或其他工作负载。为了支持此功能，已经对核心的DPDK环境抽象层（EAL）进行了一些增加。

EAL已被修改为允许不同类型的DPDK进程产生，每个DPDK进程在应用程序使用的hugepage内存上具有不同的权限。现在可以指定两种类型的进程:

- primary processes, 可以初始化，拥有共享内存的完全权限
- secondary processes, 不能初始化共享内存，但可以附加到预初始化的共享内存并在其中创建对象。

独立DPDK进程是primary processes，而secondary processes只能与主进程一起运行，或者主进程已经为其配置了hugepage共享内存。

为了支持这两种进程类型以及稍后描述的其他多进程设置，EAL还提供了两个附加的命令行参数:

- `--proc-type`: 用于将给定的进程实例指定为primary processes或secondary processes DPDK实例。
- `--file-prefix`: 以允许不希望协作具有不同存储器区域的进程。

DPDK提供了许多示例应用程序，演示如何可以一起使用多个DPDK进程。这些用例在《DPDK Sample Application用户指南》中的"多进程示例应用"一章中有更详尽的记录。

4.20.1 内存共享

使用DPDK的多进程应用程序工作的关键要素是确保内存资源在构成多进程应用程序的进程之间正确共享。一旦存在可以通过多个进程访问的共享存储器块，则诸如进程间通信（IPC）的问题就变得简单得多。

在独立进程或者primary processes启动时，DPDK向内存映射文件中记录其使用的内存配置的详细信息，包括正在使用的hugepages，映射的虚拟地址，存在的内存通道数等。当secondary processes启动时，这些文件被读取，并且EAL在secondary processes中重新创建相同的内存配置，以便所有内存区域在进程之间共享，并且所有指向该内存的指针都是有效的，并且指向相同的对象。

Note: 有关Linux内核地址空间布局随机化（ASLR）如何影响内存共享的详细信息参考 多进程限制 。

Fig. 4.46: Memory Sharing in the DPDK Multi-process Sample Application

EAL还支持自动检测模式（由EAL -proc-type = auto标志设置），如果主实例已经在运行，则DPDK进程作为辅助实例启动。

4.20.2 部署模式

对称/对等进程

DPDK多进程支持可用于创建一组对等进程，每个进程执行相同的工作负载。该模型相当于具有多个线程，每个线程都运行相同的主循环功能，如大多数提供的DPDK示例应用程序中所完成的一样。 在此模型中，应使用–proc-type = primary EAL标志生成第一个生成的进程，而所有后续实例都应使用–proc-type = secondary标志生成。

simple_mp和symmetric_mp示例应用程序演示了此模型的用法。它们在《DPDK Sample Application用户指南》中"多进程示例应用"一章中有描述。

非对称/非对等进程

可用于多进程应用程序的替代部署模型是具有单个primary process实例，充当负载均衡器或distributor，在作为secondary processes运行的worker或客户机线程之间分发接收到的数据包。在这种情况下，广泛使用rte_ring对象，它们位于共享的hugepage内存中。

client_server_mp示例应用程序显示此模型用法。在《DPDK Sample Application用户指南》中"多进程示例应用"一章中有描述。

运行多个独立的DPDK应用程序

除了涉及多个DPDK进程的上述情况之外，可以并行运行多个DPDK进程，这些进程都可以独立工作。使用EAL的–file-prefix参数提供对此使用场景的支持。

默认情况下，EAL使用rtemap_X文件名在每个hugetlbfs文件系统上创建hugepage文件，其中X的范围为0到最大的hugepages -1。同样，当以root身份运行（或以非root用户身份运行时为$ HOME / .rte_config），如果文件系统和设备权限为空，则会在每个进程中使用/var/run/.rte_config文件名创建共享配置文件）。以上每个文件名的部分可以使用file-prefix参数进行配置。

除了指定file-prefix参数外，并行运行的任何DPDK应用程序都必须明确限制其内存使用。这通过将-m标志传递给每个进程来指定每个进程可以使用多少hugepage内存（以兆字节为单位）（或通过–socket-mem来指定每个进程可以使用每个套接字的多少hugepage内存）。

Note: 在单台机器上并行运行的独立DPDK实例无法共享任何网络端口。一个进程使用的任何网络端口都应该在其他进程中列入黑名单。

运行多个独立的DPDK应用程序组

以同样的方式，可以在单个系统上并行运行独立的DPDK应用程序，这也可以简单地扩展到并行运行DPDK应用程序的多进程组。在这种情况下，secondary processes必须使用与其共享内存连接的primary process相同的–file-prefix参数。

Note: 并行运行的多个独立DPDK进程的所有限制和问题也适用于此使用场景。

4.20.3 多进程限制

运行DPDK多进程应用程序时存在一些限制。其中一些记录如下：

- 多进程功能要求在所有应用程序中都存在完全相同的hugepage内存映射。Linux安全功能，地址空间布局随机化（ASLR）可能会干扰此映射，因此可能需要禁用此功能才能可靠地运行多进程应用程序。

Warning: 禁用地址空间布局随机化（ASLR）可能具有安全隐患，因此建议仅在绝对必要时才被禁用，并且只有在了解了此更改的含义时。

- 作为单个应用程序运行并使用共享内存的所有DPDK进程必须具有不同的coremask /corelist参数。任何相同的逻辑内核不可能拥有primary和secondary实例或两个secondary实例。尝试这样做可能会导致内存池缓存的损坏等问题。
- 中断的传递，如Ethernet设备链路状态中断，在secondary process中不起作用。所有中断仅在primary process内触发。在多个进程中需要中断通知的任何应用程序都应提供自己的机制，将中断信息从primary process转移到需要该信息的任何secondary process。
- 不支持使用基于不同编译二进制文件运行的多个进程之间的函数指针，因为在一个进程中给定函数的位置可能与其中一个进程的位置不同。这样可以防止librte_hash库在多线程实例中正常运行，因为它在内部使用了一个指向散列函数的指针。

要解决此问题，建议多进程应用程序通过直接从代码中调用散列函数，然后使用rte_hash_add_with_hash()/rte_hash_lookup_with_hash()函数来执行哈希计算，而不是内部执行散列的函数，例如rte_hash_add()/rte_hash_lookup()。

- 根据所使用的硬件和所使用的DPDK进程的数量，可能无法在每个DPDK实例中都使用HPET定时器。可用于Linux 用户空间的HPET comparators的最小数量可能只有一个，这意味着只有第一个primary DPDK进程实例可以打开和mmap/dev/hpet。如果所需DPDK进程的数量超过可用的HPETcomparators数量，则必须使用TSC（此版本中的默认计时器）而不是HPET。

4.21 内核网络接口卡接口

DPDK Kernel NIC Interface（KNI）允许用户空间应用程序访问Linux *控制面。

使用DPDK KNI的好处是：

- 比现有的Linux TUN / TAP接口更快（通过消除系统调用和copy_to_user()/copy_from_user()操作）。
- 允许使用标准Linux网络工具（如ethtool，ifconfig和tcpdump）管理DPDK端口。
- 允许与内核网络堆栈的接口。

使用DPDK内核NIC接口的应用程序的组件如图所示。 Fig. 4.47.

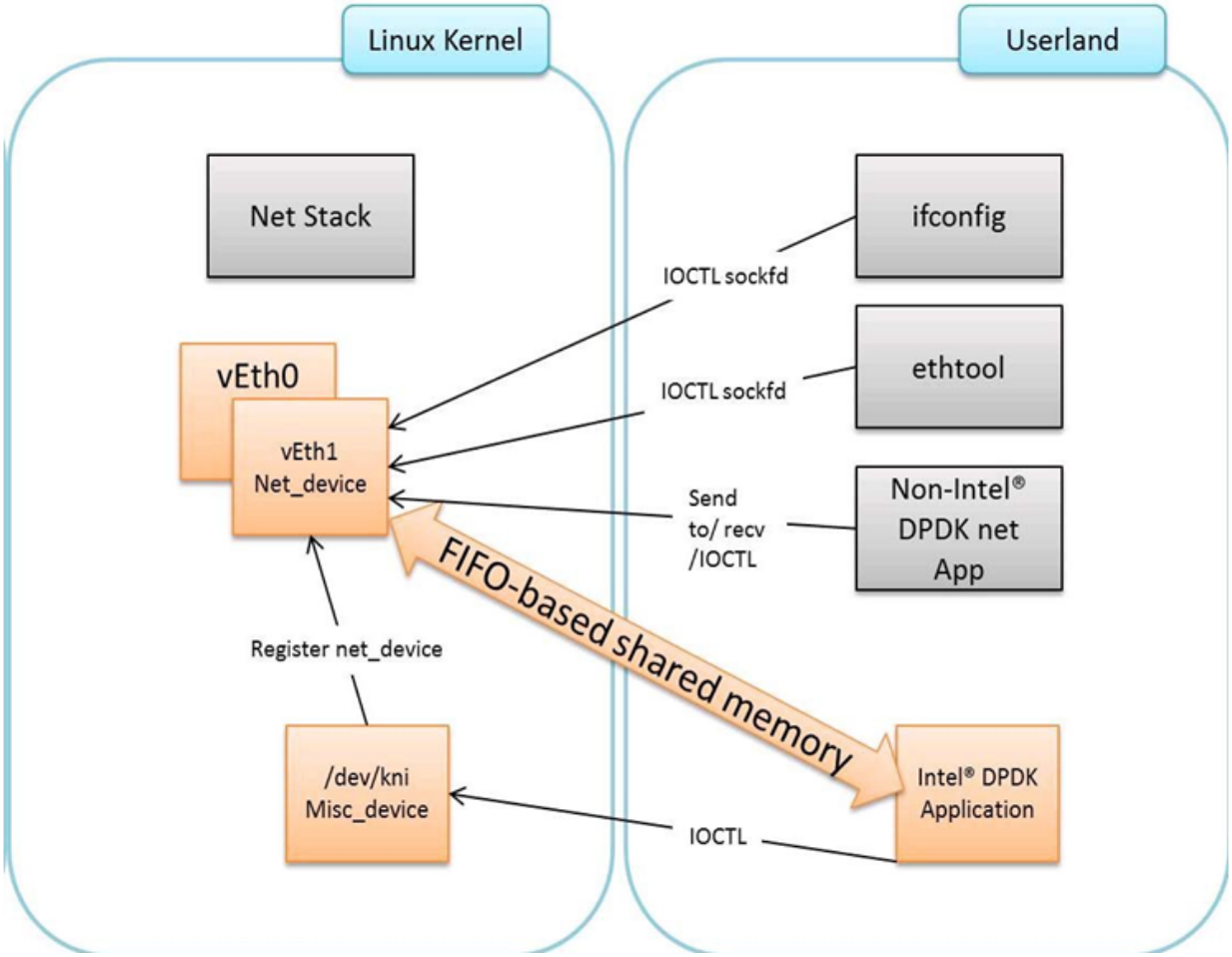


Fig. 4.47: Components of a DPDK KNI Application

4.21.1 DPDK KNI内核模块

KNI内核可加载模块支持两种类型的设备：

- 其他设备：(/dev/kni)
  - 创建网络设备（通过ioctl调用）。

  - 维护所有KNI实例共享的内核线程上下文（模拟网络驱动程序的RX端）。
  - 对于单内核线程模式，维护所有KNI实例共享的内核线程上下文（模拟网络驱动程序的RX端）。
  - 对于多个内核线程模式，为每个KNI实例（模拟新驱动程序的RX侧）维护一个内核线程上下文。
- 网络设备：
  - 通过实现由struct net_device定义的诸如netdev_ops，header_ops，ethtool_ops之类的几个操作提供的Net功能，包括支持DPDK mbufs和FIFO。
  - 接口名称由用户空间提供。
  - MAC地址可以是真正的NIC MAC地址或随机的。

4.21.2 KNI创建及删除

KNI接口由DPDK应用程序动态创建。接口名称和FIFO详细信息由应用程序通过ioctl调用使用rte_kni_device_info结构提供，该结构包含：

- 接口名称。
- 相关FIFO的相应存储器的物理地址。
- Mbuf mempool详细信息，包括物理和虚拟（计算mbuf指针的偏移量）。
- PCI信息。
- Core。

有关详细信息，请参阅DPDK源代码中的rte_kni_common.h。

物理地址将重新映射到内核地址空间，并存储在单独的KNI上下文中。

内核RX线程（单线程和多线程模式）的亲和力由force_bind和core_id配置参数控制。

创建后，DPDK应用程序可以动态删除KNI接口。此外，所有未删除的KNI接口将在杂项设备（DPDK应用程序关闭时）的释放操作中被删除。

4.21.3 DPDK缓冲区流

为了最小化在内核空间中运行的DPDK代码的数量，mbuf mempool仅在用户空间中进行管理。内核模块可以感知mbufs，但是所有mbuf分配和释放操作将仅由DPDK应用程序处理。

Fig. 4.48 shows a typical scenario with packets sent in both directions.

4.21.4 用例: Ingress

在DPDK RX侧，mbuf由PMD在RX线程上下文中分配。该线程将mbuf入队到rx_q FIFO中。 KNI线程将轮询所有KNI活动设备。如果mbuf出队，它将被转换为sk_buff，并通过netif_rx()发送到网络协议栈。必须释放出队的mbuf，将指针返回到free_q FIFO中。

RX线程在相同的主循环中轮询该FIFO，并在出队之后释放mbuf。

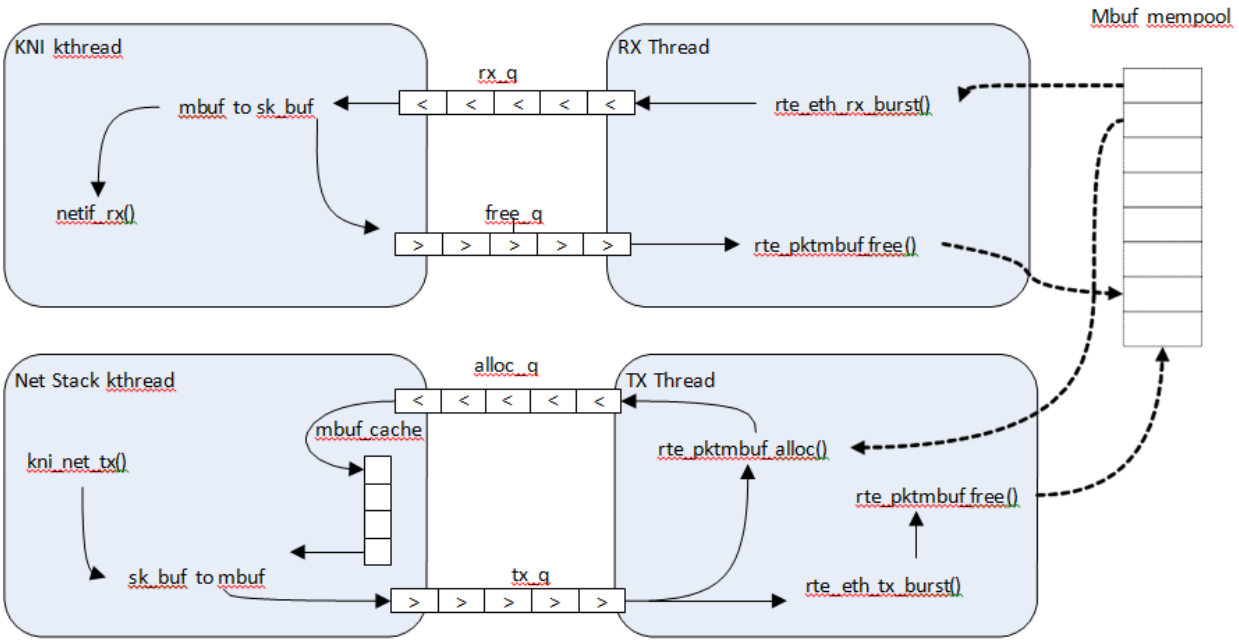


Fig. 4.48: Packet Flow via mbufs in the DPDK KNI

4.21.5 用例: Egress

对于数据包出口，DPDK应用程序必须首先入队几个mbufs才能在内核端创建一个mbuf缓存。

通过调用kni_net_tx()回调，从Linux网络堆栈接收数据包。mbuf出队（因为使用缓存，所以无需等待），并填充了来自sk_buff的数据。然后释放sk_buff，并将mbuf发送到tx_q FIFO。

DPDK TX线程执行mbuf出队，并将其发送到PMD（通过rte_eth_tx_burst()）。然后将mbuf放回缓存中。

4.21.6 以太网工具

Ethtool是Linux专用工具，在内核中具有相应的支持，每个网络设备必须为支持的操作注册自己的回调。目前的实现使用igb / ixgbe修改的Linux驱动程序进行ethtool支持。i40e和VM（VF或EM设备）不支持Ethtool。

4.21.7 链路状态及MTU改变

链路状态和MTU变化是通常通过ifconfig完成的网络接口操作。该请求是从内核端（在ifconfig进程的上下文中）发起的，由用户空间DPDK应用程序处理。应用程序轮询请求，调用应用程序处理程序并将响应返回到内核空间。

应用处理程序可以在创建接口时注册，也可以在运行时再注册/卸载。这提供了多进程方案（其中KNI在primary process中创建，在secondary process中处理回调）的灵活性。约束是单个进程可以注册和处理请求。

4.22 DPDK功能的线程安全

DPDK由几个库组成。这些库中的某些功能可以同时被多个线程安全地调用，而另一部分则不能。 本节介绍开发人员在构建自己的应用程序时考虑这些问题。

DPDK的运行时环境通常是每个逻辑核上的单个线程。但是，在某些情况下，它不仅是多线程的，而且是多进程的。通常，最好避免在在线程和/或进程之间共享数据结构。如果不可能，则执行块必须以线程安全的方式访问数据。可以使用诸如原子操作或锁的机制，这将允许执行块串行操作。但是，这可能会对应用程序的性能产生影响。

4.22.1 快速路径API

在数据面中运行的应用程序对性能敏感，但这些库中的某些函数可能不会多线程并发调用。PMD中的Hash，LPM和mempool库以及RX / TX都是这样的例子。

通过设计，Hash和LPM库线程不安全，不能并行调用，以保持性能。然而，如果需要，开发人员可以在这些库之上添加封装层以提供线程安全性。在所有情况下都不需要锁，并且在哈希和LPM库中，可以在多个线程中并行执行值的查找。但是，当访问单个哈希表或LPM表时，添加，删除或修改值不能不使用锁在多个线程中完成。锁的另一个替代方法是创建这些表的多个实例，允许每个线程自己的副本。

PMD的RX和TX是DPDK应用程序中最关键的方面，建议不要使用锁，因为它会影响性能。但是请注意，当每个线程在不同的NIC队列上执行I/O时，这些功能可以安全地从多个线程使用。如果多个线程在同一个NIC端口上使用相同的硬件队列，则需要锁定或某种其他形式的互斥。

Ring库的实现基于无锁缓冲算法，保持其原有的线程安全设计。此外，它可以为多个或单个消费者/生产者入队/出队操作提供高性能。mempool库基于DPDK无锁ring库，因此也是多线程安全的。

4.22.2 非性能敏感API

在第25.1节描述的性能敏感区域之外，DPDK为大多数其他库提供线程安全的API。例如，malloc和memzone功能可以安全地用于多线程和多进程环境中。

PMD的设置和配置不是性能敏感的，但也不是线程安全的。在多线程环境中PMD设置和配置期间的多次读/写可能会被破坏。由于这不是性能敏感的，开发人员可以选择添加自己的层，以提供线程安全的设置和配置。预计在大多数应用中，网络端口的初始配置将由启动时的单个线程完成。

4.22.3 库初始化

建议DPDK库在应用程序启动时在主线程中初始化，而不是随后在转发线程中初始化。但是，DPDK会执行检查，以确保库仅被初始化一次。如果尝试多次初始化，则返回错误。 在多进程情况下，共享内存的配置信息只能由primary process初始化。此后，primary process和secondary process都可以分配/释放最终依赖于rte_malloc或memzone的任何内存对象。

4.22.4 中断线程

DPDK在轮询模式下几乎完全用于Linux用户空间。对于诸如接收PMD链路状态改变通知的某些不经常的操作，可以在主DPDK处理线程外部的附加线程中调用回调。这些函数回调应避免操作也由普通DPDK线程管理的DPDK对象，如果需要这样做，应用程序就可以为这些对象提供适当的锁定或互斥限制。

4.23 QoS框架

本章介绍了DPDK服务质量（QoS）框架。

4.23.1 支持QoS的数据包水线

具有QoS支持的复杂报文处理流水线的示例如下图所示。

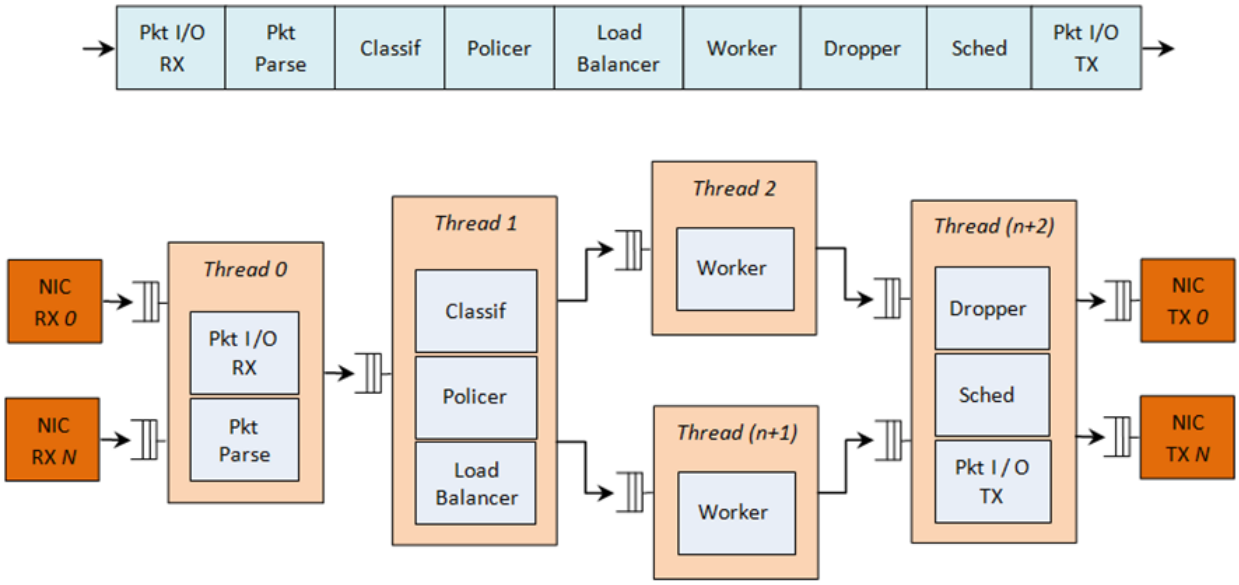


Fig. 4.49: Complex Packet Processing Pipeline with QoS Support

这个水线使用可重复使用的DPDK软件库构建。在这个流程中实现QoS的主要模块有：策略器，缓存器和调度器。下表列出了各块的功能描述。

Table 4.51: Packet Processing Pipeline Implementing QoS

#	Block	Functional Description
1	Packet I/O RX & TX	多个NIC端口的报文接收/传输。用于Intel 1GbE/10GbE NIC的轮询模式驱动程序（PMD）。
2	Packet parser	识别输入数据包的协议栈。检查数据包头部的完整性。
3	Flow classification	将输入数据包映射到已知流量上。使用可配置散列函数（jhash，CRC等）和桶逻辑来处理冲突的精确匹配表查找。
4	Policer	使用srTCM（RFC 2697）或trTCM（RFC2698）算法进行数据包测量。
5	Load Balancer	将输入数据包分发给应用程序worker。为每个worker提供统一的负载。保持流量对worker的亲和力和每个流程中的数据包顺序。
6	Worker threads	客户指定的应用工作负载的占位符（例如IP堆栈等）。
7	Dropper	拥塞管理使用随机早期检测（RED）算法（Sally Floyd-Van Jacobson的论文）或加权RED（WRED）。根据当前调度程序队列的负载级别和报文优先级丢弃报文。当遇到拥塞时，首先丢弃优先级较低的数据包。
8	Hierarchical Scheduler	具有数千（通常为64K）叶节点（队列）的5级分层调度器（级别为：输出端口，子端口，管道，流量类和队列）。实现流量整形（用于子站和管道级），严格优先级（对于流量级别）和加权循环（WRR）（用于每个管道流量类中的队列）。

整个数据包处理流程中使用的基础架构块如下表所示。

Table 4.52: Infrastructure Blocks Used by the Packet Processing Pipeline

#	Block	Functional Description
1	Buffer manager	支持全局缓冲池和专用的每线程缓存缓存。
2	Queue manager	支持水线之间的消息传递。
3	Power saving	在低活动期间支持节能。

水线块到CPU cores的映射可以根据每个特定应用程序所需的性能级别和为每个块启用的功能集进行配置。一些块可能会消耗多个CPU cores（每个CPU core在不同的输入数据包上运行同一个块的不同实例），而另外的几个块可以映射到同一个CPU core。

4.23.2 分层调度

分层调度块（当存在时）通常位于发送阶段之前的TX侧。其目的是根据每个网络节点的服务级别协议（SLA）指定的策略来实现不同用户和不同流量类别的数据包传输。

概述

分层调度类似于网络处理器使用的流量管理，通常实现每个流（或每组流）分组排队和调度。它像缓冲区一样工作，能够在传输之前临时存储大量数据包（入队操作）；由于NIC TX正在请求更多的数据包进行传输，所以这些数据包随后被移出，并且随着分组选择逻辑观察预定义的SLA（出队操作）而交给NIC TX。

分层调度针对大量报文队列进行了优化。当只需要少量的队列时，应该使用消息传递队列而不是这个模块。有关更多详细的讨论，请参阅“Worst Case Scenarios for Performance”。

调度层次

调度层次结构如下图所示。 Fig. 4.51. 层次结构的第一级是以太网TX端口1/10/40 GbE，后续层次级别定义为子端口，管道，流量类和队列。

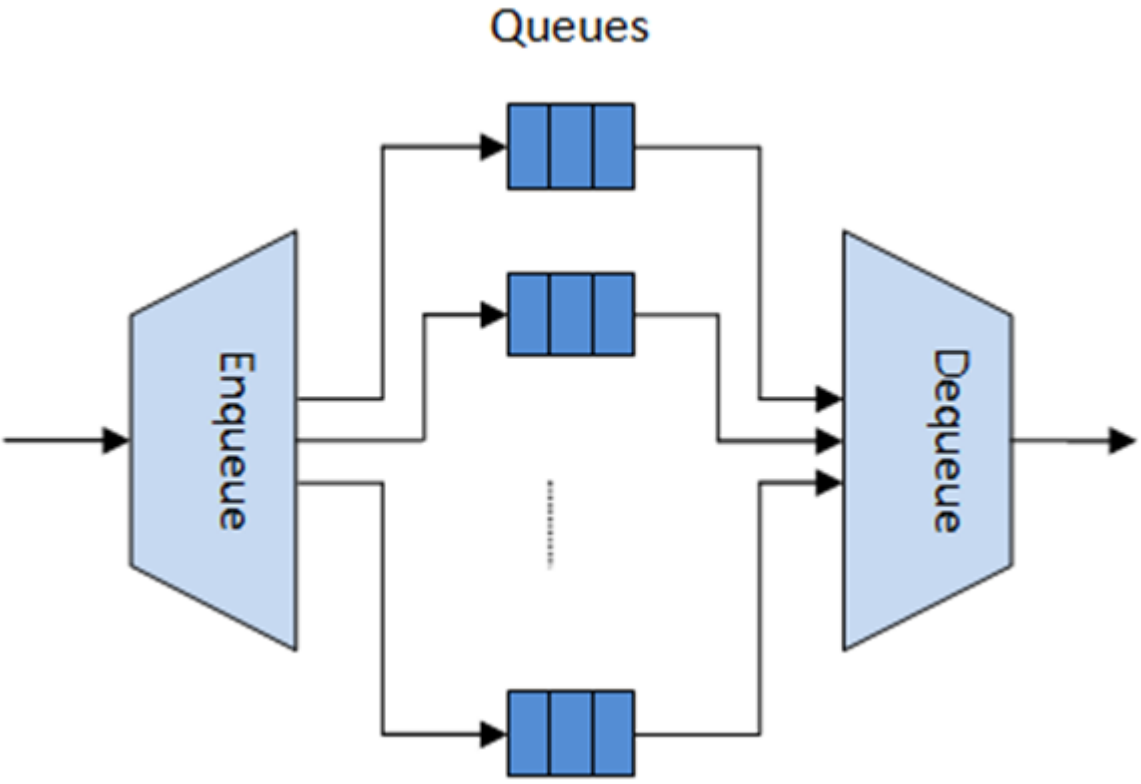


Fig. 4.50: Hierarchical Scheduler Block Internal Diagram

通常，每个子端口表示预定义的用户组，而每个管道表示单个用户/订户。每个流量类是具有特定丢失率，延迟和抖动要求（例如语音，视频或数据传输）的不同流量类型的表示。每个队列都承载属于同一用户的同一类型的一个或多个连接的数据包。

下表列出了各层次的功能。

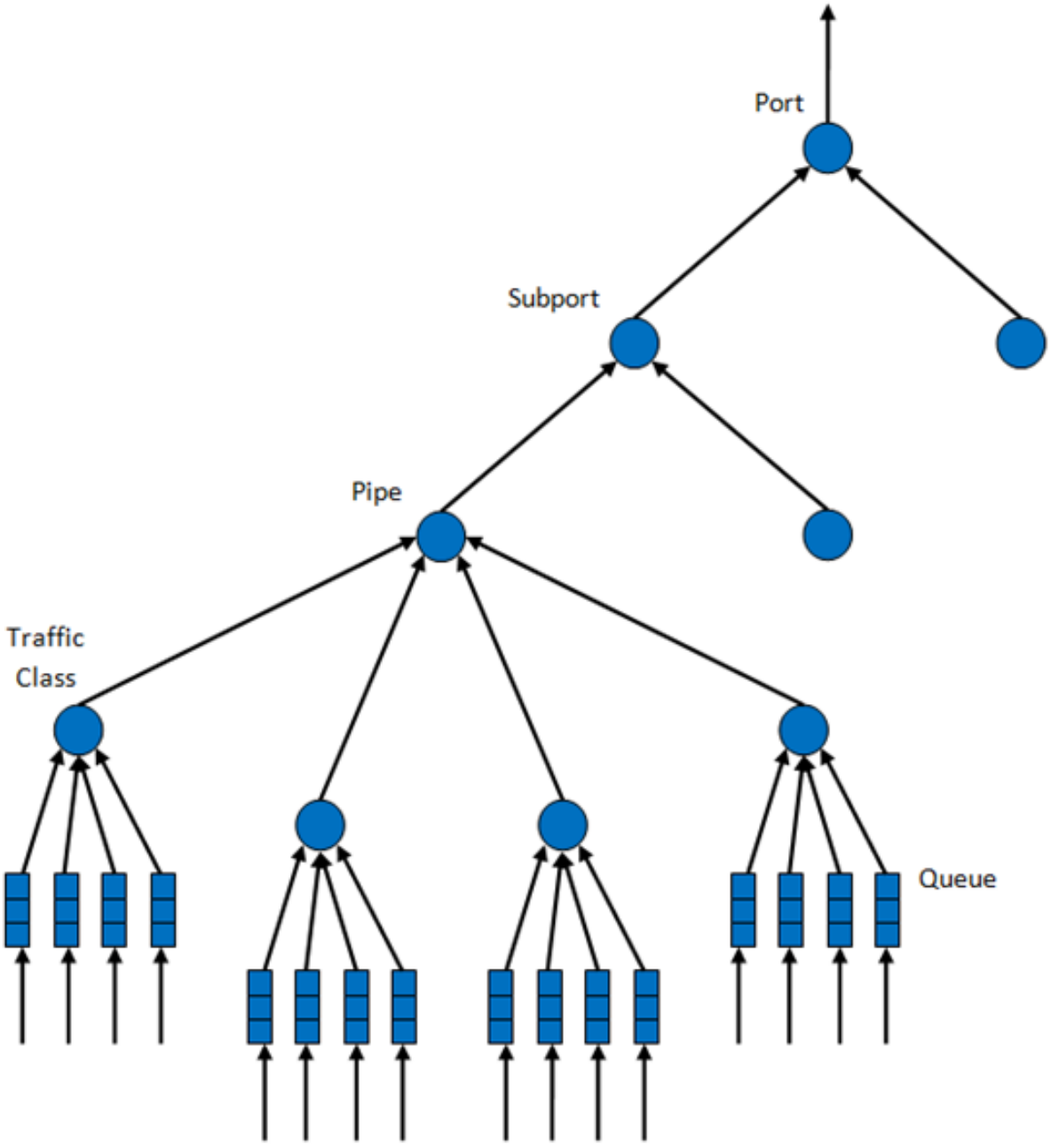


Fig. 4.51: Scheduling Hierarchy per Port

Table 4.53: Port Scheduling Hierarchy

#	Level	Siblings per Parent	Functional Description
1	Port	•	1. 输出以太网端口1/10/40 GbE 2. .多个端口以轮询方式调度，所有端口具有相同的优先级
2	Subport	Configurable (default: 8)	1. 流量整形使用令牌桶算法（每个子口一个令牌桶） 2. Subport层对每个流量类（TC）强制执行上限。 3. 较低优先级的TC能够重用较高优先级的TC当前未使用的子端口带宽
3	Pipe	Configurable (default: 4K)	1. 使用令牌桶算法进行流量整形（每个pipe一个令牌桶）
4	Traffic Class (TC)	4	1. 相同pipe的TC以严格的优先级顺序处理。 2. 在pipe级别每TC执行上限 3. 较低优先级的TC能够重用当前未被较高优先级的TC使用的pipe带宽 4. 当子卡TC超额（配置时间事件）时，管道TC上限被限制为由所有子端口管道共享的动态调整值。
5	Queue	4	1. 根据预定权重，使用加权循环（WRR）对相同TC的队列进行服务。

编程接口

PORT调度配置API

rte_sched.h文件包含port，subport和pipe的配置功能。

PORT调度入队API

Port调度入队API非常类似于DPDK PMD TX功能的API。

```
int rte_sched_port_enqueue(struct rte_sched_port *port, struct rte_mbuf **pkts,␣
↪uint32_t n_pkts);
```

PORT调度出队API

Port调度入队API非常类似于DPDK PMD RX功能的API。

```
int rte_sched_port_dequeue(struct rte_sched_port *port, struct rte_mbuf **pkts,␣
↪uint32_t n_pkts);
```

用例

```
/* File "application.c" */

#define N_PKTS_RX   64
#define N_PKTS_TX   48
#define NIC_RX_PORT 0
#define NIC_RX_QUEUE 0
#define NIC_TX_PORT 1
#define NIC_TX_QUEUE 0

struct rte_sched_port *port = NULL;
struct rte_mbuf *pkts_rx[N_PKTS_RX], *pkts_tx[N_PKTS_TX];
uint32_t n_pkts_rx, n_pkts_tx;

/* Initialization */

<initialization code>

/* Runtime */
while (1) {
    /* Read packets from NIC RX queue */

    n_pkts_rx = rte_eth_rx_burst(NIC_RX_PORT, NIC_RX_QUEUE, pkts_rx, N_PKTS_RX);

    /* Hierarchical scheduler enqueue */

    rte_sched_port_enqueue(port, pkts_rx, n_pkts_rx);

    /* Hierarchical scheduler dequeue */

    n_pkts_tx = rte_sched_port_dequeue(port, pkts_tx, N_PKTS_TX);
```

```
    /* Write packets to NIC TX queue */

    rte_eth_tx_burst(NIC_TX_PORT, NIC_TX_QUEUE, pkts_tx, n_pkts_tx);
}
```

实现

Internal Data Structures per Port

内部数据结构示意图，详细内容如下。

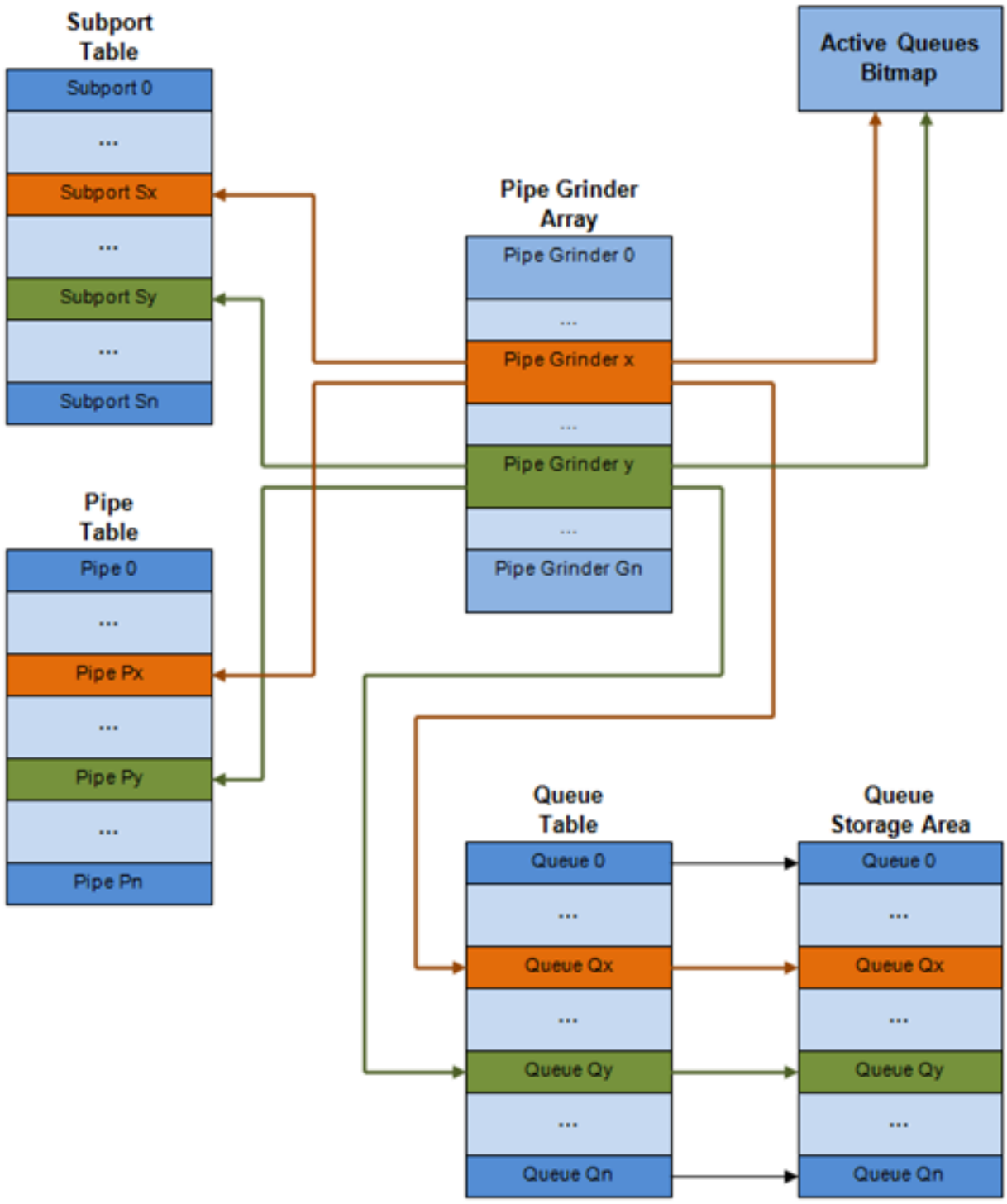


Fig. 4.52: Internal Data Structures per Port

Table 4.54: Scheduler Internal Data Structures per Port

#	Data structure	Size (bytes)	# per port	Access type		Description
				Enq	Deq	
1	Subport table entry	64	# subports per port	•	Rd, Wr	持续的子接口数据（信用，等）
2	Pipe table entry	64	# pipes per port	•	Rd, Wr	在运行时更新的pip，其TC及其队列的数据（信用等） pipe配置参数在运行时不改变。相同的pipe配置参数由多个pipe共享，因此它们不是pipe表条目的一部分。
3	Queue table entry	4	#queues per port	Rd, Wr	Rd, Wr	持续的队列数据（读写指针）。对于所有队列，每个TC的队列大小相同，允许使用快速公式计算队列基地址，因此这两个参数不是队列表条目的一部分。 任何给定pipe的队列表条目都存储在同一个高速缓存行中
4	Queue storage area	Config (default: 64 x8)	# queues per port	Wr	Rd	每个队列的元素数组;每个元素的大小是8字节mbuf指针
5	Active queues bitmap	1 bit per queue	1	Wr (Set)	Rd, Wr (Clear)	位图为每个队列维护一个状态位：队列不活动（队列为空）或队列活动（队列不为空） 队列位由调度程序入队设置，队列

多核缩放策略

多核缩放策略如下：

1. 在不同线程上操作不同的物理端口。但是同一个端口的入队和出队由同一个线程执行。
2. 通过在不同线程上操作相同物理端口（虚拟端口）的不同组的子端口，可以将相同的物理端口拆分到不同的线程。类似地，子端口可以被分割成更多个子端口，每个子端口由不同的线程运行。但是同一个端口的入队和出队由同一个线程运行。仅当出于性能考虑，不可能使用单个core处理完整端口时，才这样处理。

同一输出端口的出队和入队

上面强调过，同一个端口的出队和入队需要由同一个线程执行。因为，在不同core上对同一个输出端口执行出队和入队操作，可能会对调度程序的性能造成重大影响，因此不推荐这样做。

同一端口的入队和出队操作共享以下数据结构的访问权限：

1. 报文描述符
2. 队列表
3. 队列存储空区
4. 活动队列位图

可能存在使性能下降的原因如下：

1. 需要使队列和位图操作线程安全，这可能需要使用锁以保证访问顺序（例如，自旋锁/信号量）或使用原子操作进行无锁访问（例如，Test and Set或Compare and Swap命令等））。前一种情况对性能影响要严重得多。
2. 在两个core之间对存储共享数据结构的缓存行执行乒乓操作（由MESI协议缓存一致性CPU硬件透明地完成）。

当调度程序入队和出队操作必须在同一个线程运行，允许队列和位图操作非线程安全，并将调度程序数据结构保持在同一个core上，可以很大程度上保证性能。

性能缩放

扩展NIC端口数量只需要保证用于流量调度的CPU内核数量按比例增加即可。

入队水线

每个数据包的入队步骤：

1. 访问mbuf以读取标识数据包的目标队列所需的字段。这些字段包括port，subport，traffic class及queue，并且通常由报文分类阶段设置。
2. 访问队列结构以识别队列数组中的写入位置。如果队列已满，则丢弃该数据包。
3. 访问队列阵列位置以存储数据包（即写入mbuf指针）。

应该注意到这些步骤之间具有很强的数据依赖性，因为步骤2和3在步骤1和2的结果变得可用之前无法启动，这样就无法使用处理器乱序执行引擎上提供任何显着的性能优化。

考虑这样一种情况，给定的输入报文速率很高，队列数量大，可以想象得到，入队当前数据包需要访问的数据结构不存在于当前core的L1或L2 data cache中，此时，上述3个内存访问操作将会产生L1和L2 data cache miss。就性能考虑而言，每个数据包出现3次L1 / L2 data cache miss是不可接受的。

解决方法是提前预取所需的数据结构。预取操作具有执行延迟，在此期间处理器不应尝试访问当前正在进行预取的数据结构，此时处理器转向执行其他工作。可用的其他工作可以是对其他输入报文执行不同阶段的入队序列，从而实现入队操作的流水线实现。

Fig. 4.53 展示出了具有4级水线的入队操作实现，并且每个阶段操作2个不同的输入报文。在给定的时间点上，任何报文只能在水线某个阶段进行处理。

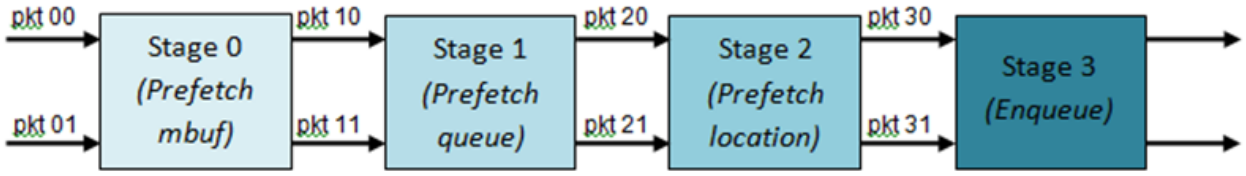


Fig. 4.53: Prefetch Pipeline for the Hierarchical Scheduler Enqueue Operation

由上图描述的入队水线实现的拥塞管理方案是非常基础的：数据包排队入队，直到指定队列变满为止；当满时，到这个队列的所有数据包将被丢弃，直到队列中有数据包出队。可以通过使用RED/WRED作为入队水线的一部分来改进，该流程查看队列占用率和报文优先级，以便产生特定数据包的入队/丢弃决定（与入队所有数据包/不加区分地丢弃所有数据包不一样）。

出队状态机

从当前pipe调度下一个数据包的步骤如下：

1. 使用位图扫描操作识别出下一个活动的pipe（prefetch pipe）。
2. 读取pipe数据结构。更新当前pipe及其subport的信用。识别当前pipe中的第一个active traffic class，使用WRR选择下一个queue，为当前pipe的所有16个queue预取队列指针。
3. 从当前WRR queue读取下一个元素，并预取其数据包描述符。
4. 从包描述符（mbuf结构）读取包长度。根据包长度和可用信用（当前pipe，pipe traffic class，subport及subport traffic class），对当前数据包进行是否调度决策。

为了避免cache miss，上述数据结构（pipe，queue，queue array，mbufs）在被访问之前被预取。隐藏预取操作的延迟的策略是在为当前pipe发出预取后立即从当前pipe（在grinder A中）切换到另一个pipe（在grinderB中）。这样就可以在执行切换回pipe（grinder A）之前，有足够的时间完成预取操作。

出pipe状态机将数据存在处理器高速缓存中，因此它尝试从相同的pipe TC和pipe（尽可能多的数据包和信用）发送尽可能多的数据包，然后再移动到下一个活动TC pipe（如果有）或另一个活动pipe。 .. _figure_pipe_prefetch_sm:

时间和同步

输出端口被建模为字节槽的传送带，需要由调度器填充用于传输的数据。对于10GbE，每秒需要由调度器填充12.5亿个字节的槽位。如果调度程序填充不够快，只要存在足够的报文和信用，则一些时隙将被闲置并且带宽将被浪费。

原则上，层次调度程序出队操作应由NIC TX触发。通常，一旦NIC TX输入队列的占用率下降到预定义的阈值以下，端口调度器将被唤醒（基于中断或基于轮询，通过连续监视队列占用）来填充更多的数据包进入队列。

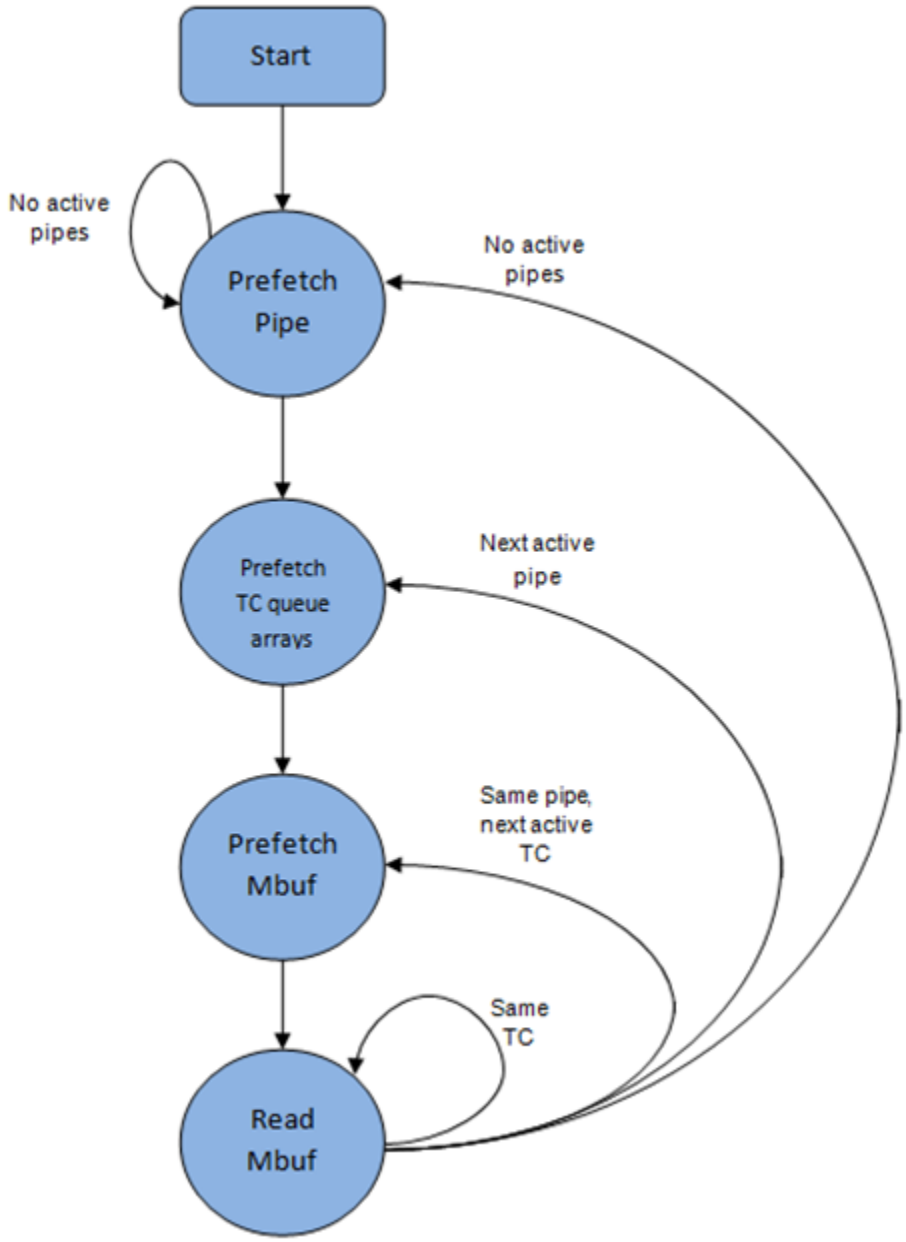


Fig. 4.54: Pipe Prefetch State Machine for the Hierarchical Scheduler Dequeue Operation

内部时间引用

调度器需要跟踪信用逻辑的时间演化，因为信用需要基于时间更新（例如，子流量和管道流量整形，流量级上限执行等）。

每当调度程序决定将数据包发送到NIC TX进行传输时，调度器将相应地增加其内部时间参考。因此，以字节为单位保持内部时间基准是方便的，其中字节表示物理接口在传输介质上发送字节所需的持续时间。这样，当报文被调度用于传输时，时间以（n + h）递增，其中n是以字节为单位的报文长度，h是每个报文的成帧开销字节数。

内部时间参考重新同步

调度器需要将其内部时间参考对齐到端口传送带的步速。原因是要确保调度程序不以比物理介质的线路速率更多的字节来馈送NIC TX，以防止数据包丢失。

调度程序读取每个出队调用的当前时间。可以通过读取时间戳计数器（TSC）寄存器或高精度事件定时器（HPET）寄存器来获取CPU时间戳。 当前CPU时间戳将CPU时钟数转换为字节数：time_bytes = time_cycles / cycles_per_byte，其中cycles_per_byte是等效于线上一个字节的传输时间的CPU周期数（例如CPU频率 2 GHz和10GbE端口， cycles_per_byte = 1.6 ）。

调度程序维护NIC time的内部时间参考。 每当分组被调度时，NIC time随分组长度（包括帧开销）增加。在每次出队调用时，调度程序将检查其NIC time的内部引用与当前时间的关系：

1. 如果NIC time未来（NIC time> =当前时间），则不需要调整NIC time。这意味着调度程序能够在NIC实际需要这些数据包之前安排NIC数据包，因此NIC TX提供了数据包;
2. 如果NIC time过去（NIC时间<当前时间），则NIC time应通过将其设置为当前时间来进行调整。 这意味着调度程序不能跟上NIC字节传送带的速度，因此由于NIC TX的数据包供应不足，所以NIC带宽被浪费了。

调度器精度和粒度

调度器往返延迟（SRTD）是指调度器在同一个pipe的两次连续检验之间的时间（CPU周期数）。

为了跟上输出端口（即避免带宽丢失），调度程序应该能够比NIC TX发送的n个数据包更快地调度n个数据包。

假设没有端口超过流量，调度程序需要跟上管道令牌桶配置的每个管道的速率。这意味着管道令牌桶的大小应该设置得足够高，以防止它由于大的SRTD而溢出，因为这将导致管道的信用损失（带宽损失）。

信用逻辑

调度决策

当满足以下所有条件时，从（subport S，pipe P，traffic class TC，queue Q）发送下一个分组的调度决定（分组被发送）：

- Subport S的Pipe P目前由一个端口调度选择;
- 流量类TC是管道P的最高优先级的主要流量类别;
- 队列Q是管道P的流量类TC内由WRR选择的下一个队列;
- 子接口S有足够的信用来发送数据包;
- 子接口S具有足够的信用流量类TC来发送数据包;

- 管道P有足够的信用来发送数据包；
- 管道P具有足够的信用用于流量类TC发送数据包。

如果满足所有上述条件，则选择分组进行传输，并从子接口S，子接口S流量类TC，管道P，管道P流量类TC中减去必要的信用。

帧开销

由于所有数据包长度的最大公约数为1个字节，所以信用单位被选为1个字节。传输n个字节的报文所需的信用数量等于（n + h），其中h等于每个报文的成帧开销字节数。

Table 4.55: Ethernet Frame Overhead Fields

#	Packet field	Length (bytes)	Comments
1	Preamble	7	
2	Start of Frame Delimiter (SFD)	1	
3	Frame Check Sequence (FCS)	4	当mbuf包长度字段中不包含时这里才需要考虑开销。
4	Inter Frame Gap (IFG)	12	
5	Total	24	

Traffic Shaping

The traffic shaping for subport and pipe is implemented using a token bucket per subport/per pipe. Each token bucket is implemented using one saturated counter that keeps track of the number of available credits.

The token bucket generic parameters and operations are presented in Table 4.56 and Table 4.57.

Table 4.56: Token Bucket Generic Operations

#	Token Bucket Parameter	Unit	Description
1	bucket_rate	Credits per second	Rate of adding credits to the bucket.
2	bucket_size	Credits	Max number of credits that can be stored in the bucket.

Table 4.57: Token Bucket Generic Parameters

#	Token Bucket Operation	Description
1	Initialization	Bucket set to a predefined value, e.g. zero or half of the bucket size.
2	Credit update	Credits are added to the bucket on top of existing ones, either periodically or on demand, based on the bucket_rate. Credits cannot exceed the upper limit defined by the bucket_size, so any credits to be added to the bucket while the bucket is full are dropped.
3	Credit consumption	As result of packet scheduling, the necessary number of credits is removed from the bucket. The packet can only be sent if enough credits are in the bucket to send the full packet (packet bytes and framing overhead for the packet).

To implement the token bucket generic operations described above, the current design uses the persistent data structure presented in Table 4.58, while the implementation of the token bucket operations is described in Table 4.59.

Table 4.58: Token Bucket Persistent Data Structure

#	Token bucket field	Unit	Description
1	tb_time	Bytes	Time of the last credit update. Measured in bytes instead of seconds or CPU cycles for ease of credit consumption operation (as the current time is also maintained in bytes). See Section 26.2.4.5.1 "Internal Time Reference" for an explanation of why the time is maintained in byte units.
2	tb_period	Bytes	Time period that should elapse since the last credit update in order for the bucket to be awarded tb_credits_per_period worth or credits.
3	tb_credits_per_period	Bytes	Credit allowance per tb_period.
4	tb_size	Bytes	Bucket size, i.e. upper limit for the tb_credits.
5	tb_credits	Bytes	Number of credits currently in the bucket.

The bucket rate (in bytes per second) can be computed with the following formula:

*bucket_rate = (tb_credits_per_period / tb_period) * r*

where, r = port line rate (in bytes per second).

Table 4.59: Token Bucket Operations

#	Token bucket operation	Description
1	Initialization	*tb_credits = 0; or tb_credits = tb_size / 2;*
2	Credit update	Credit update options: • Every time a packet is sent for a port, update the credits of all the the subports and pipes of that port. Not feasible. • Every time a packet is sent, update the credits for the pipe and subport. Very accurate, but not needed (a lot of calculations). • Every time a pipe is selected (that is, picked by one of the grinders), update the credits for the pipe and its subport. The current implementation is using option 3. According to Section "Dequeue State Machine", the pipe and subport credits are updated every time a pipe is selected by the dequeue process before the pipe and subport credits are actually used. The implementation uses a tradeoff between accuracy and speed by updating the bucket credits only when at least a full *tb_period* has elapsed since the last update. • Full accuracy can be achieved by selecting the value for *tb_period* for which *tb_credits_per_period = 1*. • When full accuracy is not required, better performance is achieved by setting *tb_credits* to a larger value. Update operations: • n_periods = (time - tb_time) / tb_period; • tb_credits += n_periods * tb_credits_per_period; • tb_credits = min(tb_credits, tb_size); • tb_time += n_periods * tb_period;
3	**Credit consumption** (on packet scheduling)	As result of packet scheduling, the necessary number of credits is removed from the bucket. The packet can only be sent if enough credits are in the bucket to send the full packet (packet bytes and framing overhead for the packet). Scheduling operations: pkt_credits = pkt_len + frame_overhead; if (tb_credits >= pkt_credits){tb_credits =

Traffic Classes

Implementation of Strict Priority Scheduling

Strict priority scheduling of traffic classes within the same pipe is implemented by the pipe dequeue state machine, which selects the queues in ascending order. Therefore, queues 0..3 (associated with TC 0, highest priority TC) are handled before queues 4..7 (TC 1, lower priority than TC 0), which are handled before queues 8..11 (TC 2), which are handled before queues 12..15 (TC 3, lowest priority TC).

Upper Limit Enforcement

The traffic classes at the pipe and subport levels are not traffic shaped, so there is no token bucket maintained in this context. The upper limit for the traffic classes at the subport and pipe levels is enforced by periodically refilling the subport / pipe traffic class credit counter, out of which credits are consumed every time a packet is scheduled for that subport / pipe, as described in Table 4.60 and Table 4.61.

Table 4.60: Subport/Pipe Traffic Class Upper Limit Enforcement Persistent Data Structure

#	Subport or pipe field	Unit	Description
1	tc_time	Bytes	Time of the next update (upper limit refill) for the 4 TCs of the current subport / pipe. See Section "Internal Time Reference"for the explanation of why the time is maintained in byte units.
2	tc_period	Bytes	Time between two consecutive updates for the 4 TCs of the current subport / pipe. This is expected to be many times bigger than the typical value of the token bucket tb_period.
3	tc_credits_per_period	Bytes	Upper limit for the number of credits allowed to be consumed by the current TC during each enforcement period tc_period.
4	tc_credits	Bytes	Current upper limit for the number of credits that can be consumed by the current traffic class for the remainder of the current enforcement period.

Table 4.61: Subport/Pipe Traffic Class Upper Limit Enforcement Operations

#	Traffic Class Operation	Description
1	Initialization	tc_credits = tc_credits_per_period; tc_time = tc_period;

2	Credit update	Update operations:
		if (time >= tc_time) {
		tc_credits = tc_credits_per_period;
		tc_time = time + tc_period;
		}

3	Credit consumption (on packet scheduling)	As result of packet scheduling, the TC limit is decreased with the necessary number of credits. The packet can only be sent if enough credits are currently available in the TC limit to send the full packet (packet bytes and framing overhead for the packet).
		Scheduling operations:
		pkt_credits = pk_len + frame_overhead;
		if (tc_credits >= pkt_credits) {tc_credits -= pkt_credits;}

Weighted Round Robin (WRR)

The evolution of the WRR design solution from simple to complex is shown in Table 4.62.

Table 4.62: Weighted Round Robin (WRR)

#	All Queues Active?	Equal Weights for All Queues?	All Packets Equal?	Strategy
1	Yes	Yes	Yes	**Byte level round robin** *Next queue* queue #i, i = *(i + 1) % n*
2	Yes	Yes	No	**Packet level round robin** Consuming one byte from queue #i requires consuming exactly one token for queue #i. T(i) = Accumulated number of tokens previously consumed from queue #i. Every time a packet is consumed from queue #i, T(i) is updated as: T(i) += *pkt_len*. *Next queue* : queue with the smallest T.
3	Yes	No	No	**Packet level weighted round robin** This case can be reduced to the previous case by introducing a cost per byte that is different for each queue. Queues with lower weights have a higher cost per byte. This way, it is still meaningful to compare the consumption amongst different queues in order to select the next queue. w(i) = Weight of queue #i t(i) = Tokens per byte for queue #i, defined as the inverse weight of queue #i. For example, if w[0..3] = [1:2:4:8], then t[0..3] = [8:4:2:1]; if w[0..3] = [1:4:15:20], then t[0..3] = [60:15:4:3]. Consuming one byte

Subport Traffic Class Oversubscription

Problem Statement

Oversubscription for subport traffic class X is a configuration-time event that occurs when more bandwidth is allocated for traffic class X at the level of subport member pipes than allocated for the same traffic class at the parent subport level.

The existence of the oversubscription for a specific subport and traffic class is solely the result of pipe and subport-level configuration as opposed to being created due to dynamic evolution of the traffic load at run-time (as congestion is).

When the overall demand for traffic class X for the current subport is low, the existence of the oversubscription condition does not represent a problem, as demand for traffic class X is completely satisfied for all member pipes. However, this can no longer be achieved when the aggregated demand for traffic class X for all subport member pipes exceeds the limit configured at the subport level.

Solution Space

summarizes some of the possible approaches for handling this problem, with the third approach selected for implementation.

Table 4.63: Subport Traffic Class Oversubscription

No.	Approach	Description
1	Don't care	First come, first served. This approach is not fair amongst subport member pipes, as pipes that are served first will use up as much bandwidth for TC X as they need, while pipes that are served later will receive poor service due to bandwidth for TC X at the subport level being scarce.
2	Scale down all pipes	All pipes within the subport have their bandwidth limit for TC X scaled down by the same factor. This approach is not fair among subport member pipes, as the low end pipes (that is, pipes configured with low bandwidth) can potentially experience severe service degradation that might render their service unusable (if available bandwidth for these pipes drops below the minimum requirements for a workable service), while the service degradation for high end pipes might not be noticeable at all.
3	Cap the high demand pipes	Each subport member pipe receives an equal share of the bandwidth available at run-time for TC X at the subport level. Any bandwidth left unused by the low-demand pipes is redistributed in equal portions to the high-demand pipes. This way, the high-demand pipes are truncated while the low-demand pipes are not impacted.

Typically, the subport TC oversubscription feature is enabled only for the lowest priority traffic class (TC 3), which is typically used for best effort traffic, with the management plane preventing this condition from occurring for the other (higher priority) traffic classes.

To ease implementation, it is also assumed that the upper limit for subport TC 3 is set to 100% of the subport rate, and that the upper limit for pipe TC 3 is set to 100% of pipe rate for all subport member pipes.

Implementation Overview

The algorithm computes a watermark, which is periodically updated based on the current demand experienced by the subport member pipes, whose purpose is to limit the amount of traffic that each pipe is allowed to send for TC 3. The watermark is computed at the subport level at the beginning of each traffic class upper limit enforcement period and the same value is used by all the subport member pipes throughout the current enforcement period. illustrates how the watermark computed as subport level at the beginning of each period is propagated to all subport member pipes.

At the beginning of the current enforcement period (which coincides with the end of the previous enforcement period),

the value of the watermark is adjusted based on the amount of bandwidth allocated to TC 3 at the beginning of the previous period that was not left unused by the subport member pipes at the end of the previous period.

If there was subport TC 3 bandwidth left unused, the value of the watermark for the current period is increased to encourage the subport member pipes to consume more bandwidth. Otherwise, the value of the watermark is decreased to enforce equality of bandwidth consumption among subport member pipes for TC 3.

The increase or decrease in the watermark value is done in small increments, so several enforcement periods might be required to reach the equilibrium state. This state can change at any moment due to variations in the demand experienced by the subport member pipes for TC 3, for example, as a result of demand increase (when the watermark needs to be lowered) or demand decrease (when the watermark needs to be increased).

When demand is low, the watermark is set high to prevent it from impeding the subport member pipes from consuming more bandwidth. The highest value for the watermark is picked as the highest rate configured for a subport member pipe. Table 4.64 and Table 4.65 illustrates the watermark operation.

Table 4.64: Watermark Propagation from Subport Level to Member Pipes at the Beginning of Each Traffic Class Upper Limit Enforcement Period

No.	Subport Traffic Class Operation	Description
1	Initialization	**Subport level**: subport_period_id= 0 **Pipe level**: pipe_period_id = 0

2	Credit update	**Subport Level**:
		if (time>=subport_tc_time)
		{ subport_wm = water_mark_update();
		subport_tc_time = time + subport_tc_period;
		subport_period_id++;
		}
		Pipelevel:
		if(pipe_period_id != subport_period_id)
		{
		pipe_ov_credits = subport_wm * pipe_weight;
		pipe_period_id = subport_period_id;
		}

3	Credit consumption (on packet scheduling)	**Pipe level:**
		pkt_credits = pk_len + frame_overhead;
		if(pipe_ov_credits >= pkt_credits{
		pipe_ov_credits -= pkt_credits;
		}

Table 4.65: Watermark Calculation

No.	Subport Traffic Class Operation	Description
1	Initialization	**Subport level:** wm = WM_MAX

2	Credit update	**Subport level (water_mark_update):**
		tc0_cons = subport_tc0_credits_per_period - subport_tc0_credits;
		tc1_cons = subport_tc1_credits_per_period - subport_tc1_credits;
		tc2_cons = subport_tc2_credits_per_period - subport_tc2_credits;
		tc3_cons = subport_tc3_credits_per_period - subport_tc3_credits;
		tc3_cons_max = subport_tc3_credits_per_period - (tc0_cons + tc1_cons + tc2_cons);
		if(tc3_consumption > (tc3_consumption_max - MTU)){
		wm -= wm >> 7;
		if(wm < WM_MIN)
		wm = WM_MIN;
		} else {
		wm += (wm >> 7) + 1;
		if(wm > WM_MAX)
		wm = WM_MAX;
		}

Worst Case Scenarios for Performance

Lots of Active Queues with Not Enough Credits

The more queues the scheduler has to examine for packets and credits in order to select one packet, the lower the performance of the scheduler is.

The scheduler maintains the bitmap of active queues, which skips the non-active queues, but in order to detect whether a specific pipe has enough credits, the pipe has to be drilled down using the pipe dequeue state machine, which consumes cycles regardless of the scheduling result (no packets are produced or at least one packet is produced).

This scenario stresses the importance of the policer for the scheduler performance: if the pipe does not have enough credits, its packets should be dropped as soon as possible (before they reach the hierarchical scheduler), thus rendering the pipe queues as not active, which allows the dequeue side to skip that pipe with no cycles being spent on investigating the pipe credits that would result in a “not enough credits” status.

Single Queue with 100% Line Rate

The port scheduler performance is optimized for a large number of queues. If the number of queues is small, then the performance of the port scheduler for the same level of active traffic is expected to be worse than the performance of a small set of message passing queues.

4.23.3 Dropper

The purpose of the DPDK dropper is to drop packets arriving at a packet scheduler to avoid congestion. The dropper supports the Random Early Detection (RED), Weighted Random Early Detection (WRED) and tail drop algorithms. Fig. 4.55 illustrates how the dropper integrates with the scheduler. The DPDK currently does not support congestion management so the dropper provides the only method for congestion avoidance.

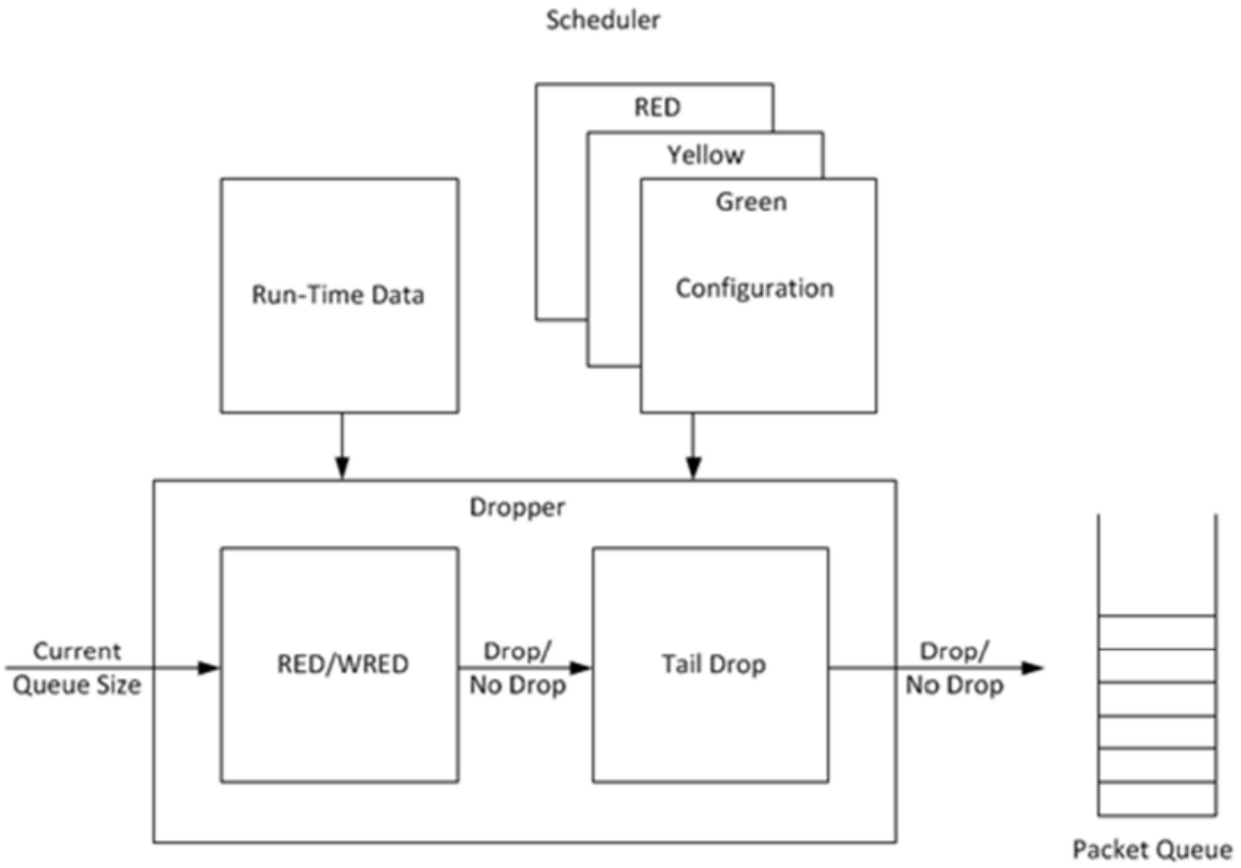


Fig. 4.55: High-level Block Diagram of the DPDK Dropper

The dropper uses the Random Early Detection (RED) congestion avoidance algorithm as documented in the reference publication. The purpose of the RED algorithm is to monitor a packet queue, determine the current congestion level in the queue and decide whether an arriving packet should be enqueued or dropped. The RED algorithm uses an Exponential Weighted Moving Average (EWMA) filter to compute average queue size which gives an indication of the current congestion level in the queue.

For each enqueue operation, the RED algorithm compares the average queue size to minimum and maximum thresholds. Depending on whether the average queue size is below, above or in between these thresholds, the RED algorithm

calculates the probability that an arriving packet should be dropped and makes a random decision based on this probability.

The dropper also supports Weighted Random Early Detection (WRED) by allowing the scheduler to select different RED configurations for the same packet queue at run-time. In the case of severe congestion, the dropper resorts to tail drop. This occurs when a packet queue has reached maximum capacity and cannot store any more packets. In this situation, all arriving packets are dropped.

The flow through the dropper is illustrated in Fig. 4.56. The RED/WRED algorithm is exercised first and tail drop second.

The use cases supported by the dropper are:

- – Initialize configuration data
- – Initialize run-time data
- – Enqueue (make a decision to enqueue or drop an arriving packet)
- – Mark empty (record the time at which a packet queue becomes empty)

The configuration use case is explained in *Section2.23.3.1*, the enqueue operation is explained in *Section 2.23.3.2* and the mark empty operation is explained in *Section 2.23.3.3*.

Configuration

A RED configuration contains the parameters given in Table 4.66.

Table 4.66: RED Configuration Parameters

Parameter	Minimum	Maximum	Typical
Minimum Threshold	0	1022	1/4 x queue size
Maximum Threshold	1	1023	1/2 x queue size
Inverse Mark Probability	1	255	10
EWMA Filter Weight	1	12	9

The meaning of these parameters is explained in more detail in the following sections. The format of these parameters as specified to the dropper module API corresponds to the format used by Cisco* in their RED implementation. The minimum and maximum threshold parameters are specified to the dropper module in terms of number of packets. The mark probability parameter is specified as an inverse value, for example, an inverse mark probability parameter value of 10 corresponds to a mark probability of 1/10 (that is, 1 in 10 packets will be dropped). The EWMA filter weight parameter is specified as an inverse log value, for example, a filter weight parameter value of 9 corresponds to a filter weight of 1/29.

Enqueue Operation

In the example shown in Fig. 4.57, q (actual queue size) is the input value, avg (average queue size) and count (number of packets since the last drop) are run-time values, decision is the output value and the remaining values are configuration parameters.

EWMA Filter Microblock

The purpose of the EWMA Filter microblock is to filter queue size values to smooth out transient changes that result from “bursty” traffic. The output value is the average queue size which gives a more stable view of the current congestion level in the queue.

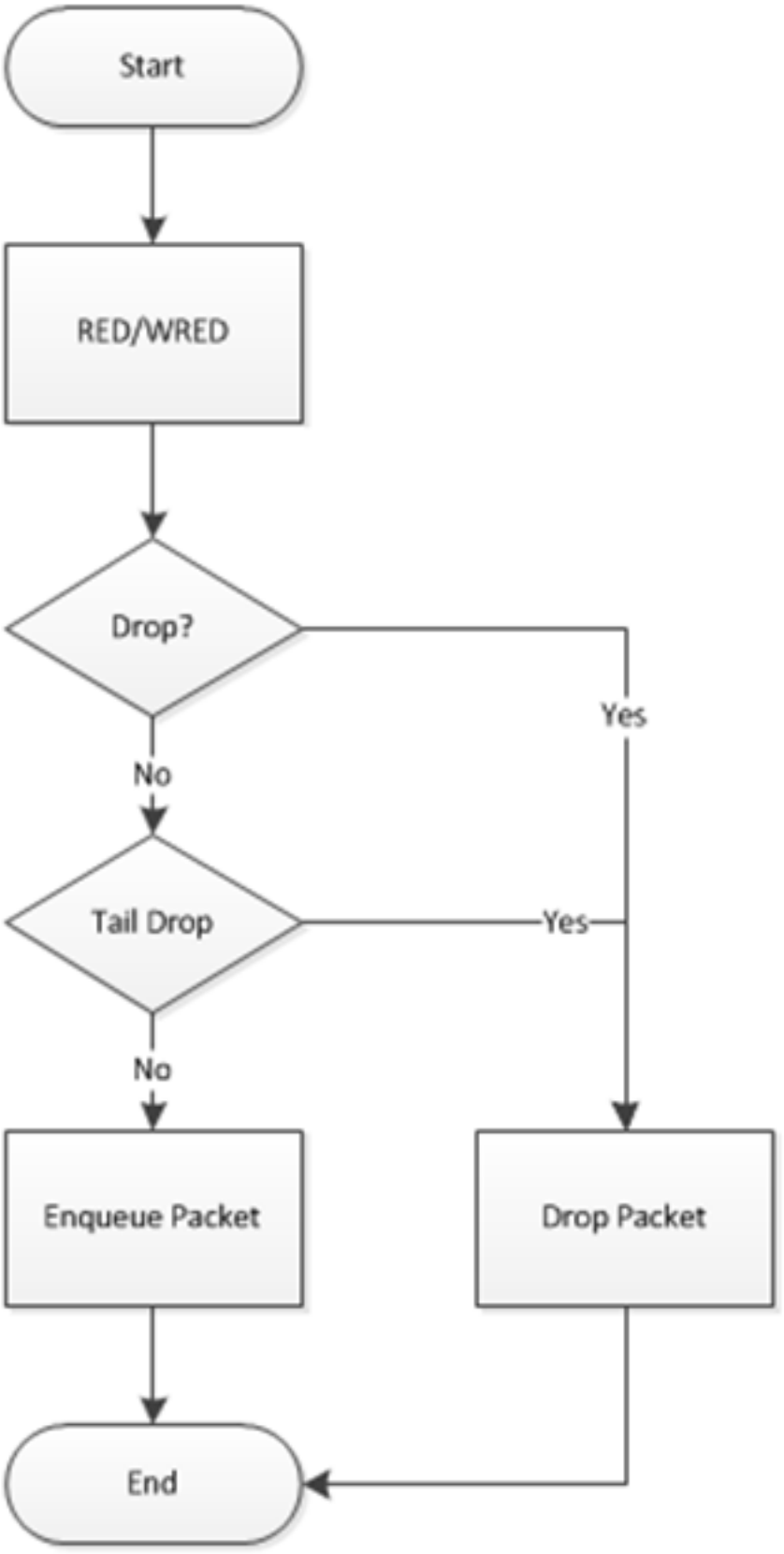


Fig. 4.56: Flow Through the Dropper

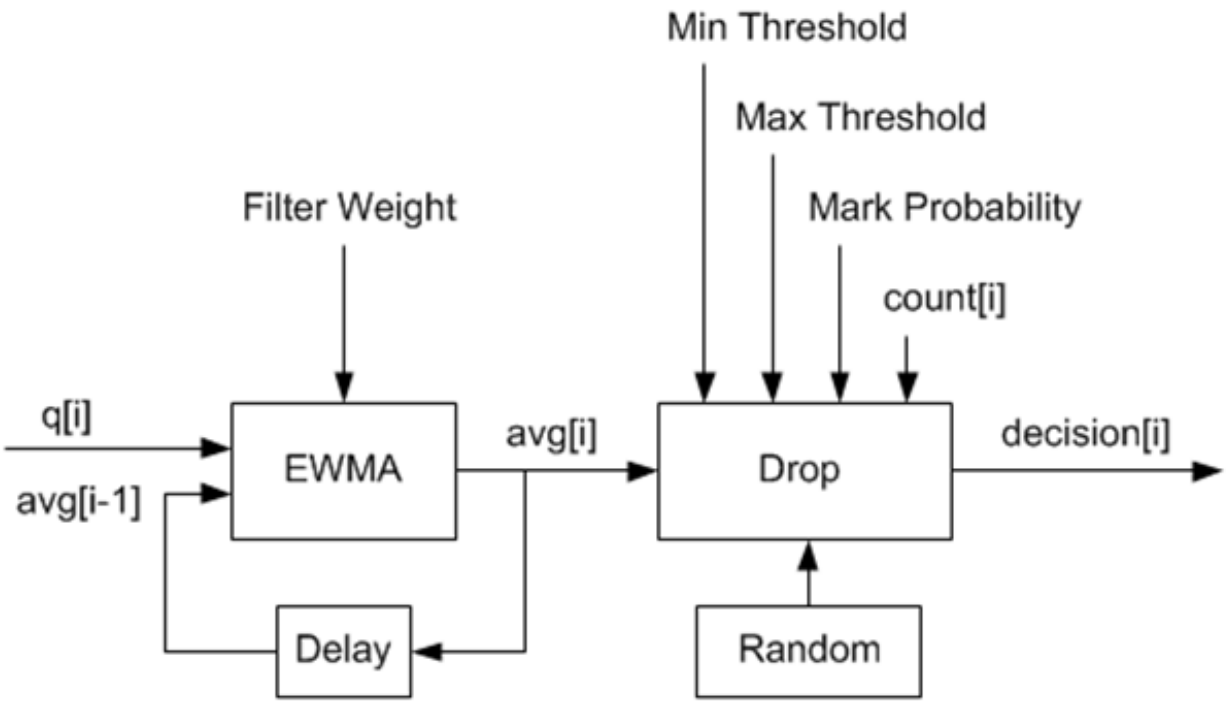


Fig. 4.57: Example Data Flow Through Dropper

The EWMA filter has one configuration parameter, filter weight, which determines how quickly or slowly the average queue size output responds to changes in the actual queue size input. Higher values of filter weight mean that the average queue size responds more quickly to changes in actual queue size.

Average Queue Size Calculation when the Queue is not Empty

The definition of the EWMA filter is given in the following equation.

$$avg[i] = (1 - w_q) \times avg[i-1] + w_q \times q[i]$$

Where:

- *avg* = average queue size
- *wq* = filter weight
- *q* = actual queue size

Note:

The filter weight, wq = 1/2^n, where n is the filter weight parameter value passed to the dropper module on configuration (see *Section2.23.3.1*).

Average Queue Size Calculation when the Queue is Empty

The EWMA filter does not read time stamps and instead assumes that enqueue operations will happen quite regularly. Special handling is required when the queue becomes empty as the queue could be empty for a short time or a long

time. When the queue becomes empty, average queue size should decay gradually to zero instead of dropping suddenly to zero or remaining stagnant at the last computed value. When a packet is enqueued on an empty queue, the average queue size is computed using the following formula:

$$avg[i] = \text{avg}[i-1] \times \left(1 - \text{w}_\text{q}\right)^m$$

Where:

- *m* = the number of enqueue operations that could have occurred on this queue while the queue was empty

In the dropper module, *m* is defined as:

$$m = \left(\frac{time - qtime}{s}\right)$$

Where:

- *time* = current time
- *qtime* = time the queue became empty
- *s* = typical time between successive enqueue operations on this queue

The time reference is in units of bytes, where a byte signifies the time duration required by the physical interface to send out a byte on the transmission medium (see Section "Internal Time Reference"). The parameter s is defined in the dropper module as a constant with the value: s=2^22. This corresponds to the time required by every leaf node in a hierarchy with 64K leaf nodes to transmit one 64-byte packet onto the wire and represents the worst case scenario. For much smaller scheduler hierarchies, it may be necessary to reduce the parameter s, which is defined in the red header source file (rte_red.h) as:

```
#define RTE_RED_S
```

Since the time reference is in bytes, the port speed is implied in the expression: *time-qtime*. The dropper does not have to be configured with the actual port speed. It adjusts automatically to low speed and high speed links.

Implementation

A numerical method is used to compute the factor (1-wq)^m that appears in Equation 2.

This method is based on the following identity:

$$a \equiv 2^{(b \times \log_2(a))}$$

This allows us to express the following:

$$\left(1 - w_q\right)^m = 2^{(m \times \log_2(1-w_q))}$$

In the dropper module, a look-up table is used to compute log2(1-wq) for each value of wq supported by the dropper module. The factor (1-wq)^m can then be obtained by multiplying the table value by *m* and applying shift operations. To avoid overflow in the multiplication, the value, *m*, and the look-up table values are limited to 16 bits. The total size of the look-up table is 56 bytes. Once the factor (1-wq)^m is obtained using this method, the average queue size can be calculated from Equation 2.

Alternative Approaches

Other methods for calculating the factor (1-wq)^m in the expression for computing average queue size when the queue is empty (Equation 2) were considered. These approaches include:

- Floating-point evaluation
- Fixed-point evaluation using a small look-up table (512B) and up to 16 multiplications (this is the approach used in the FreeBSD* ALTQ RED implementation)
- Fixed-point evaluation using a small look-up table (512B) and 16 SSE multiplications (SSE optimized version of the approach used in the FreeBSD* ALTQ RED implementation)
- Large look-up table (76 KB)

The method that was finally selected (described above in Section 26.3.2.2.1) out performs all of these approaches in terms of run-time performance and memory requirements and also achieves accuracy comparable to floating-point evaluation. Table 4.67 lists the performance of each of these alternative approaches relative to the method that is used in the dropper. As can be seen, the floating-point implementation achieved the worst performance.

Table 4.67: Relative Performance of Alternative Approaches

Method	Relative Performance
Current dropper method (see *Section 23.3.2.1.3*)	100%
Fixed-point method with small (512B) look-up table	148%
SSE method with small (512B) look-up table	114%
Large (76KB) look-up table	118%
Floating-point	595%
Note: In this case, since performance is expressed as time spent executing the operation in a specific condition, any relative performan	

Drop Decision Block

The Drop Decision block:

- Compares the average queue size with the minimum and maximum thresholds
- Calculates a packet drop probability
- Makes a random decision to enqueue or drop an arriving packet

The calculation of the drop probability occurs in two stages. An initial drop probability is calculated based on the average queue size, the minimum and maximum thresholds and the mark probability. An actual drop probability is then computed from the initial drop probability. The actual drop probability takes the count run-time value into consideration so that the actual drop probability increases as more packets arrive to the packet queue since the last packet was dropped.

Initial Packet Drop Probability

The initial drop probability is calculated using the following equation.

$$p_b = \begin{cases} 0, & avg < min_{th} \\ max_p\left(\frac{avg - min_{th}}{max_{th} - min_{th}}\right), & min_{th} \leq avg < max_{th} \\ 1, & avg \geq max_{th} \end{cases}$$

Where:

- *maxp* = mark probability
- *avg* = average queue size
- *minth* = minimum threshold
- *maxth* = maximum threshold

The calculation of the packet drop probability using Equation 3 is illustrated in Fig. 4.58. If the average queue size is below the minimum threshold, an arriving packet is enqueued. If the average queue size is at or above the maximum threshold, an arriving packet is dropped. If the average queue size is between the minimum and maximum thresholds, a drop probability is calculated to determine if the packet should be enqueued or dropped.

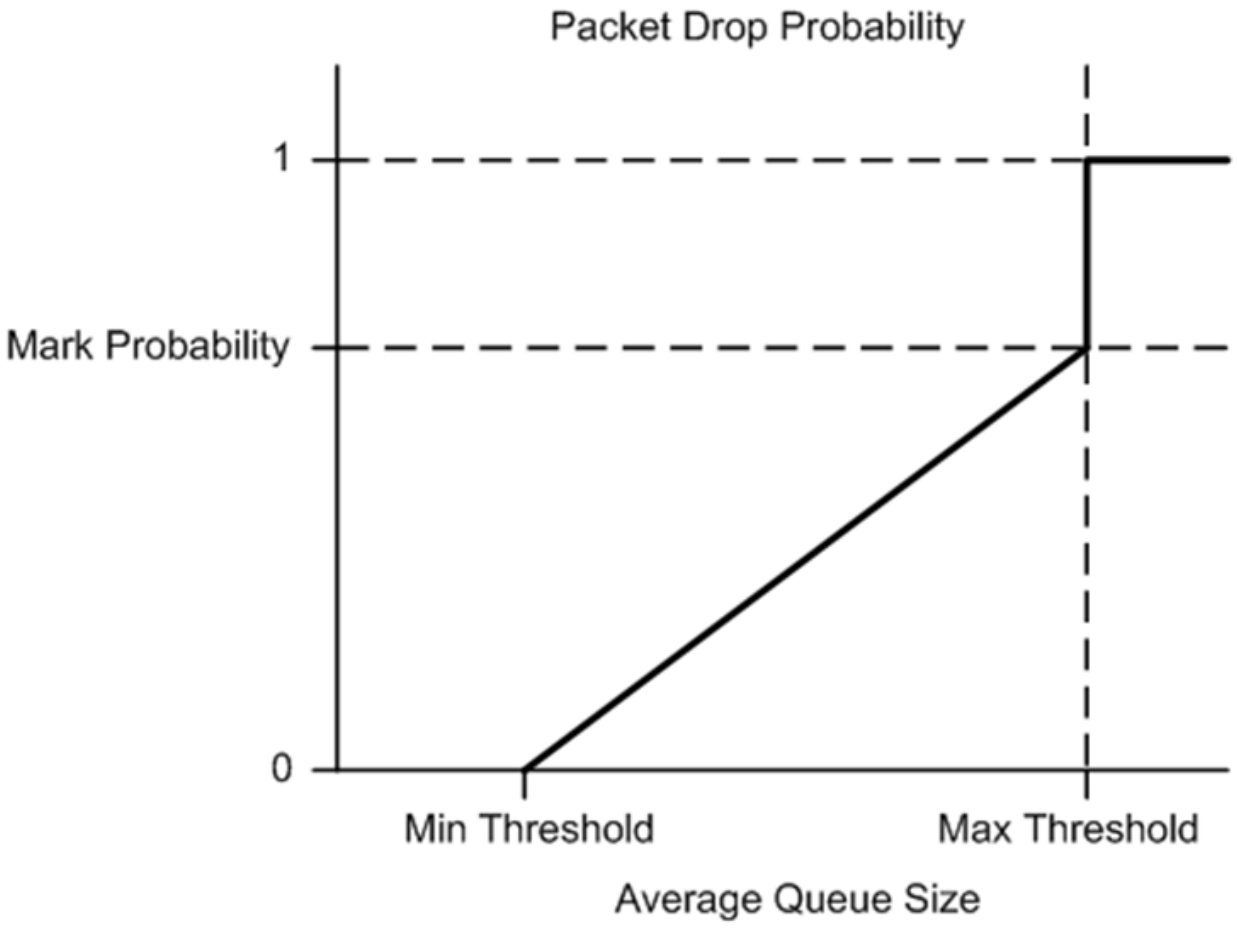


Fig. 4.58: Packet Drop Probability for a Given RED Configuration

Actual Drop Probability

If the average queue size is between the minimum and maximum thresholds, then the actual drop probability is calculated from the following equation.

$$p_a = \frac{p_b}{(2 - count \times p_b)}$$

Where:

- *Pb* = initial drop probability (from Equation 3)
- *count* = number of packets that have arrived since the last drop

The constant 2, in Equation 4 is the only deviation from the drop probability formulae given in the reference document where a value of 1 is used instead. It should be noted that the value pa computed from can be negative or greater than 1. If this is the case, then a value of 1 should be used instead.

The initial and actual drop probabilities are shown in Fig. 4.59. The actual drop probability is shown for the case where the formula given in the reference document1 is used (blue curve) and also for the case where the formula implemented in the dropper module, is used (red curve). The formula in the reference document results in a significantly higher drop rate compared to the mark probability configuration parameter specified by the user. The choice to deviate from the reference document is simply a design decision and one that has been taken by other RED implementations, for example, FreeBSD* ALTQ RED.

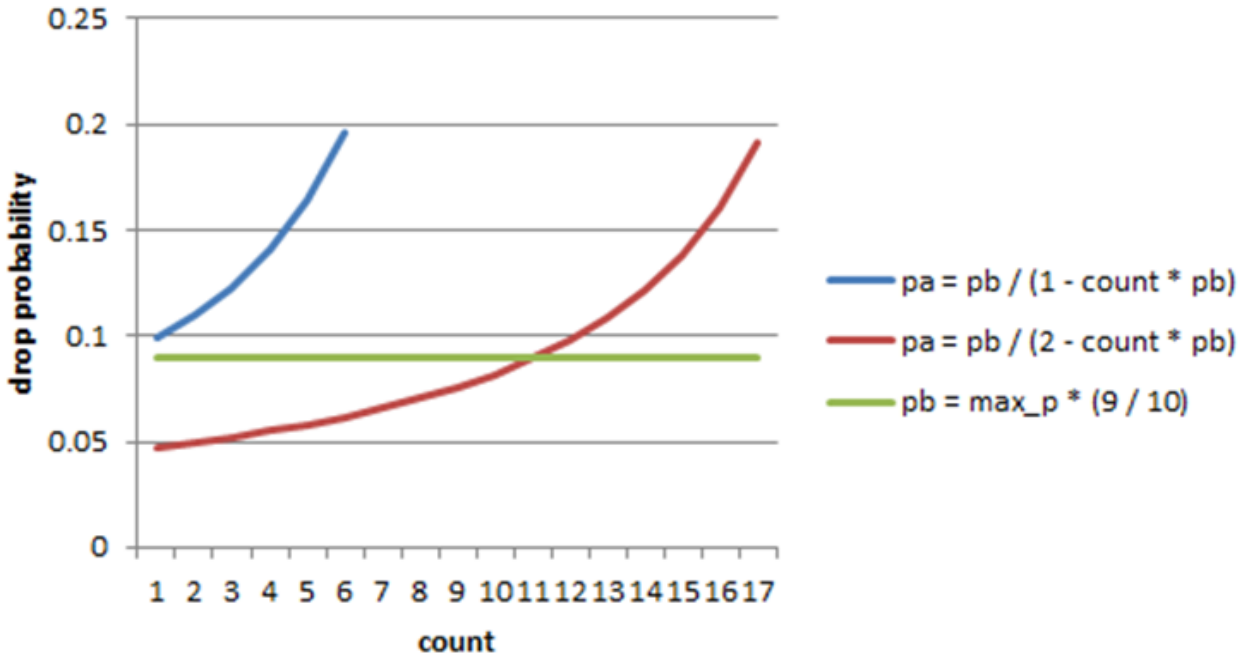


Fig. 4.59: Initial Drop Probability (pb), Actual Drop probability (pa) Computed Using a Factor 1 (Blue Curve) and a Factor 2 (Red Curve)

Queue Empty Operation

The time at which a packet queue becomes empty must be recorded and saved with the RED run-time data so that the EWMA filter block can calculate the average queue size on the next enqueue operation. It is the responsibility of the calling application to inform the dropper module through the API that a queue has become empty.

Source Files Location

The source files for the DPDK dropper are located at:

- DPDK/lib/librte_sched/rte_red.h
- DPDK/lib/librte_sched/rte_red.c

Integration with the DPDK QoS Scheduler

RED functionality in the DPDK QoS scheduler is disabled by default. To enable it, use the DPDK configuration parameter:

```
CONFIG_RTE_SCHED_RED=y
```

This parameter must be set to y. The parameter is found in the build configuration files in the DPDK/config directory, for example, DPDK/config/common_linuxapp. RED configuration parameters are specified in the rte_red_params structure within the rte_sched_port_params structure that is passed to the scheduler on initialization. RED parameters are specified separately for four traffic classes and three packet colors (green, yellow and red) allowing the scheduler to implement Weighted Random Early Detection (WRED).

Integration with the DPDK QoS Scheduler Sample Application

The DPDK QoS Scheduler Application reads a configuration file on start-up. The configuration file includes a section containing RED parameters. The format of these parameters is described in *Section2.23.3.1*. A sample RED configuration is shown below. In this example, the queue size is 64 packets.

Note: For correct operation, the same EWMA filter weight parameter (wred weight) should be used for each packet color (green, yellow, red) in the same traffic class (tc).

```
 ; RED params per traffic class and color (Green / Yellow / Red)

[red]
tc 0 wred min = 28 22 16
tc 0 wred max = 32 32 32
tc 0 wred inv prob = 10 10 10
tc 0 wred weight = 9 9 9

tc 1 wred min = 28 22 16
tc 1 wred max = 32 32 32
tc 1 wred inv prob = 10 10 10
tc 1 wred weight = 9 9 9

tc 2 wred min = 28 22 16
tc 2 wred max = 32 32 32
tc 2 wred inv prob = 10 10 10
tc 2 wred weight = 9 9 9

tc 3 wred min = 28 22 16
tc 3 wred max = 32 32 32
tc 3 wred inv prob = 10 10 10
tc 3 wred weight = 9 9 9
```

With this configuration file, the RED configuration that applies to green, yellow and red packets in traffic class 0 is shown in Table 4.68.

Table 4.68: RED Configuration Corresponding to RED Configuration File

RED Parameter	Configuration Name	Green	Yellow	Red
Minimum Threshold	tc 0 wred min	28	22	16
Maximum Threshold	tc 0 wred max	32	32	32
Mark Probability	tc 0 wred inv prob	10	10	10
EWMA Filter Weight	tc 0 wred weight	9	9	9

Application Programming Interface (API)

Enqueue API

The syntax of the enqueue API is as follows:

```
int rte_red_enqueue(const struct rte_red_config *red_cfg, struct rte_red *red, const␣
↪unsigned q, const uint64_t time)
```

The arguments passed to the enqueue API are configuration data, run-time data, the current size of the packet queue (in packets) and a value representing the current time. The time reference is in units of bytes, where a byte signifies the time duration required by the physical interface to send out a byte on the transmission medium (see Section 26.2.4.5.1 “Internal Time Reference”). The dropper reuses the scheduler time stamps for performance reasons.

Empty API

The syntax of the empty API is as follows:

```
void rte_red_mark_queue_empty(struct rte_red *red, const uint64_t time)
```

The arguments passed to the empty API are run-time data and the current time in bytes.

4.23.4 Traffic Metering

The traffic metering component implements the Single Rate Three Color Marker (srTCM) and Two Rate Three Color Marker (trTCM) algorithms, as defined by IETF RFC 2697 and 2698 respectively. These algorithms meter the stream of incoming packets based on the allowance defined in advance for each traffic flow. As result, each incoming packet is tagged as green, yellow or red based on the monitored consumption of the flow the packet belongs to.

Functional Overview

The srTCM algorithm defines two token buckets for each traffic flow, with the two buckets sharing the same token update rate:

- Committed (C) bucket: fed with tokens at the rate defined by the Committed Information Rate (CIR) parameter (measured in IP packet bytes per second). The size of the C bucket is defined by the Committed Burst Size (CBS) parameter (measured in bytes);
- Excess (E) bucket: fed with tokens at the same rate as the C bucket. The size of the E bucket is defined by the Excess Burst Size (EBS) parameter (measured in bytes).

The trTCM algorithm defines two token buckets for each traffic flow, with the two buckets being updated with tokens at independent rates:

- Committed (C) bucket: fed with tokens at the rate defined by the Committed Information Rate (CIR) parameter (measured in bytes of IP packet per second). The size of the C bucket is defined by the Committed Burst Size (CBS) parameter (measured in bytes);
- Peak (P) bucket: fed with tokens at the rate defined by the Peak Information Rate (PIR) parameter (measured in IP packet bytes per second). The size of the P bucket is defined by the Peak Burst Size (PBS) parameter (measured in bytes).

Please refer to RFC 2697 (for srTCM) and RFC 2698 (for trTCM) for details on how tokens are consumed from the buckets and how the packet color is determined.

Color Blind and Color Aware Modes

For both algorithms, the color blind mode is functionally equivalent to the color aware mode with input color set as green. For color aware mode, a packet with red input color can only get the red output color, while a packet with yellow input color can only get the yellow or red output colors.

The reason why the color blind mode is still implemented distinctly than the color aware mode is that color blind mode can be implemented with fewer operations than the color aware mode.

Implementation Overview

For each input packet, the steps for the srTCM / trTCM algorithms are:

- Update the C and E / P token buckets. This is done by reading the current time (from the CPU timestamp counter), identifying the amount of time since the last bucket update and computing the associated number of tokens (according to the pre-configured bucket rate). The number of tokens in the bucket is limited by the pre-configured bucket size;
- Identify the output color for the current packet based on the size of the IP packet and the amount of tokens currently available in the C and E / P buckets; for color aware mode only, the input color of the packet is also considered. When the output color is not red, a number of tokens equal to the length of the IP packet are subtracted from the C or E /P or both buckets, depending on the algorithm and the output color of the packet.

4.24 电源管理

DPDK电源管理功能允许用户空间应用程序通过动态调整CPU频率或进入不同的C-State来节省功耗。

- 根据RX队列的利用率动态调整CPU频率。
- 根据自适应算法进入不同层次的C-State，以推测在没有收到数据包的情况下暂停应用的短暂时间段。

调整CPU频率的接口位于电源管理库中。C-State控制是根据不同用例实现的。

4.24.1 CPU频率缩放

Linux内核提供了一个用于每个lcore的CPU频率缩放的cpufreq模块。例如，对于cpuX, /sys/devices/system/cpu/cpuX/cpufreq/具有以下用于频率缩放的sys文件:

- affected_cpus
- bios_limit
- cpuinfo_cur_freq

- cpuinfo_max_freq
- cpuinfo_min_freq
- cpuinfo_transition_latency
- related_cpus
- scaling_available_frequencies
- scaling_available_governors
- scaling_cur_freq
- scaling_driver
- scaling_governor
- scaling_max_freq
- scaling_min_freq
- scaling_setspeed

在DPDK中，scaling_governor在用户空间中配置。然后，用户空间应用程序可以通过写入scaling_setspeed来提示内核以根据用户空间应用程序定义的策略来调整CPU频率。

4.24.2 通过C-States调节Core负载

只要指定的lcore无任务执行，可以通过设置睡眠来改变Core状态。 在DPDK中，如果在轮询后没有接收到分组，则可以根据用户空间应用定义的策略来触发睡眠。

4.24.3 电源管理库API概述

电源管理库导出的主要方法是CPU频率缩放，包括:

- **频率上升**: 提示内核扩大特定lcore的频率。
- **频率下降**: 提示内核缩小特定lcore的频率。
- **频率最大**: 提示内核将特定lcore的频率最大化。
- **频率最小**: 提示内核将特定lcore的频率降至最低。
- **获取有效的频率**: 从sys文件中读取特定lcore的可用频率。
- **Freq获取**: 获取当前的特定lcore的频率。
- **频率设置**: 提示内核为特定的lcore设置频率。

4.24.4 示例

电源管理机制可用于在进行L3转发时节省功耗。

4.24.5 参考

- l3fwd-power: DPDK提供的示例应用程序，实现功耗管理下的L3转发。
- “功耗管理下的L3转发”章节请参阅《DPDK Sample Application’s User Guide》。

4.25 报文分类及访问控制

DPDK提供了一个访问控制库，它能够根据一组分类规则对输入数据包进行分类。

ACL库用于对具有多个类别的一组规则执行N元组搜索，并为每个类别找到最佳匹配（最高优先级）。库API提供以下基本操作:

- 创建一个新的访问控制（AC）上下文
- 向上下文中添加一条规则
- 对上下文中的所有规则，构建执行数据包分类所需的运行时结构
- 执行输入数据包分类
- 释放AC上下文及其运行时结构和相关的内存

4.25.1 概述

规则定义

当前的实现允许每个AC上下文的用户指定其自己的规则，通过该规则将执行数据包分类。 尽管规则字段布局几乎没有限制:

- 规则定义中的第一个字段必须是一个字节长。
- 所有后续字段必须分组为4个连续字节的集合。

这两个约束主要是出于性能原因 - 搜索函数将第一个输入字节作为流程设置的一部分进行处理，然后展开搜索函数的内部循环以一次处理四个输入字节。

为了定义AC规则中的各个字段，可以使用以下的数据结构:

```
struct rte_acl_field_def {
    uint8_t type;          /*< type - ACL_FIELD_TYPE. */
    uint8_t size;          /*< size of field 1,2,4, or 8. */
    uint8_t field_index;   /*< index of field inside the rule. */
    uint8_t input_index;   /*< 0-N input index. */
    uint32_t offset;       /*< offset to start of field. */
};
```

各个字段的含义如下:

- type 这个字段可以有如下三种选择:
  - _MASK - 用于像IP地址这种具有值和掩码字段
  - _RANGE - 用于像port这种具有上下限期间的字段
  - _BITMASK - 用于像协议标识符这种具有值和位掩码的字段
- size size参数定义了字段的字节数，可用的值为1, 2, 4, 或8字节。 注意，由于输入字节分组，必须将1或2字节的字段定义为构成4个连续输入字节的连续字段。 并且，最好将8个或更多个字节的字段定义为4个字节字段，以便构建过程可以消除额外的字段。
- field_index 表示规则中字段位置的基于零的值; 0到N-1为N字段。
- input_index 如上所示，所有输入字段，除了第一个，必须是4个连续字节的组。 输入索引则指定了该字段属于哪个输入组。
- offset 偏移定义了该字段的偏移值。该偏移从buffer参数开始算起。

举例，为了定义IPv4 5元组分类结构：

```
struct ipv4_5tuple {
    uint8_t proto;
    uint32_t ip_src;
    uint32_t ip_dst;
    uint16_t port_src;
    uint16_t port_dst;
};
```

可以使用以下数组字段定义：

```
struct rte_acl_field_def ipv4_defs[5] = {
    /* first input field - always one byte long. */
    {
        .type = RTE_ACL_FIELD_TYPE_BITMASK,
        .size = sizeof (uint8_t),
        .field_index = 0,
        .input_index = 0,
        .offset = offsetof (struct ipv4_5tuple, proto),
    },

    /* next input field (IPv4 source address) - 4 consecutive bytes. */
    {
        .type = RTE_ACL_FIELD_TYPE_MASK,
        .size = sizeof (uint32_t),
        .field_index = 1,
        .input_index = 1,
       .offset = offsetof (struct ipv4_5tuple, ip_src),
    },

    /* next input field (IPv4 destination address) - 4 consecutive bytes. */
    {
        .type = RTE_ACL_FIELD_TYPE_MASK,
        .size = sizeof (uint32_t),
        .field_index = 2,
        .input_index = 2,
       .offset = offsetof (struct ipv4_5tuple, ip_dst),
    },

    /*
     * Next 2 fields (src & dst ports) form 4 consecutive bytes.
     * They share the same input index.
     */
    {
        .type = RTE_ACL_FIELD_TYPE_RANGE,
        .size = sizeof (uint16_t),
        .field_index = 3,
        .input_index = 3,
        .offset = offsetof (struct ipv4_5tuple, port_src),
    },

    {
        .type = RTE_ACL_FIELD_TYPE_RANGE,
        .size = sizeof (uint16_t),
        .field_index = 4,
        .input_index = 3,
        .offset = offsetof (struct ipv4_5tuple, port_dst),
    },
```

```
};
```

这个IPv4 五元组的一个典型实例如下:

```
source addr/mask  destination addr/mask  source ports dest ports protocol/mask
192.168.1.0/24    192.168.2.31/32        0:65535      1234:1234  17/0xff
```

任何IPv4报文，具有协议ID为 17(UDP)，源IP为 192.168.1.[0-255]，目的IP为 192.168.2.31，源端口为 [0-65535] 且目的端口为 1234 的报文都匹配这个条目。

为了定义如下的IPv6 头部使用的2-元组分类: <protocol, IPv6 source address> :

```
struct struct ipv6_hdr {
    uint32_t vtc_flow;     /* IP version, traffic class & flow label. */
    uint16_t payload_len;  /* IP packet length - includes sizeof(ip_header). */
    uint8_t proto;         /* Protocol, next header. */
    uint8_t hop_limits;    /* Hop limits. */
    uint8_t src_addr[16];  /* IP address of source host. */
    uint8_t dst_addr[16];  /* IP address of destination host(s). */
} __attribute__((__packed__));
```

可以使用以下的数组字段:

```
struct struct rte_acl_field_def ipv6_2tuple_defs[5] = {
    {
        .type = RTE_ACL_FIELD_TYPE_BITMASK,
        .size = sizeof (uint8_t),
        .field_index = 0,
        .input_index = 0,
        .offset = offsetof (struct ipv6_hdr, proto),
    },

    {
        .type = RTE_ACL_FIELD_TYPE_MASK,
        .size = sizeof (uint32_t),
        .field_index = 1,
        .input_index = 1,
        .offset = offsetof (struct ipv6_hdr, src_addr[0]),
    },

    {
        .type = RTE_ACL_FIELD_TYPE_MASK,
        .size = sizeof (uint32_t),
        .field_index = 2,
        .input_index = 2,
        .offset = offsetof (struct ipv6_hdr, src_addr[4]),
    },

    {
        .type = RTE_ACL_FIELD_TYPE_MASK,
        .size = sizeof (uint32_t),
        .field_index = 3,
        .input_index = 3,
       .offset = offsetof (struct ipv6_hdr, src_addr[8]),
    },

    {
       .type = RTE_ACL_FIELD_TYPE_MASK,
```

```
        .size = sizeof (uint32_t),
        .field_index = 4,
        .input_index = 4,
        .offset = offsetof (struct ipv6_hdr, src_addr[12]),
    },
};
```

典型实例如下：

```
source addr/mask                                protocol/mask
2001:db8:1234:0000:0000:0000:0000:0000/48     6/0xff
```

任何IPv6报文，具有协议ID为6（TCP），且源IP在范围[2001:db8:1234:0000:0000:0000:0000:0000 - 2001:db8:1234:ffff:ffff:ffff:ffff:ffff]内的报文都将匹配这个规则。

在下面的例子中，搜索键值的最后一个元素是8bit，因此，出现输入字段的4个字节未完全占用的情况。分类结构为：

```
struct acl_key {
    uint8_t ip_proto;
    uint32_t ip_src;
    uint32_t ip_dst;
    uint8_t tos;       /*< 这里通常是32bit的元素 */
};
```

可以使用以下的数组字段：

```
struct rte_acl_field_def ipv4_defs[4] = {
    /* first input field - always one byte long. */
    {
        .type = RTE_ACL_FIELD_TYPE_BITMASK,
        .size = sizeof (uint8_t),
        .field_index = 0,
        .input_index = 0,
        .offset = offsetof (struct acl_key, ip_proto),
    },

    /* next input field (IPv4 source address) - 4 consecutive bytes. */
    {
        .type = RTE_ACL_FIELD_TYPE_MASK,
        .size = sizeof (uint32_t),
        .field_index = 1,
        .input_index = 1,
       .offset = offsetof (struct acl_key, ip_src),
    },

    /* next input field (IPv4 destination address) - 4 consecutive bytes. */
    {
        .type = RTE_ACL_FIELD_TYPE_MASK,
        .size = sizeof (uint32_t),
        .field_index = 2,
        .input_index = 2,
       .offset = offsetof (struct acl_key, ip_dst),
    },

    /*
     * 尽管tos字段只需要1个字节，但是我们仍旧要申请4字节
     */
```

```
    {
        .type = RTE_ACL_FIELD_TYPE_BITMASK,
        .size = sizeof (uint32_t), /* All the 4 consecutive bytes are allocated */
        .field_index = 3,
        .input_index = 3,
        .offset = offsetof (struct acl_key, tos),
    },
};
```

典型实例如下：

```
source addr/mask  destination addr/mask  tos/mask protocol/mask
192.168.1.0/24    192.168.2.31/32        1/0xff   6/0xff
```

任何IPv4报文，协议ID为6 (TCP)，源IP为192.168.1.[0-255]，目的IP为192.168.2.31，ToS 为1都匹配该规则。

当创建一组规则时，对于每个规则，还必须提供附加信息：

- **priority**: 衡量规则优先级的权重值，该值越大，优先级越高。 如果输入元组匹配多个规则，则返回优先级较高的规则。 请注意，如果输入元组匹配多于一个规则，并且这些规则具有相同的优先级，则未定义哪个规则作为匹配返回。 建议为每个规则分配唯一的优先级。
- **category_mask**: 每个规则使用位掩码值来选择规则的相关类别。 当执行查找时，返回每个类别的结果。 如果例如有四个不同的ACL规则集，一个用于访问控制，一个用于路由等，则通过使单个搜索能够返回多个结果来有效地提供“并行查找”。 每个集合可以被分配自己的类别，并且通过将它们组合成单个数据库，一个查找返回四个集合中的每一个的结果。
- **userdata**: 用户定义的数值。 对于每个类别，成功匹配返回最高优先级匹配规则的userdata字段。当没有规则匹配时，返回值为零。

Note: 将新规则添加到ACL上下文中时，所有字段必须是主机字节顺序（LSB）。 当为输入元组执行搜索时，该元组中的所有字段必须是网络字节顺序（MSB）。

RT 内存大小限制

构建阶段 (rte_acl_build()) 为给定的一组规则创建内部结构以供运行时遍历。 当前的实现是一组多分枝树，分枝为8. 根据规则集，可能会消耗大量的内存。 为了节省一些空间，ACL构建过程尝试将给定的规则集拆分为几个不相交的子集，并为每个子集构建一个单独的trie。 根据规则集，它可能会减少RT内存需求，但可能会增加分类时间。 在构建时有可能为给定的AC上下文指定内部RT结构的最大内存限制。 可以通过 **rte_acl_config** 结构的 **max_size** 字段来完成。 将其设置为大于0的值以指示 rte_acl_build()：

- 尝试最小化RT表中的尝试次数，但是
- 确保RT表的大小不会超过给定值。

将其设置为零可使rte_acl_build（）使用默认行为：尝试最小化RT结构的大小，但不会暴露任何硬限制。

这使用户能够对性能/空间权衡做出决定。

例如：

```
struct rte_acl_ctx * acx;
struct rte_acl_config cfg;
int ret;

/*
 * assuming that acx points to already created and
```

```
 * populated with rules AC context and cfg filled properly.
 */

/* try to build AC context, with RT structures less then 8MB. */
cfg.max_size = 0x800000;
ret = rte_acl_build(acx, &cfg);

/*
 * RT structures can't fit into 8MB for given context.
 * Try to build without exposing any hard limit.
 */
if (ret == -ERANGE) {
   cfg.max_size = 0;
   ret = rte_acl_build(acx, &cfg);
}
```

Classification 方法

在给定的AC上下文成功完成rte_acl_build()之后，它可以用于执行分类 - 搜索比输入数据高优先级的规则。

有几种分类算法实现：

- **RTE_ACL_CLASSIFY_SCALAR**: 通用实现，不需要任何特殊的硬件支持
- **RTE_ACL_CLASSIFY_SSE**: vector实现，可以实现8条流并行，需要 SSE 4.1 支持
- **RTE_ACL_CLASSIFY_AVX2**: vector实现，可以实现16条流并行，需要 AVX2 支持

纯粹是运行时决定哪种方法来选择，没有建立时间的差异。 所有实现都在相同的内部RT结构上运行，并使用类似的原理。 主要区别在于矢量实现可以手动利用IA SIMD指令并并行处理多个输入数据流。 在启动时，ACL库确定给定平台的最高可用分类方法，并将其设置为默认的。 虽然用户有能力覆盖给定ACL上下文的默认分类器功能，或使用非默认分类方法执行特定搜索。 在这种情况下，用户有责任确保给定的平台支持选定的分类实现。

4.25.2 API用法

Note: 关于 Access Control API 的更多纤细信息，请参考 *DPDK API Reference* 。

以下示例演示了更详细的多个类别的上面定义的规则的IPv4，5元组分类。

多类别报文分类

```
struct rte_acl_ctx * acx;
struct rte_acl_config cfg;
int ret;

/* define a structure for the rule with up to 5 fields. */

RTE_ACL_RULE_DEF(acl_ipv4_rule, RTE_DIM(ipv4_defs));

/* AC context creation parameters. */

struct rte_acl_param prm = {
```

```
    .name = "ACL_example",
    .socket_id = SOCKET_ID_ANY,
    .rule_size = RTE_ACL_RULE_SZ(RTE_DIM(ipv4_defs)),

    /* number of fields per rule. */

    .max_rule_num = 8, /* maximum number of rules in the AC context. */
};

struct acl_ipv4_rule acl_rules[] = {

    /* matches all packets traveling to 192.168.0.0/16, applies for categories: 0,1 */
    {
        .data = {.userdata = 1, .category_mask = 3, .priority = 1},

        /* destination IPv4 */
        .field[2] = {.value.u32 = IPv4(192,168,0,0),. mask_range.u32 = 16,},

        /* source port */
        .field[3] = {.value.u16 = 0, .mask_range.u16 = 0xffff,},

        /* destination port */
       .field[4] = {.value.u16 = 0, .mask_range.u16 = 0xffff,},
    },

    /* matches all packets traveling to 192.168.1.0/24, applies for categories: 0 */
    {
        .data = {.userdata = 2, .category_mask = 1, .priority = 2},

        /* destination IPv4 */
        .field[2] = {.value.u32 = IPv4(192,168,1,0),. mask_range.u32 = 24,},

        /* source port */
        .field[3] = {.value.u16 = 0, .mask_range.u16 = 0xffff,},

        /* destination port */
        .field[4] = {.value.u16 = 0, .mask_range.u16 = 0xffff,},
    },

    /* matches all packets traveling from 10.1.1.1, applies for categories: 1 */
    {
        .data = {.userdata = 3, .category_mask = 2, .priority = 3},

        /* source IPv4 */
        .field[1] = {.value.u32 = IPv4(10,1,1,1),. mask_range.u32 = 32,},

        /* source port */
        .field[3] = {.value.u16 = 0, .mask_range.u16 = 0xffff,},

        /* destination port */
        .field[4] = {.value.u16 = 0, .mask_range.u16 = 0xffff,},
    },

};


/* create an empty AC context  */
```

```
if ((acx = rte_acl_create(&prm)) == NULL) {

    /* handle context create failure. */

}

/* add rules to the context */

ret = rte_acl_add_rules(acx, acl_rules, RTE_DIM(acl_rules));
if (ret != 0) {
   /* handle error at adding ACL rules. */
}

/* prepare AC build config. */

cfg.num_categories = 2;
cfg.num_fields = RTE_DIM(ipv4_defs);

memcpy(cfg.defs, ipv4_defs, sizeof (ipv4_defs));

/* build the runtime structures for added rules, with 2 categories. */

ret = rte_acl_build(acx, &cfg);
if (ret != 0) {
   /* handle error at build runtime structures for ACL context. */
}
```

对于源IP地址：10.1.1.1和目标IP地址：192.168.1.15的元组，一旦执行如下操作：

```
uint32_t results[4]; /* make classify for 4 categories. */

rte_acl_classify(acx, data, results, 1, 4);
```

结果数组包含：

```
results[4] = {2, 3, 0, 0};
```

- 对于类别0，规则1和2都匹配，但规则2具有较高的优先级，因此[0]包含规则2的用户数据。
- 对于类别1，规则1和3都匹配，但规则3具有较高的优先级，因此[1]包含规则3的用户数据。
- 对于类别2和3，没有匹配，结果[2]和结果[3]包含零，这表明没有找到匹配的那些类别。

对于源IP地址为192.168.1.1和目标IP地址：192.168.2.11的元组，一旦执行：

```
uint32_t results[4]; /* make classify by 4 categories. */

rte_acl_classify(acx, data, results, 1, 4);
```

结果数组包含：

```
results[4] = {1, 1, 0, 0};
```

- 对于0和1类，只有规则1匹配。
- 对于类别2和3，没有匹配。

对于源IP地址：10.1.1.1和目标IP地址：201.212.111.12的元组，一旦执行：

```
uint32_t results[4]; /* make classify by 4 categories. */
rte_acl_classify(acx, data, results, 1, 4);
```

结果数组包含：

```
results[4] = {0, 3, 0, 0};
```

- 对于类别1，只有规则3匹配。
- 对于0,2和3类，没有匹配。

4.26 报文框架

4.26.1 设计目标

DPDK数据包框架的主要设计目标是：

- 提供标准方法来构建复杂的数据包处理流水线。 为常用的流水线功能模块提供可重复使用和可扩展的模板；
- 提供在同一流水线功能模块上在纯软件和硬件加速实现之间的切换能力；
- 提供灵活性和高性能之间的最佳权衡。 硬编码流水线通常提供最佳性能，但是不够灵活，而开发灵活的框架通常性能又较低；
- 提供一个逻辑上类似于Open Flow的框架。

4.26.2 概述

报文处理应用程序通常被设计为多级流水线，每个阶段的逻辑围绕查找表进行。 对于每个传入的数据包，查找表定义了要应用于数据包的一组操作，以及数据包处理的下一个阶段。

DPDK数据包框架通过定义流水线开发的标准方法，以及为常用的流水线模块提供可重用的模板库来最大限度地减少 构建数据包流水线所需要的开发工作量。

将一组输入端口和一组输出端口通过树形拓扑中的一组查找表来连接以构成流水线。 作为当前报文查找表的查找结果，其中一个表条目（查找命中）或默认条目（查找缺失）提供了要对当前数据包应用的一组操作， 以及数据包的下一跳，可以是另一个表，输出端口或者是丢弃报文。

数据包处理流程的一个例子如下 Fig. 4.60:

4.26.3 端口库设计

端口类型

Table 4.69 是可以使用Packet Framework实现的端口的非穷尽列表。

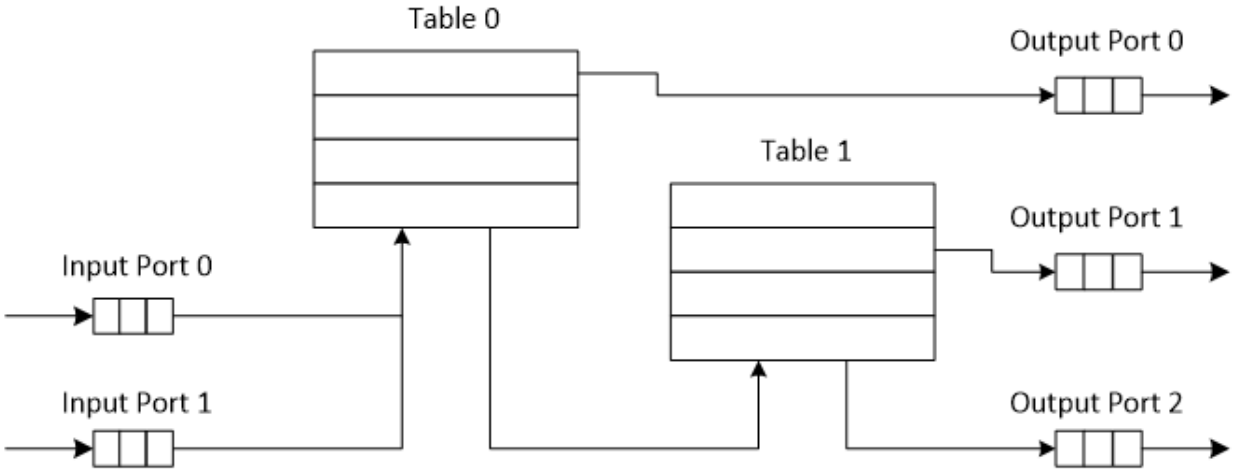


Fig. 4.60: 输入端口0和1通过表0和表1与输出端口0，1和2连接的数据包流水线示例

Table 4.69: 端口类型

#	端口类型	描述
1	SW ring	软件环形缓冲区用于应用程序之间的消息传递。使用DPDK的rte_ring。可能也是最常用的port类型
2	HW ring	用于与NIC、交换机或加速端口交互的缓冲区描述符队列。对于NIC端口，它使用rte_eth_rx_queue 或者rte_eth_tx_queue
3	IP reassembly	输入数据包是完整的数据报或者片段。输出数据包则是完整的IP数据报。
4	IP fragmentation	输入数据包是Jumbo帧 (IP 数据报，长度大于 MTU) 或者非Jumbo帧。输出数据包是非Jumbo帧。
5	Traffic manager	连接到特定NIC输出端口的流量管理器，根据预定义的SLA执行拥塞管理和分级调度。
6	KNI	接收/发送数据包到/从Linux内核空间
7	Source	输入端口用作数据包生成器。 类似于Linux内核 /dev/zero 设备
8	Sink	输出端口，用于删除所有的输入数据包。类似于Linux内核的 /dev/null 字符设备。

端口操作

每个端口是单向的，即输入端口或输出端口。 需要每个输入/输出端口来实现定义端口的初始化和运行时操作的抽象接口。 端口抽象接口描述于

Table 4.70: 20 端口抽象接口

#	端口操作	描述
1	Create	创建低级端口对象 (如，队列)，可以内部分配内存。
2	Free	释放低级端口对象使用的资源 (如，内存)
3	RX	读取一串输入数据包。只有输入端口才有这个操作。
4	TX	写一串输入数据包。非阻塞操作，只有输出端口有这个操作。
5	Flush	刷新输出缓冲区，只有输出端口有这个操作。

4.26.4 表库设计

表类型

Table 4.71 是可以用Packet Framework实现的表类型的非穷举列表。

Table 4.71: 表类型

#	表类型	描述
1	Hash table	查找关键字是n-元组。 通常，查找key使用哈希算法以产生用于标识查找结果的索引值。 与每个数据包查找关键字相关的数据可以从数据包中读取（预先计算的），或者在表查找时计算。 表查找、添加条目和删除条目操作，以及预先计算Key的任何流水线模块 都必须使用相同的哈希算法。 哈希表通常用于实现流分类表、ARP缓存、隧道协议路由表等。
2	Longest Prefix Match (LPM)	查找键值是IP地址。 表中的每个条目具有一个相关联的IP前缀。 表查找操作选择由查找键值匹配的IP前缀；在多个匹配的情况下，具有 最长前缀匹配的条目获胜。 通常用于实现IP路由表。
3	Access Control List (ACLs)	查找键值是7-元组，包括两个VLAN/MPLS 标签，目的IP，源IP，L4协议，L4目的端口，L4源端口。 每个表条目具有相关联的ACL优先级。ACL包含VLAN/ MPLS标签的位掩码，IP目的地址的IP前缀，IP源地址的IP前缀，L4协议和位掩码，L4目的端 口和位掩码，L4源端口和位掩码。 表查找操作选择与查找键匹配的ACL; 在多个匹配的情况下，优先级最高 的条目胜出。 通常用于实现防火墙等规则数据库。
4	Pattern matching search	查找键值为报文负载。 表示一个模式数据库，每个模式都有一个相关联的优先级。 表查找操作选择与输入报文匹配的模式，在多个匹配的情况下，最高优 先级匹配胜出
5	Array	查询键是表条目索引本身。

表操作接口

每个表都需要实现一个定义表的初始化和运行时操作的抽象接口。 表的抽象接口如下所述 Table 4.72.

Table 4.72: 表抽象接口

#	表操作	描述
1	Create	创建查找表的低级数据结构。 可以内部分配内存。
2	Free	释放查找表使用的所有资源。
3	Add entry	向查找表添加新条目。
4	Delete entry	从查找表中删除特定条目。
5	Lookup	查找一组输入数据包，并返回一个指定每个数据包的查找操作结果的位掩码 一个位表示相应数据包的查找命中，而一个清除位被查找错过 对于每个查找命中数据包，查找操作也返回指向被命中的表条目的指针， 其中包含要应用于数据包的操作和任何关联的元数据。 对于每个查找缺失数据包，要应用于数据包的操作和任何关联的元数据由预先配置为 查找缺失的默认表条目指定

哈希表设计

哈希表概述

哈希表很重要，因为查找操作针对速度进行了优化：搜索操作仅限于表中的某个哈希桶，而不是在表中所有元素间进行线性查找。

关联数组

关联数组是一个可以被指定为一组（键，值）对的函数，每个键最多可以存在一个可能的输入键集合。 对于给定的一个关联数组，可能的操作如下：

1. 添加 *(key, value)*: 当没有value与当前 *key*相关联时，*(key, *value*)* 关联将被创建。 当 *key* 已经关联了 *value0*，那么 (*key*, *value0*) 将被移除，并重新创建关联 *(key, value)* 。
2. 删除 *key*: 假如当前没有value关联到 *key*，这个操作将不起作用。 当 *key* 已经关联了 *value*，那么 *(key, value)* 将被移除。
3. 查找 *key*: 假如当前 *key*没有关联的*value*，那么这个操作返回查找缺失。 当 **key* 关联 *value*，那么这个操作将返回 *value*。 键值对 *(key, value)* 不做任何改变。

用于将输入key与关联数组中的key进行匹配的规则是 精确匹配，也就是说，key的大小及key值都必须精确匹配。

哈希函数

哈希函数确定性地将可变长度（密钥）的数据映射到固定大小的数据（散列值或密钥签名）。 通常地，key的大小要大于散列值的大小。 散列函数基本上将长key压缩成短哈希值。 几个key可以共享相同的哈希值，这就是哈希碰撞（哈希冲突）。

高质量散列函数可以做到均匀分布。 对于大量的key，当将哈希值的空间划分成固定数量的相等间隔（哈希桶）时，希望将哈希值均匀分布在这些间隔（均匀分布）上，而不是大多数哈希值 只分布在几个哈希桶中，其余的哈希桶在很大程度上没有使用（不均匀分布）。

哈希表

哈希表是使用散列函数进行操作的关联数组。 使用散列函数的原因是通过最小化必须与输入键进行比较的表键的数量来优化查找操作的性能。

哈希表不是将(key, value)对存储在单个链表中，而是保留多个链表（哈希桶）。 对于任意给定的key，存在单个哈希桶，并且该桶是基于key的哈希值唯一标识的。 一旦计算了哈希值，并且标识了哈希桶，key或者位于该桶中，或者根本不存在哈希表中，因此，根据key搜索可以从当前哈希表中唯一确认一个值。

哈希表查找的性能大大提高，前提是哈希表均匀分布在各个哈希桶之间，这个可以使用均匀分布的哈希函数来实现。 将key映射成哈希值的规则就是哈希函数，最简单的获取哈希桶的方式方式如下：

bucket_id = f_hash(key) % n_buckets;

通过选择桶的数量为2的幂，模运算符可以由按位AND逻辑来代替：

bucket_id = f_hash(key) & (n_buckets - 1);

为了减少哈希冲突，需要增加哈希表中哈希桶的数目。

在数据包处理上下文中，哈希表操作设计的操作顺序如下所示 Fig. 4.61:

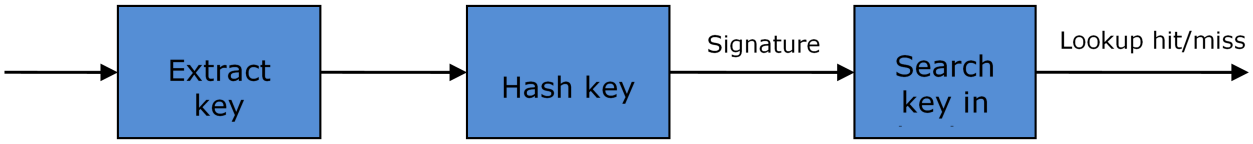


Fig. 4.61: 报文处理上下文中哈希表操作的步骤顺序

哈希表用例

流分类

描述*:* 对于每个输入数据包，流分类至少执行一次。 此操作将每个输入的数据包映射到通常包含数百万条流的流数据库中的某一条已知流上。

哈希表名称*:* 流分类表

keys 数目*:* 百万个以上

Key 格式*:* 报文字段n元组，用于唯一标识一条流/连接。 例如: DiffServ 5元组(源IP地址、目的IP地址、L4协议、L4源端口、L4目的端口)。 对于IPv4协议，且L4协议如TCP、UDP或者SCTP，DiffServ 5元组的大小是13B，对于IP6协议则是37B。

Key 值*:* 用于描述对当前流的报文应用什么样的处理动作和动作元数据。 与每个业务流相关的数据大小可以从8B到1KB不等。

ARP

描述*:* 一旦IP数据包的路由找到，也就是说输出接口和下一个中继站的IP地址是已知的，那么就需要下一个中继站的MAC地址，以便将数据包发到下一站。 下一跳的MAC地址成为输出以太网帧的目标MAC地址。

哈希表名称*:* ARP表

keys 数目*:* 数千个

Key 格式*:* 键值对(输出接口，下一跳IP地址)，通常IPv4是5B，IPv6是17B。

Key 值*:* 下一跳MAC地址6B。

哈希表类型

Table 4.73 列出了所有不同散列表类型共享的散列表配置参数。

Table 4.73: 所有散列表类型的通用配置参数

#	参数	描述
1	Key size	按照字节数来衡量，所有的Key具有相同的大小。
2	Key value (key data) size	按照字节数来衡量
3	Number of buckets	必须是2的幂次.
4	Maximum number of keys	必须是2的幂次.
5	Hash function	如: jhash, CRC hash, etc.
6	Hash function seed	传递给哈希函数的参数。
7	Key offset	存储在分组缓冲器中的分组元数据内的查找键字节阵列的偏移。

哈希桶溢出问题

在初始化时，为每个哈希表的桶分配4个keys的空间。 随着keys被添加到哈希表中，可能出现某个哈希桶中已经有4个keys的情况。 可以使用的方法有:

1. **LRU哈希表** 哈希桶中现有的key之一将被删除以添加新的key到他的位置。 每个哈希桶中的key数目不会超过4个。选择要丢弃的key的规则是LRU。 哈希表查找操作维护同一个哈希桶中不同key命中的顺序，所以，每当命中key时，该key就成为最近使用的key（MRU），因此LRU的key通常在链表尾部。当一个key被添加到哈希桶中时，它也成为新的MRU。 当需要选取和丢弃一个key时，第一个丢弃候选者，即当前的LRU Key总是被挑选出来丢弃。 LRU逻辑需要维护每个桶的特殊数据结构。
2. **可扩展桶的哈希表.** 哈希桶可以扩展空间，以存储4个以上的key。 这是通过在表初始化时分配额外的内存来实现的，这个内存用于创建一个空闲的key池（这个池的大小可配置，总是是4的倍数）。在添加key操作中，可以分配一组（4个key）的空间，如果空间不足，则添加失败。 在删除key操作中，当要删除的key是一组4个key中唯一使用的key时，将密钥删除，并将这组空间释放回key池。 在查找key操作中，如果当前存储的哈希桶处于扩展状态，并且在第一组4个key中找不到匹配项，则搜索将在后续的key中继续进行，知道桶中所有的key都被检查。 可扩展桶的哈希表需要维护每个表和每个存储哈希桶的特定数据结构。

Table 4.74: 可扩展桶散列表特定的配置参数

#	Parameter	Details
1	Number of additional keys	需要是2的幂次，至少是4

哈希值计算

哈希值计算的可用方法包括:

1. **预选计算的哈希值** Key查找操作被拆分到两个cpu core上。 第一个cpu core（通常是执行数据包接收的cpu core）从输入数据包中提取key，计算哈希值，并肩key和哈希值保存在接受数据包的缓冲区中作为数据包元数据。 第二个cpu core从数据包元数据中读取key和哈希值，并执行key查找操作。
2. **查找过程中计算的哈希值** 相同的cpu core从数据包元数据中读取key，用它来计算哈希值，并执行key查找操作。

Table 4.75: 预先计算哈希值的哈希表配置参数

#	Parameter	Details
1	Signature offset	数据包元数据内预先计算的哈希值的偏移

Key大小优化的哈希表

对于特定的key大小，key查找操作的数据结构和算法可以进行特殊的处理，以进一步提高性能，因此有如下选项：

1. **支持可配置密钥大小的实现**
2. **实现支持单个密钥大小** 通常key大小为8B或者16B。

可配置Key大小的哈希表查找操作

哈希桶搜索的性能是影响key查找的最要因素之一。 数据结构和算法旨在充分利用Intel CPU架构资源如：缓冲区存储结构，缓冲区存储带宽，外部存储器带宽，并行工作的多个执行单元，无序指令执行，特殊CPU指令等等。

哈希桶搜索逻辑并行处理多个输入数据包。 它被构建为几个阶段（3或者4阶段）流水线，每个流水线阶段处理来自突发输入的两个报文。 在每个流水线迭代中，数据包被推送到下一个流水线阶段：对于4阶段的流水线，两个数据包（刚刚完成阶段3）退出流水线，两个数据包（刚刚完成阶段2）正在执行阶段3，两个数据包（刚刚完成阶段1）正在执行阶段2，两个数据包（刚刚完成阶段0）正在执行阶段1，两个数据包（从输入数据包中读取）正在执行阶段0。 流水线持续迭代，直到来自输入分组的所有报文全部出流水线。

哈希桶搜索逻辑在存储器访问边界处分成流水线的不同阶段。 每个流水线阶段（高概率）使用存储在当前CPU core的L1/L2 cache中的数据结构，并在算法要求的下一个存储器访问之前终止。 当前流水线阶段通过预取下一个阶段需要的数据结构来完成，当下一个流水线阶段执行时，可以直接从L1/L2 cache中读取数据，从而避免L2/L3 cache miss造成的性能损失。

通过预取下一个水线阶段需要的数据结构，并且切换到针对不同分组的另一个流水线阶段，L2/L3 cache miss会大大减少。 这是因为在存储器读取L2 /L3 cache miss的数据成本很高，通常由于指令之间的数据依赖性，CPU执行单元必须停止，直到从L3高速缓冲存储器或外部DRAM存储器完成读取操作。 通过使用预取指令，存储器读取访问的延迟是隐藏的，只要在相应的数据结构被实际使用之前足够早地执行。

通过将处理分成在不同分组上执行的几个阶段（来自输入突发的分组交错），创建足够的工作以允许预取指令成功完成（在预取的数据结构被实际访问之前）以及数据指令之间的依赖关系被松动了。例如，对于4级流水线，对包0和1执行阶段0，然后在使用相同包0和1之前（即，在包0和1上执行阶1之前），使用不同的包：包2和3（执行阶段1），分组4和5（执行阶段2）以及分组6和7（执行阶段3）。 通过在将数据结构带入L1或L2高速缓冲存储器的同时执行有用的工作，隐藏了读取存储器访问的等待时间。 通过增加对同一数据结构的两次连续访问之间的差距，减轻了指令之间的数据依赖性;这允许最大限度地利用超标量和无序执行CPU架构，因为处于活动状态的CPU核心执行单元的数量（而不是由于指令之间的数据依赖性约束而空闲或停滞）被最大化。

哈希桶搜索逻辑也是在不是用任何分支指令的情况下实现的。 这避免了在每个分支错误预测实例上刷新CPU core执行管道的相关消耗。

可配置Key大小的哈希表

Fig. 4.62, Table 4.76 and Table 4.77 详细介绍用于实现可配置Key大小的哈希表的主要数据结构。

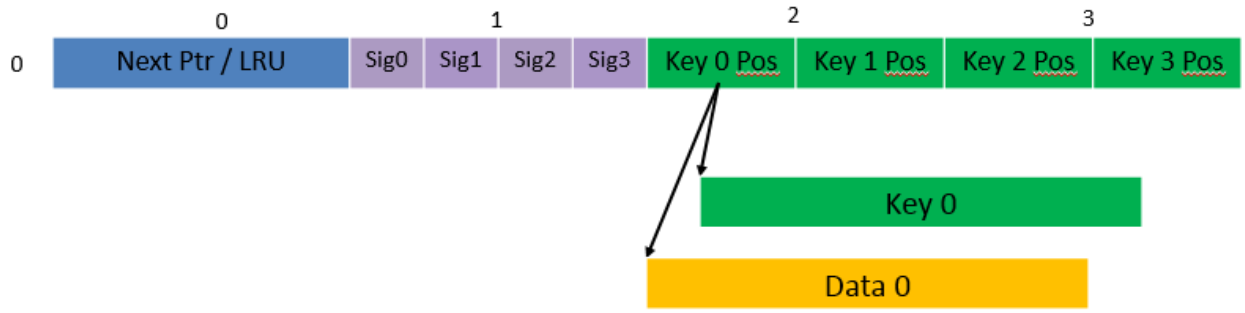


Fig. 4.62: 可配置Key大小的散列表的数据结构

Table 4.76: Main Large Data Structures (Arrays) used for Configurable Key Size Hash Tables

#	数组名	条目数	条目大小 (字节)	描述
1	Bucket array	n_buckets (可配置)	32	哈希表的桶数目
2	Bucket extensions array	n_buckets_ext (可配置)	32	只有可扩展哈希桶才会有
3	Key array	n_keys	key_size (可配置)	Keys
4	Data array	n_keys	entry_size (可配置)	Key values

Table 4.77: 数组输入的字段描述（可配置的密钥大小哈希表）

#	Field name	Field size (bytes)	Description
1	Next Ptr/LRU	8	对于LRU表，这些字段表示当前哈希桶的LRU列表 每个存储为2B的4个条目数组。条目0存储MRU Key的索引（0..3），而条目3存储LRU Key的索引。 对于可扩展桶表，该字段表示下一个指针（即指向链接到当前桶的下一组4个Key的指针）。如果存储桶当前已扩展，则下一个指针不为NULL 如果下一个指针不为NULL，则将该字段的位0设置为1，否则置位0。
2	Sig[0 .. 3]	4 x 2	如果 key X (X = 0 .. 3) 有效，则 sig X 的 bits 15 .. 1 存储 哈希值的最高 15 bits，而sig X bit 0 设置为1。 如果 key X 无效， sig X 被设置为0。
3	Key Pos [0 .. 3]	4 x 4	如果 key X (X = 0 .. 3)有效，那么 Key Pos X 代表 存储Key X的数组的索引，以及存储与Key X相关联的值的数据数组索引 如果 key X 无效，Key Pos X 的值未定义。

Fig. 4.63 and Table 4.78 详细说明桶搜索流水线阶段(LRU或可扩展桶，预先计算哈希值或"do-sig")。 对于每个流水线阶段，所描述的操作被应用于由该阶段处理的两个报文中的任何一个。

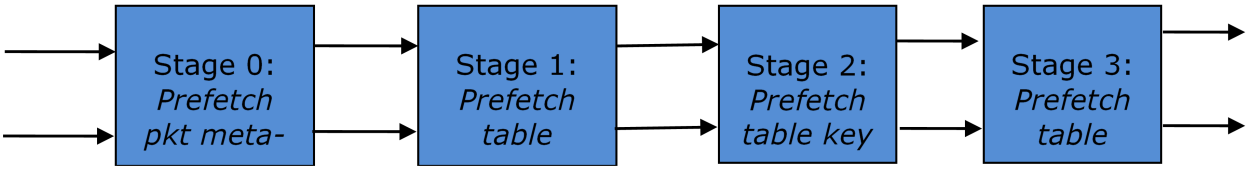


Fig. 4.63: 用于Key查找操作的流水线(可配置Key大小的哈希表)

Table 4.78: 桶搜索流水线阶段的描述(可配置Key大小的哈希表)

#	Stage name	描述
0	预取报文元数据	从输入数据包的突发中选择接下来的两个数据包。 预取包含Key和哈希值的数据包元数据。
1	Prefetch table bucket	从报文元数据中读取哈希值（对于可扩展表），从报文元数据中读取Key（LRU表） 使用哈希值识别桶ID。 设置哈希值的bit 0 为1 (用于匹配表中哈希值有效的Key） 预取桶。
2	Prefetch table key	从桶中读取哈希值。 将哈希值与报文中读取的哈希值进行对比，可能产生如下几种结果: *match* = TRUE（如果至少有一个哈希值匹配），FALSE（无哈希值匹配） *match_many* = TRUE（不止一个哈希值匹配，最多可以是4个），否则为FALSE。 *match_pos* = 哈希值匹配的第一个Key索引（当match为TRUE是才有效） 对于桶扩展的哈希表，如果next pointer有效设置*match_many*为TRUE 预取由 *match_pos* 标识的Key。
3	Prefetch table data	读取由 *match_pos* 标识的Key。 将该Key与输入的Key进行对比，产生如下结果：*match_key* = TRUE（如果两个key匹配），否则为FALSE。 当且仅当 *match* 和 *match_key* 都为TRUE时报告查找命中，否则未命中。 对于LRU表。使用无分支逻辑来更新桶的LRU表（当查找命中时，当前Key更改为MRU） 预取Key值（与当前Key关联的数据域）。

额外注意:

1. 桶搜索的流水线版本只有在输入突发中至少有7个包时才被执行。 如果输入突发中少于7个分组，则执行分组搜索算法的非优化实现。
2. 一旦针对输入突发中的所有分组已经执行了桶搜索算法的流水线版本，则对不产生查找命中的任何分组，如果 *match_many* 已经设置了，那么将同时执行桶优化算法的非优化实现。 作为执行非优化版的结果，这些分组中的一些可能产生查找命中或者未命中。 这并不会影响Key查找操作的性能，因为在同一组4个Key中匹配多个哈希值的概率或者处于扩展状态的桶的概率相对较小。

哈希值比较逻辑

哈希值比较逻辑描述如下 Table 4.79.

Table 4.79: Lookup Tables for Match, Match_Many and Match_Pos

#	mask	match (1 bit)	match_many (1 bit)	match_pos (2 bits)
0	0000	0	0	00
1	0001	1	0	00
2	0010	1	0	01
3	0011	1	1	00
4	0100	1	0	10
5	0101	1	1	00
6	0110	1	1	01
7	0111	1	1	00
8	1000	1	0	11
9	1001	1	1	00
10	1010	1	1	01
11	1011	1	1	00
12	1100	1	1	10
13	1101	1	1	00
14	1110	1	1	01
15	1111	1	1	00

输入的 *mask* 哈希 bit X (X = 0 .. 3) 设置为 1，如果输入的哈希值等于桶的哈希值X，否则则设置为0。 输出的 *match*, *match_many* 及 *match_pos* 是 1 bit, 1 bit 和 2 bits大小，其意义如上表描述。

如 Table 4.80 所描述的， *match* 和 *match_many* 的查找表可以折叠成一个32bit的值，*match_pos* 可以折叠成一个64bit的值。 给定输入的 *mask* ， *match* 的值， *match_many* 和 *match_pos* 的值可以通过索引他们各自的比特数来获得，分别用无分支逻辑取1，1和2 bits。

Table 4.80: Collapsed Lookup Tables for Match, Match_Many and Match_Pos

	Bit array	Hexadecimal value
match	1111_1111_1111_1110	0xFFFELLU
match_many	1111_1110_1110_1000	0xFEE8LLU
match_pos	0001_0010_0001_0011__0001_0010_0001_0000	0x12131210LLU

计算match, match_many 和 match_pos 的伪代码:

```
match = (0xFFFELLU >> mask) & 1;

match_many = (0xFEE8LLU >> mask) & 1;

match_pos = (0x12131210LLU >> (mask << 1)) & 3;
```

单一Key大小的哈希表

Fig. 4.64, Fig. 4.65, Table 4.81 and Table 4.82 详细描述了用于实现8B和16B Key的哈希表的主要的数据结构(包括LRU或扩展桶表，预先计算哈希值或"do-sig")。

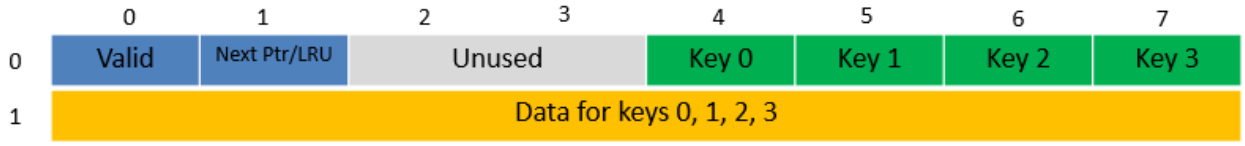


Fig. 4.64: 8B Key哈希表数据结构

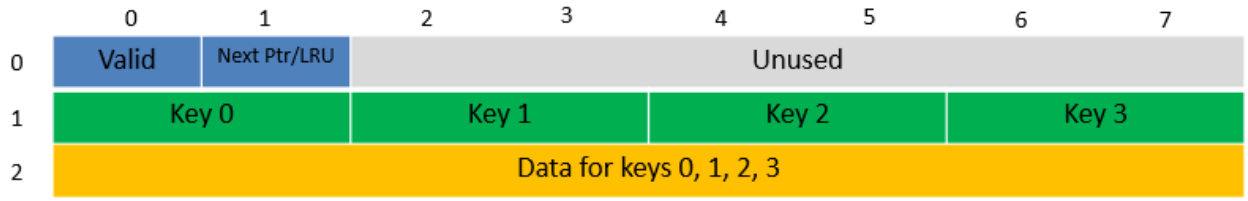


Fig. 4.65: 16B Key哈希表数据结构

Table 4.81: 用于8B和16B Key大小的哈希表的主要数据结构

#	Array name	Number of entries	Entry size (bytes)	Description
1	Bucket array	n_buckets (configurable)	*8-byte key size:* 64 + 4 x entry_size *16-byte key size:* 128 + 4 x entry_size	该哈希表的桶
2	Bucket extensions array	n_buckets_ext (configurable)	*8-byte key size:* 64 + 4 x entry_size *16-byte key size:* 128 + 4 x entry_size	仅用于扩展桶的哈希表

Table 4.82: 桶数组条目字段说明(8B和16B Key大小的哈希表)

#	Field name	Field size (bytes)	描述
1	Valid	8	如果Key X有效，那么Bit X (X = 0 .. 3) 设置为1，否则为0。 Bit 4 仅用于扩展桶的哈希表，用来帮助实现无分支逻辑。在这种情况下，如果next pointer有效，bit 4 设置为1，否则为0。
2	Next Ptr/LRU	8	对于LRU表，这个字段代表了当前桶中的LRU表。以2B代表4个条目存储为数组。条目 0 存储 MRU key (0 .. 3)，条目3存储LRU Key。 对于可扩展桶表，该字段表示下一个指针（即指向链接到当前桶的下一组4个键的指针）如果存储桶当前被扩展，则下一个指针不为NULL;否则为NULL。
3	Key [0 .. 3]	4 x key_size	Full keys.
4	Data [0 .. 3]	4 x entry_size	Full key values (key data) associated with keys 0 .. 3.

详细介绍用于实现8B和16B大小的Key的哈希表(包括LRU或可扩展桶表，预先计算哈希值或者”do-sig”)。对于每个流水线阶段，所描述的操作被应用于由该阶段处理的两个分组中的每一个。

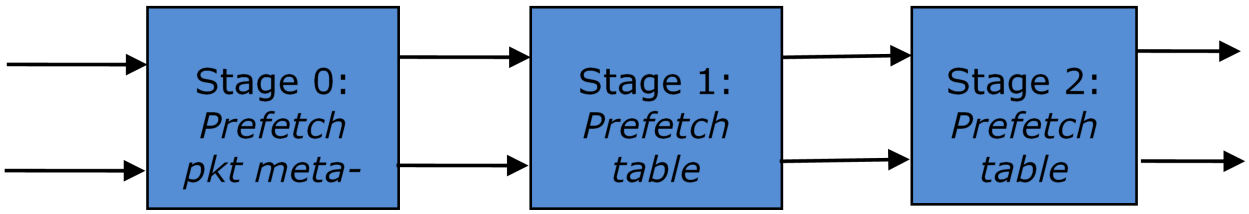


Fig. 4.66: 用于Key查找操作的桶搜索水线(单一Key大小的哈希表)

Table 4.83: 桶搜索流水线阶段的描述（8B和16B的Key散列表）

#	Stage name	Description
0	Prefetch packet meta-data	1. 从输入数据包的突发中选择接下来的两个数据包。 2. 预取包含Key和哈希值的数据包元数据。
1	Prefetch table bucket	1. 从报文元数据中读取哈希值（对于可扩展桶表），从报文元数据中读取Key(LRU表) 2. 使用哈希值来识别bucket ID。 3. 预取bucket。
2	Prefetch table data	1. 读取bucket。 2. 将输入的key与4个 bucket keys对比。 3. 如果有一个匹配，则报告查找命中。 4. 对于LRU表，使用无分支逻辑来更新存储区LRU列表（如果匹配当前Key变为MRU） 5. 预取与匹配Key相关联的键值（键数据）（这在查找命中和未命中时完成）。

额外注意：

1. 桶搜索算法的流水线版本只有在输入突发中至少有5个包时才会执行。 如果在输入分组突发中少于5个分组，则执行分组搜索算法的非优化实现。
2. 对于可扩展的分组哈希表，一旦已经对输入分组的突发中的所有分组执行了桶搜索算法的流水线版本，对于没有产生的任何分组但有扩展状态的桶，也执行桶搜索算法的非优化实现 查找命中。 作为执行非优化版本的结果，这些分组中的一些可能产生查找命中或查找未命中。 这不影响密钥查找操作的性能，因为处于扩展状态的桶的概率相对较小。

4.26.5 流水线库设计

一个流水线由如下几个元素定义：

1. 一组输入端口；
2. 一组输出端口；
3. 一组查找表；
4. 一组动作集。

输入端口通过互连表格的树状拓扑连接到输出端口。 表项包含定义在输入数据包上执行的动作和管道内的数据包流。

端口和表的连接

为了避免对流水线创建顺序的依赖，流水线元素的连通性在所有水线输入端口、输出端口和表创建完之后被定义。

一般的连接规则如下：

1. 每个输入端口连接到一个表，没有输入端口是悬空的；
2. 与其他表或输出端口的表连接由每个表条目和默认表条目的下一跳动作来调节。 表连接性是流畅的，因为表项和默认表项可以在运行时更新。
    - 一个表可以有多个条目（包括默认条目）连接到同一个输出端口。 一个表可以有不同的条目连接到不同的输出端口。 不同的表可以有连接到同一个输出端口的条目（包括默认条目）。
    - 一个表可以有多个条目（包括默认条目）连接到另一个表，在这种情况下，所有这些条目都必须指向同一个表。 这个约束是由API强制的，并且防止了树状拓扑的建立（只允许表连接），目的是简化流水线运行时执行引擎的实现。

端口动作

端口动作处理

可以为每个输出/输出端口分配一个操作处理程序，以定义在端口接收到的每个输入数据包上执行的操作。为特定的输入输出端口定义动作处理程序是可选的。（即可以禁用动作处理程序）

对于输入端口，操作处理程序在RX功能之后执行。对于输出端口，动作处理程序在TX功能之前执行。

操作处理程序可以快速丢弃数据包。

表动作

表动作处理

每个输入数据包上执行的操作处理程序可以分配给每个表。 为特定表定义动作处理程序是可选的（即可以禁用动作处理程序）。

在执行表查找操作之后执行动作处理程序，并且识别与每个输入分组相关联的表项。 操作处理程序只能处理用户定义的操作，而保留的操作（如下一跳操作）则由分组框架处理。 操作处理程序可以决定丢弃输入数据包。

预留动作

保留的动作有数据包框架直接处理，用户无法通过表动作处理程序配置更改其含义。 保留动作的一个特殊类别由下一跳动作来表示，它通过流水线来调节输入端口、表格和输出端口之间的数据流。 Table 4.84 列出了下一跳动作。

Table 4.84: Next Hop Actions (Reserved)

#	Next hop action	描述
1	Drop	丢弃当前报文。
2	Send to output port	发送当前报文到指定的输出端口。输出端口ID是存储在表元素中的元素据。
3	Send to table	发送当前报文到指定的表，表D是存储在表元素中的元数据。

用户动作

对于每个表，用户动作的含义都是通过表操作处理程序的配置来定义的。 不同的表可以配置不同的操作处理程序，因此用户动作及其相关元数据的含义对于每个表是私有的。 在同一个表中，所有表项（包括表默认项）共享用户动作及其相关元数据的相同定义，每个表项具有其自己的一组启用的用户动作以及它自己的操作副本元数据。

Table 4.85 包含用户动作的部分列表。

Table 4.85: 用户动作实例

#	User action	描述
1	Metering	使用srTCM和trTCM算法的每流量计量。
2	Statistics	更新每个流维护的统计信息计数器。
3	App ID	每个流状态机在流初始化时通过可变长度的分组序列进行馈送， 以识别流量类型和应用。
4	Push/pop labels	对当前报文执行VLAN/MPLS标签的入栈和出栈
5	Network Address Translation (NAT)	内部和外部IP地址（源和目的）的转化，L4协议源/目的端口转换
6	TTL update	递减TP TTL值，及更新IPv4报文的校验和。

4.26.6 多核处理

一个复杂的程序通常分成多核处理，多核之间通过SW队列进行通信。 由于以下硬件约束，在同一CPU内核上可以安装的表查找操作的数量通常有性能限制：了用的CPU周期，高速缓冲区的大小、高数缓存带宽、存储器传输带宽等。

由于应用程序跨越多个CPU核心，数据包框架便于创建多个流水线，将每个这样的流水线分配给不同的核心，并将所有的CPU核心级别的流水线互联为单个应用级复杂流水线。 例如，如果CPU核心A被分配运行流水线P1和CPU核心B流水线P2，则P1和P2的相互连接可以通过使相同的一组SW队列作为P1的输出端口和P2的输入端口来实现。

这种方法可以使用流水线，运行到完成（集群）或混合（混合）模型来开发应用程序。

允许同一个内核运行多个管道，但不允许多个内核运行相同的管道。

共享的数据结构

执行表查询的线程实际上是写线程，不仅仅是读操作。 即便指定的表查找算法是多线程安全的读者(如搜索算法数据结构的只读访问足以进行查找操作)， 一旦识别出当前报文的表项，通常期望线程更新存储在表项中的元数据(如增加命中该表项的数据包的计数器等)，这写操作将修改表项。 在此线程访问表项期间（写入或读取；持续时间与应用程序相关），由于数据一致性原因，不允许其他线程（执行查表或者添加删除表项操作）来修改此表项。

在多个线程之间共享一个表的机制：

1. **多个写线程** 线程需要使用类似信号量（每个表项不同的信号量）或原子操作的同步原语。 信号量的耗时通常很高。 原子指令的耗时通常高于普通指令。
2. **多个写线程，其中单个线程执行表查找操作，其他线程执行表添加、删除操作** 执行表添加、删除操作的线程向读取器发送表更新请求（通常是通过消息队列传递），这些请求执行实际的表更新，然后将相应发送回请求发起者。

3. **单个写线程执行表项添加、删除操作，多个度线程执行表查找操作，该查表操作只读表项，没有修改表项信息** 读线程使用主表的副本，而写线程更新镜像副本。 一旦写更新操作完成，写线程发信号给读线程，并等待所有的读线程切换到镜像副本上。

4.26.7 加速器

在初始化阶段通常通过检查作为系统一部分的HW设备（如通过PCI枚举操作）来检测加速器的存在。

具有加速功能的典型设备：

- 内联加速器：网卡、交换机、FPGA等；
- 外置加速器：芯片组、FPGA等

通常，为了支持特定功能模块，必须为每个加速器提供Packet Framework表、端口、动作的特定实现，所有实现共享相同的API：纯SW实现（无加速）、使用加速器A、使用加速器B等等。 这些实现之间的选择可以在构建或者在运行时完成，而不需要该变应用程序。

4.27 Vhost 库

Vhost库实现了一个用户空间virtio网络服务器，允许用户直接操作virtio。 换句话说，它允许用户通过VM virtio网络设备获取/发送数据包。 为了达到这个功能，一个vhost库需要实现：

- 访问guest内存：

  对于QEMU，这是通过使用 `-object memory-backend-file,share=on,...` 选项实现的。 这意味着QEMU将创建一个文件作为guest RAM。 选项 `share=on` 允许另一个进程映射该文件，这意味着该进程可以访问这个guest RAM。

- 知道关于vring所有必要的信息：

  诸如可用环形存储链表的存储空间。Vhost定义了一些消息（通过Unix套接字传递）来告诉后端所有需要知道如何操作vring的信息。

4.27.1 Vhost API 概述

以下是一些关键的Vhost API函数概述：

- `rte_vhost_driver_register(path, flags)`

  此函数将vhost驱动程序注册到系统中。`path` 指定Unix套接字的文件路径。

  当前支持的flags包括：

  - `RTE_VHOST_USER_CLIENT`

    当使用该flag时，DPDK vhost-user 作为客户端。 请参阅以下说明。

  - `RTE_VHOST_USER_NO_RECONNECT`

    当 DPDK vhost-user 作为客户端时，它将不断尝试连接到服务端（QEMU），知道成功。 这在以下两个情况中是非常有用的：

    * 当 QEMU 还没启动时
    * 当 QEMU 重启时(如guset OS 重启)

    这个重新连接选项是默认启用的，但是，可以通过设置这个标志来关闭它。

- RTE_VHOST_USER_DEQUEUE_ZERO_COPY

  设置此flag时将启用出队了零复制。默认情况下是禁用的。

  在设置此标志时，需要知道以下原则：

  * 零拷贝对于小数据包（小于512）是不好的。
  * 零拷贝对VM2VM情况比较好。对于两个虚拟机之间的ipref，提升性能可能高达70%(当TSO使能时).
  * 对于VM2NIC情况，nb_tx_desc 必须足够小：如果未启动virtio间接特性则 <=64，否则 <= 128。

    这是因为，当启用出队列零拷贝时，只有当相应的mbuf被释放时，客户端TX使用的vring才会被更新。因此，nb_tx_desc必须足够小，以便PMD驱动程序将耗尽可用的TX描述符，并及时释放mbufs。否则，guset TX vring将无mbuf使用。
  * Guest的内存应该使用应该使用huge page支持以获得更好的性能。最好使用1G大小的页面。

    当启用出队零拷贝时，必须建立guest 物理地址和host物理地址之间的映射。使用non-huge page则意味着更多的页面细分。为了简单起见，DPDK vhost对这些段进行了线性搜索，因此，段越少，我们得到的映射就越快。注意：将来我们可能使用树搜索来提升速度。

- rte_vhost_driver_set_features(path, features)

  此函数设置vhost-user驱动支持的功能位。vhost-user驱动可以是vhost-user net，但也可以是其他的，例如vhost-user SCSI。

- rte_vhost_driver_callback_register(path, vhost_device_ops)

  此函数注册一组回调函数，以便在发生某些事件时让DPDK应用程序采取适当的操作。目前支持以下事件：

  - new_device(int vid)

    这个回调在virtio设备准备就绪时调用，vid 是虚拟设备ID。

  - destroy_device(int vid)

    当virtio设备关闭时（或vhost连接中断），调用此函数处理。

  - vring_state_changed(int vid, uint16_t queue_id, int enable)

    当特定队列的状态发生改变，如启用或禁用，将调用此回调。

  - features_changed(int vid, uint64_t features)

    这个函数在feature改变时被调用。例如，VHOST_F_LOG_ALL 将分别在实时迁移的开始/结束时设置/清除。

- rte_vhost_driver_disable/enable_features(path, features))

  该函数禁用或启用某些功能。例如，可以使用它来禁用可合并的缓冲区和TSO功能，这两个功能默认都是启用的。

- rte_vhost_driver_start(path)

  这个函数触发vhost-user协商。它应该在初始化一个vhost-user驱动程序结束时被调用。

- rte_vhost_enqueue_burst(vid, queue_id, pkts, count)

  传输（入队）从host到guest的 count 包。

- rte_vhost_dequeue_burst(vid, queue_id, mbuf_pool, pkts, count)

  接收（出队）来自guest的 count 包，并将它们存储在 pkts。

4.27.2 Vhost-user 实现

Vhost-user 使用Unix套接字来传递消息。这意味着DPDK vhost-user的实现具有两种角色:

- DPDK vhost-user作为server:

  DPDK 将创建一个Unix套接字服务器文件，并监听来自前端的连接。

  注意，这是默认模式，也是DPDK v16.07之前的唯一模式。

- DPDK vhost-user最为client:

  与服务器模式不同，此模式不会创建套接字文件; 它只是试图连接到服务器（而不是创建文件的响应）。

  当DPDK vhost-user应用程序重新启动时，DPDK vhost-user将尝试再次连接到服务器。这是"重新连接"功能的工作原理。

  Note:

  – "重连" 功能需要 **QEMU v2.7** 及以上的版本。

  – vhost支持的功能在重新启动之前和之后必须完全相同。例如，如果TSO被禁用，但是重启之后被启用了，将导致未定义的错误。

无论使用哪种模式，建立连接之后，DPDK vhost-user 都将开始接收和处理来自QEMU的vhost消息。

对于带有文件描述符的消息，文件描述符可以直接在vhost进程中使用，因为它已经被Unix套接字安装了。

当前支持的vhost 消息包括:

- `VHOST_SET_MEM_TABLE`
- `VHOST_SET_VRING_KICK`
- `VHOST_SET_VRING_CALL`
- `VHOST_SET_LOG_FD`
- `VHOST_SET_VRING_ERR`

对于 `VHOST_SET_MEM_TABLE` 消息，QEMU将在消息的辅助数据中为每个存储区域及其文件描述符发送信息。 文件描述符用于映射该区域。

`VHOST_SET_VRING_KICK` 用作将vhost设备放入数据面的信号， `VHOST_GET_VRING_BASE` 用作从数据面移除vhost设备的信号。

当套接字连接关闭，vhost将销毁设备。

4.27.3 支持Vhost的vSwitch

有关更多vhost详细信息以及如何在vSwitch中支持vhost，请参阅《DPDK Sample Applications Guide》。

4.28 Metrics 库

Metrics 库实现了一个机制，通过这个机制，*producers* 可以发布numeric信息，供 *consumers* 后续查询。 实际上，生产者通常是其他库或者主进程，而消费者通常是应用程序。

Metrics 本身是一个静态值，并不是有PMD产生的。 Metric 信息是由推送模型填充的，其中生产者通过调用相关的更新函数来跟新metric库中包含的值。 消费者通过查询共享内存中的metric数据来获取metric信息。

对于每个mettic，为每个端口ID保留一个单独的值，并且在发布metric时，生产者需要指定哪个端口正在更新。 此外，还有一个特殊的ID `RTE_METRICS_GLOBAL`， 用于全局统计，不与任何单个设备关联。 由于metric库是自包含的，因此，对端口号的唯一限制是他们小于 `RTE_MAX_ETHPORTS`，不需要实际端口存在。

4.28.1 初始化库

在使用库之前，必须通过调用在共享内存中设置mettic存储的 `rte_metrics_init()` 来初始化它。 这也就是生产者将metric信息发布到哪里以及消费者从哪里查新metric信息。

```
rte_metrics_init(rte_socket_id());
```

这个初始化函数必须在主函数中调用，否则生产者和消费者可能在主程序或次进程中多次调用。？？

4.28.2 注册metrics

Metrics 必须先注册，这是生产者声明他们将要发布的metric的方式。 注册可以单独完成，也可以将一组metric标注为一个组。 单独注册使用接口 `rte_metrics_reg_name()` 实现:

```
id_1 = rte_metrics_reg_name("mean_bits_in");
id_2 = rte_metrics_reg_name("mean_bits_out");
id_3 = rte_metrics_reg_name("peak_bits_in");
id_4 = rte_metrics_reg_name("peak_bits_out");
```

一组metric注册使用 `rte_metrics_reg_names()` 完成:

```
const char * const names[] = {
    "mean_bits_in", "mean_bits_out",
    "peak_bits_in", "peak_bits_out",
};
id_set = rte_metrics_reg_names(&names[0], 4);
```

如果返回负数，表示注册失败。否则，返回值表示更新metic时使用的 *key* 值。 可以使用 `rte_metrics_get_names()` 获得将这些key值与metric名称映射起来的映射表。

4.28.3 更新 metric 值

一旦注册，生产者可以使用 `rte_metrics_update_value()` 函数更新给定端口的metric。 这个函数使用metric注册时返回的key值，也可以使用 `rte_metrics_get_names()` 查找。

```
rte_metrics_update_value(port_id, id_1, values[0]);
rte_metrics_update_value(port_id, id_2, values[1]);
rte_metrics_update_value(port_id, id_3, values[2]);
rte_metrics_update_value(port_id, id_4, values[3]);
```

如果metric被注册为一个集合，则可以使用 `rte_metrics_update_value()` 单独更新他们，或者使用 `rte_metrics_update_values()` 一起更新:

```
rte_metrics_update_value(port_id, id_set, values[0]);
rte_metrics_update_value(port_id, id_set + 1, values[1]);
rte_metrics_update_value(port_id, id_set + 2, values[2]);
```

```
rte_metrics_update_value(port_id, id_set + 3, values[3]);

rte_metrics_update_values(port_id, id_set, values, 4);
```

注意，`rte_metrics_update_values()` 不能用来更新 *multiple sets* 的metric，因为不能保证两个集合一个接一个地注册了连续的ID值。

4.28.4 查询 metrics

消费者可以通过使用返回 `struct rte_metric_value` 数组的接口 `rte_metrics_get_values()` 来查询metric库。 该数组中的每个条目都包含一个metric值及其关联的key。 key值和名称的映射可以使用 `rte_metrics_get_names()` 函数来获得，该函数返回由key索引的 `struct rte_metric_name` 数组。 以下将打印给定端口的所有metric：

```
void print_metrics() {
    struct rte_metric_name *names;
    int len;

    len = rte_metrics_get_names(NULL, 0);
    if (len < 0) {
        printf("Cannot get metrics count\n");
        return;
    }
    if (len == 0) {
        printf("No metrics to display (none have been registered)\n");
        return;
    }
    metrics = malloc(sizeof(struct rte_metric_value) * len);
    names =  malloc(sizeof(struct rte_metric_name) * len);
    if (metrics == NULL || names == NULL) {
        printf("Cannot allocate memory\n");
        free(metrics);
        free(names);
        return;
    }
    ret = rte_metrics_get_values(port_id, metrics, len);
    if (ret < 0 || ret > len) {
        printf("Cannot get metrics values\n");
        free(metrics);
        free(names);
        return;
    }
    printf("Metrics for port %i:\n", port_id);
    for (i = 0; i < len; i++)
        printf("  %s: %"PRIu64"\n",
            names[metrics[i].key].name, metrics[i].value);
    free(metrics);
    free(names);
}
```

4.28.5 Bit-rate 统计库

Bit-rate 库计算每个活动端口（即网络设备）的指数加权平均值和峰值比特率。 这些统计信息通过metric库使用以下名称进行发布：

- mean_bits_in: 平均入站比特率
- mean_bits_out: 平均出站比特率
- ewma_bits_in: 平均入站比特率 (EWMA 平滑)
- ewma_bits_out: 平均出站比特率 (EWMA 平滑)
- peak_bits_in: 峰值入站比特率
- peak_bits_out: 峰值出站比特率

一旦初始化，并以适当的频率计时，可以通过查询metric库来获取metric值。

初始化

在使用库之前，必须通过接口 rte_stats_bitrate_create() 来初始化，这个函数返回一个bit-rate计算对象。 由于bit-rate库使用metric来报告计算的统计量，因此bit-rate库需要将计算的统计量与metric库一起注册。 这通过辅助函数 rte_stats_bitrate_reg() 完成。

```
struct rte_stats_bitrates *bitrate_data;

bitrate_data = rte_stats_bitrate_create();
if (bitrate_data == NULL)
    rte_exit(EXIT_FAILURE, "Could not allocate bit-rate data.\n");
rte_stats_bitrate_reg(bitrate_data);
```

控制采样速率

由于库通过定期采样来工作，而不是使用内部线程，应用程序必须定期调用 rte_stats_bitrate_calc() 。 这个函数被调用的频率应该是计算统计所需要的预期采样频率。 例如，需要按秒统计，那么应该每秒钟调用一次这个函数。

```
tics_datum = rte_rdtsc();
tics_per_1sec = rte_get_timer_hz();

while( 1 ) {
    /* ... */
    tics_current = rte_rdtsc();
    if (tics_current - tics_datum >= tics_per_1sec) {
        /* Periodic bitrate calculation */
        for (idx_port = 0; idx_port < cnt_ports; idx_port++)
                rte_stats_bitrate_calc(bitrate_data, idx_port);
            tics_datum = tics_current;
        }
    /* ... */
}
```

4.28.6 延迟统计库

延迟统计库计算DPDK应用程序的数据包处理延迟，报告数据包处理所需的最小，平均和最大纳秒，以及处理延迟中的抖动。 使用以下名称通过metric库报告这些统计信息:

- min_latency_ns: 最小处理延迟（纳秒）
- avg_latency_ns: 平均处理延迟（纳秒）

- mac_latency_ns: 最大处理延迟（纳秒）
- jitter_ns: 处理等待时间的变化（纳秒）

一旦初始化并以适当的频率采样，可以通过查询metric库来获得这些统计数据。

初始化

使用库之前，需要调用函数 rte_latencystats_init() 进行初始化。

```
lcoreid_t latencystats_lcore_id = -1;

int ret = rte_latencystats_init(1, NULL);
if (ret)
    rte_exit(EXIT_FAILURE, "Could not allocate latency data.\n");
```

触发统计值更新

需要定期调用 rte_latencystats_update() 函数，以便更新延迟统计值信息。

```
if (latencystats_lcore_id == rte_lcore_id())
    rte_latencystats_update();
```

关闭库

完成之后，需要调用 rte_latencystats_uninit() 来关闭延迟统计库。

```
rte_latencystats_uninit();
```

4.29 端口热插拔框架

端口热插拔框架为DPDK应用程序提供了运行时添加、移除端口的能力。 由于框架一来PMD实现，所以热插拔的端口必须是PMD支持的端口才行。 此外，从DPDK程序中移除端口之后，框架并不提供从系统中删除设备的方法。 对于由物理网卡支持的端口，内核需要支持PCI热插拔功能。

4.29.1 概述

端口热插拔框架的基本要求:

- 使用端口热插拔框架的DPDK应用程序需要管理其自己的端口。

  端口热插拔矿机被实现为允许DPDK应用程序管理自己的端口。 例如，当应用程序调用添加端口的功能时，将返回添加的端口号。 DPDK应用程序也可以通过端口号移除该端口。

- 内核需要支持待添加、移除的物理设备端口。

  为了添加新的物理设备端口，设备首先被内核中的用户框架IO驱动识别。 然后DPDK应用程序可以调用端口热插拔功能来连接端口。 移除过程步骤刚好相反。

- 移除之前，必须先停止并关闭端口。

  DPDK应用程序在移除端口之前，必须调用 “rte_eth_dev_stop()” 和 “rte_eth_dev_close()” 函数。 这些函数将启动PMD的反初始化过程。

- 本框架不会影响传统的DPDK应用程序的行为。

  如果端口热插拔的功能没有被调用，所有传统的DPDK应用程序仍然可以不加修改地工作。

4.29.2 端口热插拔API概述

- 添加一个端口

  “rte_eth_dev_attach()” API 将端口添加到DPDK应用程序，并返回添加的端口号。 在调用API之前，设备应该被用户空间驱动IO框架识别。 API接收一个类似 “0000:01:00.0” 的pci地址或者是 “net_pcap0,iface=eth0” 这样的虚拟设备名称。 在虚拟设备名称情况下，格式与DPDK的一般‘-vdev’选项相同。

- 移除一个端口

  “rte_eth_dev_detach()” API 从DPDK应用程序中移除一个端口，并返回移除的设备的pci地址或虚拟设备名称。

4.29.3 引用

“testpmd” 支持端口热插拔框架。

4.29.4 限制

- 端口热插拔API并不是线程安全的。
- 本框架只能在Linux下使能，BSD并不支持。
- 为了移除端口，端口必须是igb_uio或VFIO管理的设备端口。
- 并非所有的PMD都支持移除功能。要知道PMD是否支持移除，请搜索 rte_eth_dev::data::dev_flags 中的 “RTE_ETH_DEV_DETACHABLE” 标志。 如果在PMD中定义该标志，则表示支持。

Part 2: Development Environment

4.30 源码组织

本节介绍DPDK框架中的源码组织结构。

4.30.1 Makefiles 和 Config

Note: 在本文的描述中 `RTE_SDK` 作为环境变量指向DPDK源码包解压出来的文件根目录。 参看 构建系统提供的有用变量 获取更多的变量描述。

由DPDK库和应用程序提供的 Makefiles 位于 `$(RTE_SDK)/mk` 中。

配置模板位于 `$(RTE_SDK)/config` 。这些模板描述了为每个目标启用的选项。 配置文件许多可以为DPDK库启用或禁用的选项，包括调试选项。 用户应该查看配置文件并熟悉这些选项。 配置文件同样也用于创建头文件，创建的头文件将位于新生成的目录中。

4.30.2 库

库文件源码位于目录 `$(RTE_SDK)/lib` 中。 按照惯例，库指的是为应用程序提供API的任何代码。 通常，它会生成一个 (`.a`) 文件，这个目录中可能也保存一些内核模块。

Lib目标包含如下项目

```
lib
+-- librte_cmdline       # 命令行接口
+-- librte_distributor   # 报文分发器
+-- librte_eal           # 环境抽象层
+-- librte_ether         # PMD通用接口
+-- librte_hash          # 哈希库
+-- librte_ip_frag       # IP分片库
+-- librte_kni           # 内核NIC接口
+-- librte_kvargs        # 参数解析库
+-- librte_lpm           # 最长前缀匹配库
+-- librte_mbuf          # 报文及控制缓冲区操作库
+-- librte_mempool       # 内存池管理器
+-- librte_meter         # QoS metering 库
+-- librte_net           # IP相关的一些头部
+-- librte_power         # 电源管理库
+-- librte_ring          # 软件无锁环形缓冲区
+-- librte_sched         # QoS调度器和丢包器库
+-- librte_timer         # 定时器库
```

4.30.3 驱动

驱动程序是为设备（硬件设备或者虚拟设备）提供轮询模式驱动程序实现的特殊库。 他们包含在 *drivers* 子目录中，按照类型分类，各自编译成一个库，其格式为 `librte_pmd_X.a` ，其中 `X` 是驱动程序的名称。

驱动程序目录下有个 *net* 子目录，包括如下项目:

```
drivers/net
+-- af_packet           # 基于Linux af_packet的pmd
+-- bonding             # 绑定pmd驱动
+-- cxgbe               # Chelsio Terminator 10GbE/40GbE pmd
+-- e1000               # 1GbE pmd (igb and em)
+-- enic                # Cisco VIC Ethernet NIC Poll-mode Driver
+-- fm10k               # Host interface PMD driver for FM10000 Series
+-- i40e                # 40GbE poll mode driver
+-- ixgbe               # 10GbE poll mode driver
+-- mlx4                # Mellanox ConnectX-3 poll mode driver
+-- null                # NULL poll mode driver for testing
+-- pcap                # PCAP poll mode driver
+-- ring                # Ring poll mode driver
+-- szedata2            # SZEDATA2 poll mode driver
+-- virtio              # Virtio poll mode driver
+-- vmxnet3             # VMXNET3 poll mode driver
+-- xenvirt             # Xen virtio poll mode driver
```

Note: 部分 `driver/net` 目录包含一个 `base` 子目录，这个目录通常包含用户不能直接修改的代码。 任何修订或增强都应该 `X_osdep.c` 或 `X_osdep.h` 文件完成。 请参阅base目录中本地的自述文件以获取更多的信息。

4.30.4 应用程序

应用程序是包含 `main()` 函数的源文件。 他们位于 `$(RTE_SDK)/app` 和 `$(RTE_SDK)/examples` 目录中。

应用程序目录包含用于测试DPPDK（如自动测试）或轮询模式驱动程序（test-pmd）的实例应用程序:

```
app
+-- chkincs             # Test program to check include dependencies
+-- cmdline_test        # Test the commandline library
+-- test                # Autotests to validate DPDK features
+-- test-acl            # Test the ACL library
+-- test-pipeline       # Test the IP Pipeline framework
+-- test-pmd            # Test and benchmark poll mode drivers
```

Example目录包含示例应用程序，显示了如何使用库:

```
examples
+-- cmdline             # Example of using the cmdline library
+-- exception_path      # Sending packets to and from Linux TAP device
+-- helloworld          # Basic Hello World example
+-- ip_reassembly       # Example showing IP reassembly
+-- ip_fragmentation    # Example showing IPv4 fragmentation
+-- ipv4_multicast      # Example showing IPv4 multicast
+-- kni                 # Kernel NIC Interface (KNI) example
+-- l2fwd               # L2 forwarding with and without SR-IOV
+-- l3fwd               # L3 forwarding example
+-- l3fwd-power         # L3 forwarding example with power management
+-- l3fwd-vf            # L3 forwarding example with SR-IOV
+-- link_status_interrupt # Link status change interrupt example
+-- load_balancer       # Load balancing across multiple cores/sockets
+-- multi_process       # Example apps using multiple DPDK processes
+-- qos_meter           # QoS metering example
+-- qos_sched           # QoS scheduler and dropper example
+-- timer               # Example of using librte_timer library
+-- vmdq_dcb            # Example of VMDQ and DCB receiving
+-- vmdq                # Example of VMDQ receiving
+-- vhost               # Example of userspace vhost and switch
```

Note: 实际的实例目录可能与上面显示的有所出入。 相关详细信息，请参考最新的DPDK代码。

4.31 开发套件构建系统

DPDK 需要一个构建系统用于编译等操作。 本节介绍 DPDK 框架中使用的约束和机制。

这个框架有两个使用场景:

- 编译DPDK库和示例应用程序，该框架生成特定的二进制库，包含文件和示例应用程序。
- 使用安装的DPDK二进制编译外部的应用程序或库。

4.31.1 编译DPDK二进制文件

以下提供了如何构建DPDK二进制文件。

建立目录概念

安装之后，将创建一个构建目录结构。 每个构件目录包含文件、库和应用程序。

构建目录特定于配置的体系结构、执行环境、工具链。 可以存在几个构建目录共享源码，但是配置不一样的情况。

例如，要使用配置模板 config/defconfig_x86_64-linuxapp 创建一个名为 my_sdk_build_dir 的构建目录，我们使用如下命令：

```
cd ${RTE_SDK}
make config T=x86_64-native-linuxapp-gcc O=my_sdk_build_dir
```

这会创建一个新的 new my_sdk_build_dir 目录，之后，我们可以使用如下的命令进行编译：

```
cd my_sdk_build_dir
make
```

相当于：

```
make O=my_sdk_build_dir
```

目录 my_sdk_build_dir 的内容是：

```
-- .config                          # used configuration

-- Makefile                         # wrapper that calls head Makefile
                                    # with $PWD as build directory


    -- build                                #All temporary files used during build
    +--app                                  # process, including . o, .d, and .cmd␣
↪files.
        |  +-- test                         # For libraries, we have the .a file.
        |  +-- test.o                       # For applications, we have the elf file.
        |  `-- ...
        +-- lib
            +-- librte_eal
            |   `-- ...
            +-- librte_mempool
            |  +--  mempool-file1.o
            |  +--  .mempool-file1.o.cmd
            |  +--  .mempool-file1.o.d
            |  +--   mempool-file2.o
            |  +--  .mempool-file2.o.cmd
            |  +--  .mempool-file2.o.d
            |  `--  mempool.a
            `-- ...

-- include                 # All include files installed by libraries
    +-- librte_mempool.h    # and applications are located in this
    +-- rte_eal.h           # directory. The installed files can depend
    +-- rte_spinlock.h      # on configuration if needed (environment,
    +-- rte_atomic.h        # architecture, ..)
    `-- \*.h ...

-- lib                     # all compiled libraries are copied in this
    +-- librte_eal.a        # directory
    +-- librte_mempool.a
```

```
    `-- \*.a ...

-- app                      # All compiled applications are installed
+ --test                    # here. It includes the binary in elf format
```

请参阅 *Development Kit Root Makefile Help* 获取更详细的信息。

4.31.2 构建外部应用程序

由于DPDK本质上是一个开发工具包，所以最终用户的第一个目标就是使用这个SDK创建新的应用程序。 要编译应用程序，用户必须设置 RTE_SDK 和 RTE_TARGET 环境变量。

```
export RTE_SDK=/opt/DPDK
export RTE_TARGET=x86_64-native-linuxapp-gcc
cd /path/to/my_app
```

对于一个新的应用程序，用户必须创建新的 Makefile 并包含指定的 .mk 文件，如 ${RTE_SDK}/mk/rte.vars.mk 和 ${RTE_SDK}/mk/rte.app.mk。 这部分内容描述请参考 *Building Your Own Application*.

根据 Makefile 所选定的目标（架构、机器、执行环境、工具链）或环境变量，应用程序和库将使用适当的h头文件进行编译，并和适当的a库链接。 这些文件位于 ${RTE_SDK}/arch-machine-execenv-toolchain，由 ${RTE_BIN_SDK} 内部引用。

为了编译应用程序，用户只需要调用make命令。编译结果将置于 /path/to/my_app/build 目录。

示例应用程序在example目录中提供。

4.31.3 Makefile 描述

DPDK Makefiles 的通用规则

在DPDK中，Makefiles始终遵循相同的方案:

1. 起始处包含 $(RTE_SDK)/mk/rte.vars.mk 文件。
2. 为RTE构建系统定义特殊的变量。
3. 包含指定的 $(RTE_SDK)/mk/rte.XYZ.mk 文件，其中 XYZ 可以是 app、lib、extapp, extlib、obj、gnuconfigure等等，取决于要编译什么样的目标文件。 请参阅 *See Makefile Types* 描述。
4. 包含用户定义的规则及变量。

   以下是一个简单的例子，用于便于一个外部应用程序:

```
include $(RTE_SDK)/mk/rte.vars.mk

# binary name
APP = helloworld

# all source are stored in SRCS-y
SRCS-y := main.c

CFLAGS += -O3
CFLAGS += $(WERROR_FLAGS)
```

```
include $(RTE_SDK)/mk/rte.extapp.mk
```

Makefile 类型

根据Makefile最后包含的 .mk 文件，Makefile将具有不同的角色。 注意到，并不能在同一个Makefile文件中同时编译库和应用程序。 因此，用户必须创建两个独立的Makefile文件，最好是置于两个不同的目录中。

无论如何，rte.vars.mk 文件必须包含用户Makefile。

应用程序

这些 Makefiles 生成一个二进制应用程序。

- rte.app.mk: DPDK框架中的应用程序。
- rte.extapp.mk: 外部应用程序。
- rte.hostapp.mk: 建立DPDK的先决条件和工具。

库

创建一个 .a 库。

- rte.lib.mk: DPDK中的库。
- rte.extlib.mk: 外部库。
- rte.hostlib.mk: DPDK中的host库。

安装

- rte.install.mk: 不构建任何东西，只是用于创建链接或者将文件复制到安装目录。 这对于开发包框架中包含的文件非常有用。

内核模块

- rte.module.mk: 构建DPDK内核模块。

对象

- rte.obj.mk: DPDK中的目标文件聚合（合并一些o文件成一个）。
- rte.extobj.mk: 外部目标文件聚合（合并外部的一些o文件）。

杂

- rte.doc.mk: DPDK中的文档。
- rte.gnuconfigure.mk: 构建一个基于配置的应用程序。
- rte.subdir.mk: 构建几个目录。

内部生成的构建工具

app/dpdk-pmdinfogen

dpdk-pmdinfogen 扫描各种总所周知的符号名称对象文件。这些目标文件由各种宏定义，用于导出关于pmd文件的硬件支持和使用的重要信息。 例如宏定义：

```
RTE_PMD_REGISTER_PCI(name, drv)
```

创建以下的符号：

```
static char this_pmd_name0[] __attribute__((used)) = "<name>";
```

将被 dpdk-pmdinfogen 扫描。使用这个虚拟系，可以从目标文件中导出其他相关位信息，并用于产生硬件支持描述， 然后 dpdk-pmdinfogen 按照以下格式编码成 json 格式的字符串：

```
static char <name_pmd_string>="PMD_INFO_STRING=\"{'name' : '<name>', ...}\"";
```

然后可以通过外部工具搜索这些字符串，以确定给定库或应用程序的硬件支持。

构建系统提供的有用变量

- RTE_SDK: DPDK源码绝对路径。 编译DPDK时，该变量由框架自动设置。 如果编译外部应用程序，它必须由用户定义为环境变量。
- RTE_SRCDIR: DPDK源码根路径。 当编译DPDK时，RTE_SRCDIR = RTE_SDK。 当编译外部应用程序时，该变量指向外部应用程序源码的跟目录。
- RTE_OUTPUT: 输出文件的路径。 通常情况下，他是 $(RTE_SRCDIR)/build，但是可以通过make命令中的 O= 选项来重新指定。
- RTE_TARGET: 一个字符串，用于我们正在构建的目标。 格式是arch-machine-execenv-toolchain。 当编译DPDK时，目标是有构建系统从配置(.config)中推导出来的。 当构建外部应用程序时，必须由用户在Makefile中指定或作为环境变量。
- RTE_SDK_BIN: 参考 $(RTE_SDK)/$(RTE_TARGET)。
- RTE_ARCH: 定义架构(i686, x86_64)。 它与 CONFIG_RTE_ARCH 相同，但是没有字符串的双引号。
- RTE_MACHINE: 定义机器。 它与 CONFIG_RTE_MACHINE 相同，但是没有字符串的双引号。
- RTE_TOOLCHAIN: 定义工具链 (gcc , icc)。 它与 CONFIG_RTE_TOOLCHAIN 相同，但是没有字符串的双引号。
- RTE_EXEC_ENV: 定义运行环境 (linuxapp)。 它与 CONFIG_RTE_EXEC_ENV 相同，但是没有字符串的双引号。
- RTE_KERNELDIR: 这个变量包含了将被用于编译内核模块的内核源的绝对路径。 内核头文件必须与目标机器（将运行应用程序的机器）上使用的头文件相同。 默认情况下，变量设置为 /lib/modules/$(shell uname -r)/build，当目标机器也是构建机器时，这是正确的。

- RTE_DEVEL_BUILD: 更严格的选项（停止警告）。 它在默认的git树中。

只能在Makefile中设置/覆盖的变量

- VPATH: 构建系统将搜索源码的路径列表。默认情况下，RTE_SRCDIR将被包含在VPATH中。
- CFLAGS: 用于C编译的标志。用户应该使用+=在这个变量中附加数据。
- LDFLAGS: 用于链接的标志。用户应该使用+=在这个变量中附加数据。
- ASFLAGS: 用于汇编的标志。用户应该使用+=在这个变量中附加数据。
- CPPFLAGS: 用于给C预处理器赋予标志的标志（仅在汇编.S文件时有用）。用户应该使用+=在这个变量中附加数据。
- LDLIBS: 在应用程序中，链接的库列表（例如，-L / path / to / libfoo -lfoo）。用户应该使用+=在这个变量中附加数据。
- SRC-y: 在应用程序，库或对象Makefiles的情况下，源文件列表（.c，.S或.o，如果源是二进制文件）。源文件必须可从VPATH获得。
- INSTALL-y-$(INSTPATH): 需要安装到 $(INSTPATH) 的文件列表。 这些文件必须在VPATH中可用，并将复制到 $(RTE_OUTPUT)/$(INSTPATH)。几乎可以在任何 RTE Makefile 中可用。
- SYMLINK-y-$(INSTPATH): 需要安装到 $(INSTPATH) 的文件列表。 这些文件必须在VPATH中可用并将链接到 (symbolically) 在 $(RTE_OUTPUT)/$(INSTPATH)。 几乎可以在任何 RTE Makefile 中可用。
- PREBUILD: 构建之前要采取的先决条件列表。用户应该使用+=在这个变量中附加数据。
- POSTBUILD: 主构建之后要执行的操作列表。用户应该使用+=在这个变量中附加数据。
- PREINSTALL: 安装前要执行的先决条件操作的列表。 用户应该使用+=在这个变量中附加数据。
- POSTINSTALL: 安装后要执行的操作列表。用户应该使用+=在这个变量中附加数据。
- PRECLEAN: 清除前要执行的先决条件操作列表。用户应该使用+=在这个变量中附加数据。
- POSTCLEAN: 清除后要执行的先决条件操作列表。用户应该使用+=在这个变量中附加数据。
- DEPDIRS-$(DIR): 仅在开发工具包框架中用于指定当前目录的构建是否依赖于另一个的构建。这是正确支持并行构建所必需的。

只能在命令行上由用户设置/覆盖的变量

一些变量可以用来配置构建系统的行为。在文件 *Development Kit Root Makefile Help* 及 *External Application/Library Makefile Help* 中有描述。

- WERROR_CFLAGS: 默认情况下，它被设置为一个依赖于编译器的特定值。 鼓励用户使用这个变量，如下所示：

```
CFLAGS += $(WERROR_CFLAGS)
```

这避免了根据编译器（icc或gcc）使用不同的情况。而且，这个变量可以从命令行覆盖，这允许绕过标志用于测试目的。

可以在Makefile或命令行中由用户设置/覆盖的变量

- CFLAGS_my_file.o: 为my_file.c的C编译添加的特定标志。
- LDFLAGS_my_app: 链接my_app时添加的特定标志。
- EXTRA_CFLAGS: 在编译时，这个变量的内容被附加在CFLAGS之后。

- EXTRA_LDFLAGS: 链接后，将此变量的内容添加到LDFLAGS之后。
- EXTRA_LDLIBS: 链接后，此变量的内容被添加到LDLIBS之后。
- EXTRA_ASFLAGS: 组装后这个变量的内容被附加在ASFLAGS之后。
- EXTRA_CPPFLAGS: 在汇编文件上使用C预处理器时，此变量的内容将附加在CPPFLAGS之后。

4.32 DPDK 根目录 Makefile 理解

DPDK提供了一个根目录级别的Makefile，包含配置，构建，清理，测试，安装等目的。 这些操作将在下面的部分中进行解释。

4.32.1 配置 Targets

配置 target 需要使用 T=mytarget 指定target的名称，这个操作不能省略。 可用的target列表位于 $(RTE_SDK)/config 中(移除defconfig _ 前缀)。

配置target还支持使用 O=mybuilddir 来指定输出目录的名称。 这是一个可选配置，默认的输出目录是build。

- Config

  这将创建一个构建目录，并从模板中生成一个配置。 同时会在构建目录下生成一个 Makefile 文件。

  例如：

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

4.32.2 构建 Targets

构建 targets 支持输出目录名称可选规则，使用 O=mybuilddir。 默认的输出目录是build。

- all, build or just make

  在前面由make config创建的目录上构建DPDK。

  例如：

```
make O=mybuild
```

- clean

  清除所有由 make build 生成的目标文件。

  例如：

```
make clean O=mybuild
```

- %_sub

  只构建某个目录，而不管对其他目录的依赖性。

  例如：

```
make lib/librte_eal_sub O=mybuild
```

- %_clean

  清除对子目录的构建操作结果。

  例如：

```
make lib/librte_eal_clean O=mybuild
```

4.32.3 安装 Targets

- Install

  可用的 targets 列表位于 $(RTE_SDK)/config (移除 defconfig_ 前缀).

  可以使用 GNU 标准的变量： http://gnu.org/prep/standards/html_node/Directory-Variables.html 和 http://gnu.org/prep/standards/html_node/DESTDIR.html

  例如：

```
make install DESTDIR=myinstall prefix=/usr
```

4.32.4 测试 Targets

- test

  对使用 O=mybuilddir 指定的构建目录启动自动测试。 这是可选的，默认的输出目录是build。

  例如：

```
make test O=mybuild
```

4.32.5 文档 Targets

- doc

  生成文档（API和指南）。

- doc-api-html

  在html中生成Doxygen API文档。

- doc-guides-html

  在html中生成指南文档。

- doc-guides-pdf

  用pdf生成指南文档。

4.32.6 其他 Targets

- help

  显示快速帮助。

4.32.7 其他有用的命令行变量

以下变量可以在命令行中指定：

- V=

  启用详细构建（显示完整的编译命令行和一些中间命令）。

- D=

  启用依赖关系调试。 这提供了一些关于为什么构建目标的有用信息。

- EXTRA_CFLAGS=, EXTRA_LDFLAGS=, EXTRA_LDLIBS=, EXTRA_ASFLAGS=, EXTRA_CPPFLAGS=

  附加特定的编译，链接或汇编标志。

- CROSS=

  指定一个交叉工具链头部，该头部将作为所有gcc/binutils应用程序的前缀。这只适用于使用gcc。

4.32.8 在需要构建的目录中执行Make

上面描述的所有目标都是从SDK根目录 $(RTE_SDK) 调用的。 也可以在build目录中运行相同的Makefile target。 例如，下面的命令：

```
cd $(RTE_SDK)
make config O=mybuild T=x86_64-native-linuxapp-gcc
make O=mybuild
```

相当于：

```
cd $(RTE_SDK)
make config O=mybuild T=x86_64-native-linuxapp-gcc
cd mybuild

# no need to specify O= now
make
```

4.32.9 编译为调试 Target

要编译包含调试信息和优化级别设置为0的DPDK和示例应用程序，应在编译之前设置EXTRA_CFLAGS环境变量，如下所示：

```
export EXTRA_CFLAGS='-O0 -g'
```

4.33 扩展 DPDK

本章描述了开发者如何通过扩展DPDK来提供一个新的库、目标文件或者支持新的开发板。

4.33.1 示例：添加新的库 libfoo

要添加新的库到DPDK，按照如下操作：

1. 添加新的配置选项：

```
for f in config/\*; do \
    echo CONFIG_RTE_LIBFOO=y >> $f; done
```

2. 创建新的源码目录：

```
mkdir ${RTE_SDK}/lib/libfoo
touch ${RTE_SDK}/lib/libfoo/foo.c
touch ${RTE_SDK}/lib/libfoo/foo.h
```

3. 源码添加 foo() 函数。

   函数定义于 foo.c:

```
void foo(void)
{
}
```

   函数声明于 foo.h:

```
extern void foo(void);
```

4. 更新文件 lib/Makefile:

```
vi ${RTE_SDK}/lib/Makefile
# add:
# DIRS-$(CONFIG_RTE_LIBFOO) += libfoo
```

5. 为新的库创建新的 Makefile，如派生自 mempool Makefile，进行修改：

```
cp ${RTE_SDK}/lib/librte_mempool/Makefile ${RTE_SDK}/lib/libfoo/

vi ${RTE_SDK}/lib/libfoo/Makefile
# replace:
# librte_mempool -> libfoo
# rte_mempool -> foo
```

6. 更新文件 mk/DPDK.app.mk，添加 -lfoo 选项到 LDLIBS 变量中。 链接DPDK应用程序时会自动添加此标志。

7. 添加此新库之后，重新构建DPDK (此处仅显示这个特殊的部分)：

```
cd ${RTE_SDK}
make config T=x86_64-native-linuxapp-gcc
make
```

8. 检测这个库被正确安装了：

```
ls build/lib
ls build/include
```

示例：在测试用例中使用新库 libfoo

测试应用程序用于验证DPDK的所有功能。 一旦添加了一个库，应该在测试用例程序中添加一个用例。

- 新的测试文件 test_foo.c 被添加，包含头文件 foo.h 并调用 foo() 函数。 当测试通过时，test_foo() 函数需要返回0。
- 为了处理新的测试用例，Makefile, test.h 和 commands.c 必须同时更新。
- 测试报告生成：autotest.py 是一个脚本，用于生成文件 ${RTE_SDK}/doc/rst/test_report/autotests 目录中指定的测试用例报告。 如果libfoo处于新的测试家族，链接 ${RTE_SDK}/doc/rst/test_report/test_report.rst 需要更新。
- 重新构建DPDK库，添加新的测试应用程序：

```
cd ${RTE_SDK}
make config T=x86_64-native-linuxapp-gcc
make
```

4.34 构建你自己的应用程序

4.34.1 在DPDK中编译一个示例程序

当编译示例应用程序(如 hello world)时，需要导出变量：RTE_SDK 和 RTE_TARGET。

```
~/DPDK$ cd examples/helloworld/
~/DPDK/examples/helloworld$ export RTE_SDK=/home/user/DPDK
~/DPDK/examples/helloworld$ export RTE_TARGET=x86_64-native-linuxapp-gcc
~/DPDK/examples/helloworld$ make
    CC main.o
    LD helloworld
    INSTALL-APP helloworld
    INSTALL-MAP helloworld.map
```

生成的二进制文件默认放在build目录下：

```
~/DPDK/examples/helloworld$ ls build/app
helloworld helloworld.map
```

4.34.2 在DPDK外构建自己的应用程序

示例应用程序(Hello World)可以复制到一个新的目录中作为开发目录：

```
~$ cp -r DPDK/examples/helloworld my_rte_app
~$ cd my_rte_app/
~/my_rte_app$ export RTE_SDK=/home/user/DPDK
~/my_rte_app$ export RTE_TARGET=x86_64-native-linuxapp-gcc
~/my_rte_app$ make
    CC main.o
    LD helloworld
    INSTALL-APP helloworld
    INSTALL-MAP helloworld.map
```

4.34.3 定制 Makefiles

应用程序 Makefile

示例应用程序默认的makefile可以作为一个很好的起点，我们可以直接修订使用。它包括：

- 起始处包含 $(RTE_SDK)/mk/rte.vars.mk
- 终止处包含 $(RTE_SDK)/mk/rte.extapp.mk

用户必须配置几个变量：

- APP: 应用程序的名称
- SRCS-y: 源文件列表(*.c, *.S)。

库 Makefile

同样的方法也可以用于构建库：

- 起始处包含 $(RTE_SDK)/mk/rte.vars.mk
- 终止处包含 $(RTE_SDK)/mk/rte.extlib.mk

唯一的不同之处就是用LIB名称替换APP的名称，例如：libfoo.a。

定制 Makefile 动作

可以通过定制一些变量来制定 Makefile 动作。常用的动作列表可以参考文档 *Development Kit Build System* 章节 *Makefile Description* 。

- VPATH: 构建系统将搜索的源文件目录，默认情况下 RTE_SRCDIR 将被包含在 VPATH 中。
- CFLAGS_my_file.o: 编译c文件时指定的编译flag标志。
- CFLAGS: C编译标志。
- LDFLAGS: 链接标志。
- CPPFLAGS: 预处理器标志（只是用于汇编.s文件）。
- LDLIBS: 链接库列表(如 -L /path/to/libfoo - lfoo)。

4.35 外部应用程序/库的 Makefile

外部的应用程序或库必须包含RTE_SDK指定的位于mk目录中的Makefiles文件。 这些Makefiles包括：

- ${RTE_SDK}/mk/rte.extapp.mk: 构建一个应用程序。
- ${RTE_SDK}/mk/rte.extlib.mk: 构建一个静态库。
- ${RTE_SDK}/mk/rte.extobj.mk: 购件一个目标文件。

4.35.1 前提

必须定义以下变量：

- ${RTE_SDK}: 指向DPDK根目录。
- ${RTE_TARGET}: 指向用于编译的目标编译器(如x86_64-native-linuxapp-gcc)。

4.35.2 构建 Targets

支持构建target时指定输出文件的目录，使用 O=mybuilddir 选项。 这是可选的，默认的输出目录是build。

- all, “nothing” (仅make)

  编译应用程序或库到指定的输出目录中。

  例如：

```
make O=mybuild
```

- clean

  清除make操作产生的所有目标文件。

  例如：

```
make clean O=mybuild
```

4.35.3 Help Targets

- help

  显示帮助信息。

4.35.4 其他有用的命令行变量

以下变量可以在命令行中指定：

- S=

  指定源文件的位置。默认情况下是当前目录。

- M=

  指定需要被调用的Makefile。默认情况下使用 $(S)/Makefile。

- V=

  使能详细编译（显示完全编译命令及一些中间命令过程）。

- D=

  启用依赖关系调试。提供了一些有用的信息。

- EXTRA_CFLAGS=, EXTRA_LDFLAGS=, EXTRA_ASFLAGS=, EXTRA_CPPFLAGS=

  添加的编译、连接或汇编标志。

- CROSS=

  指定一个交叉工具链，该前缀将作为所有gcc/binutils应用程序的前缀。只有在gcc下才起作用。

4.35.5 从其他目录中编译

通过指定输出和源目录，可以从另一个目录运行Makefile。 例如：

```
export RTE_SDK=/path/to/DPDK
export RTE_TARGET=x86_64-native-linuxapp-icc
make -f /path/to/my_app/Makefile S=/path/to/my_app O=/path/to/build_dir
```

Part 3: Performance Optimization

4.36 性能优化指南

4.36.1 介绍

以下各节将介绍DPDK中使用的优化以及新应用程序应考虑的优化。

还强调了在开发使用DPDK的应用程序时应该及不应该使用的影响性能的编码技术。

最后，介绍了使用英特尔性能分析器进行应用程序分析以优化软件。

4.37 编写高效代码

本章提供了一些使用DPDK开发高效代码的技巧。 有关其他更详细的信息，请参阅 *Intel® 64 and IA-32 Architectures Optimization Reference Manual* ，这是编写高效代码的宝贵参考。

4.37.1 内存

本节介绍在DPDK环境中开发应用程序时使用内存的一些关键注意事项。

内存拷贝：不要在数据面程序中使用libc

通过Linux应用程序环境，DPDK中可以使用许多libc函数。 这可以简化应用程序的移植和控制平面的开发。 但是，这些功能中有许多不是为了性能而设计的。 诸如memcpy() 或 strcpy() 之类的函数不应该在数据平面中使用。 要复制小型结构体，首选方法是编译器可以优化一个更简单的技术。 请参阅英特尔新出版的 *VTune™ Performance Analyzer Essentials* 以获取建议。

对于经常调用的特定函数，提供一个自制的优化函数也是一个好主意，该函数应声明为静态内联。

DPDK API提供了一个优化的rte_memcpy() 函数。

内存申请

libc的其他功能，如malloc()，提供了一种灵活的方式来分配和释放内存。 在某些情况下，使用动态分配是必要的，但是建议不要在数据层面使用类似malloc的函数，因为管理碎片堆可能代价高昂，并且分配器可能无法针对并行分配进行优化。

如果您确实需要在数据平面中进行动态分配，最好使用固定大小对象的内存池。 这个API由librte_mempool提供。 这个数据结构提供了一些提高性能的服务，比如对象的内存对齐，对对象的无锁访问，NUMA感知，批量get/put和percore缓存。 rte_malloc() 函数对mempools使用类似的概念。

内存区域的并发访问

几个lcore对同一个内存区域进行的读写（RW）访问操作可能会产生大量的数据高速缓存未命中，这代价非常昂贵。 通常可以使用per-lcore变量来解决这类问题。例如，在统计的情况下。 至少有两个解决方案:

- 使用 RTE_PER_LCORE 变量。注意，在这种情况下，处于lcore x的数据在lcore y上是无效的。
- 使用一个表结构（每个lcore一个）。在这种情况下，每个结构都必须缓存对齐。

如果在同一缓存行中没有RW变量，那么读取主要变量可以在不损失性能的情况下在内核之间共享。

NUMA

在NUMA系统上，由于远程内存访问速度较慢，所以最好访问本地内存。在DPDK中，memzone，ring，rte_malloc和mempool API提供了在特定内存槽上创建内存池的方法。

有时候，复制数据以优化速度可能是一个好主意。 对于经常访问的大多数读取变量，将它们保存在一个socket中应该不成问题，因为数据将存在于缓存中。

跨存储器通道分配

现代内存控制器具有许多内存通道，可以支持并行数据读写操作。 根据内存控制器及其配置，通道数量和内存在通道中的分布方式会有所不同。 每个通道都有带宽限制，这意味着如果所有的存储器访问都在同一通道上完成，则存在潜在的性能瓶颈。

默认情况下， *Mempool Library* 分配对象在内存通道中的地址。

4.37.2 lcore之间的通信

为了在内核之间提供基于消息的通信，建议使用提供无锁环实现的DPDK ring API。

该环支持批量访问和突发访问，这意味着只需要一次昂贵的原子操作即可从环中读取多个元素（请参阅 *Ring 库* ）。

使用批量访问操作时，性能会大大提高。

出队消息的代码算法可能类似于以下内容:

```
#define MAX_BULK 32

while (1) {
    /* Process as many elements as can be dequeued. */
    count = rte_ring_dequeue_burst(ring, obj_table, MAX_BULK, NULL);
    if (unlikely(count == 0))
        continue;

    my_process_bulk(obj_table, count);
}
```

4.37.3 PMD 驱动

DPDK轮询模式驱动程序（PMD）也能够在批量/突发模式下工作，允许在发送或接收功能中对每个呼叫的一些代码进行分解。

避免部分写入。 当PCI设备通过DMA写入系统存储器时，如果写入操作位于完全缓存行而不是部分写入操作，则其花费较少。 在PMD代码中，已采取了尽可能避免部分写入的措施。

低报文延迟

传统上，吞吐量和延迟之间有一个折衷。 可以调整应用程序以实现高吞吐量，但平均数据包的端到端延迟通常会因此而增加。 类似地，可以将应用程序调整为平均具有低端到端延迟，但代价是较低的吞吐量。

为了实现更高的吞吐量，DPDK尝试通过突发处理数据包来合并单独处理每个数据包的成本。

以testpmd应用程序为例，突发大小可以在命令行上设置为16（也是默认值）。 这允许应用程序一次从PMD请求16个数据包。 然后，testpmd应用程序立即尝试传输所有接收到的数据包，在这种情况下是全部16个数据包。

在网络端口的相应的TX队列上更新尾指针之前，不发送分组。 当调整高吞吐量时，这种行为是可取的，因为对RX和TX队列的尾指针更新的成本可以分布在16个分组上， 有效地隐藏了写入PCIe 设备的相对较慢的MMIO成本。 但是，当调优为低延迟时，这不是很理想，因为接收到的第一个数据包也必须等待另外15个数据包才能被接收。 直到其他15个数据包也被处理完毕才能被发送，因为直到TX尾指针被更新，NIC才知道要发送数据包，直到所有的16个数据包都被处理完毕才被发送。

为了始终如一地实现低延迟，即使在系统负载较重的情况下，应用程序开发人员也应避免处理数据包。 testpmd应用程序可以从命令行配置使用突发值1。 这将允许一次处理单个数据包，提供较低的延迟，但是增加了较低吞吐量的成本。

4.37.4 锁和原子操作

原子操作意味着在指令之前有一个锁定前缀，导致处理器的LOCK＃信号在执行下一条指令时被断言。 这对多核环境中的性能有很大的影响。

可以通过避免数据平面中的锁定机制来提高性能。 它通常可以被其他解决方案所取代，比如percore变量。 而且，一些锁定技术比其他锁定技术更有效率。 例如，Read-Copy-Update（RCU）算法可以经常替换简单的rwlock

4.37.5 编码考虑

内联函数

小函数可以在头文件中声明为静态内联。 这避免了调用指令的成本（和关联的上下文保存）。 但是，这种技术并不总是有效的。 它取决于许多因素，包括编译器。

分支预测

英特尔的C/C ++编译器icc/gcc内置的帮助函数likely()和unlikely()允许开发人员指出是否可能采取代码分支。 例如：

```
if (likely(x > 1))
    do_stuff();
```

4.37.6 设置目标CPU类型

DPDK通过DPDK配置文件中的CONFIG_RTE_MACHINE选项支持CPU微体系结构特定的优化。 优化程度取决于编译器针对特定微架构进行优化的能力，因此，只要有可能，最好使用最新的编译器版本。

如果编译器版本不支持特定的功能集（例如，英特尔®AVX指令集），则编译过程将优雅地降级到编译器支持的任何最新功能集。

由于构建和运行时目标可能不相同，因此生成的二进制文件还包含在main()函数之前运行的平台检查，并检查当前机器是否适合运行二进制文件。

除编译器优化之外，一组预处理器定义会自动添加到构建过程中（不管编译器版本如何）。 这些定义对应于目标CPU应该能够支持的指令集。 例如，为任何支持SSE4.2的处理器编译的二进制文件将定义RTE_MACHINE_CPUFLAG_SSE4_2，从而为不同的平台启用编译时代码路径选择。

4.38 配置你的应用程序

以下各节介绍了在不同体系结构上配置DPDK应用程序的方法。

4.38.1 X86

英特尔处理器提供性能计数器来监视事件 英特尔提供的某些工具（如VTune）可用于对应用程序进行配置和基准测试。 欲了解更多信息，请参阅英特尔 *VTune Performance Analyzer Essentials* 。

对于DPDK应用程序，这只能在Linux应用程序环境中完成。

应通过事件计数器监测的主要情况是：

- Cache misses
- Branch mis-predicts
- DTLB misses
- Long latency instructions and exceptions

请参考 Intel Performance Analysis Guide 获取更多的信息。

4.38.2 ARM64

使用 Linux perf

ARM64体系结构提供性能计数器来监视事件。 Linux `perf` 工具可以用来分析和测试应用程序。 除了标准事件之外，还可以使用 `perf` 通过原始事件（ `-e -rXX` ）来分析arm64特定的PMU（性能监视单元）事件。

更多详细信息请参考 ARM64 specific PMU events enumeration.

高分辨率的cycle计数器

基于 `rte_rdtsc()` 的默认 `cntvct_el0` 提供了一种便携的方式来获取用户空间中的时钟计数器。通常它运行在<= 100MHz下。

为高分辨时钟计数器启用 `rte_rdtsc()` 的替代方法是通过armv8 PMU子系统。 PMU周期计数器以CPU频率运行。但是，在arm64 linux内核中，默认情况下，不能从用户空间访问PMU周期计数器。 通过从特权模式（内核空间）配置PMU，可以为用户空间访问启用循环计数器。

默认情况下，rte_rdtsc() 实现使用一个可移植的 cntvct_el0 方案。 应用程序可以使用“CONFIG_RTE_ARM_EAL_RDTSC_USE_PMU”选择基于PMU的实现。

下面的示例显示了在armv8机器上配置基于PMU的循环计数器的步骤。

```
git clone https://github.com/jerinjacobk/armv8_pmu_cycle_counter_el0
cd armv8_pmu_cycle_counter_el0
make
sudo insmod pmu_el0_cycle_counter.ko
cd $DPDK_DIR
make config T=arm64-armv8a-linuxapp-gcc
echo "CONFIG_RTE_ARM_EAL_RDTSC_USE_PMU=y" >> build/.config
make
```

Warning: The PMU based scheme is useful for high accuracy performance profiling with rte_rdtsc(). However, this method can not be used in conjunction with Linux userspace profiling tools like perf as this scheme alters the PMU registers state.

4.39 术语

ACL 访问控制列表

API 应用程序编程接口

ASLR Linux内核地址空间布局

BSD 伯克利软件

Clr Clear

CIDR 无类别域间路由

Control Plane 控制面

Core 如果处理器支持超线程，则内核可能包含多个内核或线程。

Core Components DPDK提供的一组库，包括 eal, ring, mempool, mbuf, timers等。

CPU Central Processing Unit

CRC Cyclic Redundancy Check

ctrlmbuf 携带控制数据的 *mbuf* 。

Data Plane 数据面，在网络架构层中对应于控制面，负责转发报文。这层必须具有很高的性能。

DIMM Dual In-line Memory Module

Doxygen DPDK中用于生成API参考的文档生成器。

DPDK Data Plane Development Kit

DRAM Dynamic Random Access Memory

EAL The Environment Abstraction Layer (EAL) provides a generic interface that hides the environment specifics from the applications and libraries. The services expected from the EAL are: development kit loading and launching, core affinity/ assignment procedures, system memory allocation/description, PCI bus access, inter-partition communication.

FIFO First In First Out

FPGA Field Programmable Gate Array

GbE Gigabit Ethernet

HW Hardware

HPET High Precision Event Timer; a hardware timer that provides a precise time reference on x86 platforms.

ID Identifier

IOCTL Input/Output Control

I/O Input/Output

IP Internet Protocol

IPv4 Internet Protocol version 4

IPv6 Internet Protocol version 6

lcore A logical execution unit of the processor, sometimes called a *hardware thread*.

KNI Kernel Network Interface

L1 Layer 1

L2 Layer 2

L3 Layer 3

L4 Layer 4

LAN Local Area Network

LPM Longest Prefix Match

master lcore The execution unit that executes the main() function and that launches other lcores.

mbuf An mbuf is a data structure used internally to carry messages (mainly network packets). The name is derived from BSD stacks. To understand the concepts of packet buffers or mbuf, refer to *TCP/IP Illustrated, Volume 2: The Implementation*.

MESI Modified Exclusive Shared Invalid (CPU cache coherency protocol)

MTU Maximum Transfer Unit

NIC Network Interface Card

OOO Out Of Order (execution of instructions within the CPU pipeline)

NUMA Non-uniform Memory Access

PCI Peripheral Connect Interface

PHY An abbreviation for the physical layer of the OSI model.

pktmbuf An *mbuf* carrying a network packet.

PMD Poll Mode Driver

QoS Quality of Service

RCU Read-Copy-Update algorithm, an alternative to simple rwlocks.

Rd Read

RED Random Early Detection

RSS Receive Side Scaling

RTE Run Time Environment. Provides a fast and simple framework for fast packet processing, in a lightweight environment as a Linux* application and using Poll Mode Drivers (PMDs) to increase speed.

Rx Reception

Slave lcore Any *lcore* that is not the *master lcore*.

Socket A physical CPU, that includes several *cores*.

SLA Service Level Agreement

srTCM Single Rate Three Color Marking

SRTD Scheduler Round Trip Delay

SW Software

Target In the DPDK, the target is a combination of architecture, machine, executive environment and toolchain. For example: i686-native-linuxapp-gcc.

TCP Transmission Control Protocol

TC Traffic Class

TLB Translation Lookaside Buffer

TLS Thread Local Storage

trTCM Two Rate Three Color Marking

TSC Time Stamp Counter

Tx Transmission

TUN/TAP TUN 和 TAP 是虚拟的网络内核设备。

VLAN Virtual Local Area Network

Wr Write

WRED 加权随机早丢弃

WRR 加权轮询

Figures

Fig. 4.1 *Core Components Architecture*

Fig. 4.2 *EAL在Linux APP*环境中被初始化。

Fig. 4.3 *Malloc*库中*malloc heap* 和 *malloc elements*。

Fig. 4.4 *Ring* 结构

Fig. 4.5 *Enqueue first step*

Fig. 4.6 *Enqueue second step*

Fig. 4.7 *Enqueue last step*

Fig. 4.8 *Dequeue last step*

Fig. 4.9 *Dequeue second step*

Fig. 4.10 *Dequeue last step*

Fig. 4.11 *Multiple producer enqueue first step*

Fig. 4.12 *Multiple producer enqueue second step*

Fig. 4.13 *Multiple producer enqueue third step*

Fig. 4.14 *Multiple producer enqueue fourth step*

Fig. 4.15 *Multiple producer enqueue last step*

Fig. 4.16 *Modulo 32-bit indexes - Example 1*

Fig. 4.17 *Modulo 32-bit indexes - Example 2*

Fig. 4.18 *Two Channels and Quad-ranked DIMM Example*

Fig. 4.19 *Three Channels and Two Dual-ranked DIMM Example*

Fig. 4.20 *A mempool in Memory with its Associated Ring*

Fig. 4.21 *An mbuf with One Segment*

Fig. 4.22 *An mbuf with Three Segments*

Fig. 4.46 *Memory Sharing in the DPDK Multi-process Sample Application*

Fig. 4.47 *Components of a DPDK KNI Application*

Fig. 4.48 *Packet Flow via mbufs in the DPDK KNI*

Fig. 4.49 *Complex Packet Processing Pipeline with QoS Support*

Fig. 4.50 *Hierarchical Scheduler Block Internal Diagram*

Fig. 4.51 *Scheduling Hierarchy per Port*

Fig. 4.52 *Internal Data Structures per Port*

Fig. 4.53 *Prefetch Pipeline for the Hierarchical Scheduler Enqueue Operation*

Fig. 4.55 *High-level Block Diagram of the DPDK Dropper*

Fig. 4.56 *Flow Through the Dropper*

Fig. 4.57 *Example Data Flow Through Dropper*

Fig. 4.58 *Packet Drop Probability for a Given RED Configuration*

Fig. 4.59 *Initial Drop Probability (pb), Actual Drop probability (pa) Computed Using a Factor 1 (Blue Curve) and a Factor 2 (Red Curve)*

Fig. 4.60 输入端口*0*和*1*通过表*0*和表*1*与输出端口*0*，*1*和*2*连接的数据包流水线示例

Fig. 4.61 报文处理上下文中哈希表操作的步骤顺序

Fig. 4.62 可配置*Key*大小的散列表的数据结构

Fig. 4.63 用于*Key*查找操作的流水线*(*可配置*Key*大小的哈希表*)*

Fig. 4.64 *8B Key*哈希表数据结构

Fig. 4.65 *16B Key*哈希表数据结构

Fig. 4.66 用于*Key*查找操作的桶搜索水线*(*单一*Key*大小的哈希表*)*

Fig. 4.30 *Load Balancing Using Front End Node*

Fig. 4.31 *Consistent Hashing*

Fig. 4.32 *Table Based Flow Distribution*

Fig. 4.33 *Searching for Perfect Hash Function*

Fig. 4.34 *Divide and Conquer for Millions of Keys*

Fig. 4.35 *EFD as a Flow-Level Load Balancer*

Fig. 4.36 *Group Assignment*

Fig. 4.37 *Perfect Hash Search - Assigned Keys & Target Value*

Fig. 4.38 *Perfect Hash Search - Satisfy Target Values*

Fig. 4.39 *Finding Hash Index for Conflict Free lookup_table*

Fig. 4.40 *EFD Lookup Operation*

Tables

Table 4.51 *Packet Processing Pipeline Implementing QoS*

Table 4.52 *Infrastructure Blocks Used by the Packet Processing Pipeline*

Table 4.53 *Port Scheduling Hierarchy*

Table 4.54 *Scheduler Internal Data Structures per Port*

Table 4.55 *Ethernet Frame Overhead Fields*

Table 4.56 *Token Bucket Generic Operations*

Table 4.57 *Token Bucket Generic Parameters*

Table 4.58 *Token Bucket Persistent Data Structure*

Table 4.59 *Token Bucket Operations*

Table 4.60 *Subport/Pipe Traffic Class Upper Limit Enforcement Persistent Data Structure*

Table 4.61 *Subport/Pipe Traffic Class Upper Limit Enforcement Operations*

Table 4.62 *Weighted Round Robin (WRR)*

Table 4.63 *Subport Traffic Class Oversubscription*

Table 4.64 *Watermark Propagation from Subport Level to Member Pipes at the Beginning of Each Traffic Class Upper Limit Enforcement Period*

Table 4.65 *Watermark Calculation*

Table 4.66 *RED Configuration Parameters*

Table 4.67 *Relative Performance of Alternative Approaches*

Table 4.68 *RED Configuration Corresponding to RED Configuration File*

Table 4.69 端口类型

Table 4.70 *20* 端口抽象接口

Table 4.71 表类型

Table 4.73 所有散列表类型的通用配置参数

Table 4.74 可扩展桶散列表特定的配置参数

Table 4.75 预先计算哈希值的哈希表配置参数

Table 4.76 *Main Large Data Structures (Arrays) used for Configurable Key Size Hash Tables*

Table 4.77 数组输入的字段描述（可配置的密钥大小哈希表）

Table 4.78 桶搜索流水线阶段的描述*(*可配置*Key*大小的哈希表*)*

Table 4.79 *Lookup Tables for Match, Match_Many and Match_Pos*

Table 4.80 *Collapsed Lookup Tables for Match, Match_Many and Match_Pos*

Table 4.81 用于*8B*和*16B Key*大小的哈希表的主要数据结构

Table 4.82 桶数组条目字段说明*(8B*和*16B Key*大小的哈希表*)*

Table 4.83 桶搜索流水线阶段的描述（*8B*和*16B*的*Key*散列表）

Table 4.84 *Next Hop Actions (Reserved)*

Table 4.85 用户动作实例

Table 4.49 *Entry distribution measured with an example table with 1024 random entries using jhash algorithm*

Table 4.50 *Entry distribution measured with an example table with 1 million random entries using jhash algorithm*

CHAPTER 5

HowTo Guides

5.1 Live Migration of VM with SR-IOV VF

5.1.1 Overview

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

To get around this problem the bonding PMD is used.

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

5.1.2 Test Setup

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

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

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

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

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

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

The ip address of host_server_1 is 10.237.212.46

The ip address of host_server_2 is 10.237.212.131

5.1.3 Live Migration steps

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

On host_server_1: Terminal 1

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

For Fortville NIC

```
./vm_virtio_vf_i40e_212_46.sh
```

For Niantic NIC

```
./vm_virtio_vf_one_212_46.sh
```

On host_server_1: Terminal 2

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

On host_server_1: Terminal 1

In VM on host_server_1:

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

testpmd> show port info all
```

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

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

The syntax of the `testpmd` command is:

Create bonded device (mode) (socket).

Mode 1 is active backup.

Virtio is port 0 (P0).

VF is port 1 (P1).

Bonding is port 2 (P2).

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

```
testpmd> add bonding slave 1 2
testpmd> show bonding config 2
```

The syntax of the `testpmd` command is:

set bonding primary (slave id) (port id)

Set primary to P1 before starting bonding port.

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

testpmd> show bonding config 2
```

Primary is now P1. There are 2 active slaves.

Use P2 only for forwarding.

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

Primary is now P1. There are 2 active slaves.

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

VF traffic is seen at P1 and P2.

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

Primary is now P0. There is 1 active slave.

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

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

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

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

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

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

No VF traffic is seen at P0 and P2.

On host_server_1: Terminal 2

```
(qemu) device_del vf1
```

On host_server_1: Terminal 1

In VM on host_server_1:

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

Primary is now P0. There is 1 active slave.

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

On host_server_2: Terminal 1

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

On host_server_2: Terminal 2

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

On host_server_1: Terminal 2

Check that the switch is up before migrating.

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

For the Niantic NIC.

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

```
total ram: 1590088 kbytes
duplicate: 301620 pages
skipped: 0 pages
normal: 96433 pages
normal bytes: 385732 kbytes
dirty sync count: 2
(qemu) quit
```

For the Fortville NIC.

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

On host_server_2: Terminal 1

In VM on host_server_2:

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

```
testpmd>
```

On host_server_2: Terminal 2

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

For the Niantic NIC.

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

For the Fortville NIC.

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

On host_server_2: Terminal 1

In VM on host_server_2:

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

testpmd> port start 1
```

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

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

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

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

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

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

5.1.4 Sample host scripts

setup_vf_on_212_46.sh

Set up Virtual Functions on host_server_1

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

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

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

vm_virtio_vf_one_212_46.sh

Setup Virtual Machine on host_server_1

```
#!/bin/sh

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

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

# Number of guest cpus
VCPUS_NR="4"

# Memory
MEM=1536

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

setup_bridge_on_212_46.sh

Setup bridge on host_server_1

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

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

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

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

ifconfig ens3f0 up
```

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

connect_to_qemu_mon_on_host.sh

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

telnet 0 3333
```

setup_vf_on_212_131.sh

Set up Virtual Functions on host_server_2

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

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

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

vm_virtio_one_migrate.sh

Setup Virtual Machine on host_server_2

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

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

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

# Number of guest cpus
VCPUS_NR="4"

# Memory
MEM=1536

taskset -c 1-5 $KVM_PATH \
```

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

setup_bridge_on_212_131.sh

Setup bridge on host_server_2

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

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

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

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

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

5.1.5 Sample VM scripts

setup_dpdk_in_vm.sh

Set up DPDK in the Virtual Machine

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

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

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

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

rmmod virtio-pci ixgbevf

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

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

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

run_testpmd_bonding_in_vm.sh

Run testpmd in the Virtual Machine.

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

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

# use for bonding of virtio and vf tests in VM

/root/dpdk/x86_64-default-linuxapp-gcc/app/testpmd \
-l 0-3 -n 4 --socket-mem 350 --  --i --port-topology=chained
```

5.1.6 Sample switch configuration

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

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

Log in as root in both windows.

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

On Switch: Terminal 1

run TestPointShared

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

On Switch: Terminal 2

execute run_switch.sh

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

On Switch: Terminal 1

load switch configuration

```
load /root/switch_config
```

Sample switch configuration script

The `/root/switch_config` script:

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

5.2 Live Migration of VM with Virtio on host running vhost_user

5.2.1 Overview

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

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

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

5.2.2 Test Setup

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

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

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

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

The ip address of host_server_1 is 10.237.212.46

The ip address of host_server_2 is 10.237.212.131

5.2.3 Live Migration steps

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

On host_server_1: Terminal 1

Setup DPDK on host_server_1

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

On host_server_1: Terminal 2

Bind the Niantic or Fortville NIC to igb_uio on host_server_1.

For Fortville NIC.

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

For Niantic NIC.

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

On host_server_1: Terminal 3

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

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

On host_server_1: Terminal 1

Start the VM on host_server_1

```
./vm_virtio_vhost_user.sh
```

On host_server_1: Terminal 4

Connect to the QEMU monitor on host_server_1.

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

On host_server_1: Terminal 1

In VM on host_server_1:

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

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

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

Virtio traffic is seen at P1 and P2.

On host_server_2: Terminal 1

Set up DPDK on the host_server_2.

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

On host_server_2: Terminal 2

Bind the Niantic or Fortville NIC to igb_uio on host_server_2.

For Fortville NIC.

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

For Niantic NIC.

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

On host_server_2: Terminal 3

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

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

On host_server_2: Terminal 1

Start the VM on host_server_2.

```
./vm_virtio_vhost_user_migrate.sh
```

On host_server_2: Terminal 4

Connect to the QEMU monitor on host_server_2.

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

On host_server_1: Terminal 4

Check that switch is up before migrating the VM.

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

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

On host_server_2: Terminal 1

In VM on host_server_2:

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

```
testpmd>
```

On host_server_2: Terminal 4

In QEMU monitor on host_server_2

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

On host_server_2: Terminal 1

In VM on host_server_2:

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

Virtio traffic is seen at P0 and P1.

5.2.4 Sample host scripts

reset_vf_on_212_46.sh

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

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

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

vm_virtio_vhost_user.sh

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

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

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

# Number of guest cpus
VCPUS_NR="6"

# Memory
MEM=1024

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

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

```
taskset -c 2-7 $KVM_PATH \
 -enable-kvm \
 -m $MEM \
 -smp $VCPUS_NR \
 -object memory-backend-file,id=mem,size=1024M,mem-path=/mnt/huge,share=on \
 -numa node,memdev=mem,nodeid=0 \
 -cpu host \
 -name VM1 \
 -no-reboot \
 -net none \
 -vnc none \
 -nographic \
 -hda $DISK_IMG \
 -chardev socket,id=chr0,path=$SOCKET_PATH \
 -netdev type=vhost-user,id=net1,chardev=chr0,vhostforce \
 -device virtio-net-pci,netdev=net1,mac=CC:BB:BB:BB:BB:BB,$VIRTIO_OPTIONS \
 -chardev socket,id=chr1,path=$SOCKET_PATH \
 -netdev type=vhost-user,id=net2,chardev=chr1,vhostforce \
 -device virtio-net-pci,netdev=net2,mac=DD:BB:BB:BB:BB:BB,$VIRTIO_OPTIONS \
 -monitor telnet::3333,server,nowait
```

connect_to_qemu_mon_on_host.sh

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

telnet 0 3333
```

reset_vf_on_212_131.sh

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

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

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

vm_virtio_vhost_user_migrate.sh

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

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

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

# Number of guest cpus
VCPUS_NR="6"

# Memory
MEM=1024

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

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

taskset -c 2-7 $KVM_PATH \
 -enable-kvm \
 -m $MEM \
 -smp $VCPUS_NR \
 -object memory-backend-file,id=mem,size=1024M,mem-path=/mnt/huge,share=on \
 -numa node,memdev=mem,nodeid=0 \
 -cpu host \
 -name VM1 \
 -no-reboot \
 -net none \
 -vnc none \
 -nographic \
 -hda $DISK_IMG \
 -chardev socket,id=chr0,path=$SOCKET_PATH \
 -netdev type=vhost-user,id=net1,chardev=chr0,vhostforce \
 -device virtio-net-pci,netdev=net1,mac=CC:BB:BB:BB:BB:BB,$VIRTIO_OPTIONS \
 -chardev socket,id=chr1,path=$SOCKET_PATH \
 -netdev type=vhost-user,id=net2,chardev=chr1,vhostforce \
 -device virtio-net-pci,netdev=net2,mac=DD:BB:BB:BB:BB:BB,$VIRTIO_OPTIONS \
 -incoming tcp:0:5555 \
 -monitor telnet::3333,server,nowait
```

5.2.5 Sample VM scripts

setup_dpdk_virtio_in_vm.sh

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

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

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

rmmod virtio-pci

modprobe uio
```

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

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

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

run_testpmd_in_vm.sh

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

/root/dpdk/x86_64-default-linuxapp-gcc/app/testpmd \
-l 0-5 -n 4 --socket-mem 350 -- --burst=64 --i --disable-hw-vlan-filter
```

5.3 Flow Bifurcation How-to Guide

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

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

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

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

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

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

Fig. 5.1: Flow Bifurcation Overview

5.3.1 Using Flow Bifurcation on IXGBE in Linux

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

The typical procedure to achieve this is as follows:

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

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

3. Enable and set flow filters:

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

   Where:

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

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

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

6. Run a DPDK application on the VFs:

```
testpmd -l 0-7 -n 4 -- -i -w 01:10.0 -w 01:10.1 --forward-mode=mac
```

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

Note:

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

5.3.2 Using Flow Bifurcation on I40E in Linux

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

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

The typical procedure to achieve this is as follows:

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

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

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

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

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

5. Examples of enabling and setting flow filters:
   - L3 VEB filter, for a route whose destination IP is 192.168.50.108 to VF 0's queue 2.

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

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

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

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

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

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

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

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

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

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

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

Note:

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

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

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

8. run DPDK application on VFs:

```
testpmd -l 0-7 -n 4 -- -i -w 01:10.0 -w 01:10.1 --forward-mode=mac
```

Note:

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

5.4 PVP reference benchmark setup using testpmd

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

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

5.4.1 Setup overview

Fig. 5.2: PVP setup using 2 NICs

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

5.4.2 Host setup

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

Host tuning

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

```
intel_pstate=disable mce=ignore_ce default_hugepagesz=1G hugepagesz=1G␣
↪hugepages=6 isolcpus=2-7 rcu_nocbs=2-7 nohz_full=2-7 iommu=pt intel_iommu=on
```

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

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

4. Disable NMIs:

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

5. Exclude isolated CPUs from the writeback cpumask:

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

6. Isolate CPUs from IRQs:

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

Qemu build

Build Qemu:

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

DPDK build

Build DPDK:

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

Testpmd launch

1. Assign NICs to DPDK:

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

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

2. Launch the testpmd application:

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

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

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

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

VM launch

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

Qemu way

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

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

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

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

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

Libvirt way

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

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

```
cat /etc/selinux/config

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

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

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

```
user = "root"
```

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

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

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

5.4.3 Guest setup

Guest tuning

1. Append these options to the Kernel command line:

```
default_hugepagesz=1G hugepagesz=1G hugepages=1 intel_iommu=on iommu=pt␣
↪isolcpus=1,2 rcu_nocbs=1,2 nohz_full=1,2
```

2. Disable NMIs:

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

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

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

4. Isolate CPUs from IRQs:

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

DPDK build

Build DPDK:

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

Testpmd launch

Probe vfio module without iommu:

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

Bind the virtio-net devices to DPDK:

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

Start testpmd:

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

5.4.4 Results template

Below template should be used when sharing results:

```
Traffic Generator: <Test equipment (e.g. IXIA, Moongen, ...)>
Acceptable Loss: <n>%
Validation run time: <n>min
Host DPDK version/commit: <version, SHA-1>
```

```
Guest DPDK version/commit: <version, SHA-1>
Patches applied: <link to patchwork>
QEMU version/commit: <version>
Virtio features: <features (e.g. mrg_rxbuf='off', leave empty if default)>
CPU: <CPU model>, <CPU frequency>
NIC: <NIC model>
Result: <n> Mpps
```

5.5 VF daemon (VFd)

VFd (the VF daemon) is a mechanism which can be used to configure features on a VF (SR-IOV Virtual Function) without direct access to the PF (SR-IOV Physical Function). VFd is an *EXPERIMENTAL* feature which can only be used in the scenario of DPDK PF with a DPDK VF. If the PF port is driven by the Linux kernel driver then the VFd feature will not work. Currently VFd is only supported by the ixgbe and i40e drivers.

In general VF features cannot be configured directly by an end user application since they are under the control of the PF. The normal approach to configuring a feature on a VF is that an application would call the APIs provided by the VF driver. If the required feature cannot be configured by the VF directly (the most common case) the VF sends a message to the PF through the mailbox on ixgbe and i40e. This means that the availability of the feature depends on whether the appropriate mailbox messages are defined.

DPDK leverages the mailbox interface defined by the Linux kernel driver so that compatibility with the kernel driver can be guaranteed. The downside of this approach is that the availability of messages supported by the kernel become a limitation when the user wants to configure features on the VF.

VFd is a new method of controlling the features on a VF. The VF driver doesn't talk directly to the PF driver when configuring a feature on the VF. When a VF application (i.e., an application using the VF ports) wants to enable a VF feature, it can send a message to the PF application (i.e., the application using the PF port, which can be the same as the VF application). The PF application will configure the feature for the VF. Obviously, the PF application can also configure the VF features without a request from the VF application.

Fig. 5.3: VF daemon (VFd) Overview

Compared with the traditional approach the VFd moves the negotiation between VF and PF from the driver level to application level. So the application should define how the negotiation between the VF and PF works, or even if the control should be limited to the PF.

It is the application's responsibility to use VFd. Consider for example a KVM migration, the VF application may transfer from one VM to another. It is recommended in this case that the PF control the VF features without participation from the VF. Then the VF application has no capability to configure the features. So the user doesn't need to define the interface between the VF application and the PF application. The service provider should take the control of all the features.

The following sections describe the VFd functionality.

Note: Although VFd is supported by both ixgbe and i40e, please be aware that since the hardware capability is different, the functions supported by ixgbe and i40e are not the same.

5.5.1 Preparing

VFd only can be used in the scenario of DPDK PF + DPDK VF. Users should bind the PF port to `igb_uio`, then create the VFs based on the DPDK PF host.

The typical procedure to achieve this is as follows:

1. Boot the system without iommu, or with `iommu=pt`.
2. Bind the PF port to `igb_uio`, for example:

```
dpdk-devbind.py -b igb_uio 01:00.0
```

3. Create a Virtual Function:

```
echo 1 > /sys/bus/pci/devices/0000:01:00.0/max_vfs
```

4. Start a VM with the new VF port bypassed to it.
5. Run a DPDK application on the PF in the host:

```
testpmd -l 0-7 -n 4 -- -i --txqflags=0
```

6. Bind the VF port to `igb_uio` in the VM:

```
dpdk-devbind.py -b igb_uio 03:00.0
```

7. Run a DPDK application on the VF in the VM:

```
testpmd -l 0-7 -n 4 -- -i --txqflags=0
```

5.5.2 Common functions of IXGBE and I40E

The following sections show how to enable PF/VF functionality based on the above testpmd setup.

TX loopback

Run a testpmd runtime command on the PF to set TX loopback:

```
set tx loopback 0 on|off
```

This sets whether the PF port and all the VF ports that belong to it are allowed to send the packets to other virtual ports.

Although it is a VFd function, it is the global setting for the whole physical port. When using this function, the PF and all the VFs TX loopback will be enabled/disabled.

VF MAC address setting

Run a testpmd runtime command on the PF to set the MAC address for a VF port:

```
set vf mac addr 0 0 A0:36:9F:7B:C3:51
```

This testpmd runtime command will change the MAC address of the VF port to this new address. If any other addresses are set before, they will be overwritten.

VF MAC anti-spoofing

Run a testpmd runtime command on the PF to enable/disable the MAC anti-spoofing for a VF port:

```
set vf mac antispoof 0 0 on|off
```

When enabling the MAC anti-spoofing, the port will not forward packets whose source MAC address is not the same as the port.

VF VLAN anti-spoofing

Run a testpmd runtime command on the PF to enable/disable the VLAN anti-spoofing for a VF port:

```
set vf vlan antispoof 0 0 on|off
```

When enabling the VLAN anti-spoofing, the port will not send packets whose VLAN ID does not belong to VLAN IDs that this port can receive.

VF VLAN insertion

Run a testpmd runtime command on the PF to set the VLAN insertion for a VF port:

```
set vf vlan insert 0 0 1
```

When using this testpmd runtime command, an assigned VLAN ID can be inserted to the transmitted packets by the hardware.

The assigned VLAN ID can be 0. It means disabling the VLAN insertion.

VF VLAN stripping

Run a testpmd runtime command on the PF to enable/disable the VLAN stripping for a VF port:

```
set vf vlan stripq 0 0 on|off
```

This testpmd runtime command is used to enable/disable the RX VLAN stripping for a specific VF port.

VF VLAN filtering

Run a testpmd runtime command on the PF to set the VLAN filtering for a VF port:

```
rx_vlan add 1 port 0 vf 1
rx_vlan rm  1 port 0 vf 1
```

These two testpmd runtime commands can be used to add or remove the VLAN filter for several VF ports. When the VLAN filters are added only the packets that have the assigned VLAN IDs can be received. Other packets will be dropped by hardware.

5.5.3 The IXGBE specific VFd functions

The functions in this section are specific to the ixgbe driver.

All queues drop

Run a testpmd runtime command on the PF to enable/disable the all queues drop:

```
set all queues drop on|off
```

This is a global setting for the PF and all the VF ports of the physical port.

Enabling the `all queues drop` feature means that when there is no available descriptor for the received packets they are dropped. The `all queues drop` feature should be enabled in SR-IOV mode to avoid one queue blocking others.

VF packet drop

Run a testpmd runtime command on the PF to enable/disable the packet drop for a specific VF:

```
set vf split drop 0 0 on|off
```

This is a similar function as `all queues drop`. The difference is that this function is per VF setting and the previous function is a global setting.

VF rate limit

Run a testpmd runtime command on the PF to all queues' rate limit for a specific VF:

```
set port 0 vf 0 rate 10 queue_mask 1
```

This is a function to set the rate limit for all the queues in the `queue_mask` bitmap. It is not used to set the summary of the rate limit. The rate limit of every queue will be set equally to the assigned rate limit.

VF RX enabling

Run a testpmd runtime command on the PF to enable/disable packet receiving for a specific VF:

```
set port 0 vf 0 rx on|off
```

This function can be used to stop/start packet receiving on a VF.

VF TX enabling

Run a testpmd runtime command on the PF to enable/disable packet transmitting for a specific VF:

```
set port 0 vf 0 tx on|off
```

This function can be used to stop/start packet transmitting on a VF.

VF RX mode setting

Run a testpmd runtime command on the PF to set the RX mode for a specific VF:

```
set port 0 vf 0 rxmode AUPE|ROPE|BAM|MPE on|off
```

This function can be used to enable/disable some RX modes on the VF, including:

- If it accept untagged packets.
- If it accepts packets matching the MAC filters.
- If it accept MAC broadcast packets,
- If it enables MAC multicast promiscuous mode.

5.5.4 The I40E specific VFd functions

The functions in this section are specific to the i40e driver.

VF statistics

This provides an API to get the a specific VF's statistic from PF.

VF statistics resetting

This provides an API to rest the a specific VF's statistic from PF.

VF link status change notification

This provide an API to let a specific VF know if the physical link status changed.

Normally if a VF received this notification, the driver should notify the application to reset the VF port.

VF MAC broadcast setting

Run a testpmd runtime command on the PF to enable/disable MAC broadcast packet receiving for a specific VF:

```
set vf broadcast 0 0 on|off
```

VF MAC multicast promiscuous mode

Run a testpmd runtime command on the PF to enable/disable MAC multicast promiscuous mode for a specific VF:

```
set vf allmulti 0 0 on|off
```

VF MAC unicast promiscuous mode

Run a testpmd runtime command on the PF to enable/disable MAC unicast promiscuous mode for a specific VF:

```
set vf promisc 0 0 on|off
```

VF max bandwidth

Run a testpmd runtime command on the PF to set the TX maximum bandwidth for a specific VF:

```
set vf tx max-bandwidth 0 0 2000
```

The maximum bandwidth is an absolute value in Mbps.

VF TC bandwidth allocation

Run a testpmd runtime command on the PF to set the TCs (traffic class) TX bandwidth allocation for a specific VF:

```
set vf tc tx min-bandwidth 0 0 (20,20,20,40)
```

The allocated bandwidth should be set for all the TCs. The allocated bandwidth is a relative value as a percentage. The sum of all the bandwidth should be 100.

VF TC max bandwidth

Run a testpmd runtime command on the PF to set the TCs TX maximum bandwidth for a specific VF:

```
set vf tc tx max-bandwidth 0 0 0 10000
```

The maximum bandwidth is an absolute value in Mbps.

TC strict priority scheduling

Run a testpmd runtime command on the PF to enable/disable several TCs TX strict priority scheduling:

```
set tx strict-link-priority 0 0x3
```

The 0 in the TC bitmap means disabling the strict priority scheduling for this TC. To enable use a value of 1.

5.6 Virtio_user for Container Networking

Container becomes more and more popular for strengths, like low overhead, fast boot-up time, and easy to deploy, etc. How to use DPDK to accelerate container networking becomes a common question for users. There are two use models of running DPDK inside containers, as shown in Fig. 5.4.

Fig. 5.4: Use models of running DPDK inside container

This page will only cover aggregation model.

5.6.1 Overview

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

The overview of accelerating container networking by virtio-user is shown in Fig. 5.5.

Fig. 5.5: Overview of accelerating container networking by virtio-user

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

Virtio, in essence, is a shm-based solution to transmit/receive packets. How is memory shared? In VM's case, qemu always shares the whole physical layout of VM to vhost backend. But it's not feasible for a container, as a process, to share all virtual memory regions to backend. So only those virtual memory regions (aka, hugepages initialized in DPDK) are sent to backend. It restricts that only addresses in these areas can be used to transmit or receive packets.

5.6.2 Sample Usage

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

1. Compile DPDK.

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

2. Write a Dockerfile like below.

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

3. Build a Docker image.

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

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

```
$(testpmd) -l 0-1 -n 4 --socket-mem 1024,1024 \
    --vdev 'eth_vhost0,iface=/tmp/sock0' \
    --file-prefix=host --no-pci -- -i
```

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

```
docker run -i -t -v /tmp/sock0:/var/run/usvhost \
    -v /dev/hugepages:/dev/hugepages \
    dpdk-app-testpmd testpmd -l 6-7 -n 4 -m 1024 --no-pci \
    --vdev=virtio_user0,path=/var/run/usvhost \
    --file-prefix=container \
    -- -i --txqflags=0xf00 --disable-hw-vlan
```

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

5.6.3 Limitations

We have below limitations in this solution:

- Cannot work with –huge-unlink option. As we need to reopen the hugepage file to share with vhost backend.
- Cannot work with –no-huge option. Currently, DPDK uses anonymous mapping under this option which cannot be reopened to share with vhost backend.
- Cannot work when there are more than VHOST_MEMORY_MAX_NREGIONS(8) hugepages. In another word, do not use 2MB hugepage so far.

- Applications should not use file name like HUGEFILE_FMT ("%smap_%d"). That will bring confusion when sharing hugepage files with backend by name.
- Root privilege is a must. DPDK resolves physical addresses of hugepages which seems not necessary, and some discussions are going on to remove this restriction.

5.7 Virtio_user as Exceptional Path

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

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

- Maintenance

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

- Features

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

- Performance

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

The overview of an application using virtio-user as exceptional path is shown in Fig. 5.6.

Fig. 5.6: Overview of a DPDK app using virtio-user as excpetional path

5.7.1 Sample Usage

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

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

   This physical NIC is for communicating with outside.

2. Run testpmd.

```
$(testpmd) -l 2-3 -n 4 \
        --vdev=virtio_user0,path=/dev/vhost-net,queue_size=1024 \
        -- -i --txqflags=0x0 --disable-hw-vlan --enable-lro \
        --enable-rx-cksum --rxd=1024 --txd=1024
```

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

- `--enable-lro`

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

- `--enable-rx-cksum`

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

- `queue_size`

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

- `queues`

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

```
$(testpmd) -l 2-3 -n 4 \
        --vdev=virtio_user0,path=/dev/vhost-net,queues=2,queue_size=1024 \
        -- -i --txqflags=0x0 --disable-hw-vlan --enable-lro \
        --enable-rx-cksum --txq=2 --rxq=2 --rxd=1024 \
        --txd=1024
```

1. Start testpmd:

```
(testpmd) start
```

2. Configure IP address and start tap:

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

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

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

5.7.2 Limitations

This solution is only available on Linux systems.

CHAPTER 6

DPDK Tools User Guides

6.1 dpdk-procinfo Application

The dpdk-procinfo application is a Data Plane Development Kit (DPDK) application that runs as a DPDK secondary process and is capable of retrieving port statistics, resetting port statistics and printing DPDK memory information. This application extends the original functionality that was supported by dump_cfg.

6.1.1 Running the Application

The application has a number of command line options:

```
./$(RTE_TARGET)/app/dpdk-procinfo -- -m | [-p PORTMASK] [--stats | --xstats |
--stats-reset | --xstats-reset]
```

Parameters

-p PORTMASK: Hexadecimal bitmask of ports to configure.

–stats The stats parameter controls the printing of generic port statistics. If no port mask is specified stats are printed for all DPDK ports.

–xstats The xstats parameter controls the printing of extended port statistics. If no port mask is specified xstats are printed for all DPDK ports.

–stats-reset The stats-reset parameter controls the resetting of generic port statistics. If no port mask is specified, the generic stats are reset for all DPDK ports.

–xstats-reset The xstats-reset parameter controls the resetting of extended port statistics. If no port mask is specified xstats are reset for all DPDK ports.

-m: Print DPDK memory information.

6.2 dpdk-pdump Application

The dpdk-pdump tool is a Data Plane Development Kit (DPDK) tool that runs as a DPDK secondary process and is capable of enabling packet capture on dpdk ports.

Note:

- The dpdk-pdump tool can only be used in conjunction with a primary application which has the packet capture framework initialized already.
- The dpdk-pdump tool depends on libpcap based PMD which 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 libpcap based PMD can be enabled by setting CONFIG_RTE_LIBRTE_PMD_PCAP=y and recompiling the DPDK.

6.2.1 Running the Application

The tool has a number of command line options:

```
./build/app/dpdk-pdump --
                       --pdump '(port=<port id> | device_id=<pci id or vdev name>),
                                (queue=<queue_id>),
                                (rx-dev=<iface or pcap file> |
                                 tx-dev=<iface or pcap file>),
                                [ring-size=<ring size>],
                                [mbuf-size=<mbuf data size>],
                                [total-num-mbufs=<number of mbufs>]'
                       [--server-socket-path=<server socket dir>]
                       [--client-socket-path=<client socket dir>]
```

The --pdump command line option is mandatory and it takes various sub arguments which are described in below section.

Note:

- Parameters inside the parentheses represents mandatory parameters.
- Parameters inside the square brackets represents optional parameters.
- Multiple instances of --pdump can be passed to capture packets on different port and queue combinations.

The --server-socket-path command line option is optional. This represents the server socket directory. If no value is passed default values are used i.e. /var/run/.dpdk/ for root users and ~/.dpdk/ for non root users.

The --client-socket-path command line option is optional. This represents the client socket directory. If no value is passed default values are used i.e. /var/run/.dpdk/ for root users and ~/.dpdk/ for non root users.

The --pdump parameters

port: Port id of the eth device on which packets should be captured.

device_id: PCI address (or) name of the eth device on which packets should be captured.

Note:

- As of now the `dpdk-pdump` tool cannot capture the packets of virtual devices in the primary process due to a bug in the ethdev library. Due to this bug, in a multi process context, when the primary and secondary have different ports set, then the secondary process (here the `dpdk-pdump` tool) overwrites the `rte_eth_devices[]` entries of the primary process.

`queue`: Queue id of the eth device on which packets should be captured. The user can pass a queue value of `*` to enable packet capture on all queues of the eth device.

`rx-dev`: Can be either a pcap file name or any Linux iface.

`tx-dev`: Can be either a pcap file name or any Linux iface.

Note:

- To receive ingress packets only, `rx-dev` should be passed.
- To receive egress packets only, `tx-dev` should be passed.
- To receive ingress and egress packets separately `rx-dev` and `tx-dev` should both be passed with the different file names or the Linux iface names.
- To receive ingress and egress packets together, `rx-dev` and `tx-dev` should both be passed with the same file name or the same Linux iface name.

`ring-size`: Size of the ring. This value is used internally for ring creation. The ring will be used to enqueue the packets from the primary application to the secondary. This is an optional parameter with default size 16384.

`mbuf-size`: Size of the mbuf data. This is used internally for mempool creation. Ideally this value must be same as the primary application's mempool's mbuf data size which is used for packet RX. This is an optional parameter with default size 2176.

`total-num-mbufs`: Total number mbufs in mempool. This is used internally for mempool creation. This is an optional parameter with default value 65535.

6.2.2 Example

```
$ sudo ./build/app/dpdk-pdump -- --pdump 'port=0,queue=*,rx-dev=/tmp/rx.pcap'
```

6.3 dpdk-pmdinfo Application

The `dpdk-pmdinfo` tool is a Data Plane Development Kit (DPDK) utility that can dump a PMDs hardware support info.

6.3.1 Running the Application

The tool has a number of command line options:

```
dpdk-pmdinfo [-hrtp] [-d <pci id file] <elf-file>

-h, --help            Show a short help message and exit
```

```
-r, --raw             Dump as raw json strings
-d FILE, --pcidb=FILE Specify a pci database to get vendor names from
-t, --table           Output information on hw support as a hex table
-p, --plugindir       Scan dpdk for autoload plugins
```

Note:

- Parameters inside the square brackets represents optional parameters.

6.4 dpdk-devbind Application

The `dpdk-devbind` tool is a Data Plane Development Kit (DPDK) utility that helps binding and unbinding devices from specific drivers. As well as checking their status in that regard.

6.4.1 Running the Application

The tool has a number of command line options:

```
dpdk-devbind [options] DEVICE1 DEVICE2 ....
```

6.4.2 OPTIONS

- `--help, --usage`

  Display usage information and quit

- `-s, --status`

  Print the current status of all known network interfaces. For each device, it displays the PCI domain, bus, slot and function, along with a text description of the device. Depending upon whether the device is being used by a kernel driver, the `igb_uio` driver, or no driver, other relevant information will be displayed: - the Linux interface name e.g. `if=eth0` - the driver being used e.g. `drv=igb_uio` - any suitable drivers not currently using that device e.g. `unused=igb_uio` NOTE: if this flag is passed along with a bind/unbind option, the status display will always occur after the other operations have taken place.

- `-b driver, --bind=driver`

  Select the driver to use or "none" to unbind the device

- `-u, --unbind`

  Unbind a device (Equivalent to `-b none`)

- `--force`

  By default, devices which are used by Linux - as indicated by having routes in the routing table - cannot be modified. Using the `--force` flag overrides this behavior, allowing active links to be forcibly unbound. WARNING: This can lead to loss of network connection and should be used with caution.

Warning: Due to the way VFIO works, there are certain limitations to which devices can be used with VFIO. Mainly it comes down to how IOMMU groups work. Any Virtual Function device can be used with VFIO on its own, but physical devices will require either all ports bound to VFIO, or some of them bound to VFIO while others not being bound to anything at all.

If your device is behind a PCI-to-PCI bridge, the bridge will then be part of the IOMMU group in which your device is in. Therefore, the bridge driver should also be unbound from the bridge PCI device for VFIO to work with devices behind the bridge.

Warning: While any user can run the `dpdk-devbind.py` script to view the status of the network ports, binding or unbinding network ports requires root privileges.

6.4.3 Examples

To display current device status:

```
dpdk-devbind --status
```

To bind eth1 from the current driver and move to use igb_uio:

```
dpdk-devbind --bind=igb_uio eth1
```

To unbind 0000:01:00.0 from using any driver:

```
dpdk-devbind -u 0000:01:00.0
```

To bind 0000:02:00.0 and 0000:02:00.1 to the ixgbe kernel driver:

```
dpdk-devbind -b ixgbe 02:00.0 02:00.1
```

To check status of all network ports, assign one to the igb_uio driver and check status again:

```
# Check the status of the available devices.
dpdk-devbind --status
Network devices using DPDK-compatible driver
============================================
<none>

Network devices using kernel driver
===================================
0000:0a:00.0 '82599ES 10-Gigabit' if=eth2 drv=ixgbe unused=


# Bind the device to igb_uio.
sudo dpdk-devbind -b igb_uio 0000:0a:00.0


# Recheck the status of the devices.
dpdk-devbind --status
Network devices using DPDK-compatible driver
============================================
0000:0a:00.0 '82599ES 10-Gigabit' drv=igb_uio unused=
```

6.5 dpdk-test-crypto-perf Application

The dpdk-test-crypto-perf tool is a Data Plane Development Kit (DPDK) utility that allows measuring performance parameters of PMDs available in the crypto tree. There are available two measurement types: throughput and latency. User can use multiply cores to run tests on but only one type of crypto PMD can be measured during single application execution. Cipher parameters, type of device, type of operation and chain mode have to be specified in the command line as application parameters. These parameters are checked using device capabilities structure.

6.5.1 Limitations

On hardware devices the cycle-count doesn't always represent the actual offload cost. The cycle-count only represents the offload cost when the hardware accelerator is not fully loaded, when loaded the cpu cycles freed up by the offload are still consumed by the test tool and included in the cycle-count. These cycles are consumed by retries and inefficient API calls enqueuing and dequeuing smaller bursts than specified by the cmdline parameter. This results in a larger cycle-count measurement and should not be interpreted as an offload cost measurement.

On hardware devices the throughput measurement is not necessarily the maximum possible for the device, e.g. it may be necessary to use multiple cores to keep the hardware accelerator fully loaded and so measure maximum throughput.

6.5.2 Compiling the Application

Step 1: PMD setting

The dpdk-test-crypto-perf tool depends on crypto device drivers PMD which are disabled by default in the build configuration file common_base. The crypto device drivers PMD which should be tested can be enabled by setting:

```
CONFIG_RTE_LIBRTE_PMD_<name>=y
```

Setting example for open ssl PMD:

```
CONFIG_RTE_LIBRTE_PMD_OPENSSL=y
```

Step 2: Linearization setting

It is possible linearized input segmented packets just before crypto operation for devices which doesn't support scatter-gather, and allows to measure performance also for this use case.

To set on the linearization options add below definition to the cperf_ops.h file:

```
#define CPERF_LINEARIZATION_ENABLE
```

Step 3: Build the application

Execute the dpdk-setup.sh script to build the DPDK library together with the dpdk-test-crypto-perf applcation.

Initially, the user must select a DPDK target to choose the correct target type and compiler options to use when building the libraries. The user must have all libraries, modules, updates and compilers installed in the system prior to this, as described in the earlier chapters in this Getting Started Guide.

6.5.3 Running the Application

The tool application has a number of command line options:

```
dpdk-test-crypto-perf [EAL Options] -- [Application Options]
```

EAL Options

The following are the EAL command-line options that can be used in conjunction with the `dpdk-test-crypto-perf` applcation. See the DPDK Getting Started Guides for more information on these options.

- `-c <COREMASK>` or `-l <CORELIST>`

  Set the hexadecimal bitmask of the cores to run on. The corelist is a list cores to use.

- `-w <PCI>`

  Add a PCI device in white list.

- `--vdev <driver><id>`

  Add a virtual device.

Appication Options

The following are the appication command-line options:

- `--ptest type`

  Set test type, where `type` is one of the following:

  ```
  throughput
  latency
  verify
  ```

- `--silent`

  Disable options dump.

- `--pool-sz <n>`

  Set the number of mbufs to be allocated in the mbuf pool.

- `--total-ops <n>`

  Set the number of total operations performed.

- `--burst-sz <n>`

  Set the number of packets per burst.

  This can be set as:

  - Single value (i.e. `--burst-sz 16`)
  - Range of values, using the following structure `min:inc:max`, where `min` is minimum size, `inc` is the increment size and `max` is the maximum size (i.e. `--burst-sz 16:2:32`)
  - List of values, up to 32 values, separated in commas (i.e. `--burst-sz 16,24,32`)

- `--buffer-sz <n>`

  Set the size of single packet (plaintext or ciphertext in it).

  This can be set as:

  - Single value (i.e. `--buffer-sz 16`)
  - Range of values, using the following structure `min:inc:max`, where `min` is minimum size, `inc` is the increment size and `max` is the maximum size (i.e. `--buffer-sz 16:2:32`)
  - List of values, up to 32 values, separated in commas (i.e. `--buffer-sz 32,64,128`)

- `--segments-nb <n>`

  Set the number of segments per packet.

- `--devtype <name>`

  Set device type, where `name` is one of the following:

```
crypto_null
crypto_aesni_mb
crypto_aesni_gcm
crypto_openssl
crypto_qat
crypto_snow3g
crypto_kasumi
crypto_zuc
```

- `--optype <name>`

  Set operation type, where `name` is one of the following:

```
cipher-only
auth-only
cipher-then-auth
auth-then-cipher
aead
```

  For GCM/CCM algorithms you should use aead flag.

- `--sessionless`

  Enable session-less crypto operations mode.

- `--out-of-place`

  Enable out-of-place crypto operations mode.

- `--test-file <name>`

  Set test vector file path. See the Test Vector File chapter.

- `--test-name <name>`

  Set specific test name section in the test vector file.

- `--cipher-algo <name>`

  Set cipher algorithm name, where `name` is one of the following:

```
3des-cbc
3des-ecb
3des-ctr
aes-cbc
aes-ccm
aes-ctr
aes-ecb
aes-gcm
```

```
aes-f8
aes-xts
arc4
null
kasumi-f8
snow3g-uea2
zuc-eea3
```

- `--cipher-op <mode>`

  Set cipher operation mode, where `mode` is one of the following:

```
encrypt
decrypt
```

- `--cipher-key-sz <n>`

  Set the size of cipher key.

- `--cipher-iv-sz <n>`

  Set the size of cipher iv.

- `--auth-algo <name>`

  Set authentication algorithm name, where `name` is one of the following:

```
3des-cbc
aes-cbc-mac
aes-ccm
aes-cmac
aes-gcm
aes-gmac
aes-xcbc-mac
md5
md5-hmac
sha1
sha1-hmac
sha2-224
sha2-224-hmac
sha2-256
sha2-256-hmac
sha2-384
sha2-384-hmac
sha2-512
sha2-512-hmac
kasumi-f9
snow3g-uia2
zuc-eia3
```

- `--auth-op <mode>`

  Set authentication operation mode, where `mode` is one of the following:

```
verify
generate
```

- `--auth-key-sz <n>`

  Set the size of authentication key.

- `--auth-digest-sz <n>`

  Set the size of authentication digest.

- `--auth-aad-sz <n>`

  Set the size of authentication aad.

- `--csv-friendly`

  Enable test result output CSV friendly rather than human friendly.

Test Vector File

The test vector file is a text file contain information about test vectors. The file is made of the sections. The first section doesn't have header. It contain global information used in each test variant vectors - typicaly information about plaintext, ciphertext, cipher key, aut key, initial vector. All other sections begin header. The sections contain particular information typicaly digest.

Format of the file:

Each line beginig with sign '#' contain comment and it is ignored by parser:

```
# <comment>
```

Header line is just name in square bracket:

```
[<section name>]
```

Data line contain information tocken then sign '=' and a string of bytes in C byte array format:

```
<tocken> = <C byte array>
```

Tockens list:

- `plaintext`

  Original plaintext to be crypted.

- `ciphertext`

  Encrypted plaintext string.

- `cipher_key`

  Key used in cipher operation.

- `auth_key`

  Key used in auth operation.

- `iv`

  Initial vector.

- `aad`

  Additional data.

- `digest`

  Digest string.

6.5.4 Examples

Call application for performance throughput test of single Aesni MB PMD for cipher encryption aes-cbc and auth generation sha1-hmac, one milion operations, burst size 32, packet size 64:

```
dpdk-test-crypto-perf -l 6-7 --vdev crypto_aesni_mb_pmd -w 0000:00:00.0 --
--ptest throughput --devtype crypto_aesni_mb --optype cipher-then-auth
--cipher-algo aes-cbc --cipher-op encrypt --cipher-key-sz 16 --auth-algo
sha1-hmac --auth-op generate --auth-key-sz 64 --auth-digest-sz 12
--total-ops 10000000 --burst-sz 32 --buffer-sz 64
```

Call application for performance latency test of two Aesni MB PMD executed on two cores for cipher encryption aes-cbc, ten operations in silent mode:

```
dpdk-test-crypto-perf -l 4-7 --vdev crypto_aesni_mb_pmd1
--vdev crypto_aesni_mb_pmd2 -w 0000:00:00.0 -- --devtype crypto_aesni_mb
--cipher-algo aes-cbc --cipher-key-sz 16 --cipher-iv-sz 16
--cipher-op encrypt --optype cipher-only --silent
--ptest latency --total-ops 10
```

Call application for verification test of single open ssl PMD for cipher encryption aes-gcm and auth generation aes-gcm,ten operations in silent mode, test vector provide in file "test_aes_gcm.data" with packet verification:

```
dpdk-test-crypto-perf -l 4-7 --vdev crypto_openssl -w 0000:00:00.0 --
--devtype crypto_openssl --cipher-algo aes-gcm --cipher-key-sz 16
--cipher-iv-sz 16 --cipher-op encrypt --auth-algo aes-gcm --auth-key-sz 16
--auth-digest-sz 16 --auth-aad-sz 16 --auth-op generate --optype aead
--silent --ptest verify --total-ops 10
--test-file test_aes_gcm.data
```

Test vector file for cipher algorithm aes cbc 256 with authorization sha:

```
# Global Section
plaintext =
0xff, 0xca, 0xfb, 0xf1, 0x38, 0x20, 0x2f, 0x7b, 0x24, 0x98, 0x26, 0x7d, 0x1d, 0x9f,␣
↪0xb3, 0x93,
0xd9, 0xef, 0xbd, 0xad, 0x4e, 0x40, 0xbd, 0x60, 0xe9, 0x48, 0x59, 0x90, 0x67, 0xd7,␣
↪0x2b, 0x7b,
0x8a, 0xe0, 0x4d, 0xb0, 0x70, 0x38, 0xcc, 0x48, 0x61, 0x7d, 0xee, 0xd6, 0x35, 0x49,␣
↪0xae, 0xb4,
0xaf, 0x6b, 0xdd, 0xe6, 0x21, 0xc0, 0x60, 0xce, 0x0a, 0xf4, 0x1c, 0x2e, 0x1c, 0x8d,␣
↪0xe8, 0x7b
ciphertext =
0x77, 0xF9, 0xF7, 0x7A, 0xA3, 0xCB, 0x68, 0x1A, 0x11, 0x70, 0xD8, 0x7A, 0xB6, 0xE2,␣
↪0x37, 0x7E,
0xD1, 0x57, 0x1C, 0x8E, 0x85, 0xD8, 0x08, 0xBF, 0x57, 0x1F, 0x21, 0x6C, 0xAD, 0xAD,␣
↪0x47, 0x1E,
0x0D, 0x6B, 0x79, 0x39, 0x15, 0x4E, 0x5B, 0x59, 0x2D, 0x76, 0x87, 0xA6, 0xD6, 0x47,␣
↪0x8F, 0x82,
0xB8, 0x51, 0x91, 0x32, 0x60, 0xCB, 0x97, 0xDE, 0xBE, 0xF0, 0xAD, 0xFC, 0x23, 0x2E,␣
↪0x22, 0x02
cipher_key =
0xE4, 0x23, 0x33, 0x8A, 0x35, 0x64, 0x61, 0xE2, 0x49, 0x03, 0xDD, 0xC6, 0xB8, 0xCA,␣
↪0x55, 0x7A,
0xd0, 0xe7, 0x4b, 0xfb, 0x5d, 0xe5, 0x0c, 0xe7, 0x6f, 0x21, 0xb5, 0x52, 0x2a, 0xbb,␣
↪0xc7, 0xf7
auth_key =
0xaf, 0x96, 0x42, 0xf1, 0x8c, 0x50, 0xdc, 0x67, 0x1a, 0x43, 0x47, 0x62, 0xc7, 0x04,␣
↪0xab, 0x05,
```

```
0xf5, 0x0c, 0xe7, 0xa2, 0xa6, 0x23, 0xd5, 0x3d, 0x95, 0xd8, 0xcd, 0x86, 0x79, 0xf5,␣
↪0x01, 0x47,
0x4f, 0xf9, 0x1d, 0x9d, 0x36, 0xf7, 0x68, 0x1a, 0x64, 0x44, 0x58, 0x5d, 0xe5, 0x81,␣
↪0x15, 0x2a,
0x41, 0xe4, 0x0e, 0xaa, 0x1f, 0x04, 0x21, 0xff, 0x2c, 0xf3, 0x73, 0x2b, 0x48, 0x1e,␣
↪0xd2, 0xf7
iv =
0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D,␣
↪0x0E, 0x0F
# Section sha 1 hmac buff 32
[sha1_hmac_buff_32]
digest =
0x36, 0xCA, 0x49, 0x6A, 0xE3, 0x54, 0xD8, 0x4F, 0x0B, 0x76, 0xD8, 0xAA, 0x78, 0xEB,␣
↪0x9D, 0x65,
0x2C, 0xCA, 0x1F, 0x97
# Section sha 256 hmac buff 32
[sha256_hmac_buff_32]
digest =
0x1C, 0xB2, 0x3D, 0xD1, 0xF9, 0xC7, 0x6C, 0x49, 0x2E, 0xDA, 0x94, 0x8B, 0xF1, 0xCF,␣
↪0x96, 0x43,
0x67, 0x50, 0x39, 0x76, 0xB5, 0xA1, 0xCE, 0xA1, 0xD7, 0x77, 0x10, 0x07, 0x43, 0x37,␣
↪0x05, 0xB4
```

CHAPTER 7

Testpmd Application User Guide

7.1 Introduction

This document is a user guide for the `testpmd` example application that is shipped as part of the Data Plane Development Kit.

The `testpmd` application can be used to test the DPDK in a packet forwarding mode and also to access NIC hardware features such as Flow Director. It also serves as a example of how to build a more fully-featured application using the DPDK SDK.

The guide shows how to build and run the testpmd application and how to configure the application from the command line and the run-time environment.

7.2 Compiling the Application

The `testpmd` application is compiled as part of the main compilation of the DPDK libraries and tools. Refer to the DPDK Getting Started Guides for details. The basic compilation steps are:

1. Set the required environmental variables and go to the source directory:

```
export RTE_SDK=/path/to/rte_sdk
cd $RTE_SDK
```

2. Set the compilation target. For example:

```
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

3. Build the application:

```
make install T=$RTE_TARGET
```

   The compiled application will be located at:

```
$RTE_SDK/$RTE_TARGET/app/testpmd
```

7.3 Running the Application

7.3.1 EAL Command-line Options

The following are the EAL command-line options that can be used in conjunction with the testpmd, or any other DPDK application. See the DPDK Getting Started Guides for more information on these options.

- `-c COREMASK`

  Set the hexadecimal bitmask of the cores to run on.

- `-l CORELIST`

  List of cores to run on

  The argument format is `<c1>[-c2][,c3[-c4],...]` where `c1`, `c2`, etc are core indexes between 0 and 128.

- `--lcores COREMAP`

  Map lcore set to physical cpu set

  The argument format is:

  ```
  <lcores[@cpus]>[<,lcores[@cpus]>...]
  ```

  Lcore and CPU lists are grouped by `(` and `)` Within the group. The `-` character is used as a range separator and `,` is used as a single number separator. The grouping `()` can be omitted for single element group. The `@` can be omitted if cpus and lcores have the same value.

- `--master-lcore ID`

  Core ID that is used as master.

- `-n NUM`

  Set the number of memory channels to use.

- `-b, --pci-blacklist domain:bus:devid.func`

  Blacklist a PCI device to prevent EAL from using it. Multiple -b options are allowed.

- `-d LIB.so`

  Load an external driver. Multiple -d options are allowed.

- `-w, --pci-whitelist domain:bus:devid:func`

  Add a PCI device in white list.

- `-m MB`

  Memory to allocate. See also `--socket-mem`.

- `-r NUM`

  Set the number of memory ranks (auto-detected by default).

- `-v`

  Display the version information on startup.

- `--xen-dom0`

  Support application running on Xen Domain0 without hugetlbfs.

- `--syslog`

  Set the syslog facility.

- `--socket-mem`

  Set the memory to allocate on specific sockets (use comma separated values).

- `--huge-dir`

  Specify the directory where the hugetlbfs is mounted.

- `--proc-type`

  Set the type of the current process.

- `--file-prefix`

  Prefix for hugepage filenames.

- `-vmware-tsc-map`

  Use VMware TSC map instead of native RDTSC.

- `--vdev`

  Add a virtual device using the format:

```
<driver><id>[,key=val, ...]
```

  For example:

```
--vdev 'net_pcap0,rx_pcap=input.pcap,tx_pcap=output.pcap'
```

- `--base-virtaddr`

  Specify base virtual address.

- `--create-uio-dev`

  Create `/dev/uioX` (usually done by hotplug).

- `--no-shconf`

  No shared config (mmap-ed files).

- `--no-pci`

  Disable pci.

- `--no-hpet`

  Disable hpet.

- `--no-huge`

  Use malloc instead of hugetlbfs.

7.3.2 Testpmd Command-line Options

The following are the command-line options for the testpmd applications. They must be separated from the EAL options, shown in the previous section, with a `--` separator:

```
sudo ./testpmd -l 0-3 -n 4 -- -i --portmask=0x1 --nb-cores=2
```

The commandline options are:

- `-i, --interactive`

  Run testpmd in interactive mode. In this mode, the testpmd starts with a prompt that can be used to start and stop forwarding, configure the application and display stats on the current packet processing session. See *Testpmd Runtime Functions* for more details.

  In non-interactive mode, the application starts with the configuration specified on the command-line and immediately enters forwarding mode.

- `-h, --help`

  Display a help message and quit.

- `-a, --auto-start`

  Start forwarding on initialization.

- `--nb-cores=N`

  Set the number of forwarding cores, where 1 <= N <= "number of cores" or `CONFIG_RTE_MAX_LCORE` from the configuration file. The default value is 1.

- `--nb-ports=N`

  Set the number of forwarding ports, where 1 <= N <= "number of ports" on the board or `CONFIG_RTE_MAX_ETHPORTS` from the configuration file. The default value is the number of ports on the board.

- `--coremask=0xXX`

  Set the hexadecimal bitmask of the cores running the packet forwarding test. The master lcore is reserved for command line parsing only and cannot be masked on for packet forwarding.

- `--portmask=0xXX`

  Set the hexadecimal bitmask of the ports used by the packet forwarding test.

- `--numa`

  Enable NUMA-aware allocation of RX/TX rings and of RX memory buffers (mbufs).

- `--port-numa-config=(port,socket)[,(port,socket)]`

  Specify the socket on which the memory pool to be used by the port will be allocated.

- `--ring-numa-config=(port,flag,socket)[,(port,flag,socket)]`

  Specify the socket on which the TX/RX rings for the port will be allocated. Where flag is 1 for RX, 2 for TX, and 3 for RX and TX.

- `--socket-num=N`

  Set the socket from which all memory is allocated in NUMA mode, where 0 <= N < number of sockets on the board.

- `--mbuf-size=N`

  Set the data size of the mbufs used to N bytes, where N < 65536. The default value is 2048.

- `--total-num-mbufs=N`

  Set the number of mbufs to be allocated in the mbuf pools, where N > 1024.

- `--max-pkt-len=N`

  Set the maximum packet size to N bytes, where N >= 64. The default value is 1518.

- `--eth-peers-configfile=name`

  Use a configuration file containing the Ethernet addresses of the peer ports. The configuration file should contain the Ethernet addresses on separate lines:

```
XX:XX:XX:XX:XX:01
XX:XX:XX:XX:XX:02
...
```

- `--eth-peer=N,XX:XX:XX:XX:XX:XX`

  Set the MAC address `XX:XX:XX:XX:XX:XX` of the peer port N, where 0 <= N < `CONFIG_RTE_MAX_ETHPORTS` from the configuration file.

- `--pkt-filter-mode=mode`

  Set Flow Director mode where mode is either `none` (the default), `signature` or `perfect`. See *flow_director_filter* for more details.

- `--pkt-filter-report-hash=mode`

  Set Flow Director hash match reporting mode where mode is `none`, `match` (the default) or `always`.

- `--pkt-filter-size=N`

  Set Flow Director allocated memory size, where N is 64K, 128K or 256K. Sizes are in kilobytes. The default is 64.

- `--pkt-filter-flexbytes-offset=N`

  Set the flexbytes offset. The offset is defined in words (not bytes) counted from the first byte of the destination Ethernet MAC address, where N is 0 <= N <= 32. The default value is 0x6.

- `--pkt-filter-drop-queue=N`

  Set the drop-queue. In perfect filter mode, when a rule is added with queue = -1, the packet will be enqueued into the RX drop-queue. If the drop-queue does not exist, the packet is dropped. The default value is N=127.

- `--disable-crc-strip`

  Disable hardware CRC stripping.

- `--enable-lro`

  Enable large receive offload.

- `--enable-rx-cksum`

  Enable hardware RX checksum offload.

- `--enable-scatter`

  Enable scatter (multi-segment) RX.

- `--disable-hw-vlan`

  Disable hardware VLAN.

- `--disable-hw-vlan-filter`

  Disable hardware VLAN filter.

- `--disable-hw-vlan-strip`

  Disable hardware VLAN strip.

- `--disable-hw-vlan-extend`

  Disable hardware VLAN extend.

- `--enable-drop-en`

  Enable per-queue packet drop for packets with no descriptors.

- `--disable-rss`

  Disable RSS (Receive Side Scaling).

- `--port-topology=mode`

  Set port topology, where mode is `paired` (the default) or `chained`.

  In `paired` mode, the forwarding is between pairs of ports, for example: (0,1), (2,3), (4,5).

  In `chained` mode, the forwarding is to the next available port in the port mask, for example: (0,1), (1,2), (2,0).

  The ordering of the ports can be changed using the portlist testpmd runtime function.

- `--forward-mode=mode`

  Set the forwarding mode where `mode` is one of the following:

```
io (the default)
mac
mac_swap
flowgen
rxonly
txonly
csum
icmpecho
ieee1588
```

- `--rss-ip`

  Set RSS functions for IPv4/IPv6 only.

- `--rss-udp`

  Set RSS functions for IPv4/IPv6 and UDP.

- `--rxq=N`

  Set the number of RX queues per port to N, where 1 <= N <= 65535. The default value is 1.

- `--rxd=N`

  Set the number of descriptors in the RX rings to N, where N > 0. The default value is 128.

- `--txq=N`

  Set the number of TX queues per port to N, where 1 <= N <= 65535. The default value is 1.

- `--txd=N`

  Set the number of descriptors in the TX rings to N, where N > 0. The default value is 512.

- `--burst=N`

  Set the number of packets per burst to N, where 1 <= N <= 512. The default value is 16.

- `--mbcache=N`

  Set the cache of mbuf memory pools to N, where 0 <= N <= 512. The default value is 16.

- `--rxpt=N`

  Set the prefetch threshold register of RX rings to N, where N >= 0. The default value is 8.

- `--rxht=N`

  Set the host threshold register of RX rings to N, where N >= 0. The default value is 8.

- `--rxfreet=N`

  Set the free threshold of RX descriptors to N, where 0 <= N < value of –rxd. The default value is 0.

- `--rxwt=N`

  Set the write-back threshold register of RX rings to N, where N >= 0. The default value is 4.

- `--txpt=N`

  Set the prefetch threshold register of TX rings to N, where N >= 0. The default value is 36.

- `--txht=N`

  Set the host threshold register of TX rings to N, where N >= 0. The default value is 0.

- `--txwt=N`

  Set the write-back threshold register of TX rings to N, where N >= 0. The default value is 0.

- `--txfreet=N`

  Set the transmit free threshold of TX rings to N, where 0 <= N <= value of `--txd`. The default value is 0.

- `--txrst=N`

  Set the transmit RS bit threshold of TX rings to N, where 0 <= N <= value of `--txd`. The default value is 0.

- `--txqflags=0xXXXXXXXX`

  Set the hexadecimal bitmask of TX queue flags, where 0 <= N <= 0x7FFFFFFF. The default value is 0.

  Note: When using hardware offload functions such as vlan or checksum add `txqflags=0` to force the full-featured TX code path. In some PMDs this may already be the default.

- `--rx-queue-stats-mapping=(port,queue,mapping)[,(port,queue,mapping)]`

  Set the RX queues statistics counters mapping 0 <= mapping <= 15.

- `--tx-queue-stats-mapping=(port,queue,mapping)[,(port,queue,mapping)]`

  Set the TX queues statistics counters mapping 0 <= mapping <= 15.

- `--no-flush-rx`

  Don't flush the RX streams before starting forwarding. Used mainly with the PCAP PMD.

- `--txpkts=X[,Y]`

  Set TX segment sizes or total packet length. Valid for `tx-only` and `flowgen` forwarding modes.

- `--disable-link-check`

  Disable check on link status when starting/stopping ports.

7.4 Testpmd Runtime Functions

Where the testpmd application is started in interactive mode, (`-i|--interactive`), it displays a prompt that can be used to start and stop forwarding, configure the application, display statistics (including the extended NIC statistics aka xstats) , set the Flow Director and other tasks:

```
testpmd>
```

The testpmd prompt has some, limited, readline support. Common bash command-line functions such as `Ctrl+a` and `Ctrl+e` to go to the start and end of the prompt line are supported as well as access to the command history via the up-arrow.

There is also support for tab completion. If you type a partial command and hit `<TAB>` you get a list of the available completions:

```
testpmd> show port <TAB>

    info [Mul-choice STRING]: show|clear port info|stats|xstats|fdir|stat_qmap|dcb_
↪tc|cap X
    info [Mul-choice STRING]: show|clear port info|stats|xstats|fdir|stat_qmap|dcb_
↪tc|cap all
    stats [Mul-choice STRING]: show|clear port info|stats|xstats|fdir|stat_qmap|dcb_
↪tc|cap X
    stats [Mul-choice STRING]: show|clear port info|stats|xstats|fdir|stat_qmap|dcb_
↪tc|cap all
    ...
```

Note: Some examples in this document are too long to fit on one line are are shown wrapped at "\" for display purposes:

```
testpmd> set flow_ctrl rx (on|off) tx (on|off) (high_water) (low_water) \
         (pause_time) (send_xon) (port_id)
```

In the real `testpmd>` prompt these commands should be on a single line.

7.4.1 Help Functions

The testpmd has on-line help for the functions that are available at runtime. These are divided into sections and can be accessed using help, help section or help all:

```
testpmd> help

    help control    : Start and stop forwarding.
    help display    : Displaying port, stats and config information.
    help config     : Configuration information.
    help ports      : Configuring ports.
    help registers  : Reading and setting port registers.
    help filters    : Filters configuration help.
    help all        : All of the above sections.
```

7.4.2 Control Functions

start

Start packet forwarding with current configuration:

```
testpmd> start
```

start tx_first

Start packet forwarding with current configuration after sending specified number of bursts of packets:

```
testpmd> start tx_first ("""|burst_num)
```

The default burst number is 1 when `burst_num` not presented.

stop

Stop packet forwarding, and display accumulated statistics:

```
testpmd> stop
```

quit

Quit to prompt:

```
testpmd> quit
```

7.4.3 Display Functions

The functions in the following sections are used to display information about the testpmd configuration or the NIC status.

show port

Display information for a given port or all ports:

```
testpmd> show port (info|stats|xstats|fdir|stat_qmap|dcb_tc|cap) (port_id|all)
```

The available information categories are:

- `info`: General port information such as MAC address.
- `stats`: RX/TX statistics.
- `xstats`: RX/TX extended NIC statistics.
- `fdir`: Flow Director information and statistics.
- `stat_qmap`: Queue statistics mapping.
- `dcb_tc`: DCB information such as TC mapping.
- `cap`: Supported offload capabilities.

For example:

```
testpmd> show port info 0

********************* Infos for port 0 *********************

MAC address: XX:XX:XX:XX:XX:XX
Connect to socket: 0
memory allocation on the socket: 0
Link status: up
Link speed: 40000 Mbps
Link duplex: full-duplex
Promiscuous mode: enabled
Allmulticast mode: disabled
Maximum number of MAC addresses: 64
Maximum number of MAC addresses of hash filtering: 0
VLAN offload:
    strip on
    filter on
    qinq(extend) off
Redirection table size: 512
Supported flow types:
  ipv4-frag
  ipv4-tcp
  ipv4-udp
  ipv4-sctp
  ipv4-other
  ipv6-frag
  ipv6-tcp
  ipv6-udp
  ipv6-sctp
  ipv6-other
  l2_payload
  port
  vxlan
  geneve
  nvgre
```

show port rss reta

Display the rss redirection table entry indicated by masks on port X:

```
testpmd> show port (port_id) rss reta (size) (mask0, mask1...)
```

size is used to indicate the hardware supported reta size

show port rss-hash

Display the RSS hash functions and RSS hash key of a port:

```
testpmd> show port (port_id) rss-hash ipv4|ipv4-frag|ipv4-tcp|ipv4-udp|ipv4-sctp|ipv4-
↪other|ipv6|ipv6-frag|ipv6-tcp|ipv6-udp|ipv6-sctp|ipv6-other|l2-payload|ipv6-ex|ipv6-
↪tcp-ex|ipv6-udp-ex [key]
```

clear port

Clear the port statistics for a given port or for all ports:

```
testpmd> clear port (info|stats|xstats|fdir|stat_qmap) (port_id|all)
```

For example:

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

show (rxq|txq)

Display information for a given port's RX/TX queue:

```
testpmd> show (rxq|txq) info (port_id) (queue_id)
```

show config

Displays the configuration of the application. The configuration comes from the command-line, the runtime or the application defaults:

```
testpmd> show config (rxtx|cores|fwd|txpkts)
```

The available information categories are:

- `rxtx`: RX/TX configuration items.
- `cores`: List of forwarding cores.
- `fwd`: Packet forwarding configuration.
- `txpkts`: Packets to TX configuration.

For example:

```
testpmd> show config rxtx

io packet forwarding - CRC stripping disabled - packets/burst=16
nb forwarding cores=2 - nb forwarding ports=1
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
```

set fwd

Set the packet forwarding mode:

```
testpmd> set fwd (io|mac|macswap|flowgen| \
                  rxonly|txonly|csum|icmpecho) (""|retry)
```

`retry` can be specified for forwarding engines except `rx_only`.

The available information categories are:

- `io`: Forwards packets "as-is" in I/O mode. This is the fastest possible forwarding operation as it does not access packets data. This is the default mode.
- `mac`: Changes the source and the destination Ethernet addresses of packets before forwarding them. Default application behaviour is to set source Ethernet address to that of the transmitting interface, and destination address to a dummy value (set during init). The user may specify a target destination Ethernet address via the 'eth-peer' or 'eth-peer-configfile' command-line options. It is not currently possible to specify a specific source Ethernet address.
- `macswap`: MAC swap forwarding mode. Swaps the source and the destination Ethernet addresses of packets before forwarding them.
- `flowgen`: Multi-flow generation mode. Originates a number of flows (with varying destination IP addresses), and terminate receive traffic.
- `rxonly`: Receives packets but doesn't transmit them.
- `txonly`: Generates and transmits packets without receiving any.
- `csum`: Changes the checksum field with hardware or software methods depending on the offload flags on the packet.
- `icmpecho`: Receives a burst of packets, lookup for IMCP echo requests and, if any, send back ICMP echo replies.
- `ieee1588`: Demonstrate L2 IEEE1588 V2 PTP timestamping for RX and TX. Requires `CONFIG_RTE_LIBRTE_IEEE1588=y`.

Note: TX timestamping is only available in the "Full Featured" TX path. To force `testpmd` into this mode set `--txqflags=0`.

Example:

```
testpmd> set fwd rxonly

Set rxonly packet forwarding mode
```

read rxd

Display an RX descriptor for a port RX queue:

```
testpmd> read rxd (port_id) (queue_id) (rxd_id)
```

For example:

```
testpmd> read rxd 0 0 4
     0x0000000B - 0x001D0180 / 0x0000000B - 0x001D0180
```

read txd

Display a TX descriptor for a port TX queue:

```
testpmd> read txd (port_id) (queue_id) (txd_id)
```

For example:

```
testpmd> read txd 0 0 4
     0x00000001 - 0x24C3C440 / 0x000F0000 - 0x2330003C
```

show vf stats

Display VF statistics:

```
testpmd> show vf stats (port_id) (vf_id)
```

clear vf stats

Reset VF statistics:

```
testpmd> clear vf stats (port_id) (vf_id)
```

7.4.4 Configuration Functions

The testpmd application can be configured from the runtime as well as from the command-line.

This section details the available configuration functions that are available.

Note: Configuration changes only become active when forwarding is started/restarted.

set default

Reset forwarding to the default configuration:

```
testpmd> set default
```

set verbose

Set the debug verbosity level:

```
testpmd> set verbose (level)
```

Currently the only available levels are 0 (silent except for error) and 1 (fully verbose).

set nbport

Set the number of ports used by the application:

set nbport (num)

This is equivalent to the `--nb-ports` command-line option.

set nbcore

Set the number of cores used by the application:

```
testpmd> set nbcore (num)
```

This is equivalent to the `--nb-cores` command-line option.

Note: The number of cores used must not be greater than number of ports used multiplied by the number of queues per port.

set coremask

Set the forwarding cores hexadecimal mask:

```
testpmd> set coremask (mask)
```

This is equivalent to the `--coremask` command-line option.

Note: The master lcore is reserved for command line parsing only and cannot be masked on for packet forwarding.

set portmask

Set the forwarding ports hexadecimal mask:

```
testpmd> set portmask (mask)
```

This is equivalent to the `--portmask` command-line option.

set burst

Set number of packets per burst:

```
testpmd> set burst (num)
```

This is equivalent to the `--burst command-line` option.

When retry is enabled, the transmit delay time and number of retries can also be set:

```
testpmd> set burst tx delay (microseconds) retry (num)
```

set txpkts

Set the length of each segment of the TX-ONLY packets or length of packet for FLOWGEN mode:

```
testpmd> set txpkts (x[,y]*)
```

Where x[,y]* represents a CSV list of values, without white space.

set txsplit

Set the split policy for the TX packets, applicable for TX-ONLY and CSUM forwarding modes:

```
testpmd> set txsplit (off|on|rand)
```

Where:

- `off` disable packet copy & split for CSUM mode.
- `on` split outgoing packet into multiple segments. Size of each segment and number of segments per packet is determined by `set txpkts` command (see above).
- `rand` same as 'on', but number of segments per each packet is a random value between 1 and total number of segments.

set corelist

Set the list of forwarding cores:

```
testpmd> set corelist (x[,y]*)
```

For example, to change the forwarding cores:

```
testpmd> set corelist 3,1
testpmd> show config fwd

io packet forwarding - ports=2 - cores=2 - streams=2 - NUMA support disabled
Logical Core 3 (socket 0) forwards packets on 1 streams:
RX P=0/Q=0 (socket 0) -> TX P=1/Q=0 (socket 0) peer=02:00:00:00:00:01
Logical Core 1 (socket 0) forwards packets on 1 streams:
RX P=1/Q=0 (socket 0) -> TX P=0/Q=0 (socket 0) peer=02:00:00:00:00:00
```

Note: The cores are used in the same order as specified on the command line.

set portlist

Set the list of forwarding ports:

```
testpmd> set portlist (x[,y]*)
```

For example, to change the port forwarding:

```
testpmd> set portlist 0,2,1,3
testpmd> show config fwd

io packet forwarding - ports=4 - cores=1 - streams=4
Logical Core 3 (socket 0) forwards packets on 4 streams:
RX P=0/Q=0 (socket 0) -> TX P=2/Q=0 (socket 0) peer=02:00:00:00:00:01
RX P=2/Q=0 (socket 0) -> TX P=0/Q=0 (socket 0) peer=02:00:00:00:00:00
RX P=1/Q=0 (socket 0) -> TX P=3/Q=0 (socket 0) peer=02:00:00:00:00:03
RX P=3/Q=0 (socket 0) -> TX P=1/Q=0 (socket 0) peer=02:00:00:00:00:02
```

set tx loopback

Enable/disable tx loopback:

```
testpmd> set tx loopback (port_id) (on|off)
```

set drop enable

set drop enable bit for all queues:

```
testpmd> set all queues drop (port_id) (on|off)
```

set split drop enable (for VF)

set split drop enable bit for VF from PF:

```
testpmd> set vf split drop (port_id) (vf_id) (on|off)
```

set mac antispoof (for VF)

Set mac antispoof for a VF from the PF:

```
testpmd> set vf mac antispoof  (port_id) (vf_id) (on|off)
```

set macsec offload

Enable/disable MACsec offload:

```
testpmd> set macsec offload (port_id) on encrypt (on|off) replay-protect (on|off)
testpmd> set macsec offload (port_id) off
```

set macsec sc

Configure MACsec secure connection (SC):

```
testpmd> set macsec sc (tx|rx) (port_id) (mac) (pi)
```

Note: The pi argument is ignored for tx. Check the NIC Datasheet for hardware limits.

set macsec sa

Configure MACsec secure association (SA):

```
testpmd> set macsec sa (tx|rx) (port_id) (idx) (an) (pn) (key)
```

Note: The IDX value must be 0 or 1. Check the NIC Datasheet for hardware limits.

set broadcast mode (for VF)

Set broadcast mode for a VF from the PF:

```
testpmd> set vf broadcast (port_id) (vf_id) (on|off)
```

vlan set strip

Set the VLAN strip on a port:

```
testpmd> vlan set strip (on|off) (port_id)
```

vlan set stripq

Set the VLAN strip for a queue on a port:

```
testpmd> vlan set stripq (on|off) (port_id,queue_id)
```

vlan set stripq (for VF)

Set VLAN strip for all queues in a pool for a VF from the PF:

```
testpmd> set vf vlan stripq (port_id) (vf_id) (on|off)
```

vlan set insert (for VF)

Set VLAN insert for a VF from the PF:

```
testpmd> set vf vlan insert (port_id) (vf_id) (vlan_id)
```

vlan set tag (for VF)

Set VLAN tag for a VF from the PF:

```
testpmd> set vf vlan tag (port_id) (vf_id) (on|off)
```

vlan set antispoof (for VF)

Set VLAN antispoof for a VF from the PF:

```
testpmd> set vf vlan antispoof (port_id) (vf_id) (on|off)
```

vlan set filter

Set the VLAN filter on a port:

```
testpmd> vlan set filter (on|off) (port_id)
```

vlan set qinq

Set the VLAN QinQ (extended queue in queue) on for a port:

```
testpmd> vlan set qinq (on|off) (port_id)
```

vlan set tpid

Set the inner or outer VLAN TPID for packet filtering on a port:

```
testpmd> vlan set (inner|outer) tpid (value) (port_id)
```

Note: TPID value must be a 16-bit number (value <= 65536).

rx_vlan add

Add a VLAN ID, or all identifiers, to the set of VLAN identifiers filtered by port ID:

```
testpmd> rx_vlan add (vlan_id|all) (port_id)
```

Note: VLAN filter must be set on that port. VLAN ID < 4096. Depending on the NIC used, number of vlan_ids may be limited to the maximum entries in VFTA table. This is important if enabling all vlan_ids.

rx_vlan rm

Remove a VLAN ID, or all identifiers, from the set of VLAN identifiers filtered by port ID:

```
testpmd> rx_vlan rm (vlan_id|all) (port_id)
```

rx_vlan add (for VF)

Add a VLAN ID, to the set of VLAN identifiers filtered for VF(s) for port ID:

```
testpmd> rx_vlan add (vlan_id) port (port_id) vf (vf_mask)
```

rx_vlan rm (for VF)

Remove a VLAN ID, from the set of VLAN identifiers filtered for VF(s) for port ID:

```
testpmd> rx_vlan rm (vlan_id) port (port_id) vf (vf_mask)
```

tunnel_filter add

Add a tunnel filter on a port:

```
testpmd> tunnel_filter add (port_id) (outer_mac) (inner_mac) (ip_addr) \
         (inner_vlan) (vxlan|nvgre|ipingre) (imac-ivlan|imac-ivlan-tenid|\
         imac-tenid|imac|omac-imac-tenid|oip|iip) (tenant_id) (queue_id)
```

The available information categories are:

- `vxlan`: Set tunnel type as VXLAN.
- `nvgre`: Set tunnel type as NVGRE.
- `ipingre`: Set tunnel type as IP-in-GRE.
- `imac-ivlan`: Set filter type as Inner MAC and VLAN.
- `imac-ivlan-tenid`: Set filter type as Inner MAC, VLAN and tenant ID.
- `imac-tenid`: Set filter type as Inner MAC and tenant ID.
- `imac`: Set filter type as Inner MAC.
- `omac-imac-tenid`: Set filter type as Outer MAC, Inner MAC and tenant ID.
- `oip`: Set filter type as Outer IP.
- `iip`: Set filter type as Inner IP.

Example:

```
testpmd> tunnel_filter add 0 68:05:CA:28:09:82 00:00:00:00:00:00 \
         192.168.2.2 0 ipingre oip 1 1

Set an IP-in-GRE tunnel on port 0, and the filter type is Outer IP.
```

tunnel_filter remove

Remove a tunnel filter on a port:

```
testpmd> tunnel_filter rm (port_id) (outer_mac) (inner_mac) (ip_addr) \
         (inner_vlan) (vxlan|nvgre|ipingre) (imac-ivlan|imac-ivlan-tenid|\
         imac-tenid|imac|omac-imac-tenid|oip|iip) (tenant_id) (queue_id)
```

rx_vxlan_port add

Add an UDP port for VXLAN packet filter on a port:

```
testpmd> rx_vxlan_port add (udp_port) (port_id)
```

rx_vxlan_port remove

Remove an UDP port for VXLAN packet filter on a port:

```
testpmd> rx_vxlan_port rm (udp_port) (port_id)
```

tx_vlan set

Set hardware insertion of VLAN IDs in packets sent on a port:

```
testpmd> tx_vlan set (port_id) vlan_id[, vlan_id_outer]
```

For example, set a single VLAN ID (5) insertion on port 0:

```
tx_vlan set 0 5
```

Or, set double VLAN ID (inner: 2, outer: 3) insertion on port 1:

```
tx_vlan set 1 2 3
```

tx_vlan set pvid

Set port based hardware insertion of VLAN ID in packets sent on a port:

```
testpmd> tx_vlan set pvid (port_id) (vlan_id) (on|off)
```

tx_vlan reset

Disable hardware insertion of a VLAN header in packets sent on a port:

```
testpmd> tx_vlan reset (port_id)
```

csum set

Select hardware or software calculation of the checksum when transmitting a packet using the `csum` forwarding engine:

```
testpmd> csum set (ip|udp|tcp|sctp|outer-ip) (hw|sw) (port_id)
```

Where:

- `ip|udp|tcp|sctp` always relate to the inner layer.
- `outer-ip` relates to the outer IP layer (only for IPv4) in the case where the packet is recognized as a tunnel packet by the forwarding engine (vxlan, gre and ipip are supported). See also the `csum parse-tunnel` command.

Note: Check the NIC Datasheet for hardware limits.

csum parse-tunnel

Define how tunneled packets should be handled by the csum forward engine:

```
testpmd> csum parse-tunnel (on|off) (tx_port_id)
```

If enabled, the csum forward engine will try to recognize supported tunnel headers (vxlan, gre, ipip).

If disabled, treat tunnel packets as non-tunneled packets (a inner header is handled as a packet payload).

Note: The port argument is the TX port like in the `csum set` command.

Example:

Consider a packet in packet like the following:

```
eth_out/ipv4_out/udp_out/vxlan/eth_in/ipv4_in/tcp_in
```

- If parse-tunnel is enabled, the `ip|udp|tcp|sctp` parameters of `csum set` command relate to the inner headers (here `ipv4_in` and `tcp_in`), and the `outer-ip parameter` relates to the outer headers (here `ipv4_out`).
- **If parse-tunnel is disabled, the `ip|udp|tcp|sctp` parameters of `csum set`** command relate to the outer headers, here `ipv4_out` and `udp_out`.

csum show

Display tx checksum offload configuration:

```
testpmd> csum show (port_id)
```

tso set

Enable TCP Segmentation Offload (TSO) in the `csum` forwarding engine:

```
testpmd> tso set (segsize) (port_id)
```

Note: Check the NIC datasheet for hardware limits.

---

tso show

Display the status of TCP Segmentation Offload:

```
testpmd> tso show (port_id)
```

mac_addr add

Add an alternative MAC address to a port:

```
testpmd> mac_addr add (port_id) (XX:XX:XX:XX:XX:XX)
```

mac_addr remove

Remove a MAC address from a port:

```
testpmd> mac_addr remove (port_id) (XX:XX:XX:XX:XX:XX)
```

mac_addr add (for VF)

Add an alternative MAC address for a VF to a port:

```
testpmd> mac_add add port (port_id) vf (vf_id) (XX:XX:XX:XX:XX:XX)
```

mac_addr set

Set the default MAC address for a port:

```
testpmd> mac_addr set (port_id) (XX:XX:XX:XX:XX:XX)
```

mac_addr set (for VF)

Set the MAC address for a VF from the PF:

```
testpmd> set vf mac addr (port_id) (vf_id) (XX:XX:XX:XX:XX:XX)
```

set port-uta

Set the unicast hash filter(s) on/off for a port:

```
testpmd> set port (port_id) uta (XX:XX:XX:XX:XX:XX|all) (on|off)
```

set promisc

Set the promiscuous mode on for a port or for all ports. In promiscuous mode packets are not dropped if they aren't for the specified MAC address:

```
testpmd> set promisc (port_id|all) (on|off)
```

set allmulti

Set the allmulti mode for a port or for all ports:

```
testpmd> set allmulti (port_id|all) (on|off)
```

Same as the ifconfig (8) option. Controls how multicast packets are handled.

set promisc (for VF)

Set the unicast promiscuous mode for a VF from PF. It's supported by Intel i40e NICs now. In promiscuous mode packets are not dropped if they aren't for the specified MAC address:

```
testpmd> set vf promisc (port_id) (vf_id) (on|off)
```

set allmulticast (for VF)

Set the multicast promiscuous mode for a VF from PF. It's supported by Intel i40e NICs now. In promiscuous mode packets are not dropped if they aren't for the specified MAC address:

```
testpmd> set vf allmulti (port_id) (vf_id) (on|off)
```

set tx max bandwidth (for VF)

Set TX max absolute bandwidth (Mbps) for a VF from PF:

```
testpmd> set vf tx max-bandwidth (port_id) (vf_id) (max_bandwidth)
```

set tc tx min bandwidth (for VF)

Set all TCs' TX min relative bandwidth (%) for a VF from PF:

```
testpmd> set vf tc tx min-bandwidth (port_id) (vf_id) (bw1, bw2, ...)
```

set tc tx max bandwidth (for VF)

Set a TC's TX max absolute bandwidth (Mbps) for a VF from PF:

```
testpmd> set vf tc tx max-bandwidth (port_id) (vf_id) (tc_no) (max_bandwidth)
```

set tc strict link priority mode

Set some TCs' strict link priority mode on a physical port:

```
testpmd> set tx strict-link-priority (port_id) (tc_bitmap)
```

set tc tx min bandwidth

Set all TCs' TX min relative bandwidth (%) globally for all PF and VFs:

```
testpmd> set tc tx min-bandwidth (port_id) (bw1, bw2, ...)
```

set flow_ctrl rx

Set the link flow control parameter on a port:

```
testpmd> set flow_ctrl rx (on|off) tx (on|off) (high_water) (low_water) \
         (pause_time) (send_xon) mac_ctrl_frame_fwd (on|off) \
         autoneg (on|off) (port_id)
```

Where:

- `high_water` (integer): High threshold value to trigger XOFF.
- `low_water` (integer): Low threshold value to trigger XON.
- `pause_time` (integer): Pause quota in the Pause frame.
- `send_xon` (0/1): Send XON frame.
- `mac_ctrl_frame_fwd`: Enable receiving MAC control frames.
- `autoneg`: Change the auto-negotiation parameter.

set pfc_ctrl rx

Set the priority flow control parameter on a port:

```
testpmd> set pfc_ctrl rx (on|off) tx (on|off) (high_water) (low_water) \
         (pause_time) (priority) (port_id)
```

Where:

- `high_water` (integer): High threshold value.
- `low_water` (integer): Low threshold value.
- `pause_time` (integer): Pause quota in the Pause frame.
- `priority` (0-7): VLAN User Priority.

set stat_qmap

Set statistics mapping (qmapping 0..15) for RX/TX queue on port:

```
testpmd> set stat_qmap (tx|rx) (port_id) (queue_id) (qmapping)
```

For example, to set rx queue 2 on port 0 to mapping 5:

```
testpmd>set stat_qmap rx 0 2 5
```

set port - rx/tx (for VF)

Set VF receive/transmit from a port:

```
testpmd> set port (port_id) vf (vf_id) (rx|tx) (on|off)
```

set port - mac address filter (for VF)

Add/Remove unicast or multicast MAC addr filter for a VF:

```
testpmd> set port (port_id) vf (vf_id) (mac_addr) \
         (exact-mac|exact-mac-vlan|hashmac|hashmac-vlan) (on|off)
```

set port - rx mode(for VF)

Set the VF receive mode of a port:

```
testpmd> set port (port_id) vf (vf_id) \
         rxmode (AUPE|ROPE|BAM|MPE) (on|off)
```

The available receive modes are:

- `AUPE`: Accepts untagged VLAN.
- `ROPE`: Accepts unicast hash.
- `BAM`: Accepts broadcast packets.
- `MPE`: Accepts all multicast packets.

set port - tx_rate (for Queue)

Set TX rate limitation for a queue on a port:

```
testpmd> set port (port_id) queue (queue_id) rate (rate_value)
```

set port - tx_rate (for VF)

Set TX rate limitation for queues in VF on a port:

```
testpmd> set port (port_id) vf (vf_id) rate (rate_value) queue_mask (queue_mask)
```

set port - mirror rule

Set pool or vlan type mirror rule for a port:

```
testpmd> set port (port_id) mirror-rule (rule_id) \
         (pool-mirror-up|pool-mirror-down|vlan-mirror) \
         (poolmask|vlanid[,vlanid]*) dst-pool (pool_id) (on|off)
```

Set link mirror rule for a port:

```
testpmd> set port (port_id) mirror-rule (rule_id) \
        (uplink-mirror|downlink-mirror) dst-pool (pool_id) (on|off)
```

For example to enable mirror traffic with vlan 0,1 to pool 0:

```
set port 0 mirror-rule 0 vlan-mirror 0,1 dst-pool 0 on
```

reset port - mirror rule

Reset a mirror rule for a port:

```
testpmd> reset port (port_id) mirror-rule (rule_id)
```

set flush_rx

Set the flush on RX streams before forwarding. The default is flush `on`. Mainly used with PCAP drivers to turn off the default behavior of flushing the first 512 packets on RX streams:

```
testpmd> set flush_rx off
```

set bypass mode

Set the bypass mode for the lowest port on bypass enabled NIC:

```
testpmd> set bypass mode (normal|bypass|isolate) (port_id)
```

set bypass event

Set the event required to initiate specified bypass mode for the lowest port on a bypass enabled:

```
testpmd> set bypass event (timeout|os_on|os_off|power_on|power_off) \
         mode (normal|bypass|isolate) (port_id)
```

Where:

- `timeout`: Enable bypass after watchdog timeout.
- `os_on`: Enable bypass when OS/board is powered on.
- `os_off`: Enable bypass when OS/board is powered off.
- `power_on`: Enable bypass when power supply is turned on.
- `power_off`: Enable bypass when power supply is turned off.

set bypass timeout

Set the bypass watchdog timeout to `n` seconds where 0 = instant:

```
testpmd> set bypass timeout (0|1.5|2|3|4|8|16|32)
```

show bypass config

Show the bypass configuration for a bypass enabled NIC using the lowest port on the NIC:

```
testpmd> show bypass config (port_id)
```

set link up

Set link up for a port:

```
testpmd> set link-up port (port id)
```

set link down

Set link down for a port:

```
testpmd> set link-down port (port id)
```

E-tag set

Enable E-tag insertion for a VF on a port:

```
testpmd> E-tag set insertion on port-tag-id (value) port (port_id) vf (vf_id)
```

Disable E-tag insertion for a VF on a port:

```
testpmd> E-tag set insertion off port (port_id) vf (vf_id)
```

Enable/disable E-tag stripping on a port:

```
testpmd> E-tag set stripping (on|off) port (port_id)
```

Enable/disable E-tag based forwarding on a port:

```
testpmd> E-tag set forwarding (on|off) port (port_id)
```

Add an E-tag forwarding filter on a port:

```
testpmd> E-tag set filter add e-tag-id (value) dst-pool (pool_id) port (port_id)
```

Delete an E-tag forwarding filter on a port:: testpmd> E-tag set filter del e-tag-id (value) port (port_id)

7.4.5 Port Functions

The following sections show functions for configuring ports.

Note: Port configuration changes only become active when forwarding is started/restarted.

port attach

Attach a port specified by pci address or virtual device args:

```
testpmd> port attach (identifier)
```

To attach a new pci device, the device should be recognized by kernel first. Then it should be moved under DPDK management. Finally the port can be attached to testpmd.

For example, to move a pci device using ixgbe under DPDK management:

```
# Check the status of the available devices.
./usertools/dpdk-devbind.py --status

Network devices using DPDK-compatible driver
============================================
<none>

Network devices using kernel driver
===================================
0000:0a:00.0 '82599ES 10-Gigabit' if=eth2 drv=ixgbe unused=


# Bind the device to igb_uio.
sudo ./usertools/dpdk-devbind.py -b igb_uio 0000:0a:00.0


# Recheck the status of the devices.
./usertools/dpdk-devbind.py --status
Network devices using DPDK-compatible driver
============================================
0000:0a:00.0 '82599ES 10-Gigabit' drv=igb_uio unused=
```

To attach a port created by virtual device, above steps are not needed.

For example, to attach a port whose pci address is 0000:0a:00.0.

```
testpmd> port attach 0000:0a:00.0
Attaching a new port...
EAL: PCI device 0000:0a:00.0 on NUMA socket -1
EAL:   probe driver: 8086:10fb rte_ixgbe_pmd
EAL:   PCI memory mapped at 0x7f83bfa00000
EAL:   PCI memory mapped at 0x7f83bfa80000
PMD: eth_ixgbe_dev_init(): MAC: 2, PHY: 18, SFP+: 5
PMD: eth_ixgbe_dev_init(): port 0 vendorID=0x8086 deviceID=0x10fb
Port 0 is attached. Now total ports is 1
Done
```

For example, to attach a port created by pcap PMD.

```
testpmd> port attach net_pcap0
Attaching a new port...
PMD: Initializing pmd_pcap for net_pcap0
PMD: Creating pcap-backed ethdev on numa socket 0
Port 0 is attached. Now total ports is 1
Done
```

In this case, identifier is `net_pcap0`. This identifier format is the same as `--vdev` format of DPDK applications.

For example, to re-attach a bonded port which has been previously detached, the mode and slave parameters must be given.

```
testpmd> port attach net_bond_0,mode=0,slave=1
Attaching a new port...
EAL: Initializing pmd_bond for net_bond_0
```

```
EAL: Create bonded device net_bond_0 on port 0 in mode 0 on socket 0.
Port 0 is attached. Now total ports is 1
Done
```

port detach

Detach a specific port:

```
testpmd> port detach (port_id)
```

Before detaching a port, the port should be stopped and closed.

For example, to detach a pci device port 0.

```
testpmd> port stop 0
Stopping ports...
Done
testpmd> port close 0
Closing ports...
Done

testpmd> port detach 0
Detaching a port...
EAL: PCI device 0000:0a:00.0 on NUMA socket -1
EAL:   remove driver: 8086:10fb rte_ixgbe_pmd
EAL:   PCI memory unmapped at 0x7f83bfa00000
EAL:   PCI memory unmapped at 0x7f83bfa80000
Done
```

For example, to detach a virtual device port 0.

```
testpmd> port stop 0
Stopping ports...
Done
testpmd> port close 0
Closing ports...
Done

testpmd> port detach 0
Detaching a port...
PMD: Closing pcap ethdev on numa socket 0
Port 'net_pcap0' is detached. Now total ports is 0
Done
```

To remove a pci device completely from the system, first detach the port from testpmd. Then the device should be moved under kernel management. Finally the device can be removed using kernel pci hotplug functionality.

For example, to move a pci device under kernel management:

```
sudo ./usertools/dpdk-devbind.py -b ixgbe 0000:0a:00.0

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

Network devices using DPDK-compatible driver
============================================
<none>
```

```
Network devices using kernel driver
===================================
0000:0a:00.0 '82599ES 10-Gigabit' if=eth2 drv=ixgbe unused=igb_uio
```

To remove a port created by a virtual device, above steps are not needed.

port start

Start all ports or a specific port:

```
testpmd> port start (port_id|all)
```

port stop

Stop all ports or a specific port:

```
testpmd> port stop (port_id|all)
```

port close

Close all ports or a specific port:

```
testpmd> port close (port_id|all)
```

port start/stop queue

Start/stop a rx/tx queue on a specific port:

```
testpmd> port (port_id) (rxq|txq) (queue_id) (start|stop)
```

Only take effect when port is started.

port config - speed

Set the speed and duplex mode for all ports or a specific port:

```
testpmd> port config (port_id|all) speed␣
↪(10|100|1000|10000|25000|40000|50000|100000|auto) \
         duplex (half|full|auto)
```

port config - queues/descriptors

Set number of queues/descriptors for rxq, txq, rxd and txd:

```
testpmd> port config all (rxq|txq|rxd|txd) (value)
```

This is equivalent to the `--rxq`, `--txq`, `--rxd` and `--txd` command-line options.

port config - max-pkt-len

Set the maximum packet length:

```
testpmd> port config all max-pkt-len (value)
```

This is equivalent to the `--max-pkt-len` command-line option.

port config - CRC Strip

Set hardware CRC stripping on or off for all ports:

```
testpmd> port config all crc-strip (on|off)
```

CRC stripping is on by default.

The `off` option is equivalent to the `--disable-crc-strip` command-line option.

port config - scatter

Set RX scatter mode on or off for all ports:

```
testpmd> port config all scatter (on|off)
```

RX scatter mode is off by default.

The `on` option is equivalent to the `--enable-scatter` command-line option.

port config - TX queue flags

Set a hexadecimal bitmap of TX queue flags for all ports:

```
testpmd> port config all txqflags value
```

This command is equivalent to the `--txqflags` command-line option.

port config - RX Checksum

Set hardware RX checksum offload to on or off for all ports:

```
testpmd> port config all rx-cksum (on|off)
```

Checksum offload is off by default.

The `on` option is equivalent to the `--enable-rx-cksum` command-line option.

port config - VLAN

Set hardware VLAN on or off for all ports:

```
testpmd> port config all hw-vlan (on|off)
```

Hardware VLAN is on by default.

The `off` option is equivalent to the `--disable-hw-vlan` command-line option.

port config - VLAN filter

Set hardware VLAN filter on or off for all ports:

```
testpmd> port config all hw-vlan-filter (on|off)
```

Hardware VLAN filter is on by default.

The `off` option is equivalent to the `--disable-hw-vlan-filter` command-line option.

port config - VLAN strip

Set hardware VLAN strip on or off for all ports:

```
testpmd> port config all hw-vlan-strip (on|off)
```

Hardware VLAN strip is on by default.

The `off` option is equivalent to the `--disable-hw-vlan-strip` command-line option.

port config - VLAN extend

Set hardware VLAN extend on or off for all ports:

```
testpmd> port config all hw-vlan-extend (on|off)
```

Hardware VLAN extend is off by default.

The `off` option is equivalent to the `--disable-hw-vlan-extend` command-line option.

port config - Drop Packets

Set packet drop for packets with no descriptors on or off for all ports:

```
testpmd> port config all drop-en (on|off)
```

Packet dropping for packets with no descriptors is off by default.

The `on` option is equivalent to the `--enable-drop-en` command-line option.

port config - RSS

Set the RSS (Receive Side Scaling) mode on or off:

```
testpmd> port config all rss (all|ip|tcp|udp|sctp|ether|port|vxlan|geneve|nvgre|none)
```

RSS is on by default.

The `none` option is equivalent to the `--disable-rss` command-line option.

port config - RSS Reta

Set the RSS (Receive Side Scaling) redirection table:

```
testpmd> port config all rss reta (hash,queue)[,(hash,queue)]
```

port config - DCB

Set the DCB mode for an individual port:

```
testpmd> port config (port_id) dcb vt (on|off) (traffic_class) pfc (on|off)
```

The traffic class should be 4 or 8.

port config - Burst

Set the number of packets per burst:

```
testpmd> port config all burst (value)
```

This is equivalent to the `--burst` command-line option.

port config - Threshold

Set thresholds for TX/RX queues:

```
testpmd> port config all (threshold) (value)
```

Where the threshold type can be:

- `txpt`: Set the prefetch threshold register of the TX rings, 0 <= value <= 255.
- `txht`: Set the host threshold register of the TX rings, 0 <= value <= 255.
- `txwt`: Set the write-back threshold register of the TX rings, 0 <= value <= 255.
- `rxpt`: Set the prefetch threshold register of the RX rings, 0 <= value <= 255.
- `rxht`: Set the host threshold register of the RX rings, 0 <= value <= 255.
- `rxwt`: Set the write-back threshold register of the RX rings, 0 <= value <= 255.
- `txfreet`: Set the transmit free threshold of the TX rings, 0 <= value <= txd.
- `rxfreet`: Set the transmit free threshold of the RX rings, 0 <= value <= rxd.
- `txrst`: Set the transmit RS bit threshold of TX rings, 0 <= value <= txd.

These threshold options are also available from the command-line.

port config - E-tag

Set the value of ether-type for E-tag:

```
testpmd> port config (port_id|all) l2-tunnel E-tag ether-type (value)
```

Enable/disable the E-tag support:

```
testpmd> port config (port_id|all) l2-tunnel E-tag (enable|disable)
```

7.4.6 Link Bonding Functions

The Link Bonding functions make it possible to dynamically create and manage link bonding devices from within testpmd interactive prompt.

create bonded device

Create a new bonding device:

```
testpmd> create bonded device (mode) (socket)
```

For example, to create a bonded device in mode 1 on socket 0:

```
testpmd> create bonded 1 0
created new bonded device (port X)
```

add bonding slave

Adds Ethernet device to a Link Bonding device:

```
testpmd> add bonding slave (slave id) (port id)
```

For example, to add Ethernet device (port 6) to a Link Bonding device (port 10):

```
testpmd> add bonding slave 6 10
```

remove bonding slave

Removes an Ethernet slave device from a Link Bonding device:

```
testpmd> remove bonding slave (slave id) (port id)
```

For example, to remove Ethernet slave device (port 6) to a Link Bonding device (port 10):

```
testpmd> remove bonding slave 6 10
```

set bonding mode

Set the Link Bonding mode of a Link Bonding device:

```
testpmd> set bonding mode (value) (port id)
```

For example, to set the bonding mode of a Link Bonding device (port 10) to broadcast (mode 3):

```
testpmd> set bonding mode 3 10
```

set bonding primary

Set an Ethernet slave device as the primary device on a Link Bonding device:

```
testpmd> set bonding primary (slave id) (port id)
```

For example, to set the Ethernet slave device (port 6) as the primary port of a Link Bonding device (port 10):

```
testpmd> set bonding primary 6 10
```

set bonding mac

Set the MAC address of a Link Bonding device:

```
testpmd> set bonding mac (port id) (mac)
```

For example, to set the MAC address of a Link Bonding device (port 10) to 00:00:00:00:00:01:

```
testpmd> set bonding mac 10 00:00:00:00:00:01
```

set bonding xmit_balance_policy

Set the transmission policy for a Link Bonding device when it is in Balance XOR mode:

```
testpmd> set bonding xmit_balance_policy (port_id) (l2|l23|l34)
```

For example, set a Link Bonding device (port 10) to use a balance policy of layer 3+4 (IP addresses & UDP ports):

```
testpmd> set bonding xmit_balance_policy 10 l34
```

set bonding mon_period

Set the link status monitoring polling period in milliseconds for a bonding device.

This adds support for PMD slave devices which do not support link status interrupts. When the mon_period is set to a value greater than 0 then all PMD's which do not support link status ISR will be queried every polling interval to check if their link status has changed:

```
testpmd> set bonding mon_period (port_id) (value)
```

For example, to set the link status monitoring polling period of bonded device (port 5) to 150ms:

```
testpmd> set bonding mon_period 5 150
```

show bonding config

Show the current configuration of a Link Bonding device:

```
testpmd> show bonding config (port id)
```

For example, to show the configuration a Link Bonding device (port 9) with 3 slave devices (1, 3, 4) in balance mode with a transmission policy of layer 2+3:

```
testpmd> show bonding config 9
     Bonding mode: 2
     Balance Xmit Policy: BALANCE_XMIT_POLICY_LAYER23
     Slaves (3): [1 3 4]
     Active Slaves (3): [1 3 4]
     Primary: [3]
```

7.4.7 Register Functions

The Register Functions can be used to read from and write to registers on the network card referenced by a port number. This is mainly useful for debugging purposes. Reference should be made to the appropriate datasheet for the network card for details on the register addresses and fields that can be accessed.

read reg

Display the value of a port register:

```
testpmd> read reg (port_id) (address)
```

For example, to examine the Flow Director control register (FDIRCTL, 0x0000EE000) on an Intel 82599 10 GbE Controller:

```
testpmd> read reg 0 0xEE00
port 0 PCI register at offset 0xEE00: 0x4A060029 (1241907241)
```

read regfield

Display a port register bit field:

```
testpmd> read regfield (port_id) (address) (bit_x) (bit_y)
```

For example, reading the lowest two bits from the register in the example above:

```
testpmd> read regfield 0 0xEE00 0 1
port 0 PCI register at offset 0xEE00: bits[0, 1]=0x1 (1)
```

read regbit

Display a single port register bit:

```
testpmd> read regbit (port_id) (address) (bit_x)
```

For example, reading the lowest bit from the register in the example above:

```
testpmd> read regbit 0 0xEE00 0
port 0 PCI register at offset 0xEE00: bit 0=1
```

write reg

Set the value of a port register:

```
testpmd> write reg (port_id) (address) (value)
```

For example, to clear a register:

```
testpmd> write reg 0 0xEE00 0x0
port 0 PCI register at offset 0xEE00: 0x00000000 (0)
```

write regfield

Set bit field of a port register:

```
testpmd> write regfield (port_id) (address) (bit_x) (bit_y) (value)
```

For example, writing to the register cleared in the example above:

```
testpmd> write regfield 0 0xEE00 0 1 2
port 0 PCI register at offset 0xEE00: 0x00000002 (2)
```

write regbit

Set single bit value of a port register:

```
testpmd> write regbit (port_id) (address) (bit_x) (value)
```

For example, to set the high bit in the register from the example above:

```
testpmd> write regbit 0 0xEE00 31 1
port 0 PCI register at offset 0xEE00: 0x8000000A (2147483658)
```

7.4.8 Filter Functions

This section details the available filter functions that are available.

Note these functions interface the deprecated legacy filtering framework, superseded by *rte_flow*. See *Flow rules management*.

ethertype_filter

Add or delete a L2 Ethertype filter, which identify packets by their L2 Ethertype mainly assign them to a receive queue:

```
ethertype_filter (port_id) (add|del) (mac_addr|mac_ignr) (mac_address) \
                 ethertype (ether_type) (drop|fwd) queue (queue_id)
```

The available information parameters are:

- `port_id`: The port which the Ethertype filter assigned on.
- `mac_addr`: Compare destination mac address.

- mac_ignr: Ignore destination mac address match.
- mac_address: Destination mac address to match.
- ether_type: The EtherType value want to match, for example 0x0806 for ARP packet. 0x0800 (IPv4) and 0x86DD (IPv6) are invalid.
- queue_id: The receive queue associated with this EtherType filter. It is meaningless when deleting or dropping.

Example, to add/remove an ethertype filter rule:

```
testpmd> ethertype_filter 0 add mac_ignr 00:11:22:33:44:55 \
                          ethertype 0x0806 fwd queue 3

testpmd> ethertype_filter 0 del mac_ignr 00:11:22:33:44:55 \
                          ethertype 0x0806 fwd queue 3
```

2tuple_filter

Add or delete a 2-tuple filter, which identifies packets by specific protocol and destination TCP/UDP port and forwards packets into one of the receive queues:

```
2tuple_filter (port_id) (add|del) dst_port (dst_port_value) \
              protocol (protocol_value) mask (mask_value) \
              tcp_flags (tcp_flags_value) priority (prio_value) \
              queue (queue_id)
```

The available information parameters are:

- port_id: The port which the 2-tuple filter assigned on.
- dst_port_value: Destination port in L4.
- protocol_value: IP L4 protocol.
- mask_value: Participates in the match or not by bit for field above, 1b means participate.
- tcp_flags_value: TCP control bits. The non-zero value is invalid, when the pro_value is not set to 0x06 (TCP).
- prio_value: Priority of this filter.
- queue_id: The receive queue associated with this 2-tuple filter.

Example, to add/remove an 2tuple filter rule:

```
testpmd> 2tuple_filter 0 add dst_port 32 protocol 0x06 mask 0x03 \
                       tcp_flags 0x02 priority 3 queue 3

testpmd> 2tuple_filter 0 del dst_port 32 protocol 0x06 mask 0x03 \
                       tcp_flags 0x02 priority 3 queue 3
```

5tuple_filter

Add or delete a 5-tuple filter, which consists of a 5-tuple (protocol, source and destination IP addresses, source and destination TCP/UDP/SCTP port) and routes packets into one of the receive queues:

```
5tuple_filter (port_id) (add|del) dst_ip (dst_address) src_ip \
              (src_address) dst_port (dst_port_value) \
              src_port (src_port_value) protocol (protocol_value) \
              mask (mask_value) tcp_flags (tcp_flags_value) \
              priority (prio_value) queue (queue_id)
```

The available information parameters are:

- `port_id`: The port which the 5-tuple filter assigned on.
- `dst_address`: Destination IP address.
- `src_address`: Source IP address.
- `dst_port_value`: TCP/UDP destination port.
- `src_port_value`: TCP/UDP source port.
- `protocol_value`: L4 protocol.
- `mask_value`: Participates in the match or not by bit for field above, 1b means participate
- `tcp_flags_value`: TCP control bits. The non-zero value is invalid, when the protocol_value is not set to 0x06 (TCP).
- `prio_value`: The priority of this filter.
- `queue_id`: The receive queue associated with this 5-tuple filter.

Example, to add/remove an 5tuple filter rule:

```
testpmd> 5tuple_filter 0 add dst_ip 2.2.2.5 src_ip 2.2.2.4 \
         dst_port 64 src_port 32 protocol 0x06 mask 0x1F \
         flags 0x0 priority 3 queue 3

testpmd> 5tuple_filter 0 del dst_ip 2.2.2.5 src_ip 2.2.2.4 \
         dst_port 64 src_port 32 protocol 0x06 mask 0x1F \
         flags 0x0 priority 3 queue 3
```

syn_filter

Using the SYN filter, TCP packets whose *SYN* flag is set can be forwarded to a separate queue:

```
syn_filter (port_id) (add|del) priority (high|low) queue (queue_id)
```

The available information parameters are:

- `port_id`: The port which the SYN filter assigned on.
- `high`: This SYN filter has higher priority than other filters.
- `low`: This SYN filter has lower priority than other filters.
- `queue_id`: The receive queue associated with this SYN filter

Example:

```
testpmd> syn_filter 0 add priority high queue 3
```

flex_filter

With flex filter, packets can be recognized by any arbitrary pattern within the first 128 bytes of the packet and routed into one of the receive queues:

```
flex_filter (port_id) (add|del) len (len_value) bytes (bytes_value) \
            mask (mask_value) priority (prio_value) queue (queue_id)
```

The available information parameters are:

- `port_id`: The port which the Flex filter is assigned on.
- `len_value`: Filter length in bytes, no greater than 128.
- `bytes_value`: A string in hexadecimal, means the value the flex filter needs to match.
- `mask_value`: A string in hexadecimal, bit 1 means corresponding byte participates in the match.
- `prio_value`: The priority of this filter.
- `queue_id`: The receive queue associated with this Flex filter.

Example:

```
testpmd> flex_filter 0 add len 16 bytes 0x00000000000000000000000008060000 \
                       mask 000C priority 3 queue 3

testpmd> flex_filter 0 del len 16 bytes 0x00000000000000000000000008060000 \
                       mask 000C priority 3 queue 3
```

flow_director_filter

The Flow Director works in receive mode to identify specific flows or sets of flows and route them to specific queues.

Four types of filtering are supported which are referred to as Perfect Match, Signature, Perfect-mac-vlan and Perfect-tunnel filters, the match mode is set by the `--pkt-filter-mode` command-line parameter:

- Perfect match filters. The hardware checks a match between the masked fields of the received packets and the programmed filters. The masked fields are for IP flow.
- Signature filters. The hardware checks a match between a hash-based signature of the masked fields of the received packet.
- Perfect-mac-vlan match filters. The hardware checks a match between the masked fields of the received packets and the programmed filters. The masked fields are for MAC VLAN flow.
- Perfect-tunnel match filters. The hardware checks a match between the masked fields of the received packets and the programmed filters. The masked fields are for tunnel flow.

The Flow Director filters can match the different fields for different type of packet: flow type, specific input set per flow type and the flexible payload.

The Flow Director can also mask out parts of all of these fields so that filters are only applied to certain fields or parts of the fields.

Different NICs may have different capabilities, command show port fdir (port_id) can be used to acquire the information.

Commands to add flow director filters of different flow types:

```
flow_director_filter (port_id) mode IP (add|del|update) \
                     flow (ipv4-other|ipv4-frag|ipv6-other|ipv6-frag) \
                     src (src_ip_address) dst (dst_ip_address) \
                     tos (tos_value) proto (proto_value) ttl (ttl_value) \
                     vlan (vlan_value) flexbytes (flexbytes_value) \
                     (drop|fwd) pf|vf(vf_id) queue (queue_id) \
                     fd_id (fd_id_value)

flow_director_filter (port_id) mode IP (add|del|update) \
                     flow (ipv4-tcp|ipv4-udp|ipv6-tcp|ipv6-udp) \
                     src (src_ip_address) (src_port) \
                     dst (dst_ip_address) (dst_port) \
                     tos (tos_value) ttl (ttl_value) \
                     vlan (vlan_value) flexbytes (flexbytes_value) \
                     (drop|fwd) queue pf|vf(vf_id) (queue_id) \
                     fd_id (fd_id_value)

flow_director_filter (port_id) mode IP (add|del|update) \
                     flow (ipv4-sctp|ipv6-sctp) \
                     src (src_ip_address) (src_port) \
                     dst (dst_ip_address) (dst_port) \
                     tos (tos_value) ttl (ttl_value) \
                     tag (verification_tag) vlan (vlan_value) \
                     flexbytes (flexbytes_value) (drop|fwd) \
                     pf|vf(vf_id) queue (queue_id) fd_id (fd_id_value)

flow_director_filter (port_id) mode IP (add|del|update) flow l2_payload \
                     ether (ethertype) flexbytes (flexbytes_value) \
                     (drop|fwd) pf|vf(vf_id) queue (queue_id)
                     fd_id (fd_id_value)

flow_director_filter (port_id) mode MAC-VLAN (add|del|update) \
                     mac (mac_address) vlan (vlan_value) \
                     flexbytes (flexbytes_value) (drop|fwd) \
                     queue (queue_id) fd_id (fd_id_value)

flow_director_filter (port_id) mode Tunnel (add|del|update) \
                     mac (mac_address) vlan (vlan_value) \
                     tunnel (NVGRE|VxLAN) tunnel-id (tunnel_id_value) \
                     flexbytes (flexbytes_value) (drop|fwd) \
                     queue (queue_id) fd_id (fd_id_value)
```

For example, to add an ipv4-udp flow type filter:

```
testpmd> flow_director_filter 0 mode IP add flow ipv4-udp src 2.2.2.3 32 \
         dst 2.2.2.5 33 tos 2 ttl 40 vlan 0x1 flexbytes (0x88,0x48) \
         fwd pf queue 1 fd_id 1
```

For example, add an ipv4-other flow type filter:

```
testpmd> flow_director_filter 0 mode IP add flow ipv4-other src 2.2.2.3 \
          dst 2.2.2.5 tos 2 proto 20 ttl 40 vlan 0x1 \
          flexbytes (0x88,0x48) fwd pf queue 1 fd_id 1
```

flush_flow_director

Flush all flow director filters on a device:

```
testpmd> flush_flow_director (port_id)
```

Example, to flush all flow director filter on port 0:

```
testpmd> flush_flow_director 0
```

flow_director_mask

Set flow director's input masks:

```
flow_director_mask (port_id) mode IP vlan (vlan_value) \
                   src_mask (ipv4_src) (ipv6_src) (src_port) \
                   dst_mask (ipv4_dst) (ipv6_dst) (dst_port)

flow_director_mask (port_id) mode MAC-VLAN vlan (vlan_value)

flow_director_mask (port_id) mode Tunnel vlan (vlan_value) \
                   mac (mac_value) tunnel-type (tunnel_type_value) \
                   tunnel-id (tunnel_id_value)
```

Example, to set flow director mask on port 0:

```
testpmd> flow_director_mask 0 mode IP vlan 0xefff \
         src_mask 255.255.255.255 \
             FFFF:FFFF:FFFF:FFFF:FFFF:FFFF:FFFF:FFFF 0xFFFF \
         dst_mask 255.255.255.255 \
             FFFF:FFFF:FFFF:FFFF:FFFF:FFFF:FFFF:FFFF 0xFFFF
```

flow_director_flex_mask

set masks of flow director's flexible payload based on certain flow type:

```
testpmd> flow_director_flex_mask (port_id) \
         flow (none|ipv4-other|ipv4-frag|ipv4-tcp|ipv4-udp|ipv4-sctp| \
               ipv6-other|ipv6-frag|ipv6-tcp|ipv6-udp|ipv6-sctp| \
               l2_payload|all) (mask)
```

Example, to set flow director's flex mask for all flow type on port 0:

```
testpmd> flow_director_flex_mask 0 flow all \
         (0xff,0xff,0,0,0,0,0,0,0,0,0,0,0,0,0,0)
```

flow_director_flex_payload

Configure flexible payload selection:

```
flow_director_flex_payload (port_id) (raw|l2|l3|l4) (config)
```

For example, to select the first 16 bytes from the offset 4 (bytes) of packet's payload as flexible payload:

```
testpmd> flow_director_flex_payload 0 l4 \
         (4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19)
```

get_sym_hash_ena_per_port

Get symmetric hash enable configuration per port:

```
get_sym_hash_ena_per_port (port_id)
```

For example, to get symmetric hash enable configuration of port 1:

```
testpmd> get_sym_hash_ena_per_port 1
```

set_sym_hash_ena_per_port

Set symmetric hash enable configuration per port to enable or disable:

```
set_sym_hash_ena_per_port (port_id) (enable|disable)
```

For example, to set symmetric hash enable configuration of port 1 to enable:

```
testpmd> set_sym_hash_ena_per_port 1 enable
```

get_hash_global_config

Get the global configurations of hash filters:

```
get_hash_global_config (port_id)
```

For example, to get the global configurations of hash filters of port 1:

```
testpmd> get_hash_global_config 1
```

set_hash_global_config

Set the global configurations of hash filters:

```
set_hash_global_config (port_id) (toeplitz|simple_xor|default) \
(ipv4|ipv4-frag|ipv4-tcp|ipv4-udp|ipv4-sctp|ipv4-other|ipv6|ipv6-frag| \
ipv6-tcp|ipv6-udp|ipv6-sctp|ipv6-other|l2_payload) \
(enable|disable)
```

For example, to enable simple_xor for flow type of ipv6 on port 2:

```
testpmd> set_hash_global_config 2 simple_xor ipv6 enable
```

set_hash_input_set

Set the input set for hash:

```
set_hash_input_set (port_id) (ipv4-frag|ipv4-tcp|ipv4-udp|ipv4-sctp| \
ipv4-other|ipv6-frag|ipv6-tcp|ipv6-udp|ipv6-sctp|ipv6-other| \
l2_payload) (ovlan|ivlan|src-ipv4|dst-ipv4|src-ipv6|dst-ipv6|ipv4-tos| \
ipv4-proto|ipv6-tc|ipv6-next-header|udp-src-port|udp-dst-port| \
tcp-src-port|tcp-dst-port|sctp-src-port|sctp-dst-port|sctp-veri-tag| \
```

```
udp-key|gre-key|fld-1st|fld-2nd|fld-3rd|fld-4th|fld-5th|fld-6th|fld-7th| \
fld-8th|none) (select|add)
```

For example, to add source IP to hash input set for flow type of ipv4-udp on port 0:

```
testpmd> set_hash_input_set 0 ipv4-udp src-ipv4 add
```

set_fdir_input_set

The Flow Director filters can match the different fields for different type of packet, i.e. specific input set on per flow type and the flexible payload. This command can be used to change input set for each flow type.

Set the input set for flow director:

```
set_fdir_input_set (port_id) (ipv4-frag|ipv4-tcp|ipv4-udp|ipv4-sctp| \
ipv4-other|ipv6|ipv6-frag|ipv6-tcp|ipv6-udp|ipv6-sctp|ipv6-other| \
l2_payload) (ivlan|ethertype|src-ipv4|dst-ipv4|src-ipv6|dst-ipv6|ipv4-tos| \
ipv4-proto|ipv4-ttl|ipv6-tc|ipv6-next-header|ipv6-hop-limits| \
tudp-src-port|udp-dst-port|cp-src-port|tcp-dst-port|sctp-src-port| \
sctp-dst-port|sctp-veri-tag|none) (select|add)
```

For example to add source IP to FD input set for flow type of ipv4-udp on port 0:

```
testpmd> set_fdir_input_set 0 ipv4-udp src-ipv4 add
```

global_config

Set different GRE key length for input set:

```
global_config (port_id) gre-key-len (number in bytes)
```

For example to set GRE key length for input set to 4 bytes on port 0:

```
testpmd> global_config 0 gre-key-len 4
```

7.4.9 Flow rules management

Control of the generic flow API (*rte_flow*) is fully exposed through the `flow` command (validation, creation, destruction and queries).

Considering *rte_flow* overlaps with all *Filter Functions*, using both features simultaneously may cause undefined side-effects and is therefore not recommended.

`flow` syntax

Because the `flow` command uses dynamic tokens to handle the large number of possible flow rules combinations, its behavior differs slightly from other commands, in particular:

- Pressing *?* or the *<tab>* key displays contextual help for the current token, not that of the entire command.
- Optional and repeated parameters are supported (provided they are listed in the contextual help).

The first parameter stands for the operation mode. Possible operations and their general syntax are described below. They are covered in detail in the following sections.

- Check whether a flow rule can be created:

```
flow validate {port_id}
    [group {group_id}] [priority {level}] [ingress] [egress]
    pattern {item} [/ {item} [...]] / end
    actions {action} [/ {action} [...]] / end
```

- Create a flow rule:

```
flow create {port_id}
    [group {group_id}] [priority {level}] [ingress] [egress]
    pattern {item} [/ {item} [...]] / end
    actions {action} [/ {action} [...]] / end
```

- Destroy specific flow rules:

```
flow destroy {port_id} rule {rule_id} [...]
```

- Destroy all flow rules:

```
flow flush {port_id}
```

- Query an existing flow rule:

```
flow query {port_id} {rule_id} {action}
```

- List existing flow rules sorted by priority, filtered by group identifiers:

```
flow list {port_id} [group {group_id}] [...]
```

Validating flow rules

`flow validate` reports whether a flow rule would be accepted by the underlying device in its current state but stops short of creating it. It is bound to `rte_flow_validate()`:

```
flow validate {port_id}
   [group {group_id}] [priority {level}] [ingress] [egress]
   pattern {item} [/ {item} [...]] / end
   actions {action} [/ {action} [...]] / end
```

If successful, it will show:

```
Flow rule validated
```

Otherwise it will show an error message of the form:

```
Caught error type [...] ([...]): [...]
```

This command uses the same parameters as `flow create`, their format is described in *Creating flow rules*.

Check whether redirecting any Ethernet packet received on port 0 to RX queue index 6 is supported:

```
testpmd> flow validate 0 ingress pattern eth / end
   actions queue index 6 / end
Flow rule validated
testpmd>
```

Port 0 does not support TCPv6 rules:

```
testpmd> flow validate 0 ingress pattern eth / ipv6 / tcp / end
   actions drop / end
Caught error type 9 (specific pattern item): Invalid argument
testpmd>
```

Creating flow rules

`flow create` validates and creates the specified flow rule. It is bound to `rte_flow_create()`:

```
flow create {port_id}
   [group {group_id}] [priority {level}] [ingress] [egress]
   pattern {item} [/ {item} [...]] / end
   actions {action} [/ {action} [...]] / end
```

If successful, it will return a flow rule ID usable with other commands:

```
Flow rule #[...] created
```

Otherwise it will show an error message of the form:

```
Caught error type [...] ([...]): [...]
```

Parameters describe in the following order:

- Attributes (*group*, *priority*, *ingress*, *egress* tokens).
- A matching pattern, starting with the *pattern* token and terminated by an *end* pattern item.
- Actions, starting with the *actions* token and terminated by an *end* action.

These translate directly to *rte_flow* objects provided as-is to the underlying functions.

The shortest valid definition only comprises mandatory tokens:

```
testpmd> flow create 0 pattern end actions end
```

Note that PMDs may refuse rules that essentially do nothing such as this one.

All unspecified object values are automatically initialized to 0.

Attributes

These tokens affect flow rule attributes (`struct rte_flow_attr`) and are specified before the `pattern` token.

- `group {group id}`: priority group.
- `priority {level}`: priority level within group.
- `ingress`: rule applies to ingress traffic.
- `egress`: rule applies to egress traffic.

Each instance of an attribute specified several times overrides the previous value as shown below (group 4 is used):

```
testpmd> flow create 0 group 42 group 24 group 4 [...]
```

Note that once enabled, `ingress` and `egress` cannot be disabled.

While not specifying a direction is an error, some rules may allow both simultaneously.

Most rules affect RX therefore contain the `ingress` token:

```
testpmd> flow create 0 ingress pattern [...]
```

Matching pattern

A matching pattern starts after the `pattern` token. It is made of pattern items and is terminated by a mandatory `end` item.

Items are named after their type (*RTE_FLOW_ITEM_TYPE_* from `enum rte_flow_item_type`).

The `/` token is used as a separator between pattern items as shown below:

```
testpmd> flow create 0 ingress pattern eth / ipv4 / udp / end [...]
```

Note that protocol items like these must be stacked from lowest to highest layer to make sense. For instance, the following rule is either invalid or unlikely to match any packet:

```
testpmd> flow create 0 ingress pattern eth / udp / ipv4 / end [...]
```

More information on these restrictions can be found in the *rte_flow* documentation.

Several items support additional specification structures, for example `ipv4` allows specifying source and destination addresses as follows:

```
testpmd> flow create 0 ingress pattern eth / ipv4 src is 10.1.1.1
   dst is 10.2.0.0 / end [...]
```

This rule matches all IPv4 traffic with the specified properties.

In this example, `src` and `dst` are field names of the underlying `struct rte_flow_item_ipv4` object. All item properties can be specified in a similar fashion.

The `is` token means that the subsequent value must be matched exactly, and assigns `spec` and `mask` fields in `struct rte_flow_item` accordingly. Possible assignment tokens are:

- `is`: match value perfectly (with full bit-mask).
- `spec`: match value according to configured bit-mask.
- `last`: specify upper bound to establish a range.
- `mask`: specify bit-mask with relevant bits set to one.
- `prefix`: generate bit-mask from a prefix length.

These yield identical results:

```
ipv4 src is 10.1.1.1
```

```
ipv4 src spec 10.1.1.1 src mask 255.255.255.255
```

```
ipv4 src spec 10.1.1.1 src prefix 32
```

```
ipv4 src is 10.1.1.1 src last 10.1.1.1 # range with a single value
```

```
ipv4 src is 10.1.1.1 src last 0 # 0 disables range
```

Inclusive ranges can be defined with `last`:

```
ipv4 src is 10.1.1.1 src last 10.2.3.4 # 10.1.1.1 to 10.2.3.4
```

Note that `mask` affects both `spec` and `last`:

```
ipv4 src is 10.1.1.1 src last 10.2.3.4 src mask 255.255.0.0
   # matches 10.1.0.0 to 10.2.255.255
```

Properties can be modified multiple times:

```
ipv4 src is 10.1.1.1 src is 10.1.2.3 src is 10.2.3.4 # matches 10.2.3.4
```

```
ipv4 src is 10.1.1.1 src prefix 24 src prefix 16 # matches 10.1.0.0/16
```

Pattern items

This section lists supported pattern items and their attributes, if any.

- `end`: end list of pattern items.
- `void`: no-op pattern item.
- `invert`: perform actions when pattern does not match.
- `any`: match any protocol for the current layer.
  - `num {unsigned}`: number of layers covered.
- `pf`: match packets addressed to the physical function.
- `vf`: match packets addressed to a virtual function ID.
  - `id {unsigned}`: destination VF ID.
- `port`: device-specific physical port index to use.
  - `index {unsigned}`: physical port index.
- `raw`: match an arbitrary byte string.
  - `relative {boolean}`: look for pattern after the previous item.
  - `search {boolean}`: search pattern from offset (see also limit).
  - `offset {integer}`: absolute or relative offset for pattern.
  - `limit {unsigned}`: search area limit for start of pattern.
  - `pattern {string}`: byte string to look for.
- `eth`: match Ethernet header.
  - `dst {MAC-48}`: destination MAC.
  - `src {MAC-48}`: source MAC.
  - `type {unsigned}`: EtherType.
- `vlan`: match 802.1Q/ad VLAN tag.
  - `tpid {unsigned}`: tag protocol identifier.

  - `tci {unsigned}`: tag control information.
  - `pcp {unsigned}`: priority code point.
  - `dei {unsigned}`: drop eligible indicator.
  - `vid {unsigned}`: VLAN identifier.
- `ipv4`: match IPv4 header.
  - `tos {unsigned}`: type of service.
  - `ttl {unsigned}`: time to live.
  - `proto {unsigned}`: next protocol ID.
  - `src {ipv4 address}`: source address.
  - `dst {ipv4 address}`: destination address.
- `ipv6`: match IPv6 header.
  - `tc {unsigned}`: traffic class.
  - `flow {unsigned}`: flow label.
  - `proto {unsigned}`: protocol (next header).
  - `hop {unsigned}`: hop limit.
  - `src {ipv6 address}`: source address.
  - `dst {ipv6 address}`: destination address.
- `icmp`: match ICMP header.
  - `type {unsigned}`: ICMP packet type.
  - `code {unsigned}`: ICMP packet code.
- `udp`: match UDP header.
  - `src {unsigned}`: UDP source port.
  - `dst {unsigned}`: UDP destination port.
- `tcp`: match TCP header.
  - `src {unsigned}`: TCP source port.
  - `dst {unsigned}`: TCP destination port.
- `sctp`: match SCTP header.
  - `src {unsigned}`: SCTP source port.
  - `dst {unsigned}`: SCTP destination port.
  - `tag {unsigned}`: validation tag.
  - `cksum {unsigned}`: checksum.
- `vxlan`: match VXLAN header.
  - `vni {unsigned}`: VXLAN identifier.
- `mpls`: match MPLS header.
  - `label {unsigned}`: MPLS label.
- `gre`: match GRE header.

- `protocol {unsigned}`: protocol type.

Actions list

A list of actions starts after the `actions` token in the same fashion as *Matching pattern*; actions are separated by `/` tokens and the list is terminated by a mandatory `end` action.

Actions are named after their type (*RTE_FLOW_ACTION_TYPE_* from `enum rte_flow_action_type`).

Dropping all incoming UDPv4 packets can be expressed as follows:

```
testpmd> flow create 0 ingress pattern eth / ipv4 / udp / end
   actions drop / end
```

Several actions have configurable properties which must be specified when there is no valid default value. For example, `queue` requires a target queue index.

This rule redirects incoming UDPv4 traffic to queue index 6:

```
testpmd> flow create 0 ingress pattern eth / ipv4 / udp / end
   actions queue index 6 / end
```

While this one could be rejected by PMDs (unspecified queue index):

```
testpmd> flow create 0 ingress pattern eth / ipv4 / udp / end
   actions queue / end
```

As defined by *rte_flow*, the list is not ordered, all actions of a given rule are performed simultaneously. These are equivalent:

```
queue index 6 / void / mark id 42 / end
```

```
void / mark id 42 / queue index 6 / end
```

All actions in a list should have different types, otherwise only the last action of a given type is taken into account:

```
queue index 4 / queue index 5 / queue index 6 / end # will use queue 6
```

```
drop / drop / drop / end # drop is performed only once
```

```
mark id 42 / queue index 3 / mark id 24 / end # mark will be 24
```

Considering they are performed simultaneously, opposite and overlapping actions can sometimes be combined when the end result is unambiguous:

```
drop / queue index 6 / end # drop has no effect
```

```
drop / dup index 6 / end # same as above
```

```
queue index 6 / rss queues 6 7 8 / end # queue has no effect
```

```
drop / passthru / end # drop has no effect
```

Note that PMDs may still refuse such combinations.

Actions

This section lists supported actions and their attributes, if any.

- `end`: end list of actions.
- `void`: no-op action.
- `passthru`: let subsequent rule process matched packets.
- `mark`: attach 32 bit value to packets.
  - `id {unsigned}`: 32 bit value to return with packets.
- `flag`: flag packets.
- `queue`: assign packets to a given queue index.
  - `index {unsigned}`: queue index to use.
- `drop`: drop packets (note: passthru has priority).
- `count`: enable counters for this rule.
- `dup`: duplicate packets to a given queue index.
  - `index {unsigned}`: queue index to duplicate packets to.
- `rss`: spread packets among several queues.
  - `queues [{unsigned} [...]] end`: queue indices to use.
- `pf`: redirect packets to physical device function.
- `vf`: redirect packets to virtual device function.
  - `original {boolean}`: use original VF ID if possible.
  - `id {unsigned}`: VF ID to redirect packets to.

Destroying flow rules

`flow destroy` destroys one or more rules from their rule ID (as returned by `flow create`), this command calls `rte_flow_destroy()` as many times as necessary:

```
flow destroy {port_id} rule {rule_id} [...]
```

If successful, it will show:

```
Flow rule #[...] destroyed
```

It does not report anything for rule IDs that do not exist. The usual error message is shown when a rule cannot be destroyed:

```
Caught error type [...] ([...]): [...]
```

`flow flush` destroys all rules on a device and does not take extra arguments. It is bound to `rte_flow_flush()`:

```
flow flush {port_id}
```

Any errors are reported as above.

Creating several rules and destroying them:

```
testpmd> flow create 0 ingress pattern eth / ipv6 / end
   actions queue index 2 / end
Flow rule #0 created
testpmd> flow create 0 ingress pattern eth / ipv4 / end
   actions queue index 3 / end
Flow rule #1 created
testpmd> flow destroy 0 rule 0 rule 1
Flow rule #1 destroyed
Flow rule #0 destroyed
testpmd>
```

The same result can be achieved using `flow flush`:

```
testpmd> flow create 0 ingress pattern eth / ipv6 / end
   actions queue index 2 / end
Flow rule #0 created
testpmd> flow create 0 ingress pattern eth / ipv4 / end
   actions queue index 3 / end
Flow rule #1 created
testpmd> flow flush 0
testpmd>
```

Non-existent rule IDs are ignored:

```
testpmd> flow create 0 ingress pattern eth / ipv6 / end
   actions queue index 2 / end
Flow rule #0 created
testpmd> flow create 0 ingress pattern eth / ipv4 / end
   actions queue index 3 / end
Flow rule #1 created
testpmd> flow destroy 0 rule 42 rule 10 rule 2
testpmd>
testpmd> flow destroy 0 rule 0
Flow rule #0 destroyed
testpmd>
```

Querying flow rules

`flow query` queries a specific action of a flow rule having that ability. Such actions collect information that can be reported using this command. It is bound to `rte_flow_query()`:

```
flow query {port_id} {rule_id} {action}
```

If successful, it will display either the retrieved data for known actions or the following message:

```
Cannot display result for action type [...] ([...])
```

Otherwise, it will complain either that the rule does not exist or that some error occurred:

```
Flow rule #[...] not found
```

```
Caught error type [...] ([...]): [...]
```

Currently only the `count` action is supported. This action reports the number of packets that hit the flow rule and the total number of bytes. Its output has the following format:

```
count:
 hits_set: [...] # whether "hits" contains a valid value
 bytes_set: [...] # whether "bytes" contains a valid value
 hits: [...] # number of packets
 bytes: [...] # number of bytes
```

Querying counters for TCPv6 packets redirected to queue 6:

```
testpmd> flow create 0 ingress pattern eth / ipv6 / tcp / end
   actions queue index 6 / count / end
Flow rule #4 created
testpmd> flow query 0 4 count
count:
 hits_set: 1
 bytes_set: 0
 hits: 386446
 bytes: 0
testpmd>
```

Listing flow rules

`flow list` lists existing flow rules sorted by priority and optionally filtered by group identifiers:

```
flow list {port_id} [group {group_id}] [...]
```

This command only fails with the following message if the device does not exist:

```
Invalid port [...]
```

Output consists of a header line followed by a short description of each flow rule, one per line. There is no output at all when no flow rules are configured on the device:

```
ID      Group   Prio    Attr    Rule
[...]   [...]   [...]   [...]   [...]
```

`Attr` column flags:

- `i` for `ingress`.
- `e` for `egress`.

Creating several flow rules and listing them:

```
testpmd> flow create 0 ingress pattern eth / ipv4 / end
   actions queue index 6 / end
Flow rule #0 created
testpmd> flow create 0 ingress pattern eth / ipv6 / end
   actions queue index 2 / end
Flow rule #1 created
testpmd> flow create 0 priority 5 ingress pattern eth / ipv4 / udp / end
   actions rss queues 6 7 8 end / end
Flow rule #2 created
testpmd> flow list 0
ID      Group   Prio    Attr    Rule
0       0       0       i-      ETH IPV4 => QUEUE
1       0       0       i-      ETH IPV6 => QUEUE
```

```
2       0       5       i-      ETH IPV4 UDP => RSS
testpmd>
```

Rules are sorted by priority (i.e. group ID first, then priority level):

```
testpmd> flow list 1
ID      Group   Prio    Attr    Rule
0       0       0       i-      ETH => COUNT
6       0       500     i-      ETH IPV6 TCP => DROP COUNT
5       0       1000    i-      ETH IPV6 ICMP => QUEUE
1       24      0       i-      ETH IPV4 UDP => QUEUE
4       24      10      i-      ETH IPV4 TCP => DROP
3       24      20      i-      ETH IPV4 => DROP
2       24      42      i-      ETH IPV4 UDP => QUEUE
7       63      0       i-      ETH IPV6 UDP VXLAN => MARK QUEUE
testpmd>
```

Output can be limited to specific groups:

```
testpmd> flow list 1 group 0 group 63
ID      Group   Prio    Attr    Rule
0       0       0       i-      ETH => COUNT
6       0       500     i-      ETH IPV6 TCP => DROP COUNT
5       0       1000    i-      ETH IPV6 ICMP => QUEUE
7       63      0       i-      ETH IPV6 UDP VXLAN => MARK QUEUE
testpmd>
```

CHAPTER 8

Network Interface Controller Drivers

8.1 Overview of Networking Drivers

The networking drivers may be classified in two categories:

- physical for real devices
- virtual for emulated devices

Some physical devices may be shaped through a virtual layer as for SR-IOV. The interface seen in the virtual environment is a VF (Virtual Function).

The ethdev layer exposes an API to use the networking functions of these devices. The bottom half part of ethdev is implemented by the drivers. Thus some features may not be implemented.

There are more differences between drivers regarding some internal properties, portability or even documentation availability. Most of these differences are summarized below.

Note: Features marked with "P" are partially supported. Refer to the appropriate NIC guide in the following sections for details.

8.2 AVP Poll Mode Driver

The Accelerated Virtual Port (AVP) device is a shared memory based device only available on virtualization platforms from Wind River Systems. The Wind River Systems virtualization platform currently uses QEMU/KVM as its hypervisor and as such provides support for all of the QEMU supported virtual and/or emulated devices (e.g., virtio, e1000, etc.). The platform offers the virtio device type as the default device when launching a virtual machine or creating a virtual machine port. The AVP device is a specialized device available to customers that require increased throughput and decreased latency to meet the demands of their performance focused applications.

The AVP driver binds to any AVP PCI devices that have been exported by the Wind River Systems QEMU/KVM hypervisor. As a user of the DPDK driver API it supports a subset of the full Ethernet device API to enable the application to use the standard device configuration functions and packet receive/transmit functions.

These devices enable optimized packet throughput by bypassing QEMU and delivering packets directly to the virtual switch via a shared memory mechanism. This provides DPDK applications running in virtual machines with significantly improved throughput and latency over other device types.

The AVP device implementation is integrated with the QEMU/KVM live-migration mechanism to allow applications to seamlessly migrate from one hypervisor node to another with minimal packet loss.

8.2.1 Features and Limitations of the AVP PMD

The AVP PMD driver provides the following functionality.

- Receive and transmit of both simple and chained mbuf packets,
- Chained mbufs may include up to 5 chained segments,
- Up to 8 receive and transmit queues per device,
- Only a single MAC address is supported,
- The MAC address cannot be modified,
- The maximum receive packet length is 9238 bytes,
- VLAN header stripping and inserting,
- Promiscuous mode
- VM live-migration
- PCI hotplug insertion and removal

8.2.2 Prerequisites

The following prerequisites apply:

- A virtual machine running in a Wind River Systems virtualization environment and configured with at least one neutron port defined with a vif-model set to “avp”.

8.2.3 Launching a VM with an AVP type network attachment

The following example will launch a VM with three network attachments. The first attachment will have a default vif-model of “virtio”. The next two network attachments will have a vif-model of “avp” and may be used with a DPDK application which is built to include the AVP PMD driver.

```
nova boot --flavor small --image my-image \
   --nic net-id=${NETWORK1_UUID} \
   --nic net-id=${NETWORK2_UUID},vif-model=avp \
   --nic net-id=${NETWORK3_UUID},vif-model=avp \
   --security-group default my-instance1
```

8.3 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.

8.3.1 Supported Features

BNX2X PMD has support for:

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

8.3.2 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
- Rx TX scatter gather

8.3.3 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.

8.3.4 Supported QLogic NICs

- 578xx

8.3.5 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.

8.3.6 Pre-Installation Configuration

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.

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/gcc49␣
↪CC=gcc49
```

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/gcc49␣
↪CC=gcc49
```

8.3.7 Linux

Linux Installation

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:

   ```
   ./usertools/dpdk-devbind.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/*

   ./usertools/dpdk-devbind.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 -l 0-3 -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>
```

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.

8.4 BNXT Poll Mode Driver

The bnxt poll mode library (**librte_pmd_bnxt**) implements support for:

- **Broadcom NetXtreme-C®/NetXtreme-E® BCM5730X and BCM5740X family of Ethernet Network Controllers**

  These adapters support Standards compliant 10/25/50Gbps 30MPPS full-duplex throughput.

  Information about the NetXtreme family of adapters can be found in the NetXtreme® Brand section of the Broadcom website.

- **Broadcom StrataGX® BCM5871X Series of Communucations Processors**

  These ARM based processors target a broad range of networking applications including virtual CPE (vCPE) and NFV appliances, 10G service routers and gateways, control plane processing for Ethernet switches and network attached storage (NAS).

  Information about the StrataGX family of adapters can be found in the StrataGX® BCM5871X Series section of the Broadcom website.

8.4.1 Limitations

With the current driver, allocated mbufs must be large enough to hold the entire received frame. If the mbufs are not large enough, the packets will be dropped. This is most limiting when jumbo frames are used.

8.5 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.

8.5.1 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

8.5.2 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.

8.5.3 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

8.5.4 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.

8.5.5 Pre-Installation Configuration

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.

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
```

8.5.6 Linux

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
```

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:

```
./usertools/dpdk-devbind.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/*

./usertools/dpdk-devbind.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 -l 0-3 -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.

8.5.7 FreeBSD

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
```

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 -l 0-3 -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.

8.5.8 Sample Application Notes

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
```

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
```

8.6 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.

8.6.1 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

8.6.2 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*.

8.6.3 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.

8.6.4 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.

8.7 ENA Poll Mode Driver

The ENA PMD is a DPDK poll-mode driver for the Amazon Elastic Network Adapter (ENA) family.

8.7.1 Overview

The ENA driver exposes a lightweight management interface with a minimal set of memory mapped registers and an extendable command set through an Admin Queue.

The driver supports a wide range of ENA adapters, is link-speed independent (i.e., the same driver is used for 10GbE, 25GbE, 40GbE, etc.), and it negotiates and supports an extendable feature set.

ENA adapters allow high speed and low overhead Ethernet traffic processing by providing a dedicated Tx/Rx queue pair per CPU core.

The ENA driver supports industry standard TCP/IP offload features such as checksum offload and TCP transmit segmentation offload (TSO).

Receive-side scaling (RSS) is supported for multi-core scaling.

Some of the ENA devices support a working mode called Low-latency Queue (LLQ), which saves several more microseconds.

8.7.2 Management Interface

ENA management interface is exposed by means of:

- Device Registers
- Admin Queue (AQ) and Admin Completion Queue (ACQ)

ENA device memory-mapped PCIe space for registers (MMIO registers) are accessed only during driver initialization and are not involved in further normal device operation.

AQ is used for submitting management commands, and the results/responses are reported asynchronously through ACQ.

ENA introduces a very small set of management commands with room for vendor-specific extensions. Most of the management operations are framed in a generic Get/Set feature command.

The following admin queue commands are supported:

- Create I/O submission queue
- Create I/O completion queue
- Destroy I/O submission queue
- Destroy I/O completion queue
- Get feature
- Set feature
- Get statistics

Refer to `ena_admin_defs.h` for the list of supported Get/Set Feature properties.

8.7.3 Data Path Interface

I/O operations are based on Tx and Rx Submission Queues (Tx SQ and Rx SQ correspondingly). Each SQ has a completion queue (CQ) associated with it.

The SQs and CQs are implemented as descriptor rings in contiguous physical memory.

Refer to `ena_eth_io_defs.h` for the detailed structure of the descriptor

The driver supports multi-queue for both Tx and Rx.

8.7.4 Configuration information

DPDK Configuration Parameters

The following configuration options are available for the ENA PMD:

- **CONFIG_RTE_LIBRTE_ENA_PMD** (default y): Enables or disables inclusion of the ENA PMD driver in the DPDK compilation.
- **CONFIG_RTE_LIBRTE_ENA_DEBUG_INIT** (default y): Enables or disables debug logging of device initialization within the ENA PMD driver.

- **CONFIG_RTE_LIBRTE_ENA_DEBUG_RX** (default n): Enables or disables debug logging of RX logic within the ENA PMD driver.
- **CONFIG_RTE_LIBRTE_ENA_DEBUG_TX** (default n): Enables or disables debug logging of TX logic within the ENA PMD driver.
- **CONFIG_RTE_LIBRTE_ENA_COM_DEBUG** (default n): Enables or disables debug logging of low level tx/rx logic in ena_com(base) within the ENA PMD driver.

ENA Configuration Parameters

- **Number of Queues**

  This is the requested number of queues upon initialization, however, the actual number of receive and transmit queues to be created will be the minimum between the maximal number supported by the device and number of queues requested.

- **Size of Queues**

  This is the requested size of receive/transmit queues, while the actual size will be the minimum between the requested size and the maximal receive/transmit supported by the device.

8.7.5 Building DPDK

See the *DPDK Getting Started Guide for Linux* for instructions on how to build DPDK.

By default the ENA PMD library will be built into the DPDK library.

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

8.7.6 Supported ENA adapters

Current ENA PMD supports the following ENA adapters including:

- `1d0f:ec20` - ENA VF
- `1d0f:ec21` - ENA VF with LLQ support

8.7.7 Supported Operating Systems

Any Linux distribution fulfilling the conditions described in `System Requirements` section of *the DPDK documentation* or refer to *DPDK Release Notes*.

8.7.8 Supported features

- Jumbo frames up to 9K
- Port Hardware Statistics
- IPv4/TCP/UDP checksum offload
- TSO offload
- Multiple receive and transmit queues
- RSS
- Low Latency Queue for Tx

8.7.9 Unsupported features

The features supported by the device and not yet supported by this PMD include:

- Asynchronous Event Notification Queue (AENQ)

8.7.10 Prerequisites

1. Prepare the system as recommended by DPDK suite. This includes environment variables, hugepages configuration, tool-chains and configuration
2. Insert igb_uio kernel module using the command 'modprobe igb_uio'
3. Bind the intended ENA device to igb_uio module

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

8.7.11 Usage example

This section demonstrates how to launch **testpmd** with Amazon ENA devices managed by librte_pmd_ena.

1. Load the kernel modules:

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

Note: Currently Amazon ENA PMD driver depends on igb_uio user space I/O kernel module

2. Mount and request huge pages:

```
mount -t hugetlbfs nodev /mnt/hugepages
echo 1024 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
```

3. Bind UIO driver to ENA device (using provided by DPDK binding tool):

```
./usertools/dpdk-devbind.py --bind=igb_uio 0000:02:00.1
```

4. Start testpmd with basic parameters:

```
./x86_64-native-linuxapp-gcc/app/testpmd -l 0-3 -n 4 -- -i
```

Example output:

```
[...]
EAL: PCI device 0000:02:00.1 on NUMA socket -1
EAL:   probe driver: 1d0f:ec20 rte_ena_pmd
EAL:   PCI memory mapped at 0x7f9b6c400000
PMD: eth_ena_dev_init(): Initializing 0:2:0.1
Interactive-mode selected
Configuring Port 0 (socket 0)
Port 0: 00:00:00:11:00:01
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Done
testpmd>
```

8.8 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.

8.8.1 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

8.8.2 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 queues (RQs), work queues (WQs) and completion queues (CQs) are configurable on a per vNIC basis through the Cisco UCS Manager (CIMC or UCSM).

    These values should be configured as follows:

    * The number of WQs should be greater or equal to the value of the expected nb_tx_q parameter in the call to the rte_eth_dev_configure()
    * The number of RQs configured in the vNIC should be greater or equal to *twice* the value of the expected nb_rx_q parameter in the call to rte_eth_dev_configure(). With the addition of rx scatter, a pair of RQs on the vnic is needed for each receive queue used by DPDK, even if rx scatter is not being used. Having a vNIC with only 1 RQ is not a valid configuration, and will fail with an error message.
    * The number of CQs should set so that there is one CQ for each WQ, and one CQ for each pair of RQs.

    For example: If the application requires 3 Rx queues, and 3 Tx queues, the vNIC should be configured to have at least 3 WQs, 6 RQs (3 pairs), and 6 CQs (3 for use by WQs + 3 for use by the 3 pairs of RQs).

  - **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. An application requesting more than the set size will be limited to that size.

    Unless there is a lack of resources due to creating many vNICs, it is recommended that the WQ and RQ sizes be set to the maximum. This gives the application the greatest amount of flexibility in its queue configuration.

* *Note*: Since the introduction of rx scatter, for performance reasons, this PMD uses two RQs on the vNIC per receive queue in DPDK. One RQ holds descriptors for the start of a packet the second RQ holds the descriptors for the rest of the fragments of a packet. This means that the nb_rx_desc parameter to rte_eth_rx_queue_setup() can be a greater than 4096. The exact amount will depend on the size of the mbufs being used for receives, and the MTU size.

  For example: If the mbuf size is 2048, and the MTU is 9000, then receiving a full size packet will take 5 descriptors, 1 from the start of packet queue, and 4 from the second queue. Assuming that the RQ size was set to the maximum of 4096, then the application can specify up to 1024 + 4096 as the nb_rx_desc parameter to rte_eth_rx_queue_setup().

– **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 link status and errors in the fast path.

8.8.3 Flow director support

Advanced filtering support was added to 1300 series VIC firmware starting with version 2.0.13 for C-series UCS servers and version 3.1.2 for UCSM managed blade servers. In order to enable advanced filtering the 'Advanced filter' radio button should be enabled via CIMC or UCSM followed by a reboot of the server.

With advanced filters, perfect matching of all fields of IPv4, IPv6 headers as well as TCP, UDP and SCTP L4 headers is available through flow director. Masking of these feilds for partial match is also supported.

Without advanced filter support, the flow director is limited to IPv4 perfect filtering of the 5-tuple with no masking of fields supported.

8.8.4 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);
```

- Limited flow director support on 1200 series and 1300 series Cisco VIC adapters with old firmware. Please see *Flow director support*.
- Flow director features are not supported on generation 1 Cisco VIC adapters (M81KR and P81E)

8.8.5 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.

8.8.6 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

8.8.7 Supported Operating Systems

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

8.8.8 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 IPv4 and IPv6
- 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
- Scattered Rx
- MTU update

8.8.9 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
- MTU update only works if Scattered Rx mode is disabled

8.8.10 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-devbind.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-devbind.py.
- Bind the intended vNIC to igb_uio in case the user wants ENIC PMD to use UIO framework using dpdk-devbind.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.

8.8.11 Additional Reference

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

8.8.12 Contact Information

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

- John Daley <johndale@cisco.com>
- Nelson Escobar <neescoba@cisco.com>

8.9 FM10K Poll Mode Driver

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

8.9.1 FTAG Based Forwarding of FM10K

FTAG Based Forwarding is a unique feature of FM10K. The FM10K family of NICs support the addition of a Fabric Tag (FTAG) to carry special information. The FTAG is placed at the beginning of the frame, it contains information such as where the packet comes from and goes, and the vlan tag. In FTAG based forwarding mode, the switch logic forwards packets according to glort (global resource tag) information, rather than the mac and vlan table. Currently this feature works only on PF.

To enable this feature, the user should pass a devargs parameter to the eal like "-w 84:00.0,enable_ftag=1", and the application should make sure an appropriate FTAG is inserted for every frame on TX side.

8.9.2 Vector PMD for FM10K

Vector PMD (vPMD) 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)''. The wider register gives space to hold multiple packet buffers so as to save on the number of instructions when bulk processing packets.

There is no change to the PMD API. The RX/TX handlers are the only two entries for vPMD packet I/O. They are transparently registered at runtime RX/TX execution if all required conditions are met.

1. To date, only an SSE version of FM10K vPMD is available. To ensure that vPMD is in the binary code, set `CONFIG_RTE_LIBRTE_FM10K_INC_VECTOR=y` 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.

RX Constraints

Prerequisites and Pre-conditions

For Vector RX it is assumed that the number of descriptor rings will be a power of 2. With this pre-condition, the ring pointer can easily scroll back to the head after hitting the tail without a conditional check. In addition Vector RX can use this assumption to do a bit mask using `ring_size - 1`.

Features not Supported by Vector RX PMD

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

- IEEE1588

- Flow director
- Header split
- RX checksum offload

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

- HW VLAN strip
- L3/L4 packet type

To enable via `RX_OLFLAGS` use `RTE_LIBRTE_FM10K_RX_OLFLAGS_ENABLE=y`.

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

- `hw_vlan_extend`
- `hw_ip_checksum`
- `header_split`
- `fdir_conf->mode`

RX Burst Size

As vPMD is focused on high throughput, it processes 4 packets at a time. So it assumes that the RX burst should be greater than 4 packets per burst. It returns zero if using `nb_pkt` < 4 in the receive handler. If `nb_pkt` is not a multiple of 4, a floor alignment will be applied.

TX Constraint

Features not Supported by TX Vector PMD

TX vPMD only works when `txq_flags` is set to `FM10K_SIMPLE_TX_FLAG`. This means that it does not support TX multi-segment, VLAN offload or 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`

8.9.3 Limitations

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 from Intel support. Only Testpoint is validated with DPDK, the latest version that has been validated with DPDK is 4.1.6.

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`.

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.

Statistic Polling Frequency

The FM10000 NICs expose a set of statistics via the PCI BARs. These statistics are read from the hardware registers when `rte_eth_stats_get()` or `rte_eth_xstats_get()` is called. The packet counting registers are 32 bits while the byte counting registers are 48 bits. As a result, the statistics must be polled regularly in order to ensure the consistency of the returned reads.

Given the PCIe Gen3 x8, about 50Gbps of traffic can occur. With 64 byte packets this gives almost 100 million packets/second, causing 32 bit integer overflow after approx 40 seconds. To ensure these overflows are detected and accounted for in the statistics, it is necessary to read statistic regularly. It is suggested to read stats every 20 seconds, which will ensure the statistics are accurate.

Interrupt mode

The FM10000 family of NICS need one separate interrupt for mailbox. So only drivers which support multiple interrupt vectors e.g. vfio-pci can work for fm10k interrupt mode.

8.10 I40E Poll Mode Driver

The I40E PMD (librte_pmd_i40e) provides poll mode driver support for the Intel X710/XL710/X722 10/40 Gbps family of adapters.

8.10.1 Features

Features of the I40E PMD are:

- Multiple queues for TX and RX
- Receiver Side Scaling (RSS)
- MAC/VLAN filtering
- Packet type information
- Flow director
- Cloud filter
- Checksum offload
- VLAN/QinQ stripping and inserting
- TSO offload

- Promiscuous mode
- Multicast mode
- Port hardware statistics
- Jumbo frames
- Link state information
- Link flow control
- Mirror on port, VLAN and VSI
- Interrupt mode for RX
- Scattered and gather for TX and RX
- Vector Poll mode driver
- DCB
- VMDQ
- SR-IOV VF
- Hot plug
- IEEE1588/802.1AS timestamping
- VF Daemon (VFD) - EXPERIMENTAL

8.10.2 Prerequisites

- Identifying your adapter using Intel Support and get the latest NVM/FW images.
- Follow the DPDK *Getting Started Guide for Linux* to setup the basic DPDK environment.
- To get better performance on Intel platforms, please follow the "How to get best performance with NICs on Intel platforms" section of the *Getting Started Guide for Linux*.

8.10.3 Pre-Installation Configuration

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_I40E_PMD` (default `y`)

  Toggle compilation of the `librte_pmd_i40e` driver.

- `CONFIG_RTE_LIBRTE_I40E_DEBUG_*` (default `n`)

  Toggle display of generic debugging messages.

- `CONFIG_RTE_LIBRTE_I40E_RX_ALLOW_BULK_ALLOC` (default `y`)

  Toggle bulk allocation for RX.

- `CONFIG_RTE_LIBRTE_I40E_INC_VECTOR` (default `n`)

  Toggle the use of Vector PMD instead of normal RX/TX path. To enable vPMD for RX, bulk allocation for Rx must be allowed.

- CONFIG_RTE_LIBRTE_I40E_RX_OLFLAGS_ENABLE (default y)

  Toggle to enable RX olflags. This is only meaningful when Vector PMD is used.

- CONFIG_RTE_LIBRTE_I40E_16BYTE_RX_DESC (default n)

  Toggle to use a 16-byte RX descriptor, by default the RX descriptor is 32 byte.

- CONFIG_RTE_LIBRTE_I40E_QUEUE_NUM_PER_PF (default 64)

  Number of queues reserved for PF.

- CONFIG_RTE_LIBRTE_I40E_QUEUE_NUM_PER_VF (default 4)

  Number of queues reserved for each SR-IOV VF.

- CONFIG_RTE_LIBRTE_I40E_QUEUE_NUM_PER_VM (default 4)

  Number of queues reserved for each VMDQ Pool.

- CONFIG_RTE_LIBRTE_I40E_ITR_INTERVAL (default -1)

  Interrupt Throttling interval.

Driver Compilation

To compile the I40E PMD see *Getting Started Guide for Linux* or *Getting Started Guide for FreeBSD* depending on your platform.

8.10.4 Linux

Running testpmd

This section demonstrates how to launch testpmd with Intel XL710/X710 devices managed by librte_pmd_i40e in the Linux operating system.

1. Load igb_uio or vfio-pci driver:

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

   or

```
modprobe vfio-pci
```

2. Bind the XL710/X710 adapters to igb_uio or vfio-pci loaded in the previous step:

```
./usertools/dpdk-devbind.py --bind igb_uio 0000:83:00.0
```

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

```
./usertools/dpdk-devbind.py --bind vfio-pci 0000:83:00.0
```

3. Start testpmd with basic parameters:

```
./x86_64-native-linuxapp-gcc/app/testpmd -l 0-3 -n 4 -w 83:00.0 -- -i
```

   Example output:

```
...
EAL: PCI device 0000:83:00.0 on NUMA socket 1
EAL: probe driver: 8086:1572 rte_i40e_pmd
EAL: PCI memory mapped at 0x7f7f80000000
EAL: PCI memory mapped at 0x7f7f80800000
PMD: eth_i40e_dev_init(): FW 5.0 API 1.5 NVM 05.00.02 eetrack 8000208a
Interactive-mode selected
Configuring Port 0 (socket 0)
...

PMD: i40e_dev_rx_queue_setup(): Rx Burst Bulk Alloc Preconditions are
satisfied.Rx Burst Bulk Alloc function will be used on port=0, queue=0.

...
Port 0: 68:05:CA:26:85:84
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Done

testpmd>
```

SR-IOV: Prerequisites and sample Application Notes

1. Load the kernel module:

   ```
   modprobe i40e
   ```

   Check the output in dmesg:

   ```
   i40e 0000:83:00.1 ens802f0: renamed from eth0
   ```

2. Bring up the PF ports:

   ```
   ifconfig ens802f0 up
   ```

3. Create VF device(s):

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

   Example:

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

4. Assign VF MAC address:

   Assign MAC address to the VF using iproute2 utility. The syntax is:

   ```
   ip link set <PF netdev id> vf <VF id> mac <macaddr>
   ```

   Example:

   ```
   ip link set ens802f0 vf 0 mac a0:b0:c0:d0:e0:f0
   ```

5. Assign VF to VM, and bring up the VM. Please see the documentation for the *I40E/IXGBE/IGB Virtual Function Driver*.

8.10.5 Sample Application Notes

Vlan filter

Vlan filter only works when Promiscuous mode is off.

To start `testpmd`, and add vlan 10 to port 0:

```
./app/testpmd -l 0-15 -n 4 -- -i --forward-mode=mac
...

testpmd> set promisc 0 off
testpmd> rx_vlan add 10 0
```

Flow Director

The Flow Director works in receive mode to identify specific flows or sets of flows and route them to specific queues. The Flow Director filters can match the different fields for different type of packet: flow type, specific input set per flow type and the flexible payload.

The default input set of each flow type is:

```
ipv4-other : src_ip_address, dst_ip_address
ipv4-frag  : src_ip_address, dst_ip_address
ipv4-tcp   : src_ip_address, dst_ip_address, src_port, dst_port
ipv4-udp   : src_ip_address, dst_ip_address, src_port, dst_port
ipv4-sctp  : src_ip_address, dst_ip_address, src_port, dst_port,
             verification_tag
ipv6-other : src_ip_address, dst_ip_address
ipv6-frag  : src_ip_address, dst_ip_address
ipv6-tcp   : src_ip_address, dst_ip_address, src_port, dst_port
ipv6-udp   : src_ip_address, dst_ip_address, src_port, dst_port
ipv6-sctp  : src_ip_address, dst_ip_address, src_port, dst_port,
             verification_tag
l2_payload : ether_type
```

The flex payload is selected from offset 0 to 15 of packet's payload by default, while it is masked out from matching.

Start `testpmd` with `--disable-rss` and `--pkt-filter-mode=perfect`:

```
./app/testpmd -l 0-15 -n 4 -- -i --disable-rss --pkt-filter-mode=perfect \
              --rxq=8 --txq=8 --nb-cores=8 --nb-ports=1
```

Add a rule to direct `ipv4-udp` packet whose `dst_ip=2.2.2.5, src_ip=2.2.2.3, src_port=32, dst_port=32` to queue 1:

```
testpmd> flow_director_filter 0 mode IP add flow ipv4-udp  \
         src 2.2.2.3 32 dst 2.2.2.5 32 vlan 0 flexbytes () \
         fwd pf queue 1 fd_id 1
```

Check the flow director status:

```
testpmd> show port fdir 0

###################### FDIR infos for port 0      ##################
  MODE:   PERFECT
  SUPPORTED FLOW TYPE:  ipv4-frag ipv4-tcp ipv4-udp ipv4-sctp ipv4-other
```

```
                    ipv6-frag ipv6-tcp ipv6-udp ipv6-sctp ipv6-other
                    l2_payload
  FLEX PAYLOAD INFO:
  max_len:       16          payload_limit: 480
  payload_unit:  2           payload_seg:   3
  bitmask_unit:  2           bitmask_num:   2
  MASK:
    vlan_tci: 0x0000,
    src_ipv4: 0x00000000,
    dst_ipv4: 0x00000000,
    src_port: 0x0000,
    dst_port: 0x0000
    src_ipv6: 0x00000000,0x00000000,0x00000000,0x00000000,
    dst_ipv6: 0x00000000,0x00000000,0x00000000,0x00000000
  FLEX PAYLOAD SRC OFFSET:
    L2_PAYLOAD:    0      1      2      3      4      5      6  ...
    L3_PAYLOAD:    0      1      2      3      4      5      6  ...
    L4_PAYLOAD:    0      1      2      3      4      5      6  ...
  FLEX MASK CFG:
    ipv4-udp:    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    ipv4-tcp:    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    ipv4-sctp:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    ipv4-other:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    ipv4-frag:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    ipv6-udp:    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    ipv6-tcp:    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    ipv6-sctp:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    ipv6-other:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    ipv6-frag:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    l2_payload:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  guarant_count: 1           best_count:    0
  guarant_space: 512         best_space:    7168
  collision:     0           free:          0
  maxhash:       0           maxlen:        0
  add:           0           remove:        0
  f_add:         0           f_remove:      0
```

Delete all flow director rules on a port:

```
testpmd> flush_flow_director 0
```

Floating VEB

The Intel® Ethernet Controller X710 and XL710 Family support a feature called "Floating VEB".

A Virtual Ethernet Bridge (VEB) is an IEEE Edge Virtual Bridging (EVB) term for functionality that allows local switching between virtual endpoints within a physical endpoint and also with an external bridge/network.

A "Floating" VEB doesn't have an uplink connection to the outside world so all switching is done internally and remains within the host. As such, this feature provides security benefits.

In addition, a Floating VEB overcomes a limitation of normal VEBs where they cannot forward packets when the physical link is down. Floating VEBs don't need to connect to the NIC port so they can still forward traffic from VF to VF even when the physical link is down.

Therefore, with this feature enabled VFs can be limited to communicating with each other but not an outside network, and they can do so even when there is no physical uplink on the associated NIC port.

To enable this feature, the user should pass a `devargs` parameter to the EAL, for example:

```
-w 84:00.0,enable_floating_veb=1
```

In this configuration the PMD will use the floating VEB feature for all the VFs created by this PF device.

Alternatively, the user can specify which VFs need to connect to this floating VEB using the `floating_veb_list` argument:

```
-w 84:00.0,enable_floating_veb=1,floating_veb_list=1;3-4
```

In this example `VF1`, `VF3` and `VF4` connect to the floating VEB, while other VFs connect to the normal VEB.

The current implementation only supports one floating VEB and one regular VEB. VFs can connect to a floating VEB or a regular VEB according to the configuration passed on the EAL command line.

The floating VEB functionality requires a NIC firmware version of 5.0 or greater.

8.10.6 Limitations or Known issues

MPLS packet classification on X710/XL710

For firmware versions prior to 5.0, MPLS packets are not recognized by the NIC. The L2 Payload flow type in flow director can be used to classify MPLS packet by using a command in testpmd like:

testpmd> flow_director_filter 0 mode IP add flow l2_payload ether 0x8847 flexbytes () fwd pf queue <N> fd_id <M>

With the NIC firmware version 5.0 or greater, some limited MPLS support is added: Native MPLS (MPLS in Ethernet) skip is implemented, while no new packet type, no classification or offload are possible. With this change, L2 Payload flow type in flow director cannot be used to classify MPLS packet as with previous firmware versions. Meanwhile, the Ethertype filter can be used to classify MPLS packet by using a command in testpmd like:

testpmd> ethertype_filter 0 add mac_ignr 00:00:00:00:00:00 ethertype 0x8847 fwd queue <M>

16 Byte Descriptor cannot be used on DPDK VF

If the Linux i40e kernel driver is used as host driver, while DPDK i40e PMD is used as the VF driver, DPDK cannot choose 16 byte receive descriptor. That is to say, user should keep `CONFIG_RTE_LIBRTE_I40E_16BYTE_RX_DESC=n` in config file.

Link down with i40e kernel driver after DPDK application exit

After DPDK application quit, and the device is bound back to Linux i40e kernel driver, the link cannot be up after `ifconfig <dev> up`. To work around this issue, `ethtool -s <dev> autoneg on` should be set first and then the link can be brought up through `ifconfig <dev> up`.

NOTE: requires Linux kernel i40e driver version >= 1.4.X

Receive packets with Ethertype 0x88A8

Due to the FW limitation, PF can receive packets with Ethertype 0x88A8 only when floating VEB is disabled.

Incorrect Rx statistics when packet is oversize

When a packet is over maximum frame size, the packet is dropped. However the Rx statistics, when calling *rte_eth_stats_get* incorrectly shows it as received.

VF & TC max bandwidth setting

The per VF max bandwidth and per TC max bandwidth cannot be enabled in parallel. The dehavior is different when handling per VF and per TC max bandwidth setting. When enabling per VF max bandwidth, SW will check if per TC max bandwidth is enabled. If so, return failure. When enabling per TC max bandwidth, SW will check if per VF max bandwidth is enabled. If so, disable per VF max bandwidth and continue with per TC max bandwidth setting.

TC TX scheduling mode setting

There're 2 TX scheduling modes for TCs, round robin and strict priority mode. If a TC is set to strict priority mode, it can consume unlimited bandwidth. It means if APP has set the max bandwidth for that TC, it comes to no effect. It's suggested to set the strict priority mode for a TC that is latency sensitive but no consuming much bandwidth.

8.11 IXGBE Driver

8.11.1 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.

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.

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 TX 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

8.11.2 Application Programming Interface

In DPDK release v16.11 an API for ixgbe specific functions has been added to the ixgbe PMD. The declarations for the API functions are in the header `rte_pmd_ixgbe.h`.

8.11.3 Sample Application Notes

testpmd

By default, using CONFIG_RTE_IXGBE_RX_OLFLAGS_ENABLE=y:

```
./x86_64-native-linuxapp-gcc/app/testpmd -l 8-9 -n 4 -- -i --burst=32 --rxfreet=32 --
↪mbcache=250 --txpt=32 --rxht=8 --rxwt=0 --txfreet=32 --txrst=32 --txqflags=0xf01
```

When CONFIG_RTE_IXGBE_RX_OLFLAGS_ENABLE=n, better performance can be achieved:

```
./x86_64-native-linuxapp-gcc/app/testpmd -l 8-9 -n 4 -- -i --burst=32 --rxfreet=32 --
↪mbcache=250 --txpt=32 --rxht=8 --rxwt=0 --txfreet=32 --txrst=32 --txqflags=0xf01 --
↪disable-hw-vlan
```

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.

8.11.4 Limitations or Known issues

Malicious Driver Detection not Supported

The Intel x550 series NICs support a feature called MDD (Malicious Driver Detection) which checks the behavior of the VF driver. If this feature is enabled, the VF must use the advanced context descriptor correctly and set the

CC (Check Context) bit. DPDK PF doesn't support MDD, but kernel PF does. We may hit problem in this scenario kernel PF + DPDK VF. If user enables MDD in kernel PF, DPDK VF will not work. Because kernel PF thinks the VF is malicious. But actually it's not. The only reason is the VF doesn't act as MDD required. There's significant performance impact to support MDD. DPDK should check if the advanced context descriptor should be set and set it. And DPDK has to ask the info about the header length from the upper layer, because parsing the packet itself is not acceptable. So, it's too expensive to support MDD. When using kernel PF + DPDK VF on x550, please make sure using the kernel driver that disables MDD or can disable MDD. (Some kernel driver can use this CLI 'insmod ixgbe.ko MDD=0,0' to disable MDD. Some kernel driver disables it by default.)

Statistics

The statistics of ixgbe hardware must be polled regularly in order for it to remain consistent. Running a DPDK application without polling the statistics will cause registers on hardware to count to the maximum value, and "stick" at that value.

In order to avoid statistic registers every reaching the maximum value, read the statistics from the hardware using `rte_eth_stats_get()` or `rte_eth_xstats_get()`.

The maximum time between statistics polls that ensures consistent results can be calculated as follows:

```
max_read_interval = UINT_MAX / max_packets_per_second
max_read_interval = 4294967295 / 14880952
max_read_interval = 288.6218096127183 (seconds)
max_read_interval = ~4 mins 48 sec.
```

In order to ensure valid results, it is recommended to poll every 4 minutes.

MTU setting

Although the user can set the MTU separately on PF and VF ports, the ixgbe NIC only supports one global MTU per physical port. So when the user sets different MTUs on PF and VF ports in one physical port, the real MTU for all these PF and VF ports is the largest value set. This behavior is based on the kernel driver behavior.

8.11.5 Supported Chipsets and NICs

- Intel 82599EB 10 Gigabit Ethernet Controller
- Intel 82598EB 10 Gigabit Ethernet Controller
- Intel 82599ES 10 Gigabit Ethernet Controller
- Intel 82599EN 10 Gigabit Ethernet Controller
- Intel Ethernet Controller X540-AT2
- Intel Ethernet Controller X550-BT2
- Intel Ethernet Controller X550-AT2
- Intel Ethernet Controller X550-AT
- Intel Ethernet Converged Network Adapter X520-SR1
- Intel Ethernet Converged Network Adapter X520-SR2
- Intel Ethernet Converged Network Adapter X520-LR1
- Intel Ethernet Converged Network Adapter X520-DA1

- Intel Ethernet Converged Network Adapter X520-DA2
- Intel Ethernet Converged Network Adapter X520-DA4
- Intel Ethernet Converged Network Adapter X520-QDA1
- Intel Ethernet Converged Network Adapter X520-T2
- Intel 10 Gigabit AF DA Dual Port Server Adapter
- Intel 10 Gigabit AT Server Adapter
- Intel 10 Gigabit AT2 Server Adapter
- Intel 10 Gigabit CX4 Dual Port Server Adapter
- Intel 10 Gigabit XF LR Server Adapter
- Intel 10 Gigabit XF SR Dual Port Server Adapter
- Intel 10 Gigabit XF SR Server Adapter
- Intel Ethernet Converged Network Adapter X540-T1
- Intel Ethernet Converged Network Adapter X540-T2
- Intel Ethernet Converged Network Adapter X550-T1
- Intel Ethernet Converged Network Adapter X550-T2

8.12 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.

8.12.1 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. 8.1.

Therefore, a NIC is logically distributed among multiple virtual machines (as shown in Fig. 8.1), 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

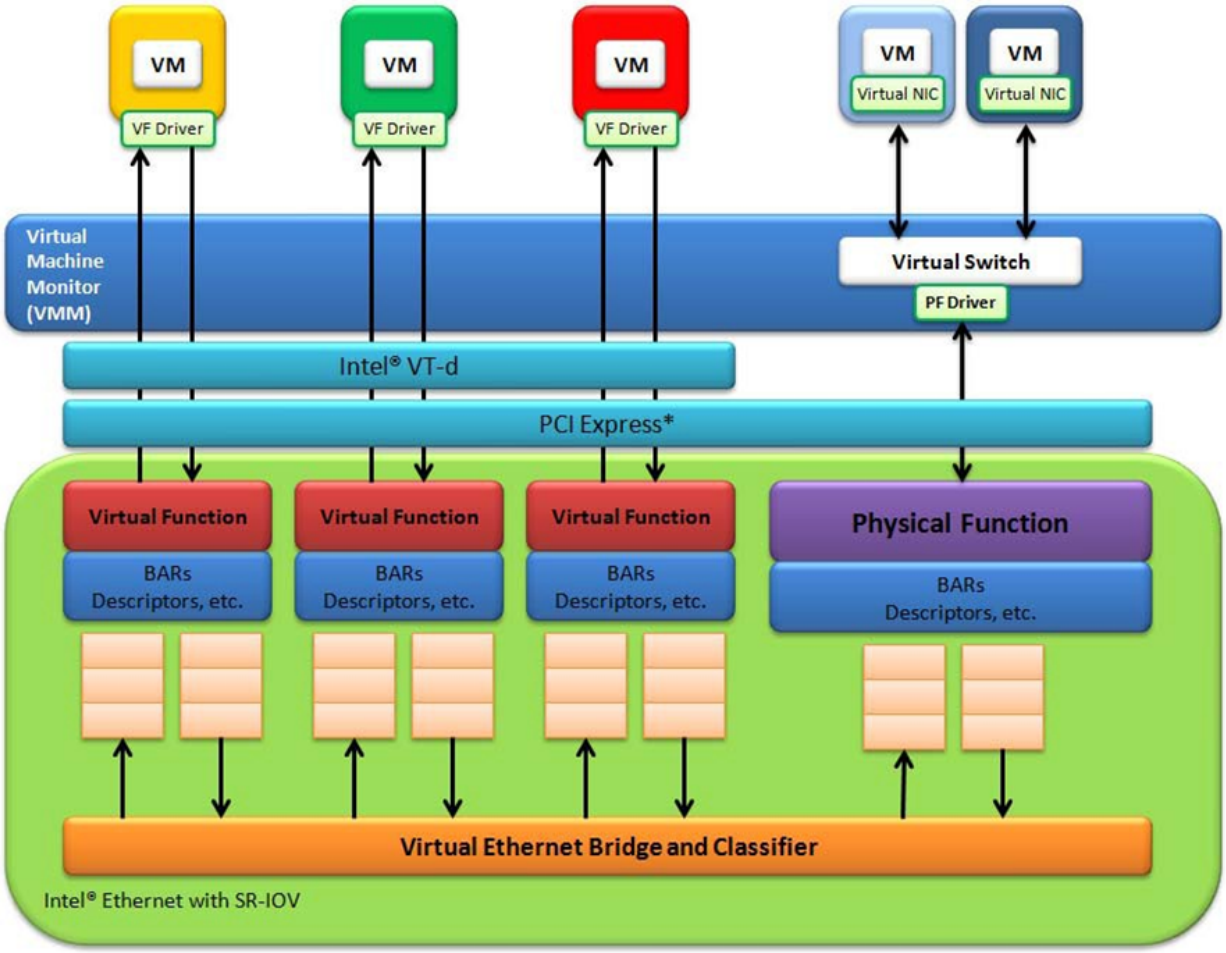


Fig. 8.1: Virtualization for a Single Port NIC in SR-IOV Mode

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-devbind.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 device)
```

  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 -l 0-15 -n 4 -- --coremask=<core-mask> --rxq=4 --txq=4 -i
```

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-devbind.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 device)
```

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 DPDK PMD PF igb driver:

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

```
insmod igb_uio
./dpdk-devbind.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 device)
```

  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.

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.

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*.

8.12.2 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-devbind.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 -
↪> ../0000:02:10.2
lrwxrwxrwx. 1 root root 0 Apr 13 05:40 /sys/bus/pci/devices/0000:02:00.0/virtfn0 -
↪> ../0000:02:10.0
```

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 -
↪> ../0000:02:10.3
lrwxrwxrwx. 1 root root 0 Apr 13 05:51 /sys/bus/pci/devices/0000:02:00.1/virtfn0 -
↪> ../0000:02:10.1
```

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-assign,host=08:10.0
```

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 value 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 -device pci-assign,host=08:10.0

(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 -l 0-3 -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. 8.2) is equivalent to a Host OS with KVM installed as described in the instructions.

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

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. 8.3 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.

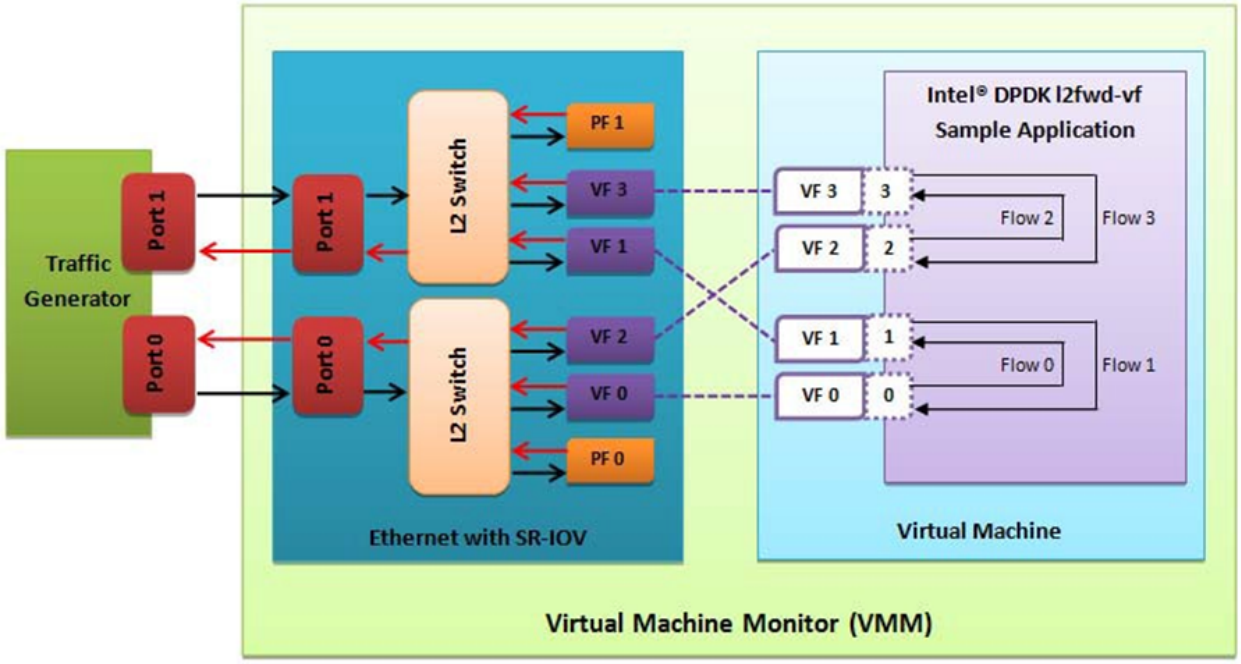


Fig. 8.2: Performance Benchmark Setup

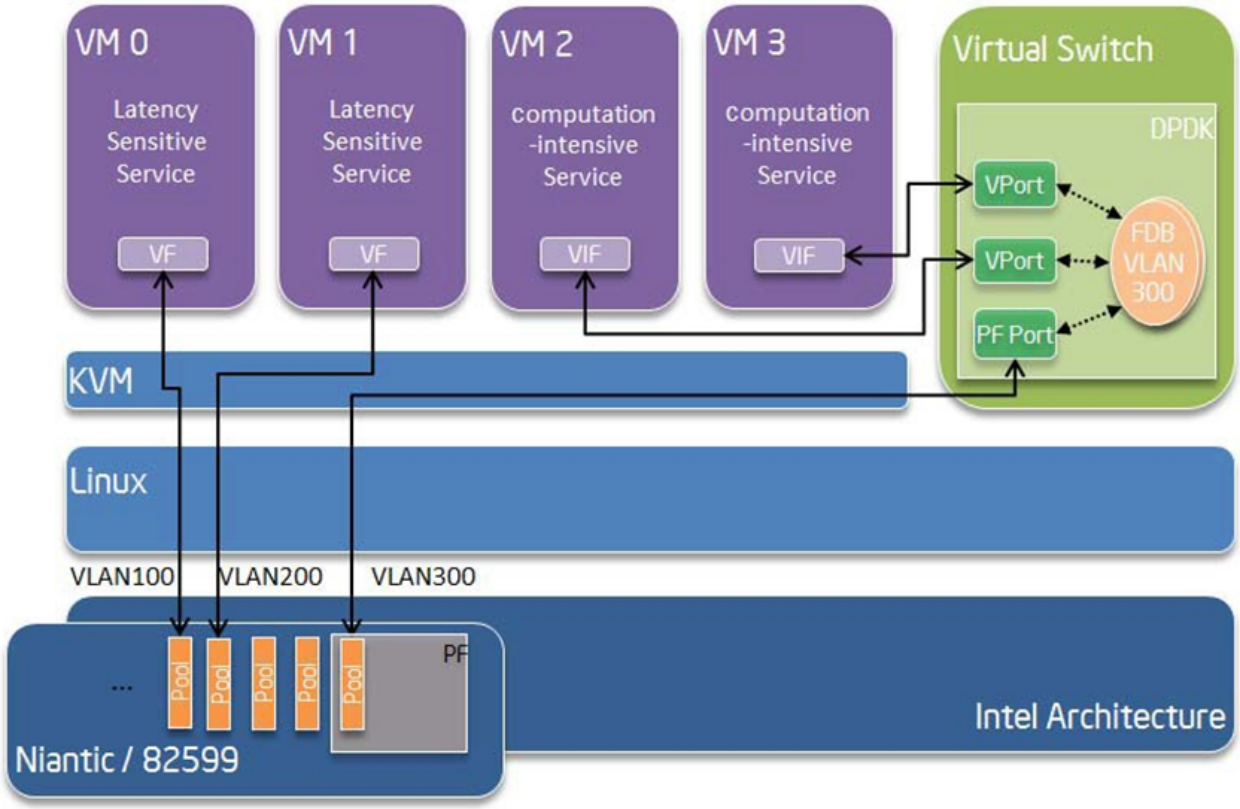


Fig. 8.3: Fast Host-based Packet Processing

8.12.4 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. 8.4 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.

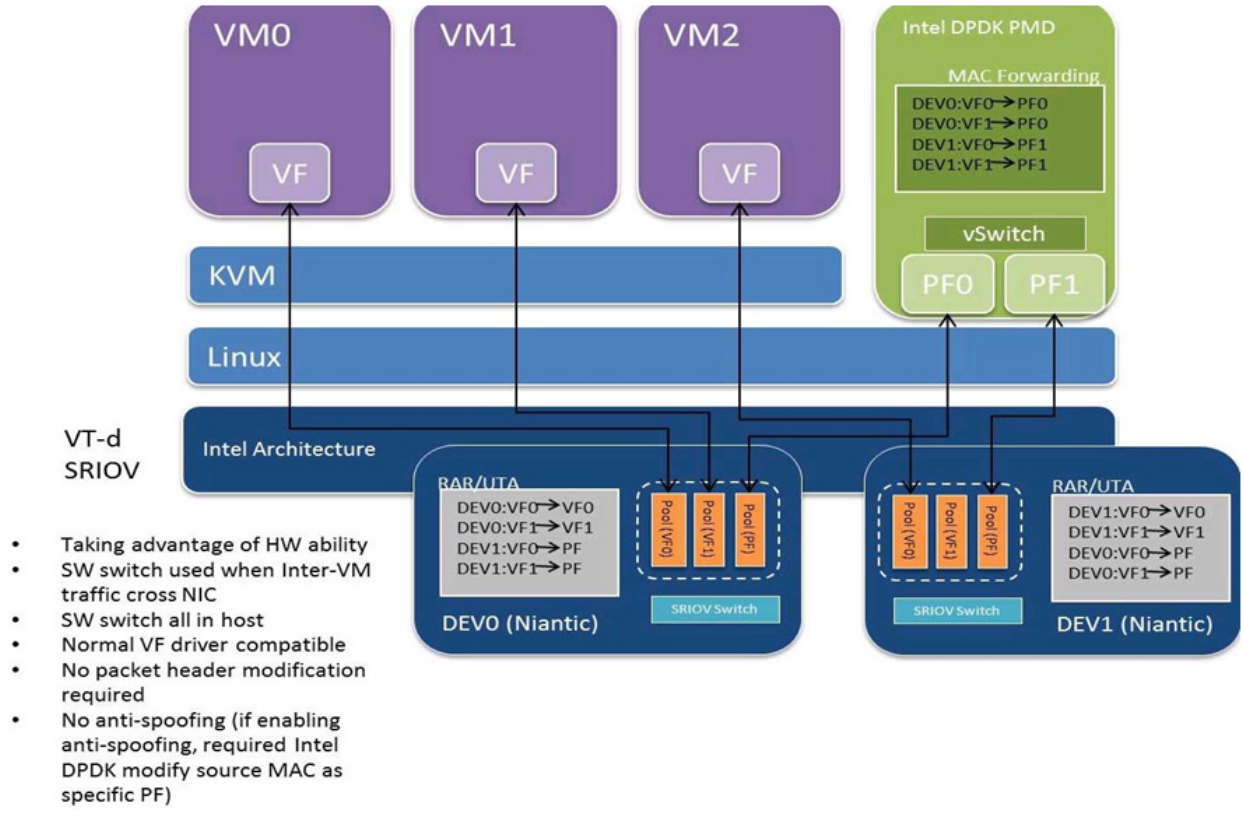


Fig. 8.4: Inter-VM Communication

8.13 KNI Poll Mode Driver

KNI PMD is wrapper to the *librte_kni* library.

This PMD enables using KNI without having a KNI specific application, any forwarding application can use PMD interface for KNI.

Sending packets to any DPDK controlled interface or sending to the Linux networking stack will be transparent to the DPDK application.

To create a KNI device `net_kni#` device name should be used, and this will create `kni#` Linux virtual network interface.

There is no physical device backend for the virtual KNI device.

Packets sent to the KNI Linux interface will be received by the DPDK application, and DPDK application may forward packets to a physical NIC or to a virtual device (like another KNI interface or PCAP interface).

To forward any traffic from physical NIC to the Linux networking stack, an application should control a physical port and create one virtual KNI port, and forward between two.

Using this PMD requires KNI kernel module be inserted.

8.13.1 Usage

EAL `--vdev` argument can be used to create KNI device instance, like:

```
testpmd --vdev=net_kni0 --vdev=net_kn1 -- -i
```

Above command will create `kni0` and `kni1` Linux network interfaces, those interfaces can be controlled by standard Linux tools.

When testpmd forwarding starts, any packets sent to `kni0` interface forwarded to the `kni1` interface and vice versa.

There is no hard limit on number of interfaces that can be created.

8.13.2 Default interface configuration

`librte_kni` can create Linux network interfaces with different features, feature set controlled by a configuration struct, and KNI PMD uses a fixed configuration:

```
Interface name: kni#
force bind kernel thread to a core : NO
mbuf size: MAX_PACKET_SZ
```

KNI control path is not supported with the PMD, since there is no physical backend device by default.

8.13.3 PMD arguments

`no_request_thread`, by default PMD creates a phtread for each KNI interface to handle Linux network interface control commands, like `ifconfig kni0 up`

With `no_request_thread` option, pthread is not created and control commands not handled by PMD.

By default request thread is enabled. And this argument should not be used most of the time, unless this PMD used with customized DPDK application to handle requests itself.

Argument usage:

```
testpmd --vdev "net_kni0,no_request_thread=1" -- -i
```

8.13.4 PMD log messages

If KNI kernel module (rte_kni.ko) not inserted, following error log printed:

```
"KNI: KNI subsystem has not been initialized. Invoke rte_kni_init() first"
```

8.13.5 PMD testing

It is possible to test PMD quickly using KNI kernel module loopback feature:

- Insert KNI kernel module with loopback support:

```
insmod build/kmod/rte_kni.ko lo_mode=lo_mode_fifo_skb
```

- Start testpmd with no physical device but two KNI virtual devices:

```
./testpmd --vdev net_kni0 --vdev net_kni1 -- -i
```

```
...
Configuring Port 0 (socket 0)
KNI: pci: 00:00:00       c580:b8
Port 0: 1A:4A:5B:7C:A2:8C
Configuring Port 1 (socket 0)
KNI: pci: 00:00:00       600:b9
Port 1: AE:95:21:07:93:DD
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Port 1 Link Up - speed 10000 Mbps - full-duplex
Done
testpmd>
```

- Observe Linux interfaces

```
$ ifconfig kni0 && ifconfig kni1
kni0: flags=4098<BROADCAST,MULTICAST>  mtu 1500
        ether ae:8e:79:8e:9b:c8  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

kni1: flags=4098<BROADCAST,MULTICAST>  mtu 1500
        ether 9e:76:43:53:3e:9b  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

- Start forwarding with tx_first:

```
testpmd> start tx_first
```

- Quit and check forwarding stats:

```
testpmd> quit
Telling cores to stop...
Waiting for lcores to finish...

---------------------- Forward statistics for port 0  ----------------------
↪--
```

```
RX-packets: 35637905       RX-dropped: 0             RX-total: 35637905
TX-packets: 35637947       TX-dropped: 0             TX-total: 35637947
----------------------------------------------------------------------------
↪--

---------------------- Forward statistics for port 1  --------------------
↪--
RX-packets: 35637915       RX-dropped: 0             RX-total: 35637915
TX-packets: 35637937       TX-dropped: 0             TX-total: 35637937
----------------------------------------------------------------------------
↪--

+++++++++++++++ Accumulated forward statistics for all␣
↪ports+++++++++++++++
RX-packets: 71275820       RX-dropped: 0             RX-total: 71275820
TX-packets: 71275884       TX-dropped: 0             TX-total: 71275884
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

8.14 LiquidIO VF Poll Mode Driver

The LiquidIO VF PMD library (librte_pmd_lio) provides poll mode driver support for Cavium LiquidIO® II server adapter VFs. PF management and VF creation can be done using kernel driver.

More information can be found at Cavium Official Website.

8.14.1 Supported LiquidIO Adapters

- LiquidIO II CN2350 210SV
- LiquidIO II CN2360 210SV

8.14.2 Pre-Installation Configuration

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

- `CONFIG_RTE_LIBRTE_LIO_PMD` (default `y`)

  Toggle compilation of LiquidIO PMD.

- `CONFIG_RTE_LIBRTE_LIO_DEBUG_DRIVER` (default `n`)

  Toggle display of generic debugging messages.

- `CONFIG_RTE_LIBRTE_LIO_DEBUG_INIT` (default `n`)

  Toggle display of initialization related messages.

- `CONFIG_RTE_LIBRTE_LIO_DEBUG_RX` (default `n`)

  Toggle display of receive fast path run-time messages.

- `CONFIG_RTE_LIBRTE_LIO_DEBUG_TX` (default `n`)

  Toggle display of transmit fast path run-time messages.

- `CONFIG_RTE_LIBRTE_LIO_DEBUG_MBOX` (default `n`)

  Toggle display of mailbox messages.

- `CONFIG_RTE_LIBRTE_LIO_DEBUG_REGS` (default `n`)

  Toggle display of register reads and writes.

8.14.3 Driver Compilation

To compile LiquidIO PMD for Linux x86_64 gcc target, run the following "make" command:

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

8.14.4 Sample Application Notes

This section demonstrates how to launch `testpmd` with LiquidIO® CN23XX device managed by `librte_pmd_lio` in Linux operating system.

1. Mount huge pages:

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

2. Request huge pages:

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

3. Load `vfio-pci` driver:

   ```
   modprobe vfio-pci
   ```

4. Bind the LiquidIO VFs to `vfio-pci` loaded in previous step:

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

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

   sudo chmod 0666 /dev/vfio/*

   ./usertools/dpdk-devbind.py --bind vfio-pci 0000:03:00.3 0000:03:08.3
   ```

5. Start `testpmd` with basic parameters:

   ```
   ./build/app/testpmd -c 0xf -n 4 -- -i
   ```

   Example output:

   ```
   [...]
   EAL: PCI device 0000:03:00.3 on NUMA socket 0
   EAL:   probe driver: 177d:9712 net_liovf
   EAL:   using IOMMU type 1 (Type 1)
   PMD: net_liovf[03:00.3]INFO: DEVICE : CN23XX VF
   EAL: PCI device 0000:03:08.3 on NUMA socket 0
   EAL:   probe driver: 177d:9712 net_liovf
   PMD: net_liovf[03:08.3]INFO: DEVICE : CN23XX VF
   ```

```
Interactive-mode selected
USER1: create a new mbuf pool <mbuf_pool_socket_0>: n=171456, size=2176, socket=0
Configuring Port 0 (socket 0)
PMD: net_liovf[03:00.3]INFO: Starting port 0
Port 0: F2:A8:1B:5E:B4:66
Configuring Port 1 (socket 0)
PMD: net_liovf[03:08.3]INFO: Starting port 1
Port 1: 32:76:CC:EE:56:D7
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Port 1 Link Up - speed 10000 Mbps - full-duplex
Done
testpmd>
```

8.14.5 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: [148 v1] Alternative Routing-ID Interpretation (ARI)
   [...]
   Capabilities: [178 v1] Single Root I/O Virtualization (SR-IOV)
   [...]
   Kernel driver in use: LiquidIO
   ```

2. Load the kernel module:

   ```
   modprobe liquidio
   ```

3. Bring up the PF ports:

   ```
   ifconfig p4p1 up
   ifconfig p4p2 up
   ```

4. Change PF MTU if required:

   ```
   ifconfig p4p1 mtu 9000
   ifconfig p4p2 mtu 9000
   ```

5. Create VF device(s):

   Echo number of VFs to be created into `"sriov_numvfs"` sysfs entry of the parent PF.

   ```
   echo 1 > /sys/bus/pci/devices/0000:03:00.0/sriov_numvfs
   echo 1 > /sys/bus/pci/devices/0000:03:00.1/sriov_numvfs
   ```

6. 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 p4p1 vf 0 mac F2:A8:1B:5E:B4:66
```

7. Assign VF(s) to VM.

   The VF devices may be passed through to the guest VM using qemu or virt-manager or virsh etc.

   Example qemu guest launch command:

```
./qemu-system-x86_64 -name lio-vm -machine accel=kvm \
-cpu host -m 4096 -smp 4 \
-drive file=<disk_file>,if=none,id=disk1,format=<type> \
-device virtio-blk-pci,scsi=off,drive=disk1,id=virtio-disk1,bootindex=1 \
-device vfio-pci,host=03:00.3 -device vfio-pci,host=03:08.3
```

8. Running testpmd

   Refer *notes above* to compile and run `testpmd` application. Use `igb_uio` instead of `vfio-pci` in VM.

8.14.6 Limitations

VF MTU

VF MTU is limited by PF MTU. Raise PF value before configuring VF for larger packet size.

VLAN offload

Tx VLAN insertion is not supported and consequently VLAN offload feature is marked partial.

Ring size

Number of descriptors for Rx/Tx ring should be in the range 128 to 512.

CRC striping

LiquidIO adapters strip ethernet FCS of every packet coming to 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`.

8.15 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.

8.15.1 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.

8.15.2 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.

8.15.3 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.

8.15.4 Configuration

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.

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.

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.
- `port` parameter [int]

  This parameter provides a physical port to probe and can be specified multiple times for additional ports. All ports are probed by default if left unspecified.

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.

8.15.5 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 **4.0-1.0.1.0**.
- Firmware version **2.40.5030**.
- Supported architectures: **x86_64** and **POWER8**.

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*.

8.15.6 Supported NICs

- Mellanox(R) ConnectX(R)-3 Pro 40G MCX354A-FCC_Ax (2*40G)

8.15.7 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 -l 8-15 -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>
```

8.16 MLX5 poll mode driver

The MLX5 poll mode driver library (**librte_pmd_mlx5**) provides support for **Mellanox ConnectX-4**, **Mellanox ConnectX-4 Lx** and **Mellanox ConnectX-5** 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.

8.16.1 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.

8.16.2 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.

- RX VLAN stripping.
- TX VLAN insertion.
- RX CRC stripping configuration.
- Promiscuous mode.
- Multicast promiscuous mode.
- Hardware checksum offloads.
- Flow director (RTE_FDIR_MODE_PERFECT, RTE_FDIR_MODE_PERFECT_MAC_VLAN and RTE_ETH_FDIR_REJECT).
- Flow API.
- Secondary process TX is supported.
- KVM and VMware ESX SR-IOV modes are supported.
- RSS hash result is supported.
- Hardware TSO.
- Hardware checksum TX offload for VXLAN and GRE.

8.16.3 Limitations

- Inner RSS for VXLAN frames is not supported yet.
- Port statistics through software counters only.
- Hardware checksum RX offloads for VXLAN inner header are not supported yet.
- Secondary process RX is not supported.

8.16.4 Configuration

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_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.

Environment variables

- `MLX5_PMD_ENABLE_PADDING`

  Enables HW packet padding in PCI bus transactions.

  When packet size is cache aligned and CRC stripping is enabled, 4 fewer bytes are written to the PCI bus. Enabling padding makes such packets aligned again.

  In cases where PCI bandwidth is the bottleneck, padding can improve performance by 10%.

  This is disabled by default since this can also decrease performance for unaligned packet sizes.

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.
- `rxq_cqe_comp_en` parameter [int]

  A nonzero value enables the compression of CQE on RX side. This feature allows to save PCI bandwidth and improve performance at the cost of a slightly higher CPU usage. Enabled by default.

  Supported on:

  - x86_64 with ConnectX4 and ConnectX4 LX
  - Power8 with ConnectX4 LX

- `txq_inline` parameter [int]

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

  It is not enabled by default (set to 0) since the additional software logic necessary to handle this mode can lower performance when back pressure is not expected.

- `txqs_min_inline` parameter [int]

  Enable inline send only when the number of TX queues is greater or equal to this value.

  This option should be used in combination with `txq_inline` above.

- `txq_mpw_en` parameter [int]

  A nonzero value enables multi-packet send (MPS) for ConnectX-4 Lx and enhanced multi-packet send (Enhanced MPS) for ConnectX-5. MPS allows the TX burst function to pack up multiple packets in a single descriptor session in order to save PCI bandwidth and improve performance at the cost of a slightly higher CPU usage. When `txq_inline` is set along with `txq_mpw_en`, TX burst function tries to copy entire packet data on to TX descriptor instead of including pointer of packet only if there is enough room remained in the descriptor. `txq_inline` sets per-descriptor space for either pointers or inlined packets. In addition, Enhanced MPS supports hybrid mode - mixing inlined packets and pointers in the same descriptor.

  This option cannot be used in conjunction with `tso` below. When `tso` is set, `txq_mpw_en` is disabled.

  It is currently only supported on the ConnectX-4 Lx and ConnectX-5 families of adapters. Enabled by default.

- `txq_mpw_hdr_dseg_en` parameter [int]

  A nonzero value enables including two pointers in the first block of TX descriptor. This can be used to lessen CPU load for memory copy.

  Effective only when Enhanced MPS is supported. Disabled by default.

- `txq_max_inline_len` parameter [int]

  Maximum size of packet to be inlined. This limits the size of packet to be inlined. If the size of a packet is larger than configured value, the packet isn't inlined even though there's enough space remained in the descriptor. Instead, the packet is included with pointer.

  Effective only when Enhanced MPS is supported. The default value is 256.

- `tso` parameter [int]

  A nonzero value enables hardware TSO. When hardware TSO is enabled, packets marked with TCP segmentation offload will be divided into segments by the hardware.

  Disabled by default.

8.16.5 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/ConnectX-5 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/ConnectX-5 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/ConnectX-5 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 version: **4.0-1.0.1.0**
- firmware version:

  - ConnectX-4: **12.18.1000**
  - ConnectX-4 Lx: **14.18.1000**
  - ConnectX-5: **16.18.1000**
  - ConnectX-5 Ex: **16.18.1000**

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*.

8.16.6 Supported NICs

- Mellanox(R) ConnectX(R)-4 10G MCX4111A-XCAT (1x10G)
- Mellanox(R) ConnectX(R)-4 10G MCX4121A-XCAT (2x10G)
- Mellanox(R) ConnectX(R)-4 25G MCX4111A-ACAT (1x25G)
- Mellanox(R) ConnectX(R)-4 25G MCX4121A-ACAT (2x25G)
- Mellanox(R) ConnectX(R)-4 40G MCX4131A-BCAT (1x40G)
- Mellanox(R) ConnectX(R)-4 40G MCX413A-BCAT (1x40G)
- Mellanox(R) ConnectX(R)-4 40G MCX415A-BCAT (1x40G)
- Mellanox(R) ConnectX(R)-4 50G MCX4131A-GCAT (1x50G)
- Mellanox(R) ConnectX(R)-4 50G MCX413A-GCAT (1x50G)
- Mellanox(R) ConnectX(R)-4 50G MCX414A-BCAT (2x50G)
- Mellanox(R) ConnectX(R)-4 50G MCX415A-GCAT (2x50G)
- Mellanox(R) ConnectX(R)-4 50G MCX416A-BCAT (2x50G)
- Mellanox(R) ConnectX(R)-4 50G MCX416A-GCAT (2x50G)
- Mellanox(R) ConnectX(R)-4 50G MCX415A-CCAT (1x100G)
- Mellanox(R) ConnectX(R)-4 100G MCX416A-CCAT (2x100G)
- Mellanox(R) ConnectX(R)-4 Lx 10G MCX4121A-XCAT (2x10G)
- Mellanox(R) ConnectX(R)-4 Lx 25G MCX4121A-ACAT (2x25G)
- Mellanox(R) ConnectX(R)-5 100G MCX556A-ECAT (2x100G)
- Mellanox(R) ConnectX(R)-5 Ex EN 100G MCX516A-CDAT (2x100G)

8.16.7 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
```

8.16.8 Usage example

This section demonstrates how to launch **testpmd** with Mellanox ConnectX-4/ConnectX-5 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 -l 8-15 -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>
```

8.17 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.

8.17.1 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 the **igb_uio** UIO driver, available with DPDK, to perform correct initialization.

8.17.2 Building the software

Netronome's PMD code is provided in the **drivers/net/nfp** directory. Although NFP PMD has Netronome´s BSP dependencies, it is possible to compile it along with other DPDK PMDs even if no BSP was installed before. Of course, a DPDK app will require such a BSP installed for using the NFP PMD.

Default PMD configuration is at **common_linuxapp configuration** file:

- **CONFIG_RTE_LIBRTE_NFP_PMD=y**

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

8.17.3 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 igb_uio kernel module (manually):** This module is part of DPDK sources and configured by default (CONFIG_RTE_EAL_IGB_UIO=y).

```
modprobe igb_uio.ko
```

The module should now be listed by the lsmod command.

Depending on which NFP modules are loaded, it could be necessary to detach NFP devices from the nfp_netvf module. If this is the case the device needs to be unbound, for example:

```
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 IGB UIO driver:

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

And then to bind the device to the igb_uio driver:

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

lspci -d19ee: -k
```

lspci should show that device bound to igb_uio driver.

5. **Using scripts 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:

   - **dpdk-setup.sh**
   - **dpdk-devbind.py**

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

8.18 QEDE Poll Mode Driver

The QEDE poll mode driver library (**librte_pmd_qede**) implements support for **QLogic FastLinQ QL4xxxx 10G/25G/40G/50G/100G CNA** family of adapters as well as their virtual functions (VF) in SR-IOV context. It is supported on several standard Linux distros like RHEL7.x, SLES12.x and Ubuntu. It is compile-tested under FreeBSD OS.

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

8.18.1 Supported Features

- Unicast/Multicast filtering
- Promiscuous mode
- Allmulti mode
- Port hardware statistics
- Jumbo frames
- VLAN offload - Filtering and stripping
- Stateless checksum offloads (IPv4/TCP/UDP)
- Multiple Rx/Tx queues
- RSS (with RETA/hash table/key)
- TSS
- Multiple MAC address
- Default pause flow control
- SR-IOV VF
- MTU change
- Multiprocess aware
- Scatter-Gather
- VXLAN tunneling offload
- N-tuple filter and flow director (limited support)
- LRO/TSO

8.18.2 Non-supported Features

- SR-IOV PF
- GENEVE and NVGRE Tunneling offloads
- NPAR

8.18.3 Supported QLogic Adapters

- QLogic FastLinQ QL4xxxx 10G/25G/40G/50G/100G CNAs.

8.18.4 Prerequisites

- Requires firmware version **8.18.x.** and management firmware version **8.18.x or higher**. Firmware may be available inbox in certain newer Linux distros under the standard directory `E.g. /lib/firmware/qed/qed_init_values-8.18.9.0.bin`
- If the required firmware files are not available then visit QLogic Driver Download Center.

Performance note

- For better performance, it is recommended to use 4K or higher RX/TX rings.

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_QEDE_PMD` (default **y**)

  Toggle compilation of QEDE PMD driver.

- `CONFIG_RTE_LIBRTE_QEDE_DEBUG_INFO` (default **n**)

  Toggle display of generic debugging messages.

- `CONFIG_RTE_LIBRTE_QEDE_DEBUG_DRIVER` (default **n**)

  Toggle display of ecore related messages.

- `CONFIG_RTE_LIBRTE_QEDE_DEBUG_TX` (default **n**)

  Toggle display of transmit fast path run-time messages.

- `CONFIG_RTE_LIBRTE_QEDE_DEBUG_RX` (default **n**)

  Toggle display of receive fast path run-time messages.

- `CONFIG_RTE_LIBRTE_QEDE_FW` (default **“”**)

  Gives absolute path of firmware file. `Eg: "/lib/firmware/qed/qed_init_values_zipped-8.18.9.0.bin"` Empty string indicates driver will pick up the firmware file from the default location.

Driver Compilation

To compile QEDE 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 QEDE 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 QEDE 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 QEDE 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/gcc49 CC=gcc49
```

Sample Application Notes

This section demonstrates how to launch `testpmd` with QLogic 4xxxx devices managed by `librte_pmd_qede` in Linux operating system.

1. Request huge pages:

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

2. Load `igb_uio` driver:

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

3. Bind the QLogic 4xxxx adapters to `igb_uio` loaded in the previous step:

```
./usertools/dpdk-devbind.py --bind igb_uio 0000:84:00.0 0000:84:00.1 \
                                      0000:84:00.2 0000:84:00.3
```

4. Start `testpmd` with basic parameters: (Enable QEDE_DEBUG_INFO=y to view informational messages)

```
  testpmd -l 0,4-11 -n 4 -- -i --nb-cores=8 --portmask=0xf --rxd=4096 \
  --txd=4096 --txfreet=4068 --enable-rx-cksum --rxq=4 --txq=4 \
  --rss-ip --rss-udp

  [...]

EAL: PCI device 0000:84:00.0 on NUMA socket 1
EAL:   probe driver: 1077:1634 rte_qede_pmd
EAL:   Not managed by a supported kernel driver, skipped
EAL: PCI device 0000:84:00.1 on NUMA socket 1
EAL:   probe driver: 1077:1634 rte_qede_pmd
EAL:   Not managed by a supported kernel driver, skipped
EAL: PCI device 0000:88:00.0 on NUMA socket 1
EAL:   probe driver: 1077:1656 rte_qede_pmd
EAL:   PCI memory mapped at 0x7f738b200000
EAL:   PCI memory mapped at 0x7f738b280000
EAL:   PCI memory mapped at 0x7f738b300000
PMD: Chip details : BB1
PMD: Driver version : QEDE PMD 8.7.9.0_1.0.0
PMD: Firmware version : 8.7.7.0
PMD: Management firmware version : 8.7.8.0
PMD: Firmware file : /lib/firmware/qed/qed_init_values_zipped-8.7.7.0.bin
[QEDE PMD: (84:00.0:dpdk-port-0)]qede_common_dev_init:macaddr \
                                                    00:0e:1e:d2:09:9c
  [...]
[QEDE PMD: (84:00.0:dpdk-port-0)]qede_tx_queue_setup:txq 0 num_desc 4096 \
                                            tx_free_thresh 4068 socket 0
[QEDE PMD: (84:00.0:dpdk-port-0)]qede_tx_queue_setup:txq 1 num_desc 4096 \
                                            tx_free_thresh 4068 socket 0
[QEDE PMD: (84:00.0:dpdk-port-0)]qede_tx_queue_setup:txq 2 num_desc 4096 \
                                             tx_free_thresh 4068 socket 0
[QEDE PMD: (84:00.0:dpdk-port-0)]qede_tx_queue_setup:txq 3 num_desc 4096 \
                                             tx_free_thresh 4068 socket 0
[QEDE PMD: (84:00.0:dpdk-port-0)]qede_rx_queue_setup:rxq 0 num_desc 4096 \
                                            rx_buf_size=2148 socket 0
[QEDE PMD: (84:00.0:dpdk-port-0)]qede_rx_queue_setup:rxq 1 num_desc 4096 \
                                            rx_buf_size=2148 socket 0
[QEDE PMD: (84:00.0:dpdk-port-0)]qede_rx_queue_setup:rxq 2 num_desc 4096 \
```

```
                                                rx_buf_size=2148 socket 0
[QEDE PMD: (84:00.0:dpdk-port-0)]qede_rx_queue_setup:rxq 3 num_desc 4096 \
                                                rx_buf_size=2148 socket 0
[QEDE PMD: (84:00.0:dpdk-port-0)]qede_dev_start:port 0
[QEDE PMD: (84:00.0:dpdk-port-0)]qede_dev_start:link status: down
  [...]
Checking link statuses...
Port 0 Link Up - speed 25000 Mbps - full-duplex
Port 1 Link Up - speed 25000 Mbps - full-duplex
Port 2 Link Up - speed 25000 Mbps - full-duplex
Port 3 Link Up - speed 25000 Mbps - full-duplex
Done
testpmd>
```

SR-IOV: Prerequisites and Sample Application Notes

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

Note: librte_pmd_qede will be used to bind to SR-IOV VF device and Linux native kernel driver (QEDE) will function as SR-IOV PF driver. Requires PF driver to be 8.10.x.x or higher.

1. Verify SR-IOV and ARI capability is 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 qede
   ```

   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`. QEDE PMD should be used to bind the VF devices in the guest VM using the instructions outlined in the Application notes above.

8.19 Solarflare libefx-based Poll Mode Driver

The SFC EFX PMD (**librte_pmd_sfc_efx**) provides poll mode driver support for **Solarflare SFN7xxx and SFN8xxx** family of 10/40 Gbps adapters. SFC EFX PMD has support for the latest Linux and FreeBSD operating systems.

More information can be found at Solarflare Communications website.

8.19.1 Features

SFC EFX PMD has support for:

- Multiple transmit and receive queues
- Link state information including link status change interrupt
- IPv4/IPv6 TCP/UDP transmit checksum offload
- Port hardware statistics
- Extended statistics (see Solarflare Server Adapter User's Guide for the statistics description)
- Basic flow control
- MTU update
- Jumbo frames up to 9K
- Promiscuous mode
- Allmulticast mode
- TCP segmentation offload (TSO)
- Multicast MAC filter
- IPv4/IPv6 TCP/UDP receive checksum offload
- Received packet type information
- Receive side scaling (RSS)
- RSS hash
- Scattered Rx DMA for packet that are larger that a single Rx descriptor
- Deferred receive and transmit queue start
- Transmit VLAN insertion (if running firmware variant supports it)
- Flow API

8.19.2 Non-supported Features

The features not yet supported include:

- Receive queue interupts
- Priority-based flow control
- Loopback
- Configurable RX CRC stripping (always stripped)
- Header split on receive
- VLAN filtering
- VLAN stripping
- LRO

8.19.3 Limitations

Due to requirements on receive buffer alignment and usage of the receive buffer for the auxiliary packet information provided by the NIC up to extra 269 (14 bytes prefix plus up to 255 bytes for end padding) bytes may be required in the receive buffer. It should be taken into account when mbuf pool for receive is created.

8.19.4 Flow API support

Supported attributes:

- Ingress

Supported pattern items:

- VOID
- ETH (exact match of source/destination addresses, individual/group match of destination address, EtherType)
- VLAN (exact match of VID, double-tagging is supported)
- IPV4 (exact match of source/destination addresses, IP transport protocol)
- IPV6 (exact match of source/destination addresses, IP transport protocol)
- TCP (exact match of source/destination ports)
- UDP (exact match of source/destination ports)

Supported actions:

- VOID
- QUEUE

Validating flow rules depends on the firmware variant.

Ethernet destinaton individual/group match

Ethernet item supports I/G matching, if only the corresponding bit is set in the mask of destination address. If destinaton address in the spec is multicast, it matches all multicast (and broadcast) packets, oherwise it matches unicast packets that are not filtered by other flow rules.

8.19.5 Supported NICs

- Solarflare Flareon [Ultra] Server Adapters:
  - Solarflare SFN8522 Dual Port SFP+ Server Adapter
  - Solarflare SFN8542 Dual Port QSFP+ Server Adapter
  - Solarflare SFN7002F Dual Port SFP+ Server Adapter
  - Solarflare SFN7004F Quad Port SFP+ Server Adapter
  - Solarflare SFN7042Q Dual Port QSFP+ Server Adapter
  - Solarflare SFN7122F Dual Port SFP+ Server Adapter
  - Solarflare SFN7124F Quad Port SFP+ Server Adapter
  - Solarflare SFN7142Q Dual Port QSFP+ Server Adapter
  - Solarflare SFN7322F Precision Time Synchronization Server Adapter

8.19.6 Prerequisites

- Requires firmware version:
  - SFN7xxx: **4.7.1.1001** or higher
  - SFN8xxx: **6.0.2.1004** or higher

Visit Solarflare Support Downloads to get Solarflare Utilities (either Linux or FreeBSD) with the latest firmware. Follow instructions from Solarflare Server Adapter User's Guide to update firmware and configure the adapter.

8.19.7 Pre-Installation Configuration

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_SFC_EFX_PMD` (default **y**)

  Enable compilation of Solarflare libefx-based poll-mode driver.

- `CONFIG_RTE_LIBRTE_SFC_EFX_DEBUG` (default **n**)

  Enable compilation of the extra run-time consistency checks.

Per-Device Parameters

The following per-device parameters can be passed via EAL PCI device whitelist option like "-w 02:00.0,arg1=value1,...".

Case-insensitive 1/y/yes/on or 0/n/no/off may be used to specify boolean parameters value.

- `rx_datapath` [auto|efx|ef10] (default **auto**)

  Choose receive datapath implementation. **auto** allows the driver itself to make a choice based on firmware features available and required by the datapath implementation. **efx** chooses libefx-based datapath which supports Rx scatter. **ef10** chooses EF10 (SFN7xxx, SFN8xxx) native datapath which is more efficient than libefx-based and provides richer packet type classification, but lacks Rx scatter support.

- `tx_datapath` [auto|efx|ef10|ef10_simple] (default **auto**)

  Choose transmit datapath implementation. **auto** allows the driver itself to make a choice based on firmware features available and required by the datapath implementation. **efx** chooses libefx-based datapath which supports VLAN insertion (full-feature firmware variant only), TSO and multi-segment mbufs. **ef10** chooses EF10 (SFN7xxx, SFN8xxx) native datapath which is more efficient than libefx-based but has no VLAN insertion and TSO support yet. **ef10_simple** chooses EF10 (SFN7xxx, SFN8xxx) native datapath which is even more faster then **ef10** but does not support multi-segment mbufs.

- `perf_profile` [auto|throughput|low-latency] (default **throughput**)

  Choose hardware tunning to be optimized for either throughput or low-latency. **auto** allows NIC firmware to make a choice based on installed licences and firmware variant configured using **sfboot**.

- `debug_init` [bool] (default **n**)

  Enable extra logging during device intialization and startup.

- `mcdi_logging` [bool] (default **n**)

  Enable extra logging of the communication with the NIC's management CPU. The logging is done using RTE_LOG() with INFO level and PMD type. The format is consumed by the Solarflare netlogdecode cross-platform tool.

- `stats_update_period_ms` [long] (default **1000**)

  Adjust period in milliseconds to update port hardware statistics. The accepted range is 0 to 65535. The value of **0** may be used to disable periodic statistics update. One should note that it's only possible to set an arbitrary value on SFN8xxx provided that firmware version is 6.2.1.1033 or higher, otherwise any positive value will select a fixed update period of **1000** milliseconds

8.20 SZEDATA2 poll mode driver library

The SZEDATA2 poll mode driver library implements support for the Netcope FPGA Boards (**NFB-***), FPGA-based programmable NICs. The SZEDATA2 PMD uses interface provided by the libsze2 library to communicate with the NFB cards over the sze2 layer.

More information about the NFB cards and used technology (Netcope Development Kit) can be found on the Netcope Technologies 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.

8.20.1 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.

8.20.2 Configuration

These configuration options can be modified before compilation in the `.config` file:

- `CONFIG_RTE_LIBRTE_PMD_SZEDATA2` default value: **n**

  Value **y** enables compilation of szedata2 PMD.

- `CONFIG_RTE_LIBRTE_PMD_SZEDATA2_AS` default value: **0**

  This option defines type of firmware address space. Currently supported value is:
  - **0** for firmwares:
    * NIC_100G1_LR4
    * HANIC_100G1_LR4
    * HANIC_100G1_SR10

8.20.3 Using the SZEDATA2 PMD

From DPDK version 16.04 the type of SZEDATA2 PMD is changed to PMD_PDEV. SZEDATA2 device is automatically recognized during EAL initialization. No special command line options are needed.

Kernel modules have to be loaded before running the DPDK application.

8.20.4 Example of usage

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

```
$RTE_TARGET/app/testpmd -l 0-3 -n 2 \
-- --port-topology=chained --rxq=2 --txq=2 --nb-cores=2 -i -a
```

Example output:

```
[...]
EAL: PCI device 0000:06:00.0 on NUMA socket -1
EAL:   probe driver: 1b26:c1c1 rte_szedata2_pmd
PMD: Initializing szedata2 device (0000:06:00.0)
PMD: SZEDATA2 path: /dev/szedataII0
PMD: Available DMA channels RX: 8 TX: 8
PMD: resource0 phys_addr = 0xe8000000 len = 134217728 virt addr = 7f48f8000000
PMD: szedata2 device (0000:06:00.0) successfully initialized
Interactive-mode selected
```

```
Auto-start selected
Configuring Port 0 (socket 0)
Port 0: 00:11:17:00:00:00
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Done
Start automatic packet forwarding
  io packet forwarding - CRC stripping disabled - packets/burst=32
  nb forwarding cores=2 - nb forwarding ports=1
  RX queues=2 - RX desc=128 - RX free threshold=0
  RX threshold registers: pthresh=0 hthresh=0 wthresh=0
  TX queues=2 - TX desc=512 - TX free threshold=0
  TX threshold registers: pthresh=0 hthresh=0 wthresh=0
  TX RS bit threshold=0 - TXQ flags=0x0
testpmd>
```

8.21 Tun/Tap Poll Mode Driver

The `rte_eth_tap.c` PMD creates a device using TUN/TAP interfaces on the local host. The PMD allows for DPDK and the host to communicate using a raw device interface on the host and in the DPDK application.

The device created is a TAP device, which sends/receives packet in a raw format with a L2 header. The usage for a TAP PMD is for connectivity to the local host using a TAP interface. When the TAP PMD is initialized it will create a number of tap devices in the host accessed via `ifconfig -a` or `ip` command. The commands can be used to assign and query the virtual like device.

These TAP interfaces can be used with Wireshark or tcpdump or Pktgen-DPDK along with being able to be used as a network connection to the DPDK application. The method enable one or more interfaces is to use the `--vdev=net_tap0` option on the DPDK application command line. Each `--vdev=net_tap1` option give will create an interface named dtap0, dtap1, and so on.

The interface name can be changed by adding the `iface=foo0`, for example:

```
--vdev=net_tap0,iface=foo0 --vdev=net_tap1,iface=foo1, ...
```

Also the speed of the interface can be changed from 10G to whatever number needed, but the interface does not enforce that speed, for example:

```
--vdev=net_tap0,iface=foo0,speed=25000
```

It is possible to specify a remote netdevice to capture packets from by adding `remote=foo1`, for example:

```
--vdev=net_tap,iface=tap0,remote=foo1
```

If a `remote` is set, the tap MAC address will be set to match the remote one just after netdevice creation. Using TC rules, traffic from the remote netdevice will be redirected to the tap. If the tap is in promiscuous mode, then all packets will be redirected. In allmulti mode, all multicast packets will be redirected.

Using the remote feature is especially useful for capturing traffic from a netdevice that has no support in the DPDK. It is possible to add explicit rte_flow rules on the tap PMD to capture specific traffic (see next section for examples).

After the DPDK application is started you can send and receive packets on the interface using the standard rx_burst/tx_burst APIs in DPDK. From the host point of view you can use any host tool like tcpdump, Wireshark, ping, Pktgen and others to communicate with the DPDK application. The DPDK application may not understand network protocols like IPv4/6, UDP or TCP unless the application has been written to understand these protocols.

If you need the interface as a real network interface meaning running and has a valid IP address then you can do this with the following commands:

```
sudo ip link set dtap0 up; sudo ip addr add 192.168.0.250/24 dev dtap0
sudo ip link set dtap1 up; sudo ip addr add 192.168.1.250/24 dev dtap1
```

Please change the IP addresses as you see fit.

If routing is enabled on the host you can also communicate with the DPDK App over the internet via a standard socket layer application as long as you account for the protocol handing in the application.

If you have a Network Stack in your DPDK application or something like it you can utilize that stack to handle the network protocols. Plus you would be able to address the interface using an IP address assigned to the internal interface.

8.21.1 Flow API support

The tap PMD supports major flow API pattern items and actions, when running on linux kernels above 4.2 ("Flower" classifier required). Supported items:

- eth: src and dst (with variable masks), and eth_type (0xffff mask).
- vlan: vid, pcp, tpid, but not eid. (requires kernel 4.9)
- ipv4/6: src and dst (with variable masks), and ip_proto (0xffff mask).
- udp/tcp: src and dst port (0xffff) mask.

Supported actions:

- DROP
- QUEUE
- PASSTHRU

It is generally not possible to provide a "last" item. However, if the "last" item, once masked, is identical to the masked spec, then it is supported.

Only IPv4/6 and MAC addresses can use a variable mask. All other items need a full mask (exact match).

As rules are translated to TC, it is possible to show them with something like:

```
tc -s filter show dev tap1 parent 1:
```

Examples of testpmd flow rules

Drop packets for destination IP 192.168.0.1:

```
testpmd> flow create 0 priority 1 ingress pattern eth / ipv4 dst is 1.1.1.1 \
         / end actions drop / end
```

Ensure packets from a given MAC address are received on a queue 2:

```
testpmd> flow create 0 priority 2 ingress pattern eth src is 06:05:04:03:02:01 \
         / end actions queue index 2 / end
```

Drop UDP packets in vlan 3:

```
testpmd> flow create 0 priority 3 ingress pattern eth / vlan vid is 3 / \
         ipv4 proto is 17 / end actions drop / end
```

8.21.2 Example

The following is a simple example of using the TUN/TAP PMD with the Pktgen packet generator. It requires that the `socat` utility is installed on the test system.

Build DPDK, then pull down Pktgen and build pktgen using the DPDK SDK/Target used to build the dpdk you pulled down.

Run pktgen from the pktgen directory in a terminal with a commandline like the following:

```
sudo ./app/app/x86_64-native-linuxapp-gcc/app/pktgen -l 1-5 -n 4        \
 --proc-type auto --log-level 8 --socket-mem 512,512 --file-prefix pg   \
 --vdev=net_tap0 --vdev=net_tap1 -b 05:00.0 -b 05:00.1                  \
 -b 04:00.0 -b 04:00.1 -b 04:00.2 -b 04:00.3                            \
 -b 81:00.0 -b 81:00.1 -b 81:00.2 -b 81:00.3                            \
 -b 82:00.0 -b 83:00.0 -- -T -P -m [2:3].0 -m [4:5].1                   \
 -f themes/black-yellow.theme
```

Verify with `ifconfig -a` command in a different xterm window, should have a `dtap0` and `dtap1` interfaces created.

Next set the links for the two interfaces to up via the commands below:

```
sudo ip link set dtap0 up; sudo ip addr add 192.168.0.250/24 dev dtap0
sudo ip link set dtap1 up; sudo ip addr add 192.168.1.250/24 dev dtap1
```

Then use socat to create a loopback for the two interfaces:

```
sudo socat interface:dtap0 interface:dtap1
```

Then on the Pktgen command line interface you can start sending packets using the commands `start 0` and `start 1` or you can start both at the same time with `start all`. The command `str` is an alias for `start all` and `stp` is an alias for `stop all`.

While running you should see the 64 byte counters increasing to verify the traffic is being looped back. You can use `set all size XXX` to change the size of the packets after you stop the traffic. Use pktgen `help` command to see a list of all commands. You can also use the `-f` option to load commands at startup in command line or Lua script in pktgen.

8.22 ThunderX NICVF Poll Mode Driver

The ThunderX NICVF PMD (**librte_pmd_thunderx_nicvf**) provides poll mode driver support for the inbuilt NIC found in the **Cavium ThunderX** SoC family as well as their virtual functions (VF) in SR-IOV context.

More information can be found at Cavium Networks Official Website.

8.22.1 Features

Features of the ThunderX PMD are:

- Multiple queues for TX and RX

- Receive Side Scaling (RSS)
- Packet type information
- Checksum offload
- Promiscuous mode
- Multicast mode
- Port hardware statistics
- Jumbo frames
- Link state information
- Scattered and gather for TX and RX
- VLAN stripping
- SR-IOV VF
- NUMA support
- Multi queue set support (up to 96 queues (12 queue sets)) per port

8.22.2 Supported ThunderX SoCs

- CN88xx
- CN81xx
- CN83xx

8.22.3 Prerequisites

- Follow the DPDK *Getting Started Guide for Linux* to setup the basic DPDK environment.

8.22.4 Pre-Installation Configuration

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_THUNDERX_NICVF_PMD` (default `y`)

  Toggle compilation of the `librte_pmd_thunderx_nicvf` driver.

- `CONFIG_RTE_LIBRTE_THUNDERX_NICVF_DEBUG_INIT` (default `n`)

  Toggle display of initialization related messages.

- `CONFIG_RTE_LIBRTE_THUNDERX_NICVF_DEBUG_RX` (default `n`)

  Toggle display of receive fast path run-time message

- `CONFIG_RTE_LIBRTE_THUNDERX_NICVF_DEBUG_TX` (default `n`)

  Toggle display of transmit fast path run-time message

- `CONFIG_RTE_LIBRTE_THUNDERX_NICVF_DEBUG_DRIVER` (default `n`)

  Toggle display of generic debugging messages

- `CONFIG_RTE_LIBRTE_THUNDERX_NICVF_DEBUG_MBOX` (default `n`)

  Toggle display of PF mailbox related run-time check messages

Driver Compilation

To compile the ThunderX NICVF PMD for Linux arm64 gcc target, run the following "make" command:

```
cd <DPDK-source-directory>
make config T=arm64-thunderx-linuxapp-gcc install
```

8.22.5 Linux

Running testpmd

This section demonstrates how to launch `testpmd` with ThunderX NIC VF device managed by `librte_pmd_thunderx_nicvf` in the Linux operating system.

1. Load `vfio-pci` driver:

   ```
   modprobe vfio-pci
   ```

2. Enable **VFIO-NOIOMMU** mode (optional):

   ```
   echo 1 > /sys/module/vfio/parameters/enable_unsafe_noiommu_mode
   ```

   Note: **VFIO-NOIOMMU** is required only when running in VM context and should not be enabled otherwise. See also *SR-IOV: Prerequisites and sample Application Notes*.

3. Bind the ThunderX NIC VF device to `vfio-pci` loaded in the previous step:

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

   ```
   ./usertools/dpdk-devbind.py --bind vfio-pci 0002:01:00.2
   ```

4. Start `testpmd` with basic parameters:

   ```
   ./arm64-thunderx-linuxapp-gcc/app/testpmd -l 0-3 -n 4 -w 0002:01:00.2 \
     -- -i --disable-hw-vlan-filter --no-flush-rx \
     --port-topology=loop
   ```

   Example output:

   ```
   ...

   PMD: rte_nicvf_pmd_init(): librte_pmd_thunderx nicvf version 1.0

   ...
   EAL:   probe driver: 177d:11 rte_nicvf_pmd
   EAL:   using IOMMU type 1 (Type 1)
   EAL:   PCI memory mapped at 0x3ffade50000
   ```

```
EAL: Trying to map BAR 4 that contains the MSI-X table.
     Trying offsets: 0x40000000000:0x0000, 0x10000:0x1f0000
EAL:   PCI memory mapped at 0x3ffadc60000
PMD: nicvf_eth_dev_init(): nicvf: device (177d:11) 2:1:0:2
PMD: nicvf_eth_dev_init(): node=0 vf=1 mode=tns-bypass sqs=false
     loopback_supported=true
PMD: nicvf_eth_dev_init(): Port 0 (177d:11) mac=a6:c6:d9:17:78:01
Interactive-mode selected
Configuring Port 0 (socket 0)
...

PMD: nicvf_dev_configure(): Configured ethdev port0 hwcap=0x0
Port 0: A6:C6:D9:17:78:01
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Done
testpmd>
```

SR-IOV: Prerequisites and sample Application Notes

Current ThunderX NIC PF/VF kernel modules maps each physical Ethernet port automatically to virtual function (VF) and presented them as PCIe-like SR-IOV device. This section provides instructions to configure SR-IOV with Linux OS.

1. Verify PF devices capabilities using `lspci`:

```
lspci -vvv
```

Example output:

```
0002:01:00.0 Ethernet controller: Cavium Networks Device a01e (rev 01)
...
Capabilities: [100 v1] Alternative Routing-ID Interpretation (ARI)
...
Capabilities: [180 v1] Single Root I/O Virtualization (SR-IOV)
...
Kernel driver in use: thunder-nic
...
```

Note: Unless `thunder-nic` driver is in use make sure your kernel config includes `CONFIG_THUNDER_NIC_PF` setting.

2. Verify VF devices capabilities and drivers using `lspci`:

```
lspci -vvv
```

Example output:

```
0002:01:00.1 Ethernet controller: Cavium Networks Device 0011 (rev 01)
...
Capabilities: [100 v1] Alternative Routing-ID Interpretation (ARI)
...
Kernel driver in use: thunder-nicvf
...
```

```
0002:01:00.2 Ethernet controller: Cavium Networks Device 0011 (rev 01)
...
Capabilities: [100 v1] Alternative Routing-ID Interpretation (ARI)
...
Kernel driver in use: thunder-nicvf
...
```

Note: Unless `thunder-nicvf` driver is in use make sure your kernel config includes `CONFIG_THUNDER_NIC_VF` setting.

3. Verify PF/VF bind using `dpdk-devbind.py`:

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

Example output:

```
...
0002:01:00.0 'Device a01e' if= drv=thunder-nic unused=vfio-pci
0002:01:00.1 'Device 0011' if=eth0 drv=thunder-nicvf unused=vfio-pci
0002:01:00.2 'Device 0011' if=eth1 drv=thunder-nicvf unused=vfio-pci
...
```

4. Load `vfio-pci` driver:

```
modprobe vfio-pci
```

5. Bind VF devices to `vfio-pci` using `dpdk-devbind.py`:

```
./usertools/dpdk-devbind.py --bind vfio-pci 0002:01:00.1
./usertools/dpdk-devbind.py --bind vfio-pci 0002:01:00.2
```

6. Verify VF bind using `dpdk-devbind.py`:

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

Example output:

```
...
0002:01:00.1 'Device 0011' drv=vfio-pci unused=
0002:01:00.2 'Device 0011' drv=vfio-pci unused=
...
0002:01:00.0 'Device a01e' if= drv=thunder-nic unused=vfio-pci
...
```

7. Pass VF device to VM context (PCIe Passthrough):

The VF devices may be passed through to the guest VM using qemu or virt-manager or virsh etc. `librte_pmd_thunderx_nicvf` or `thunder-nicvf` should be used to bind the VF devices in the guest VM in *VFIO-NOIOMMU* mode.

Example qemu guest launch command:

```
sudo qemu-system-aarch64 -name vm1 \
-machine virt,gic_version=3,accel=kvm,usb=off \
-cpu host -m 4096 \
-smp 4,sockets=1,cores=8,threads=1 \
```

```
-nographic -nodefaults \
-kernel <kernel image> \
-append "root=/dev/vda console=ttyAMA0 rw hugepagesz=512M hugepages=3" \
-device vfio-pci,host=0002:01:00.1 \
-drive file=<rootfs.ext3>,if=none,id=disk1,format=raw  \
-device virtio-blk-device,scsi=off,drive=disk1,id=virtio-disk1,bootindex=1 \
-netdev tap,id=net0,ifname=tap0,script=/etc/qemu-ifup_thunder \
-device virtio-net-device,netdev=net0 \
-serial stdio \
-mem-path /dev/huge
```

8. Refer to section *Running testpmd* for instruction how to launch `testpmd` application.

Multiple Queue Set per DPDK port configuration

There are two types of VFs:

- Primary VF
- Secondary VF

Each port consists of a primary VF and n secondary VF(s). Each VF provides 8 Tx/Rx queues to a port. When a given port is configured to use more than 8 queues, it requires one (or more) secondary VF. Each secondary VF adds 8 additional queues to the queue set.

During PMD driver initialization, the primary VF's are enumerated by checking the specific flag (see sqs message in DPDK boot log - sqs indicates secondary queue set). They are at the beginning of VF list (the remain ones are secondary VF's).

The primary VFs are used as master queue sets. Secondary VFs provide additional queue sets for primary ones. If a port is configured for more then 8 queues than it will request for additional queues from secondary VFs.

Secondary VFs cannot be shared between primary VFs.

Primary VFs are present on the beginning of the 'Network devices using kernel driver' list, secondary VFs are on the remaining on the remaining part of the list.

Note: The VNIC driver in the multiqueue setup works differently than other drivers like *ixgbe*. We need to bind separately each specific queue set device with the `usertools/dpdk-devbind.py` utility.

Note: Depending on the hardware used, the kernel driver sets a threshold `vf_id`. VFs that try to attached with an id below or equal to this boundary are considered primary VFs. VFs that try to attach with an id above this boundary are considered secondary VFs.

Example device binding

If a system has three interfaces, a total of 18 VF devices will be created on a non-NUMA machine.

Note: NUMA systems have 12 VFs per port and non-NUMA 6 VFs per port.

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

Network devices using DPDK-compatible driver
============================================
<none>

Network devices using kernel driver
===================================
0000:01:10.0 'Device a026' if= drv=thunder-BGX unused=vfio-pci,uio_pci_
↪generic
0000:01:10.1 'Device a026' if= drv=thunder-BGX unused=vfio-pci,uio_pci_
↪generic
0002:01:00.0 'Device a01e' if= drv=thunder-nic unused=vfio-pci,uio_pci_
↪generic
0002:01:00.1 'Device 0011' if=eth0 drv=thunder-nicvf unused=vfio-pci,uio_pci_
↪generic
0002:01:00.2 'Device 0011' if=eth1 drv=thunder-nicvf unused=vfio-pci,uio_pci_
↪generic
0002:01:00.3 'Device 0011' if=eth2 drv=thunder-nicvf unused=vfio-pci,uio_pci_
↪generic
0002:01:00.4 'Device 0011' if= drv=thunder-nicvf unused=vfio-pci,uio_pci_
↪generic
0002:01:00.5 'Device 0011' if= drv=thunder-nicvf unused=vfio-pci,uio_pci_
↪generic
0002:01:00.6 'Device 0011' if= drv=thunder-nicvf unused=vfio-pci,uio_pci_
↪generic
0002:01:00.7 'Device 0011' if= drv=thunder-nicvf unused=vfio-pci,uio_pci_
↪generic
0002:01:01.0 'Device 0011' if= drv=thunder-nicvf unused=vfio-pci,uio_pci_
↪generic
0002:01:01.1 'Device 0011' if= drv=thunder-nicvf unused=vfio-pci,uio_pci_
↪generic
0002:01:01.2 'Device 0011' if= drv=thunder-nicvf unused=vfio-pci,uio_pci_
↪generic
0002:01:01.3 'Device 0011' if= drv=thunder-nicvf unused=vfio-pci,uio_pci_
↪generic
0002:01:01.4 'Device 0011' if= drv=thunder-nicvf unused=vfio-pci,uio_pci_
↪generic
0002:01:01.5 'Device 0011' if= drv=thunder-nicvf unused=vfio-pci,uio_pci_
↪generic
0002:01:01.6 'Device 0011' if= drv=thunder-nicvf unused=vfio-pci,uio_pci_
↪generic
0002:01:01.7 'Device 0011' if= drv=thunder-nicvf unused=vfio-pci,uio_pci_
↪generic
0002:01:02.0 'Device 0011' if= drv=thunder-nicvf unused=vfio-pci,uio_pci_
↪generic
0002:01:02.1 'Device 0011' if= drv=thunder-nicvf unused=vfio-pci,uio_pci_
↪generic
0002:01:02.2 'Device 0011' if= drv=thunder-nicvf unused=vfio-pci,uio_pci_
↪generic

Other network devices
=====================
0002:00:03.0 'Device a01f' unused=vfio-pci,uio_pci_generic
```

We want to bind two physical interfaces with 24 queues each device, we attach two primary VFs and four secondary queues. In our example we choose two 10G interfaces eth1 (0002:01:00.2) and eth2 (0002:01:00.3). We will choose four secondary queue sets from the ending of the list (0002:01:01.7-0002:01:02.2).

1. Bind two primary VFs to the `vfio-pci` driver:

```
usertools/dpdk-devbind.py -b vfio-pci 0002:01:00.2
usertools/dpdk-devbind.py -b vfio-pci 0002:01:00.3
```

2. Bind four primary VFs to the `vfio-pci` driver:

```
usertools/dpdk-devbind.py -b vfio-pci 0002:01:01.7
usertools/dpdk-devbind.py -b vfio-pci 0002:01:02.0
usertools/dpdk-devbind.py -b vfio-pci 0002:01:02.1
usertools/dpdk-devbind.py -b vfio-pci 0002:01:02.2
```

The nicvf thunderx driver will make use of attached secondary VFs automatically during the interface configuration stage.

8.22.6 Limitations

CRC striping

The ThunderX SoC family 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`.

Maximum packet length

The ThunderX SoC family NICs support a maximum of a 9K 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 9200, frames up to 9200 bytes can still reach the host interface.

Maximum packet segments

The ThunderX SoC family NICs support up to 12 segments per packet when working in scatter/gather mode. So, setting MTU will result with `EINVAL` when the frame size does not fit in the maximum number of segments.

8.23 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.

8.23.1 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.

8.23.2 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 Rx interrupt (so far, only support 1:1 mapping for queue/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.

8.23.3 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.

8.23.4 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
```

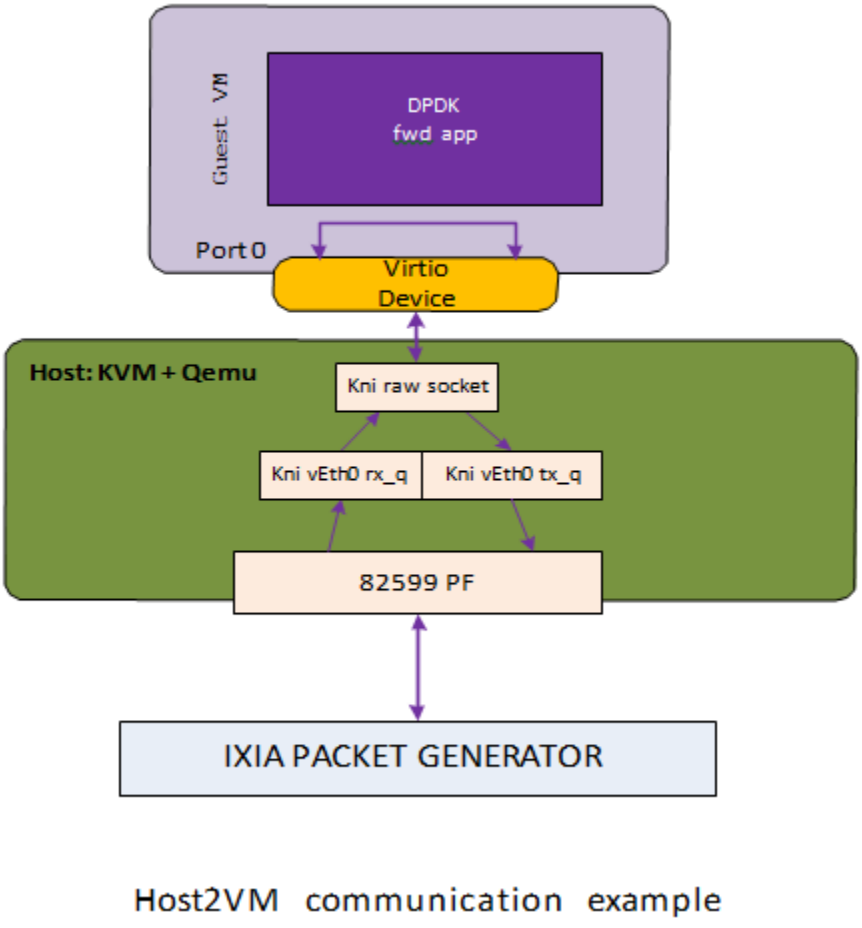


Fig. 8.5: Host2VM Communication Example Using kni vhost Back End

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 -l 0-3 -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.

For more details about kni, please refer to 内核网络接口卡接口.

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 usertools/dpdk-devbind.py -b uio_pci_generic 00:03.0
```

We use testpmd as the forwarding application in this example.

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

8.23.5 Virtio with qemu virtio Back End

```
qemu-system-x86_64 -enable-kvm -cpu host -m 2048 -smp 2 -mem-path /dev/
hugepages -mem-prealloc
-drive file=/data/DPDKVMS/dpdk-vm1
-netdev tap,id=vm1_p1,ifname=tap0,script=no,vhost=on
```

```
[root@localhost isg_cid-dpdk]# x86_64-default-linuxapp-gcc/app/testpmd  -c f -n
4 -- -i
Interactive-mode selected
Configuring Port 0 (socket -1)
Warning: nb_desc(512) is not equal to vq size (256), fall to vq size
test1
test2
test3
test4
Warning: nb_desc(128) isn't equal to  vq size (256), fall to vq size
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Done
testpmd> start _
```

Fig. 8.6: Running testpmd

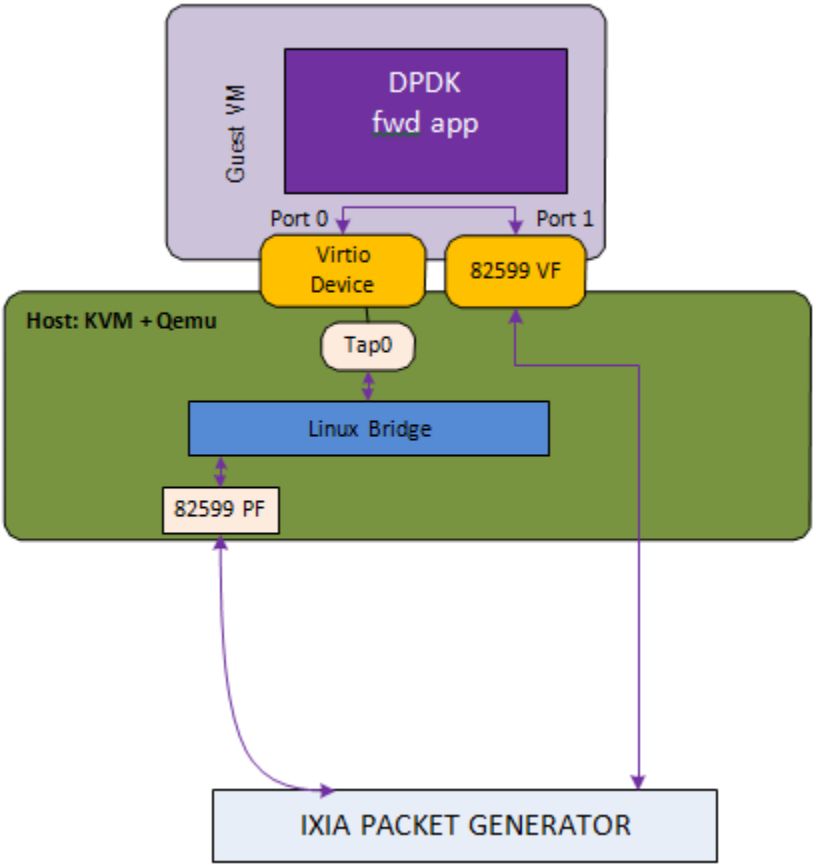


Fig. 8.7: Host2VM Communication Example Using qemu vhost Back End

```
-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

8.23.6 Virtio PMD Rx/Tx Callbacks

Virtio driver has 3 Rx callbacks and 2 Tx callbacks.

Rx callbacks:

1. `virtio_recv_pkts`: Regular version without mergeable Rx buffer support.
2. `virtio_recv_mergeable_pkts`: Regular version with mergeable Rx buffer support.
3. `virtio_recv_pkts_vec`: Vector version without mergeable Rx buffer support, also fixes the available ring indexes and uses vector instructions to optimize performance.

Tx callbacks:

1. `virtio_xmit_pkts`: Regular version.
2. `virtio_xmit_pkts_simple`: Vector version fixes the available ring indexes to optimize performance.

By default, the non-vector callbacks are used:

- For Rx: If mergeable Rx buffers is disabled then `virtio_recv_pkts` is used; otherwise `virtio_recv_mergeable_pkts`.
- For Tx: `virtio_xmit_pkts`.

Vector callbacks will be used when:

- `txq_flags` is set to `VIRTIO_SIMPLE_FLAGS` (0xF01), which implies:
  - Single segment is specified.
  - No offload support is needed.
- Mergeable Rx buffers is disabled.

The corresponding callbacks are:

- For Rx: `virtio_recv_pkts_vec`.
- For Tx: `virtio_xmit_pkts_simple`.

Example of using the vector version of the virtio poll mode driver in `testpmd`:

```
testpmd -l 0-2 -n 4 -- -i --txqflags=0xF01 --rxq=1 --txq=1 --nb-cores=1
```

8.23.7 Interrupt mode

There are three kinds of interrupts from a virtio device over PCI bus: config interrupt, Rx interrupts, and Tx interrupts. Config interrupt is used for notification of device configuration changes, especially link status (lsc). Interrupt mode is translated into Rx interrupts in the context of DPDK.

Prerequisites for Rx interrupts

To support Rx interrupts, #. Check if guest kernel supports VFIO-NOIOMMU:

Linux started to support VFIO-NOIOMMU since 4.8.0. Make sure the guest kernel is compiled with:

```
CONFIG_VFIO_NOIOMMU=y
```

1. Properly set msix vectors when starting VM:

   Enable multi-queue when starting VM, and specify msix vectors in qemu cmdline. (N+1) is the minimum, and (2N+2) is mostly recommended.

   ```
   $(QEMU) ... -device virtio-net-pci,mq=on,vectors=2N+2 ...
   ```

2. In VM, insert vfio module in NOIOMMU mode:

   ```
   modprobe vfio enable_unsafe_noiommu_mode=1
   modprobe vfio-pci
   ```

3. In VM, bind the virtio device with vfio-pci:

   ```
   python tools/dpdk-devbind.py -b vfio-pci 00:03.0
   ```

Example

Here we use l3fwd-power as an example to show how to get started.

Example:

```
$ l3fwd-power -l 0-1 -- -p 1 -P --config="(0,0,1)" \
                                    --no-numa --parse-ptype
```

8.24 Poll Mode Driver that wraps vhost library

This PMD is a thin wrapper of the DPDK vhost library. The user can handle virtqueues as one of normal DPDK port.

8.24.1 Vhost Implementation in DPDK

Please refer to Chapter "Vhost Library" of *DPDK Programmer's Guide* to know detail of vhost.

8.24.2 Features and Limitations of vhost PMD

Currently, the vhost PMD provides the basic functionality of packet reception, transmission and event handling.

- It has multiple queues support.
- It supports `RTE_ETH_EVENT_INTR_LSC` and `RTE_ETH_EVENT_QUEUE_STATE` events.
- It supports Port Hotplug functionality.
- Don't need to stop RX/TX, when the user wants to stop a guest or a virtio-net driver on guest.

8.24.3 Vhost PMD arguments

The user can specify below arguments in *–vdev* option.

1. `iface`:

   It is used to specify a path to connect to a QEMU virtio-net device.

2. `queues`:

   It is used to specify the number of queues virtio-net device has. (Default: 1)

8.24.4 Vhost PMD event handling

This section describes how to handle vhost PMD events.

The user can register an event callback handler with `rte_eth_dev_callback_register()`. The registered callback handler will be invoked with one of below event types.

1. `RTE_ETH_EVENT_INTR_LSC`:

   It means link status of the port was changed.

2. `RTE_ETH_EVENT_QUEUE_STATE`:

   It means some of queue statuses were changed. Call `rte_eth_vhost_get_queue_event()` in the callback handler. Because changing multiple statuses may occur only one event, call the function repeatedly as long as it doesn't return negative value.

8.24.5 Vhost PMD with testpmd application

This section demonstrates vhost PMD with testpmd DPDK sample application.

1. Launch the testpmd with vhost PMD:

```
./testpmd -l 0-3 -n 4 --vdev 'net_vhost0,iface=/tmp/sock0,queues=1' -- -i
```

   Other basic DPDK preparations like hugepage enabling here. Please refer to the *DPDK Getting Started Guide* for detailed instructions.

2. Launch the QEMU:

```
qemu-system-x86_64 <snip>
            -chardev socket,id=chr0,path=/tmp/sock0 \
            -netdev vhost-user,id=net0,chardev=chr0,vhostforce,queues=1 \
            -device virtio-net-pci,netdev=net0
```

This command attaches one virtio-net device to QEMU guest. After initialization processes between QEMU and DPDK vhost library are done, status of the port will be linked up.

8.25 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, 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. One can use the same device in a DPDK application with VMXNET3 PMD introduced in DPDK API.

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

8.25.1 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.

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.

8.25.2 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.

8.25.3 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.

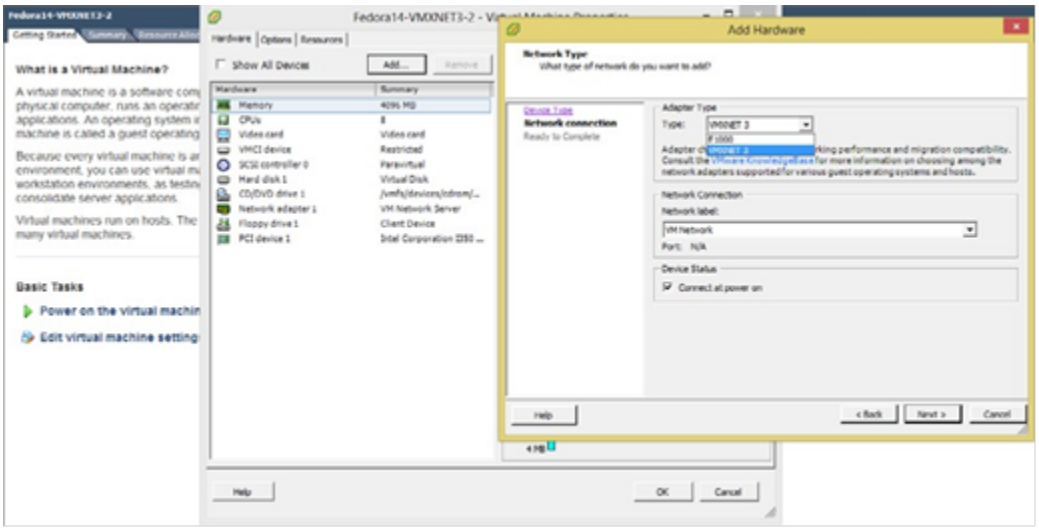


Fig. 8.8: Assigning a VMXNET3 interface to a VM using VMware vSphere Client

---

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.

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.

8.25.4 VMXNET3 with a Native NIC Connected to a vSwitch

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

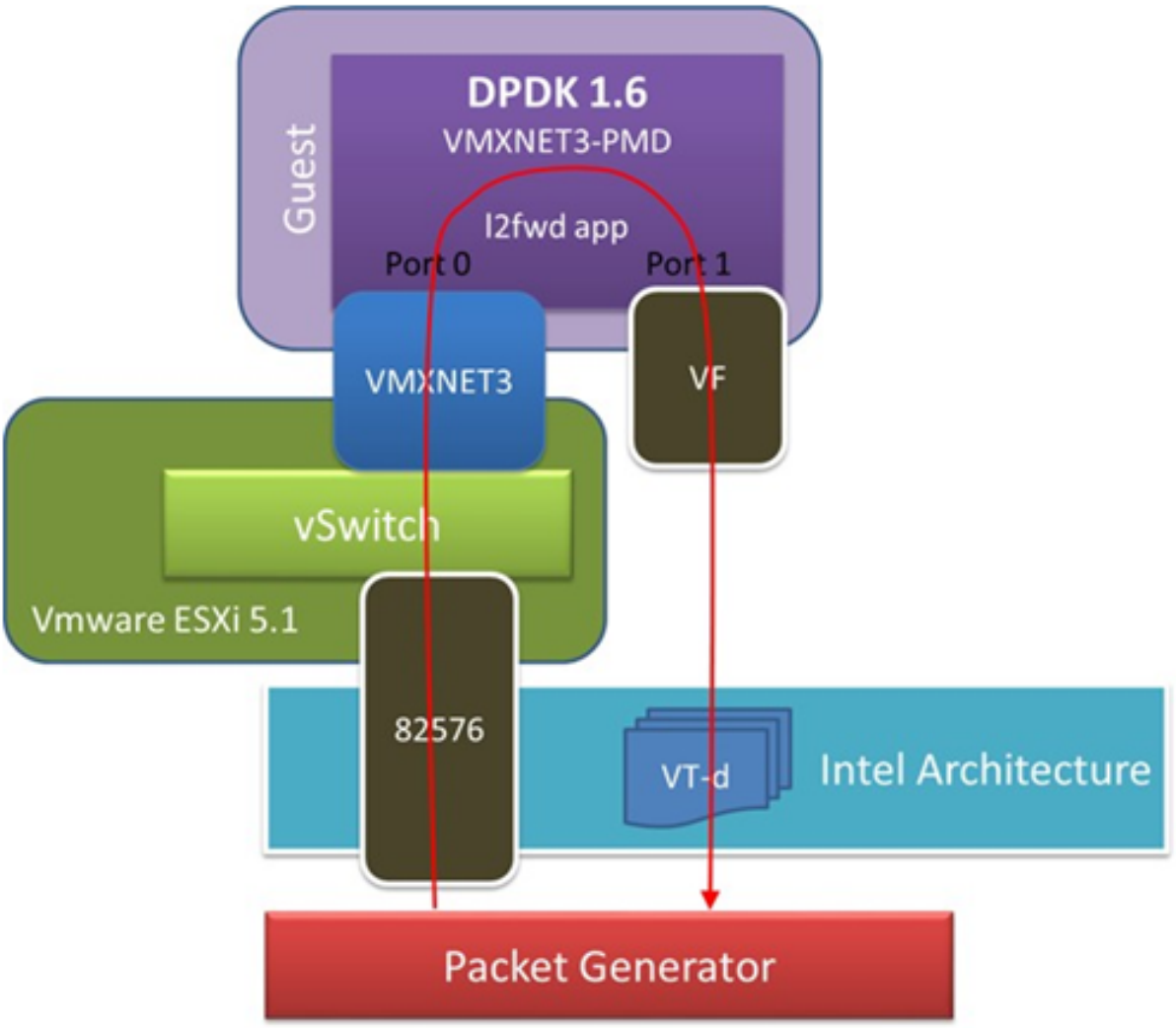


Fig. 8.9: VMXNET3 with a Native NIC Connected to a vSwitch

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
```

8.25.5 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.

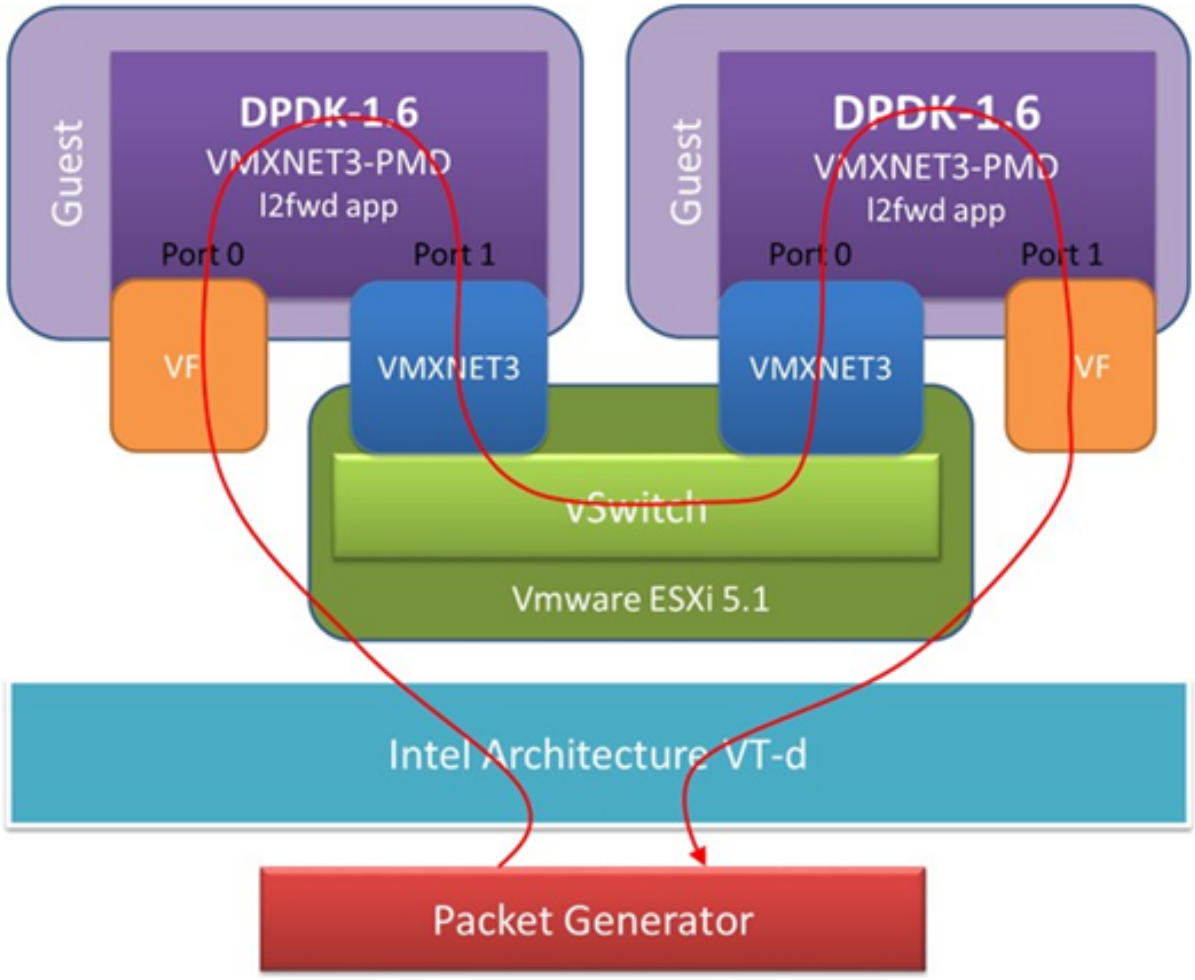


Fig. 8.10: VMXNET3 Chaining VMs Connected to a vSwitch

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:

```
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
```

8.26 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 DPDK.

8.26.1 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.

Libpcap-based PMD

Pcap-based devices can be created using the virtual device –vdev option. The device name must start with the net_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 -l 0-3 -n 4 \
    --vdev 'net_pcap0,stream_opt0=..,stream_opt1=..' \
    --vdev='net_pcap1,stream_opt0=..'
```

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 -l 0-3 -n 4 \
    --vdev 'net_pcap0,rx_pcap=file_rx.pcap,tx_pcap=file_tx.pcap' \
    -- --port-topology=chained
```

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

```
$RTE_TARGET/app/testpmd -l 0-3 -n 4 \
    --vdev 'net_pcap0,rx_iface=eth0,tx_pcap=file_tx.pcap' \
    -- --port-topology=chained
```

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

```
$RTE_TARGET/app/testpmd -l 0-3 -n 4 \
    --vdev 'net_pcap0,rx_pcap=file_rx.pcap,tx_iface=eth1' \
    -- --port-topology=chained
```

Forward packets through two network interfaces:

```
$RTE_TARGET/app/testpmd -l 0-3 -n 4 \
    --vdev 'net_pcap0,iface=eth0' --vdev='net_pcap1;iface=eth1'
```

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 -l 0-3 -n 4 \
    --vdev 'net_pcap0,rx_pcap=file_rx.pcap,tx_pcap=file_tx.pcap' \
    -- --port-topology=chained --no-flush-rx
```

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 net_ring and take no additional parameters. Multiple devices may be specified, separated by commas.

```
./testpmd -l 1-3 -n 4 --vdev=net_ring0 --vdev=net_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.
```

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("net_ring0", ring, NUM_RINGS, ring, NUM_RINGS, SOCKET0);
port1 = rte_eth_from_rings("net_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("net_ring0", &ring[0], 1, &ring[1], 1, SOCKET0);
port1 = rte_eth_from_rings("net_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

Fig. 8.1 *Virtualization for a Single Port NIC in SR-IOV Mode*

Fig. 8.2 *Performance Benchmark Setup*

Fig. 8.3 *Fast Host-based Packet Processing*

Fig. 8.4 *Inter-VM Communication*

Fig. 8.5 *Host2VM Communication Example Using kni vhost Back End*

Fig. 8.7 *Host2VM Communication Example Using qemu vhost Back End*

Fig. 8.8 *Assigning a VMXNET3 interface to a VM using VMware vSphere Client*

Fig. 8.9 *VMXNET3 with a Native NIC Connected to a vSwitch*

Fig. 8.10 *VMXNET3 Chaining VMs Connected to a vSwitch*

CHAPTER 9

Crypto Device Drivers

9.1 Crypto Device Supported Functionality Matrices

9.1.1 Supported Feature Flags

9.1.2 Supported Cipher Algorithms

9.1.3 Supported Authentication Algorithms

9.1.4 Supported AEAD Algorithms

9.2 AESN-NI Multi Buffer Crypto Poll Mode Driver

The AESNI MB PMD (**librte_pmd_aesni_mb**) provides poll mode crypto driver support for utilizing Intel multi buffer library, see the white paper Fast Multi-buffer IPsec Implementations on Intel® Architecture Processors.

The AES-NI MB PMD has current only been tested on Fedora 21 64-bit with gcc.

9.2.1 Features

AESNI MB PMD has support for:

Cipher algorithms:

- RTE_CRYPTO_CIPHER_AES128_CBC
- RTE_CRYPTO_CIPHER_AES192_CBC
- RTE_CRYPTO_CIPHER_AES256_CBC
- RTE_CRYPTO_CIPHER_AES128_CTR
- RTE_CRYPTO_CIPHER_AES192_CTR

- RTE_CRYPTO_CIPHER_AES256_CTR
- RTE_CRYPTO_CIPHER_AES_DOCSISBPI

Hash algorithms:

- RTE_CRYPTO_HASH_MD5_HMAC
- RTE_CRYPTO_HASH_SHA1_HMAC
- RTE_CRYPTO_HASH_SHA224_HMAC
- RTE_CRYPTO_HASH_SHA256_HMAC
- RTE_CRYPTO_HASH_SHA384_HMAC
- RTE_CRYPTO_HASH_SHA512_HMAC
- RTE_CRYPTO_HASH_AES_XCBC_HMAC

9.2.2 Limitations

- Chained mbufs are not supported.
- Only in-place is currently supported (destination address is the same as source address).
- Only supports session-oriented API implementation (session-less APIs are not supported).

9.2.3 Installation

To build DPDK with the AESNI_MB_PMD the user is required to download the multi-buffer library from here and compile it on their user system before building DPDK. The latest version of the library supported by this PMD is v0.45, which can be downloaded in https://github.com/01org/intel-ipsec-mb/archive/v0.45.zip.

```
make
```

As a reference, the following table shows a mapping between the past DPDK versions and the Multi-Buffer library version supported by them:

Table 9.1: DPDK and Multi-Buffer library version compatibility

DPDK version	Multi-buffer library version
2.2 - 16.11	0.43 - 0.44
17.02	0.44
17.05	0.45

9.2.4 Initialization

In order to enable this virtual crypto PMD, user must:

- Export the environmental variable AESNI_MULTI_BUFFER_LIB_PATH with the path where the library was extracted.
- Build the multi buffer library (explained in Installation section).
- Set CONFIG_RTE_LIBRTE_PMD_AESNI_MB=y in config/common_base.

To use the PMD in an application, user must:

- Call rte_eal_vdev_init("crypto_aesni_mb") within the application.
- Use –vdev="crypto_aesni_mb" in the EAL options, which will call rte_eal_vdev_init() internally.

The following parameters (all optional) can be provided in the previous two calls:

- socket_id: Specify the socket where the memory for the device is going to be allocated (by default, socket_id will be the socket where the core that is creating the PMD is running on).
- max_nb_queue_pairs: Specify the maximum number of queue pairs in the device (8 by default).
- max_nb_sessions: Specify the maximum number of sessions that can be created (2048 by default).

Example:

```
./l2fwd-crypto -l 6 -n 4 --vdev="crypto_aesni_mb,socket_id=1,max_nb_sessions=128"
```

9.3 AES-NI GCM Crypto Poll Mode Driver

The AES-NI GCM PMD (**librte_pmd_aesni_gcm**) provides poll mode crypto driver support for utilizing Intel ISA-L crypto library, which provides operation acceleration through the AES-NI instruction sets for AES-GCM authenticated cipher algorithm.

9.3.1 Features

AESNI GCM PMD has support for:

Cipher algorithms:

- RTE_CRYPTO_CIPHER_AES_GCM

Authentication algorithms:

- RTE_CRYPTO_AUTH_AES_GCM
- RTE_CRYPTO_AUTH_AES_GMAC

9.3.2 Installation

To build DPDK with the AESNI_GCM_PMD the user is required to install the `libisal_crypto` library in the build environment. For download and more details please visit https://github.com/01org/isa-l_crypto.

9.3.3 Initialization

In order to enable this virtual crypto PMD, user must:

- Install the ISA-L crypto library (explained in Installation section).
- Set CONFIG_RTE_LIBRTE_PMD_AESNI_GCM=y in config/common_base.

To use the PMD in an application, user must:

- Call rte_eal_vdev_init("crypto_aesni_gcm") within the application.
- Use –vdev="crypto_aesni_gcm" in the EAL options, which will call rte_eal_vdev_init() internally.

The following parameters (all optional) can be provided in the previous two calls:

- socket_id: Specify the socket where the memory for the device is going to be allocated (by default, socket_id will be the socket where the core that is creating the PMD is running on).
- max_nb_queue_pairs: Specify the maximum number of queue pairs in the device (8 by default).
- max_nb_sessions: Specify the maximum number of sessions that can be created (2048 by default).

Example:

```
./l2fwd-crypto -l 6 -n 4 --vdev="crypto_aesni_gcm,socket_id=1,max_nb_sessions=128"
```

9.3.4 Limitations

- Chained mbufs are supported but only out-of-place (destination mbuf must be contiguous).
- Hash only is not supported.
- Cipher only is not supported.

9.4 ARMv8 Crypto Poll Mode Driver

This code provides the initial implementation of the ARMv8 crypto PMD. The driver uses ARMv8 cryptographic extensions to process chained crypto operations in an optimized way. The core functionality is provided by a low-level library, written in the assembly code.

9.4.1 Features

ARMv8 Crypto PMD has support for the following algorithm pairs:

Supported cipher algorithms:

- `RTE_CRYPTO_CIPHER_AES_CBC`

Supported authentication algorithms:

- `RTE_CRYPTO_AUTH_SHA1_HMAC`
- `RTE_CRYPTO_AUTH_SHA256_HMAC`

9.4.2 Installation

In order to enable this virtual crypto PMD, user must:

- Download ARMv8 crypto library source code from here
- Export the environmental variable ARMV8_CRYPTO_LIB_PATH with the path where the `armv8_crypto` library was downloaded or cloned.
- Build the library by invoking:

```
make -C $ARMV8_CRYPTO_LIB_PATH/
```

- Set CONFIG_RTE_LIBRTE_PMD_ARMV8_CRYPTO=y in config/defconfig_arm64-armv8a-linuxapp-gcc

The corresponding device can be created only if the following features are supported by the CPU:

- `RTE_CPUFLAG_AES`

- RTE_CPUFLAG_SHA1
- RTE_CPUFLAG_SHA2
- RTE_CPUFLAG_NEON

9.4.3 Initialization

User can use app/test application to check how to use this PMD and to verify crypto processing.

Test name is cryptodev_sw_armv8_autotest. For performance test cryptodev_sw_armv8_perftest can be used.

9.4.4 Limitations

- Maximum number of sessions is 2048.
- Only chained operations are supported.
- AES-128-CBC is the only supported cipher variant.
- Cipher input data has to be a multiple of 16 bytes.
- Digest input data has to be a multiple of 8 bytes.

9.5 KASUMI Crypto Poll Mode Driver

The KASUMI PMD (**librte_pmd_kasumi**) provides poll mode crypto driver support for utilizing Intel Libsso library, which implements F8 and F9 functions for KASUMI UEA1 cipher and UIA1 hash algorithms.

9.5.1 Features

KASUMI PMD has support for:

Cipher algorithm:

- RTE_CRYPTO_CIPHER_KASUMI_F8

Authentication algorithm:

- RTE_CRYPTO_AUTH_KASUMI_F9

9.5.2 Limitations

- Chained mbufs are not supported.
- KASUMI(F9) supported only if hash offset field is byte-aligned.
- In-place bit-level operations for KASUMI(F8) are not supported (if length and/or offset of data to be ciphered is not byte-aligned).

9.5.3 Installation

To build DPDK with the KASUMI_PMD the user is required to download the export controlled `libsso_kasumi` library, by requesting it from https://networkbuilders.intel.com/network-technologies/dpdk. Once approval has been granted, the user needs to log in https://networkbuilders.intel.com/dpdklogin and click on "Kasumi Bit Stream crypto library" link, to download the library. After downloading the library, the user needs to unpack and compile it on their system before building DPDK:

```
make
```

Note: To build the PMD as a shared library, the libsso_kasumi library must be built as follows:

```
make KASUMI_CFLAGS=-DKASUMI_C
```

9.5.4 Initialization

In order to enable this virtual crypto PMD, user must:

- Export the environmental variable LIBSSO_KASUMI_PATH with the path where the library was extracted (kasumi folder).
- Build the LIBSSO library (explained in Installation section).
- Set CONFIG_RTE_LIBRTE_PMD_KASUMI=y in config/common_base.

To use the PMD in an application, user must:

- Call rte_eal_vdev_init("crypto_kasumi") within the application.
- Use –vdev="crypto_kasumi" in the EAL options, which will call rte_eal_vdev_init() internally.

The following parameters (all optional) can be provided in the previous two calls:

- socket_id: Specify the socket where the memory for the device is going to be allocated (by default, socket_id will be the socket where the core that is creating the PMD is running on).
- max_nb_queue_pairs: Specify the maximum number of queue pairs in the device (8 by default).
- max_nb_sessions: Specify the maximum number of sessions that can be created (2048 by default).

Example:

```
./l2fwd-crypto -l 6 -n 4 --vdev="crypto_kasumi,socket_id=1,max_nb_sessions=128"
```

9.6 OpenSSL Crypto Poll Mode Driver

This code provides the initial implementation of the openssl poll mode driver. All cryptography operations are using Openssl library crypto API. Each algorithm uses EVP interface from openssl API - which is recommended by Openssl maintainers.

For more details about openssl library please visit openssl webpage: https://www.openssl.org/

9.6.1 Features

OpenSSL PMD has support for:

Supported cipher algorithms: * RTE_CRYPTO_CIPHER_3DES_CBC * RTE_CRYPTO_CIPHER_AES_CBC * RTE_CRYPTO_CIPHER_AES_CTR * RTE_CRYPTO_CIPHER_3DES_CTR * RTE_CRYPTO_CIPHER_AES_GCM * RTE_CRYPTO_CIPHER_DES_DOCSISBPI

Supported authentication algorithms: * RTE_CRYPTO_AUTH_AES_GMAC * RTE_CRYPTO_AUTH_MD5 * RTE_CRYPTO_AUTH_SHA1 * RTE_CRYPTO_AUTH_SHA224 * RTE_CRYPTO_AUTH_SHA256 * RTE_CRYPTO_AUTH_SHA384 * RTE_CRYPTO_AUTH_SHA512 * RTE_CRYPTO_AUTH_MD5_HMAC * RTE_CRYPTO_AUTH_SHA1_HMAC * RTE_CRYPTO_AUTH_SHA224_HMAC * RTE_CRYPTO_AUTH_SHA256_HMAC * RTE_CRYPTO_AUTH_SHA384_HMAC * RTE_CRYPTO_AUTH_SHA512_HMAC

9.6.2 Installation

To compile openssl PMD, it has to be enabled in the config/common_base file and appropriate openssl packages have to be installed in the build environment.

The newest openssl library version is supported: * 1.0.2h-fips 3 May 2016. Older versions that were also verified: * 1.0.1f 6 Jan 2014 * 1.0.1 14 Mar 2012

For Ubuntu 14.04 LTS these packages have to be installed in the build system: sudo apt-get install openssl sudo apt-get install libc6-dev-i386 (for i686-native-linuxapp-gcc target)

This code was also verified on Fedora 24. This code was NOT yet verified on FreeBSD.

9.6.3 Initialization

User can use app/test application to check how to use this pmd and to verify crypto processing.

Test name is cryptodev_openssl_autotest. For performance test cryptodev_openssl_perftest can be used.

To verify real traffic l2fwd-crypto example can be used with this command:

```
sudo ./build/l2fwd-crypto -l 0-1 -n 4 --vdev "crypto_openssl"
--vdev "crypto_openssl"-- -p 0x3 --chain CIPHER_HASH
--cipher_op ENCRYPT --cipher_algo AES_CBC
--cipher_key 00:01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f
--iv 00:01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:ff
--auth_op GENERATE --auth_algo SHA1_HMAC
--auth_key 11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11
:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11
:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11
```

9.6.4 Limitations

- Maximum number of sessions is 2048.
- Chained mbufs are supported only for source mbuf (destination must be contiguous).
- Hash only is not supported for GCM and GMAC.
- Cipher only is not supported for GCM and GMAC.

9.7 Null Crypto Poll Mode Driver

The Null Crypto PMD (**librte_pmd_null_crypto**) provides a crypto poll mode driver which provides a minimal implementation for a software crypto device. As a null device it does not modify the data in the mbuf on which the crypto operation is to operate and it only has support for a single cipher and authentication algorithm.

When a burst of mbufs is submitted to a Null Crypto PMD for processing then each mbuf in the burst will be enqueued in an internal buffer for collection on a dequeue call as long as the mbuf has a valid rte_mbuf_offload operation with a valid rte_cryptodev_session or rte_crypto_xform chain of operations.

9.7.1 Features

Modes:

- RTE_CRYPTO_XFORM_CIPHER ONLY
- RTE_CRYPTO_XFORM_AUTH ONLY
- RTE_CRYPTO_XFORM_CIPHER THEN RTE_CRYPTO_XFORM_AUTH
- RTE_CRYPTO_XFORM_AUTH THEN RTE_CRYPTO_XFORM_CIPHER

Cipher algorithms:

- RTE_CRYPTO_CIPHER_NULL

Authentication algorithms:

- RTE_CRYPTO_AUTH_NULL

9.7.2 Limitations

- Only in-place is currently supported (destination address is the same as source address).

9.7.3 Installation

The Null Crypto PMD is enabled and built by default in both the Linux and FreeBSD builds.

9.7.4 Initialization

To use the PMD in an application, user must:

- Call rte_eal_vdev_init("crypto_null") within the application.
- Use –vdev="crypto_null" in the EAL options, which will call rte_eal_vdev_init() internally.

The following parameters (all optional) can be provided in the previous two calls:

- socket_id: Specify the socket where the memory for the device is going to be allocated (by default, socket_id will be the socket where the core that is creating the PMD is running on).
- max_nb_queue_pairs: Specify the maximum number of queue pairs in the device (8 by default).
- max_nb_sessions: Specify the maximum number of sessions that can be created (2048 by default).

Example:

```
./l2fwd-crypto -l 6 -n 4 --vdev="crypto_null,socket_id=1,max_nb_sessions=128"
```

9.8 Cryptodev Scheduler Poll Mode Driver Library

Scheduler PMD is a software crypto PMD, which has the capabilities of attaching hardware and/or software cryptodevs, and distributes ingress crypto ops among them in a certain manner.

Fig. 9.1: Cryptodev Scheduler Overview

The Cryptodev Scheduler PMD library (**librte_pmd_crypto_scheduler**) acts as a software crypto PMD and shares the same API provided by librte_cryptodev. The PMD supports attaching multiple crypto PMDs, software or hardware, as slaves, and distributes the crypto workload to them with certain behavior. The behaviors are categorizes as different "modes". Basically, a scheduling mode defines certain actions for scheduling crypto ops to its slaves.

The librte_pmd_crypto_scheduler library exports a C API which provides an API for attaching/detaching slaves, set/get scheduling modes, and enable/disable crypto ops reordering.

9.8.1 Limitations

- Sessionless crypto operation is not supported
- OOP crypto operation is not supported when the crypto op reordering feature is enabled.

9.8.2 Installation

To build DPDK with CRYTPO_SCHEDULER_PMD the user is required to set CONFIG_RTE_LIBRTE_PMD_CRYPTO_SCHEDULER=y in config/common_base, and recompile DPDK

9.8.3 Initialization

To use the PMD in an application, user must:

- Call rte_eal_vdev_init("crpyto_scheduler") within the application.
- Use –vdev="crpyto_scheduler" in the EAL options, which will call rte_eal_vdev_init() internally.

The following parameters (all optional) can be provided in the previous two calls:

- socket_id: Specify the socket where the memory for the device is going to be allocated (by default, socket_id will be the socket where the core that is creating the PMD is running on).
- max_nb_sessions: Specify the maximum number of sessions that can be created. This value may be overwritten internally if there are too many devices are attached.
- slave: If a cryptodev has been initialized with specific name, it can be attached to the scheduler using this parameter, simply filling the name here. Multiple cryptodevs can be attached initially by presenting this parameter multiple times.
- mode: Specify the scheduling mode of the PMD. The supported scheduling mode parameter values are specified in the "Cryptodev Scheduler Modes Overview" section.

- ordering: Specify the status of the crypto operations ordering feature. The value of this parameter can be "enable" or "disable". This feature is disabled by default.

Example:

```
... --vdev "crypto_aesni_mb_pmd,name=aesni_mb_1" --vdev "crypto_aesni_mb_pmd,
↪name=aesni_mb_2" --vdev "crypto_scheduler_pmd,slave=aesni_mb_1,slave=aesni_mb_2" ...
```

Note:

- The scheduler cryptodev cannot be started unless the scheduling mode is set and at least one slave is attached. Also, to configure the scheduler in the run-time, like attach/detach slave(s), change scheduling mode, or enable/disable crypto op ordering, one should stop the scheduler first, otherwise an error will be returned.
- The crypto op reordering feature requires using the userdata field of every mbuf to be processed to store temporary data. By the end of processing, the field is set to pointing to NULL, any previously stored value of this field will be lost.

9.8.4 Cryptodev Scheduler Modes Overview

Currently the Crypto Scheduler PMD library supports following modes of operation:

- **CDEV_SCHED_MODE_ROUNDROBIN:**

  Initialization mode parameter: **round-robin**

  Round-robin mode, which distributes the enqueued burst of crypto ops among its slaves in a round-robin manner. This mode may help to fill the throughput gap between the physical core and the existing cryptodevs to increase the overall performance.

- **CDEV_SCHED_MODE_PKT_SIZE_DISTR:**

  Initialization mode parameter: **packet-size-distr**

  Packet-size based distribution mode, which works with 2 slaves, the primary slave and the secondary slave, and distributes the enqueued crypto operations to them based on their data lengths. A crypto operation will be distributed to the primary slave if its data length is equal to or bigger than the designated threshold, otherwise it will be handled by the secondary slave.

  A typical usecase in this mode is with the QAT cryptodev as the primary and a software cryptodev as the secondary slave. This may help applications to process additional crypto workload than what the QAT cryptodev can handle on its own, by making use of the available CPU cycles to deal with smaller crypto workloads.

- **CDEV_SCHED_MODE_FAILOVER:**

  Initialization mode parameter: **fail-over**

  Fail-over mode, which works with 2 slaves, the primary slave and the secondary slave. In this mode, the scheduler will enqueue the incoming crypto operation burst to the primary slave. When one or more crypto operations fail to be enqueued, then they will be enqueued to the secondary slave.

9.9 SNOW 3G Crypto Poll Mode Driver

The SNOW 3G PMD (**librte_pmd_snow3g**) provides poll mode crypto driver support for utilizing Intel Libsso library, which implements F8 and F9 functions for SNOW 3G UEA2 cipher and UIA2 hash algorithms.

9.9.1 Features

SNOW 3G PMD has support for:

Cipher algorithm:

- RTE_CRYPTO_CIPHER_SNOW3G_UEA2

Authentication algorithm:

- RTE_CRYPTO_AUTH_SNOW3G_UIA2

9.9.2 Limitations

- Chained mbufs are not supported.
- SNOW 3G (UIA2) supported only if hash offset field is byte-aligned.
- In-place bit-level operations for SNOW 3G (UEA2) are not supported (if length and/or offset of data to be ciphered is not byte-aligned).

9.9.3 Installation

To build DPDK with the SNOW3G_PMD the user is required to download the export controlled `libsso_snow3g` library, by requesting it from https://networkbuilders.intel.com/network-technologies/dpdk. Once approval has been granted, the user needs to log in https://networkbuilders.intel.com/dpdklogin and click on "Snow3G Bit Stream crypto library" link, to download the library. After downloading the library, the user needs to unpack and compile it on their system before building DPDK:

```
make snow3G
```

9.9.4 Initialization

In order to enable this virtual crypto PMD, user must:

- Export the environmental variable LIBSSO_SNOW3G_PATH with the path where the library was extracted (snow3g folder).
- Build the LIBSSO_SNOW3G library (explained in Installation section).
- Set CONFIG_RTE_LIBRTE_PMD_SNOW3G=y in config/common_base.

To use the PMD in an application, user must:

- Call rte_eal_vdev_init("crypto_snow3g") within the application.
- Use –vdev="crypto_snow3g" in the EAL options, which will call rte_eal_vdev_init() internally.

The following parameters (all optional) can be provided in the previous two calls:

- socket_id: Specify the socket where the memory for the device is going to be allocated (by default, socket_id will be the socket where the core that is creating the PMD is running on).
- max_nb_queue_pairs: Specify the maximum number of queue pairs in the device (8 by default).
- max_nb_sessions: Specify the maximum number of sessions that can be created (2048 by default).

Example:

```
./l2fwd-crypto -l 6 -n 4 --vdev="crypto_snow3g,socket_id=1,max_nb_sessions=128"
```

9.10 Intel(R) QuickAssist (QAT) Crypto Poll Mode Driver

The QAT PMD provides poll mode crypto driver support for the following hardware accelerator devices:

- Intel QuickAssist Technology DH895xCC
- Intel QuickAssist Technology C62x
- Intel QuickAssist Technology C3xxx
- Intel QuickAssist Technology D15xx

9.10.1 Features

The QAT PMD has support for:

Cipher algorithms:

- RTE_CRYPTO_CIPHER_3DES_CBC
- RTE_CRYPTO_CIPHER_3DES_CTR
- RTE_CRYPTO_CIPHER_AES128_CBC
- RTE_CRYPTO_CIPHER_AES192_CBC
- RTE_CRYPTO_CIPHER_AES256_CBC
- RTE_CRYPTO_CIPHER_AES128_CTR
- RTE_CRYPTO_CIPHER_AES192_CTR
- RTE_CRYPTO_CIPHER_AES256_CTR
- RTE_CRYPTO_CIPHER_SNOW3G_UEA2
- RTE_CRYPTO_CIPHER_AES_GCM
- RTE_CRYPTO_CIPHER_NULL
- RTE_CRYPTO_CIPHER_KASUMI_F8
- RTE_CRYPTO_CIPHER_DES_CBC
- RTE_CRYPTO_CIPHER_AES_DOCSISBPI
- RTE_CRYPTO_CIPHER_DES_DOCSISBPI
- RTE_CRYPTO_CIPHER_ZUC_EEA3

Hash algorithms:

- RTE_CRYPTO_AUTH_SHA1_HMAC
- RTE_CRYPTO_AUTH_SHA224_HMAC
- RTE_CRYPTO_AUTH_SHA256_HMAC
- RTE_CRYPTO_AUTH_SHA384_HMAC
- RTE_CRYPTO_AUTH_SHA512_HMAC

- RTE_CRYPTO_AUTH_AES_XCBC_MAC
- RTE_CRYPTO_AUTH_SNOW3G_UIA2
- RTE_CRYPTO_AUTH_MD5_HMAC
- RTE_CRYPTO_AUTH_NULL
- RTE_CRYPTO_AUTH_KASUMI_F9
- RTE_CRYPTO_AUTH_AES_GMAC
- RTE_CRYPTO_AUTH_ZUC_EIA3

9.10.2 Limitations

- Hash only is not supported except SNOW 3G UIA2 and KASUMI F9.
- Only supports the session-oriented API implementation (session-less APIs are not supported).
- SNOW 3G (UEA2) and KASUMI (F8) supported only if cipher length, cipher offset fields are byte-aligned.
- SNOW 3G (UIA2) and KASUMI (F9) supported only if hash length, hash offset fields are byte-aligned.
- No BSD support as BSD QAT kernel driver not available.
- ZUC EEA3/EIA3 is not supported by dh895xcc devices

9.10.3 Installation

To enable QAT in DPDK, follow the instructions for modifying the compile-time configuration file as described here.

Quick instructions are as follows:

```
cd to the top-level DPDK directory
make config T=x86_64-native-linuxapp-gcc
sed -i 's,\(CONFIG_RTE_LIBRTE_PMD_QAT\)=n,\1=y,' build/.config
make
```

To use the DPDK QAT PMD an SRIOV-enabled QAT kernel driver is required. The VF devices exposed by this driver will be used by the QAT PMD. The devices and available kernel drivers and device ids are :

Table 9.2: QAT devices and drivers

Device	Driver	Kernel Module	Pci Driver	PF Did	Num PFs	Vf Did	VFs per PF
DH895xCC	01.org	icp_qa_al	n/a	435	1	443	32
DH895xCC	4.4+	qat_dh895xcc	dh895xcc	435	1	443	32
C62x	4.5+	qat_c62x	c6xx	37c8	3	37c9	16
C3xxx	4.5+	qat_c3xxx	c3xxx	19e2	1	19e3	16
D15xx	p	qat_d15xx	d15xx	6f54	1	6f55	16

The Driver column indicates either the Linux kernel version in which support for this device was introduced or a driver available on Intel's 01.org website. There are both linux and 01.org kernel drivers available for some devices. p = release pending.

If you are running on a kernel which includes a driver for your device, see *Installation using kernel.org driver* below. Otherwise see *Installation using 01.org QAT driver*.

9.10.4 Installation using kernel.org driver

The examples below are based on the C62x device, if you have a different device use the corresponding values in the above table.

In BIOS ensure that SRIOV is enabled and either:

- Disable VT-d or
- Enable VT-d and set "`intel_iommu=on iommu=pt`" in the grub file.

Check that the QAT driver is loaded on your system, by executing:

```
lsmod | grep qa
```

You should see the kernel module for your device listed, e.g.:

```
qat_c62x               5626  0
intel_qat              82336  1 qat_c62x
```

Next, you need to expose the Virtual Functions (VFs) using the sysfs file system.

First find the BDFs (Bus-Device-Function) of the physical functions (PFs) of your device, e.g.:

```
lspci -d : 37c8
```

You should see output similar to:

```
1a:00.0 Co-processor: Intel Corporation Device 37c8
3d:00.0 Co-processor: Intel Corporation Device 37c8
3f:00.0 Co-processor: Intel Corporation Device 37c8
```

Enable the VFs for each PF by echoing the number of VFs per PF to the pci driver:

```
echo 16 > /sys/bus/pci/drivers/c6xx/0000:1a:00.0/sriov_numvfs
echo 16 > /sys/bus/pci/drivers/c6xx/0000:3d:00.0/sriov_numvfs
echo 16 > /sys/bus/pci/drivers/c6xx/0000:3f:00.0/sriov_numvfs
```

Check that the VFs are available for use. For example `lspci -d:37c9` should list 48 VF devices available for a `C62x` device.

To complete the installation follow the instructions in *Binding the available VFs to the DPDK UIO driver*.

Note: If the QAT kernel modules are not loaded and you see an error like `Failed to load MMP firmware qat_895xcc_mmp.bin` in kernel logs, this may be as a result of not using a distribution, but just updating the kernel directly.

Download firmware from the kernel firmware repo.

Copy qat binaries to `/lib/firmware`:

```
cp qat_895xcc.bin /lib/firmware
cp qat_895xcc_mmp.bin /lib/firmware
```

Change to your linux source root directory and start the qat kernel modules:

```
insmod ./drivers/crypto/qat/qat_common/intel_qat.ko
insmod ./drivers/crypto/qat/qat_dh895xcc/qat_dh895xcc.ko
```

Note: If you see the following warning in `/var/log/messages` it can be ignored: `IOMMU should be enabled for SR-IOV to work correctly`.

9.10.5 Installation using 01.org QAT driver

Download the latest QuickAssist Technology Driver from 01.org. Consult the *Getting Started Guide* at the same URL for further information.

The steps below assume you are:

- Building on a platform with one `DH895xCC` device.
- Using package `qatmux.l.2.3.0-34.tgz`.
- On Fedora21 kernel `3.17.4-301.fc21.x86_64`.

In the BIOS ensure that SRIOV is enabled and VT-d is disabled.

Uninstall any existing QAT driver, for example by running:

- `./installer.sh uninstall` in the directory where originally installed.
- or `rmmod qat_dh895xcc; rmmod intel_qat`.

Build and install the SRIOV-enabled QAT driver:

```
mkdir /QAT
cd /QAT

# Copy qatmux.l.2.3.0-34.tgz to this location
tar zxof qatmux.l.2.3.0-34.tgz

export ICP_WITHOUT_IOMMU=1
./installer.sh install QAT1.6 host
```

You can use `cat /proc/icp_dh895xcc_dev0/version` to confirm the driver is correctly installed. You can use `lspci -d:443` to confirm the of the 32 VF devices available per `DH895xCC` device.

To complete the installation - follow instructions in *Binding the available VFs to the DPDK UIO driver*.

Note: If using a later kernel and the build fails with an error relating to `strict_stroul` not being available apply the following patch:

```
/QAT/QAT1.6/quickassist/utilities/downloader/Target_CoreLibs/uclo/include/linux/uclo_
↪platform.h
+ #if LINUX_VERSION_CODE >= KERNEL_VERSION(3,18,5)
+ #define STR_TO_64(str, base, num, endPtr) {endPtr=NULL; if (kstrtoul((str), (base),␣
↪(num))) printk("Error strtoull convert %s\n", str); }
+ #else
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,6,38)
#define STR_TO_64(str, base, num, endPtr) {endPtr=NULL; if (strict_strtoull((str),␣
↪(base), (num))) printk("Error strtoull convert %s\n", str); }
#else
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,6,25)
#define STR_TO_64(str, base, num, endPtr) {endPtr=NULL; strict_strtoll((str), (base),␣
↪(num));}
```

```
#else
#define STR_TO_64(str, base, num, endPtr)                                    \
     do {                                                                    \
           if (str[0] == '-')                                                \
           {                                                                 \
                *(num) = -(simple_strtoull((str+1), &(endPtr), (base)));     \
           }else {                                                           \
                *(num) = simple_strtoull((str), &(endPtr), (base));          \
           }                                                                 \
     } while(0)
+ #endif
#endif
#endif
```

Note: If the build fails due to missing header files you may need to do following:

```
sudo yum install zlib-devel
sudo yum install openssl-devel
```

Note: If the build or install fails due to mismatching kernel sources you may need to do the following:

```
sudo yum install kernel-headers-`uname -r`
sudo yum install kernel-src-`uname -r`
sudo yum install kernel-devel-`uname -r`
```

9.10.6 Binding the available VFs to the DPDK UIO driver

Unbind the VFs from the stock driver so they can be bound to the uio driver.

For an Intel(R) QuickAssist Technology DH895xCC device

The unbind command below assumes `BDFs` of `03:01.00-03:04.07`, if your VFs are different adjust the unbind command below:

```
for device in $(seq 1 4); do \
    for fn in $(seq 0 7); do \
        echo -n 0000:03:0${device}.${fn} > \
        /sys/bus/pci/devices/0000\:03\:0${device}.${fn}/driver/unbind; \
    done; \
done
```

For an Intel(R) QuickAssist Technology C62x device

The unbind command below assumes `BDFs` of `1a:01.00-1a:02.07`, `3d:01.00-3d:02.07` and `3f:01.00-3f:02.07`, if your VFs are different adjust the unbind command below:

```
for device in $(seq 1 2); do \
    for fn in $(seq 0 7); do \
        echo -n 0000:1a:0${device}.${fn} > \
        /sys/bus/pci/devices/0000\:1a\:0${device}.${fn}/driver/unbind; \

        echo -n 0000:3d:0${device}.${fn} > \
        /sys/bus/pci/devices/0000\:3d\:0${device}.${fn}/driver/unbind; \

        echo -n 0000:3f:0${device}.${fn} > \
        /sys/bus/pci/devices/0000\:3f\:0${device}.${fn}/driver/unbind; \
    done; \
done
```

For Intel(R) QuickAssist Technology C3xxx or D15xx device

The unbind command below assumes `BDFs` of `01:01.00-01:02.07`, if your VFs are different adjust the unbind command below:

```
for device in $(seq 1 2); do \
    for fn in $(seq 0 7); do \
        echo -n 0000:01:0${device}.${fn} > \
        /sys/bus/pci/devices/0000\:01\:0${device}.${fn}/driver/unbind; \
    done; \
done
```

Bind to the DPDK uio driver

Install the DPDK igb_uio driver, bind the VF PCI Device id to it and use lspci to confirm the VF devices are now in use by igb_uio kernel driver, e.g. for the C62x device:

```
cd to the top-level DPDK directory
modprobe uio
insmod ./build/kmod/igb_uio.ko
echo "8086 37c9" > /sys/bus/pci/drivers/igb_uio/new_id
lspci -vvd:37c9
```

Another way to bind the VFs to the DPDK UIO driver is by using the `dpdk-devbind.py` script:

```
cd to the top-level DPDK directory
./usertools/dpdk-devbind.py -b igb_uio 0000:03:01.1
```

9.11 ZUC Crypto Poll Mode Driver

The ZUC PMD (**librte_pmd_zuc**) provides poll mode crypto driver support for utilizing Intel Libsso library, which implements F8 and F9 functions for ZUC EEA3 cipher and EIA3 hash algorithms.

9.11.1 Features

ZUC PMD has support for:

Cipher algorithm:

- RTE_CRYPTO_CIPHER_ZUC_EEA3

Authentication algorithm:

- RTE_CRYPTO_AUTH_ZUC_EIA3

9.11.2 Limitations

- Chained mbufs are not supported.
- ZUC (EIA3) supported only if hash offset field is byte-aligned.
- ZUC (EEA3) supported only if cipher length, cipher offset fields are byte-aligned.
- ZUC PMD cannot be built as a shared library, due to limitations in in the underlying library.

9.11.3 Installation

To build DPDK with the ZUC_PMD the user is required to download the export controlled `libsso_zuc` library, by requesting it from https://networkbuilders.intel.com/network-technologies/dpdk. Once approval has been granted, the user needs to log in https://networkbuilders.intel.com/dpdklogin and click on "ZUC Library" link, to download the library. After downloading the library, the user needs to unpack and compile it on their system before building DPDK:

```
make
```

9.11.4 Initialization

In order to enable this virtual crypto PMD, user must:

- Export the environmental variable LIBSSO_ZUC_PATH with the path where the library was extracted (zuc folder).
- Build the LIBSSO_ZUC library (explained in Installation section).
- Build DPDK as follows:

```
make config T=x86_64-native-linuxapp-gcc
sed -i 's,\(CONFIG_RTE_LIBRTE_PMD_ZUC\)=n,\1=y,' build/.config
make
```

To use the PMD in an application, user must:

- Call rte_eal_vdev_init("crypto_zuc") within the application.
- Use –vdev="crypto_zuc" in the EAL options, which will call rte_eal_vdev_init() internally.

The following parameters (all optional) can be provided in the previous two calls:

- socket_id: Specify the socket where the memory for the device is going to be allocated (by default, socket_id will be the socket where the core that is creating the PMD is running on).
- max_nb_queue_pairs: Specify the maximum number of queue pairs in the device (8 by default).
- max_nb_sessions: Specify the maximum number of sessions that can be created (2048 by default).

Example:

```
./l2fwd-crypto -l 6 -n 4 --vdev="crypto_zuc,socket_id=1,max_nb_sessions=128"
```

CHAPTER 10

Event Device Drivers

The following are a list of event device PMDs, which can be used from an application trough the eventdev API.

10.1 Software Eventdev Poll Mode Driver

The software eventdev is an implementation of the eventdev API, that provides a wide range of the eventdev features. The eventdev relies on a CPU core to perform event scheduling.

10.1.1 Features

The software eventdev implements many features in the eventdev API;

Queues

- Atomic
- Ordered
- Parallel
- Single-Link

Ports

- Load balanced (for Atomic, Ordered, Parallel queues)
- Single Link (for single-link queues)

Event Priorities

- Each event has a priority, which can be used to provide basic QoS

10.1.2 Configuration and Options

The software eventdev is a vdev device, and as such can be created from the application code, or from the EAL command line:

- Call `rte_eal_vdev_init("event_sw0")` from the application
- Use `--vdev="event_sw0"` in the EAL options, which will call rte_eal_vdev_init() internally

Example:

```
./your_eventdev_application --vdev="event_sw0"
```

Scheduling Quanta

The scheduling quanta sets the number of events that the device attempts to schedule before returning to the application from the `rte_event_schedule()` function. Note that is a *hint* only, and that fewer or more events may be scheduled in a given iteration.

The scheduling quanta can be set using a string argument to the vdev create call:

```
--vdev="event_sw0,sched_quanta=64"
```

Credit Quanta

The credit quanta is the number of credits that a port will fetch at a time from the instance's credit pool. Higher numbers will cause less overhead in the atomic credit fetch code, however it also reduces the overall number of credits in the system faster. A balanced number (eg 32) ensures that only small numbers of credits are pre-allocated at a time, while also mitigating performance impact of the atomics.

Experimentation with higher values may provide minor performance improvements, at the cost of the whole system having less credits. On the other hand, reducing the quanta may cause measurable performance impact but provide the system with a higher number of credits at all times.

A value of 32 seems a good balance however your specific application may benefit from a higher or reduced quanta size, experimentation is required to verify possible gains.

```
--vdev="event_sw0,credit_quanta=64"
```

10.1.3 Limitations

The software eventdev implementation has a few limitations. The reason for these limitations is usually that the performance impact of supporting the feature would be significant.

"All Types" Queues

The software eventdev does not support creating queues that handle all types of traffic. An eventdev with this capability allows enqueueing Atomic, Ordered and Parallel traffic to the same queue, but scheduling each of them appropriately.

The reason to not allow Atomic, Ordered and Parallel event types in the same queue is that it causes excessive branching in the code to enqueue packets to the queue, causing a significant performance impact.

The `RTE_EVENT_DEV_CAP_QUEUE_ALL_TYPES` flag is not set in the `event_dev_cap` field of the `rte_event_dev_info` struct for the software eventdev.

Distributed Scheduler

The software eventdev is a centralized scheduler, requiring the `rte_event_schedule()` function to be called by a CPU core to perform the required event distribution. This is not really a limitation but rather a design decision.

The `RTE_EVENT_DEV_CAP_DISTRIBUTED_SCHED` flag is not set in the `event_dev_cap` field of the `rte_event_dev_info` struct for the software eventdev.

Dequeue Timeout

The eventdev API supports a timeout when dequeuing packets using the `rte_event_dequeue_burst` function. This allows a core to wait for an event to arrive, or until `timeout` number of ticks have passed. Timeout ticks is not supported by the software eventdev for performance reasons.

10.2 OCTEONTX SSOVF Eventdev Driver

The OCTEONTX SSOVF PMD (**librte_pmd_octeontx_ssovf**) provides poll mode eventdev driver support for the inbuilt event device found in the **Cavium OCTEONTX** SoC family as well as their virtual functions (VF) in SR-IOV context.

More information can be found at Cavium Networks Official Website.

10.2.1 Features

Features of the OCTEONTX SSOVF PMD are:

- 64 Event queues
- 32 Event ports
- HW event scheduler
- Supports 1M flows per event queue
- Flow based event pipelining
- Flow pinning support in flow based event pipelining
- Queue based event pipelining
- Supports ATOMIC, ORDERED, PARALLEL schedule types per flow
- Event scheduling QoS based on event queue priority
- Open system with configurable amount of outstanding events
- HW accelerated dequeue timeout support to enable power management
- SR-IOV VF

10.2.2 Supported OCTEONTX SoCs

- CN83xx

10.2.3 Prerequisites

There are three main pre-perquisites for executing SSOVF PMD on a OCTEONTX compatible board:

1. **OCTEONTX Linux kernel PF driver for Network acceleration HW blocks**

   The OCTEONTX Linux kernel drivers (including the required PF driver for the SSOVF) are available on Github at octeontx-kmod along with build, install and dpdk usage instructions.

2. **ARM64 Tool Chain**

   For example, the *aarch64* Linaro Toolchain, which can be obtained from here.

3. **Rootfile system**

   Any *aarch64* supporting filesystem can be used. For example, Ubuntu 15.10 (Wily) or 16.04 LTS (Xenial) userland which can be obtained from http://cdimage.ubuntu.com/ubuntu-base/releases/16.04/release/ubuntu-base-16.04.1-base-arm64.tar.gz.

   As an alternative method, SSOVF PMD can also be executed using images provided as part of SDK from Cavium. The SDK includes all the above prerequisites necessary to bring up a OCTEONTX board.

   SDK and related information can be obtained from: Cavium support site.

- Follow the DPDK *Getting Started Guide for Linux* to setup the basic DPDK environment.

10.2.4 Pre-Installation Configuration

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_PMD_OCTEONTX_SSOVF` (default `y`)

  Toggle compilation of the `librte_pmd_octeontx_ssovf` driver.

- `CONFIG_RTE_LIBRTE_PMD_OCTEONTX_SSOVF_DEBUG` (default `n`)

  Toggle display of generic debugging messages

Driver Compilation

To compile the OCTEONTX SSOVF PMD for Linux arm64 gcc target, run the following `make` command:

```
cd <DPDK-source-directory>
make config T=arm64-thunderx-linuxapp-gcc install
```

10.2.5 Initialization

The octeontx eventdev is exposed as a vdev device which consists of a set of SSO group and work-slot PCIe VF devices. On EAL initialization, SSO PCIe VF devices will be probed and then the vdev device can be created from the application code, or from the EAL command line based on the number of probed/bound SSO PCIe VF device to DPDK by

- Invoking `rte_eal_vdev_init("event_octeontx")` from the application
- Using `--vdev="event_octeontx"` in the EAL options, which will call rte_eal_vdev_init() internally

Example:

```
./your_eventdev_application --vdev="event_octeontx"
```

10.2.6 Limitations

Burst mode support

Burst mode is not supported. Dequeue and Enqueue functions accepts only single event at a time.

CHAPTER 11

Xen Guide

11.1 DPDK Xen Based Packet-Switching Solution

11.1.1 Introduction

DPDK provides a para-virtualization packet switching solution, based on the Xen hypervisor's Grant Table, Note 1, which provides simple and fast packet switching capability between guest domains and host domain based on MAC address or VLAN tag.

This solution is comprised of two components; a Poll Mode Driver (PMD) as the front end in the guest domain and a switching back end in the host domain. XenStore is used to exchange configure information between the PMD front end and switching back end, including grant reference IDs for shared Virtio RX/TX rings, MAC address, device state, and so on. XenStore is an information storage space shared between domains, see further information on XenStore below.

The front end PMD can be found in the DPDK directory lib/ librte_pmd_xenvirt and back end example in examples/vhost_xen.

The PMD front end and switching back end use shared Virtio RX/TX rings as para- virtualized interface. The Virtio ring is created by the front end, and Grant table references for the ring are passed to host. The switching back end maps those grant table references and creates shared rings in a mapped address space.

The following diagram describes the functionality of the DPDK Xen Packet- Switching Solution.

Note 1 The Xen hypervisor uses a mechanism called a Grant Table to share memory between domains (http://wiki.xen.org/wiki/Grant Table).

A diagram of the design is shown below, where "gva" is the Guest Virtual Address, which is the data pointer of the mbuf, and "hva" is the Host Virtual Address:

In this design, a Virtio ring is used as a para-virtualized interface for better performance over a Xen private ring when packet switching to and from a VM. The additional performance is gained by avoiding a system call and memory map in each memory copy with a XEN private ring.

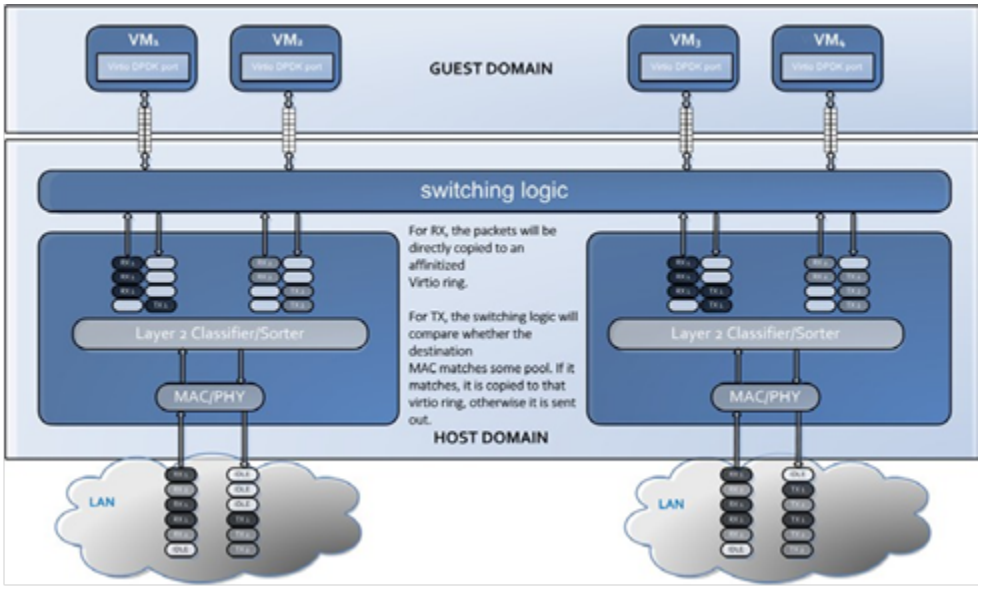


Fig. 11.1: Functionality of the DPDK Xen Packet Switching Solution.

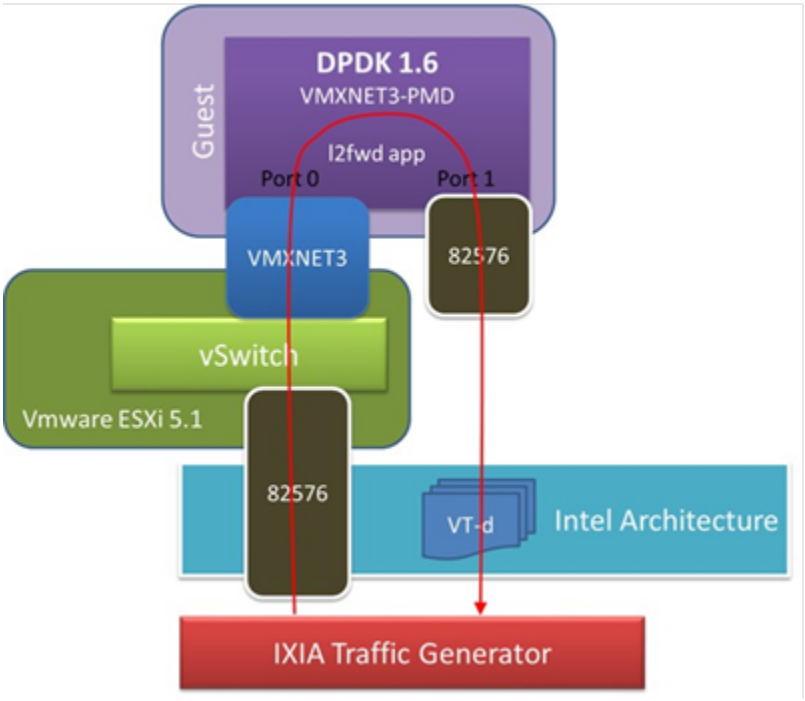


Fig. 11.2: DPDK Xen Layout

11.1.2 Device Creation

Poll Mode Driver Front End

- Mbuf pool allocation:

  To use a Xen switching solution, the DPDK application should use rte_mempool_gntalloc_create() to reserve mbuf pools during initialization. rte_mempool_gntalloc_create() creates a mempool with objects from memory allocated and managed via gntalloc/gntdev.

  The DPDK now supports construction of mempools from allocated virtual memory through the rte_mempool_xmem_create() API.

  This front end constructs mempools based on memory allocated through the xen_gntalloc driver. rte_mempool_gntalloc_create() allocates Grant pages, maps them to continuous virtual address space, and calls rte_mempool_xmem_create() to build mempools. The Grant IDs for all Grant pages are passed to the host through XenStore.

- Virtio Ring Creation:

  The Virtio queue size is defined as 256 by default in the VQ_DESC_NUM macro. Using the queue setup function, Grant pages are allocated based on ring size and are mapped to continuous virtual address space to form the Virtio ring. Normally, one ring is comprised of several pages. Their Grant IDs are passed to the host through XenStore.

  There is no requirement that this memory be physically continuous.

- Interrupt and Kick:

  There are no interrupts in DPDK Xen Switching as both front and back ends work in polling mode. There is no requirement for notification.

- Feature Negotiation:

  Currently, feature negotiation through XenStore is not supported.

- Packet Reception & Transmission:

  With mempools and Virtio rings created, the front end can operate Virtio devices, as it does in Virtio PMD for KVM Virtio devices with the exception that the host does not require notifications or deal with interrupts.

XenStore is a database that stores guest and host information in the form of (key, value) pairs. The following is an example of the information generated during the startup of the front end PMD in a guest VM (domain ID 1):

```
xenstore -ls /local/domain/1/control/dpdk
0_mempool_gref="3042,3043,3044,3045"
0_mempool_va="0x7fcbc6881000"
0_tx_vring_gref="3049"
0_rx_vring_gref="3053"
0_ether_addr="4e:0b:d0:4e:aa:f1"
0_vring_flag="3054"
...
```

Multiple mempools and multiple Virtios may exist in the guest domain, the first number is the index, starting from zero.

The idx#_mempool_va stores the guest virtual address for mempool idx#.

The idx#_ether_adder stores the MAC address of the guest Virtio device.

For idx#_rx_ring_gref, idx#_tx_ring_gref, and idx#_mempool_gref, the value is a list of Grant references. Take idx#_mempool_gref node for example, the host maps those Grant references to a continuous virtual address space.

The real Grant reference information is stored in this virtual address space, where (gref, pfn) pairs follow each other with -1 as the terminator.

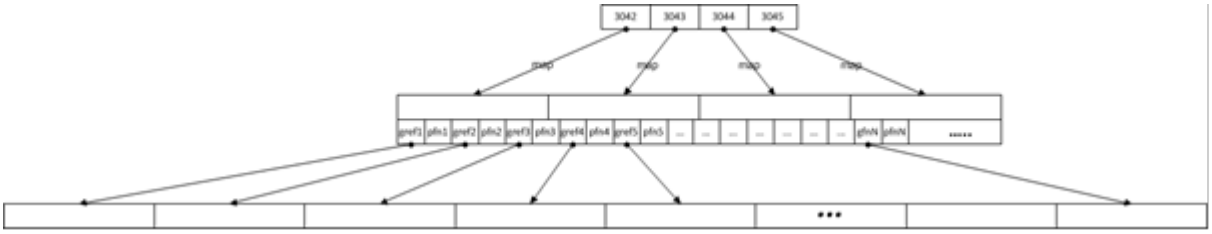


Fig. 11.3: Mapping Grant references to a continuous virtual address space

After all gref# IDs are retrieved, the host maps them to a continuous virtual address space. With the guest mempool virtual address, the host establishes 1:1 address mapping. With multiple guest mempools, the host establishes multiple address translation regions.

Switching Back End

The switching back end monitors changes in XenStore. When the back end detects that a new Virtio device has been created in a guest domain, it will:

1. Retrieve Grant and configuration information from XenStore.
2. Map and create a Virtio ring.
3. Map mempools in the host and establish address translation between the guest address and host address.
4. Select a free VMDQ pool, set its affinity with the Virtio device, and set the MAC/ VLAN filter.

Packet Reception

When packets arrive from an external network, the MAC?VLAN filter classifies packets into queues in one VMDQ pool. As each pool is bonded to a Virtio device in some guest domain, the switching back end will:

1. Fetch an available entry from the Virtio RX ring.
2. Get gva, and translate it to hva.
3. Copy the contents of the packet to the memory buffer pointed to by gva.

The DPDK application in the guest domain, based on the PMD front end, is polling the shared Virtio RX ring for available packets and receives them on arrival.

Packet Transmission

When a Virtio device in one guest domain is to transmit a packet, it puts the virtual address of the packet's data area into the shared Virtio TX ring.

The packet switching back end is continuously polling the Virtio TX ring. When new packets are available for transmission from a guest, it will:

1. Fetch an available entry from the Virtio TX ring.
2. Get gva, and translate it to hva.
3. Copy the packet from hva to the host mbuf's data area.

4. Compare the destination MAC address with all the MAC addresses of the Virtio devices it manages. If a match exists, it directly copies the packet to the matched VIrtio RX ring. Otherwise, it sends the packet out through hardware.

Note: The packet switching back end is for demonstration purposes only. The user could implement their switching logic based on this example. In this example, only one physical port on the host is supported. Multiple segments are not supported. The biggest mbuf supported is 4KB. When the back end is restarted, all front ends must also be restarted.

11.1.3 Running the Application

The following describes the steps required to run the application.

Validated Environment

Host:

Xen-hypervisor: 4.2.2

Distribution: Fedora release 18

Kernel: 3.10.0

Xen development package (including Xen, Xen-libs, xen-devel): 4.2.3

Guest:

Distribution: Fedora 16 and 18

Kernel: 3.6.11

Xen Host Prerequisites

Note that the following commands might not be the same on different Linux* distributions.

- Install xen-devel package:

```
yum install xen-devel.x86_64
```

- Start xend if not already started:

```
/etc/init.d/xend start
```

- Mount xenfs if not already mounted:

```
mount -t xenfs none /proc/xen
```

- Enlarge the limit for xen_gntdev driver:

```
modprobe -r xen_gntdev
modprobe xen_gntdev limit=1000000
```

Note: The default limit for earlier versions of the xen_gntdev driver is 1024. That is insufficient to support the mapping of multiple Virtio devices into multiple VMs, so it is necessary to enlarge the limit by reloading this module. The default limit of recent versions of xen_gntdev is 1048576. The rough calculation of this limit is:

limit=nb_mbuf# * VM#.

In DPDK examples, nb_mbuf# is normally 8192.

Building and Running the Switching Backend

1. Edit config/common_linuxapp, and change the default configuration value for the following two items:

```
CONFIG_RTE_LIBRTE_XEN_DOM0=y
CONFIG RTE_LIBRTE_PMD_XENVIRT=n
```

2. Build the target:

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

3. Ensure that RTE_SDK and RTE_TARGET are correctly set. Build the switching example:

```
make -C examples/vhost_xen/
```

4. Load the Xen DPDK memory management module and preallocate memory:

```
insmod ./x86_64-native-linuxapp-gcc/build/lib/librte_eal/linuxapp/xen_dom0/rte_
↪dom0_mm.ko
echo 2048> /sys/kernel/mm/dom0-mm/memsize-mB/memsize
```

Note: On Xen Dom0, there is no hugepage support. Under Xen Dom0, the DPDK uses a special memory management kernel module to allocate chunks of physically continuous memory. Refer to the *DPDK Getting Started Guide* for more information on memory management in the DPDK. In the above command, 4 GB memory is reserved (2048 of 2 MB pages) for DPDK.

5. Load uio_pci_generic and bind one Intel NIC controller to it:

```
modprobe uio_pci_generic
python usertools/dpdk-devbind.py -b uio_pci_generic 0000:09:00:00.0
```

In this case, 0000:09:00.0 is the PCI address for the NIC controller.

6. Run the switching back end example:

```
examples/vhost_xen/build/vhost-switch -l 0-3 -n 3 --xen-dom0 -- -p1
```

Note: The -xen-dom0 option instructs the DPDK to use the Xen kernel module to allocate memory.

Other Parameters:

- -vm2vm

  The vm2vm parameter enables/disables packet switching in software. Disabling vm2vm implies that on a VM packet transmission will always go to the Ethernet port and will not be switched to another VM

- -Stats

  The Stats parameter controls the printing of Virtio-net device statistics. The parameter specifies the interval (in seconds) at which to print statistics, an interval of 0 seconds will disable printing statistics.

Xen PMD Frontend Prerequisites

1. Install xen-devel package for accessing XenStore:

```
yum install xen-devel.x86_64
```

2. Mount xenfs, if it is not already mounted:

```
mount -t xenfs none /proc/xen
```

3. Enlarge the default limit for xen_gntalloc driver:

```
modprobe -r xen_gntalloc
modprobe xen_gntalloc limit=6000
```

Note: Before the Linux kernel version 3.8-rc5, Jan 15th 2013, a critical defect occurs when a guest is heavily allocating Grant pages. The Grant driver allocates fewer pages than expected which causes kernel memory corruption. This happens, for example, when a guest uses the v1 format of a Grant table entry and allocates more than 8192 Grant pages (this number might be different on different hypervisor versions). To work around this issue, set the limit for gntalloc driver to 6000. (The kernel normally allocates hundreds of Grant pages with one Xen front end per virtualized device). If the kernel allocates a lot of Grant pages, for example, if the user uses multiple net front devices, it is best to upgrade the Grant alloc driver. This defect has been fixed in kernel version 3.8-rc5 and later.

Building and Running the Front End

1. Edit config/common_linuxapp, and change the default configuration value:

```
CONFIG_RTE_LIBRTE_XEN_DOM0=n
CONFIG_RTE_LIBRTE_PMD_XENVIRT=y
```

2. Build the package:

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

3. Enable hugepages. Refer to the *DPDK Getting Started Guide* for instructions on how to use hugepages in the DPDK.

4. Run TestPMD. Refer to *DPDK TestPMD Application User Guide* for detailed parameter usage.

```
./x86_64-native-linuxapp-gcc/app/testpmd -l 0-3 -n 4 --vdev="net_xenvirt0,
↪mac=00:00:00:00:00:11"
testpmd>set fwd mac
testpmd>start
```

As an example to run two TestPMD instances over 2 Xen Virtio devices:

```
--vdev="net_xenvirt0,mac=00:00:00:00:00:11" --vdev="net_xenvirt1;
↪mac=00:00:00:00:00:22"
```

Usage Examples: Injecting a Packet Stream Using a Packet Generator

Loopback Mode

Run TestPMD in a guest VM:

```
./x86_64-native-linuxapp-gcc/app/testpmd -l 0-3 -n 4 --vdev="net_xenvirt0,
↪mac=00:00:00:00:00:11" -- -i --eth-peer=0,00:00:00:00:00:22
testpmd> set fwd mac
testpmd> start
```

Example output of the vhost_switch would be:

```
DATA:(0) MAC_ADDRESS 00:00:00:00:00:11 and VLAN_TAG 1000 registered.
```

The above message indicates that device 0 has been registered with MAC address 00:00:00:00:00:11 and VLAN tag 1000. Any packets received on the NIC with these values is placed on the device's receive queue.

Configure a packet stream in the packet generator, set the destination MAC address to 00:00:00:00:00:11, and VLAN to 1000, the guest Virtio receives these packets and sends them out with destination MAC address 00:00:00:00:00:22.

Inter-VM Mode

Run TestPMD in guest VM1:

```
./x86_64-native-linuxapp-gcc/app/testpmd -l 0-3 -n 4 --vdev="net_xenvirt0,
↪mac=00:00:00:00:00:11" -- -i --eth-peer=0,00:00:00:00:00:22 -- -i
```

Run TestPMD in guest VM2:

```
./x86_64-native-linuxapp-gcc/app/testpmd -l 0-3 -n 4 --vdev="net_xenvirt0,
↪mac=00:00:00:00:00:22" -- -i --eth-peer=0,00:00:00:00:00:33
```

Configure a packet stream in the packet generator, and set the destination MAC address to 00:00:00:00:00:11 and VLAN to 1000. The packets received in Virtio in guest VM1 will be forwarded to Virtio in guest VM2 and then sent out through hardware with destination MAC address 00:00:00:00:00:33.

The packet flow is:

packet generator->Virtio in guest VM1->switching backend->Virtio in guest VM2->switching backend->wire

CHAPTER 12

Contributor's Guidelines

12.1 DPDK Coding Style

12.1.1 Description

This document specifies the preferred style for source files in the DPDK source tree. It is based on the Linux Kernel coding guidelines and the FreeBSD 7.2 Kernel Developer's Manual (see man style(9)), but was heavily modified for the needs of the DPDK.

12.1.2 General Guidelines

The rules and guidelines given in this document cannot cover every situation, so the following general guidelines should be used as a fallback:

- The code style should be consistent within each individual file.
- In the case of creating new files, the style should be consistent within each file in a given directory or module.
- The primary reason for coding standards is to increase code readability and comprehensibility, therefore always use whatever option will make the code easiest to read.

Line length is recommended to be not more than 80 characters, including comments. [Tab stop size should be assumed to be 8-characters wide].

Note: The above is recommendation, and not a hard limit. However, it is expected that the recommendations should be followed in all but the rarest situations.

12.1.3 C Comment Style

Usual Comments

These comments should be used in normal cases. To document a public API, a doxygen-like format must be used: refer to *Doxygen Guidelines*.

```
/*
 * VERY important single-line comments look like this.
 */

/* Most single-line comments look like this. */

/*
 * Multi-line comments look like this.  Make them real sentences. Fill
 * them so they look like real paragraphs.
 */
```

License Header

Each file should begin with a special comment containing the appropriate copyright and license for the file. Generally this is the BSD License, except for code for Linux Kernel modules. After any copyright header, a blank line should be left before any other contents, e.g. include statements in a C file.

12.1.4 C Preprocessor Directives

Header Includes

In DPDK sources, the include files should be ordered as following:

1. libc includes (system includes first)
2. DPDK EAL includes
3. DPDK misc libraries includes
4. application-specific includes

Include files from the local application directory are included using quotes, while includes from other paths are included using angle brackets: "<>".

Example:

```
#include <stdio.h>
#include <stdlib.h>

#include <rte_eal.h>

#include <rte_ring.h>
#include <rte_mempool.h>

#include "application.h"
```

Header File Guards

Headers should be protected against multiple inclusion with the usual:

```
#ifndef _FILE_H_
#define _FILE_H_

/* Code */

#endif /* _FILE_H_ */
```

Macros

Do not `#define` or declare names except with the standard DPDK prefix: `RTE_`. This is to ensure there are no collisions with definitions in the application itself.

The names of "unsafe" macros (ones that have side effects), and the names of macros for manifest constants, are all in uppercase.

The expansions of expression-like macros are either a single token or have outer parentheses. If a macro is an inline expansion of a function, the function name is all in lowercase and the macro has the same name all in uppercase. If the macro encapsulates a compound statement, enclose it in a do-while loop, so that it can be used safely in if statements. Any final statement-terminating semicolon should be supplied by the macro invocation rather than the macro, to make parsing easier for pretty-printers and editors.

For example:

```
#define MACRO(x, y) do {                                        \
        variable = (x) + (y);                                   \
        (y) += 2;                                               \
} while(0)
```

Note: Wherever possible, enums and inline functions should be preferred to macros, since they provide additional degrees of type-safety and can allow compilers to emit extra warnings about unsafe code.

Conditional Compilation

- When code is conditionally compiled using `#ifdef` or `#if`, a comment may be added following the matching `#endif` or `#else` to permit the reader to easily discern where conditionally compiled code regions end.
- This comment should be used only for (subjectively) long regions, regions greater than 20 lines, or where a series of nested `#ifdef`'s may be confusing to the reader. Exceptions may be made for cases where code is conditionally not compiled for the purposes of lint(1), or other tools, even though the uncompiled region may be small.
- The comment should be separated from the `#endif` or `#else` by a single space.
- For short conditionally compiled regions, a closing comment should not be used.
- The comment for `#endif` should match the expression used in the corresponding `#if` or `#ifdef`.
- The comment for `#else` and `#elif` should match the inverse of the expression(s) used in the preceding `#if` and/or `#elif` statements.
- In the comments, the subexpression `defined(FOO)` is abbreviated as "FOO". For the purposes of comments, `#ifndef FOO` is treated as `#if !defined(FOO)`.

```
#ifdef KTRACE
#include <sys/ktrace.h>
#endif

#ifdef COMPAT_43
/* A large region here, or other conditional code. */
#else /* !COMPAT_43 */
/* Or here. */
#endif /* COMPAT_43 */

#ifndef COMPAT_43
/* Yet another large region here, or other conditional code. */
#else /* COMPAT_43 */
/* Or here. */
#endif /* !COMPAT_43 */
```

Note: Conditional compilation should be used only when absolutely necessary, as it increases the number of target binaries that need to be built and tested.

12.1.5 C Types

Integers

For fixed/minimum-size integer values, the project uses the form uintXX_t (from stdint.h) instead of older BSD-style integer identifiers of the form u_intXX_t.

Enumerations

- Enumeration values are all uppercase.

```
enum enumtype { ONE, TWO } et;
```

- Enum types should be used in preference to macros #defining a set of (sequential) values.
- Enum types should be prefixed with `rte_` and the elements by a suitable prefix [generally starting `RTE_<enum>_` - where <enum> is a shortname for the enum type] to avoid namespace collisions.

Bitfields

The developer should group bitfields that are included in the same integer, as follows:

```
struct grehdr {
  uint16_t rec:3,
       srr:1,
       seq:1,
       key:1,
       routing:1,
       csum:1,
       version:3,
       reserved:4,
       ack:1;
```

```
/* ... */
}
```

Variable Declarations

In declarations, do not put any whitespace between asterisks and adjacent tokens, except for tokens that are identifiers related to types. (These identifiers are the names of basic types, type qualifiers, and typedef-names other than the one being declared.) Separate these identifiers from asterisks using a single space.

For example:

```
int *x;          /* no space after asterisk */
int * const x;   /* space after asterisk when using a type qualifier */
```

- All externally-visible variables should have an `rte_` prefix in the name to avoid namespace collisions.
- Do not use uppercase letters - either in the form of ALL_UPPERCASE, or CamelCase - in variable names. Lower-case letters and underscores only.

Structure Declarations

- In general, when declaring variables in new structures, declare them sorted by use, then by size (largest to smallest), and then in alphabetical order. Sorting by use means that commonly used variables are used together and that the structure layout makes logical sense. Ordering by size then ensures that as little padding is added to the structure as possible.
- For existing structures, additions to structures should be added to the end so for backward compatibility reasons.
- Each structure element gets its own line.
- Try to make the structure readable by aligning the member names using spaces as shown below.
- Names following extremely long types, which therefore cannot be easily aligned with the rest, should be separated by a single space.

```
struct foo {
        struct foo      *next;          /* List of active foo. */
        struct mumble   amumble;        /* Comment for mumble. */
        int             bar;            /* Try to align the comments. */
        struct verylongtypename *baz;   /* Won't fit with other members */
};
```

- Major structures should be declared at the top of the file in which they are used, or in separate header files if they are used in multiple source files.
- Use of the structures should be by separate variable declarations and those declarations must be extern if they are declared in a header file.
- Externally visible structure definitions should have the structure name prefixed by `rte_` to avoid namespace collisions.

Queues

Use queue(3) macros rather than rolling your own lists, whenever possible. Thus, the previous example would be better written:

```
#include <sys/queue.h>

struct foo {
        LIST_ENTRY(foo) link;      /* Use queue macros for foo lists. */
        struct mumble   amumble;   /* Comment for mumble. */
        int             bar;       /* Try to align the comments. */
        struct verylongtypename *baz;   /* Won't fit with other members */
};
LIST_HEAD(, foo) foohead;          /* Head of global foo list. */
```

DPDK also provides an optimized way to store elements in lockless rings. This should be used in all data-path code, when there are several consumer and/or producers to avoid locking for concurrent access.

Typedefs

Avoid using typedefs for structure types.

For example, use:

```
struct my_struct_type {
/* ... */
};

struct my_struct_type my_var;
```

rather than:

```
typedef struct my_struct_type {
/* ... */
} my_struct_type;

my_struct_type my_var
```

Typedefs are problematic because they do not properly hide their underlying type; for example, you need to know if the typedef is the structure itself, as shown above, or a pointer to the structure. In addition, they must be declared exactly once, whereas an incomplete structure type can be mentioned as many times as necessary. Typedefs are difficult to use in stand-alone header files. The header that defines the typedef must be included before the header that uses it, or by the header that uses it (which causes namespace pollution), or there must be a back-door mechanism for obtaining the typedef.

Note that #defines used instead of typedefs also are problematic (since they do not propagate the pointer type correctly due to direct text replacement). For example, `#define pint int *` does not work as expected, while `typedef int *pint` does work. As stated when discussing macros, typedefs should be preferred to macros in cases like this.

When convention requires a typedef; make its name match the struct tag. Avoid typedefs ending in `_t`, except as specified in Standard C or by POSIX.

Note: It is recommended to use typedefs to define function pointer types, for reasons of code readability. This is especially true when the function type is used as a parameter to another function.

For example:

```
/**
 * Definition of a remote launch function.
 */
```

```
typedef int (lcore_function_t)(void *);

/* launch a function of lcore_function_t type */
int rte_eal_remote_launch(lcore_function_t *f, void *arg, unsigned slave_id);
```

12.1.6 C Indentation

General

- Indentation is a hard tab, that is, a tab character, not a sequence of spaces,

---

Note: Global whitespace rule in DPDK, use tabs for indentation, spaces for alignment.

- Do not put any spaces before a tab for indentation.
- If you have to wrap a long statement, put the operator at the end of the line, and indent again.
- For control statements (if, while, etc.), continuation it is recommended that the next line be indented by two tabs, rather than one, to prevent confusion as to whether the second line of the control statement forms part of the statement body or not. Alternatively, the line continuation may use additional spaces to line up to an appropriately point on the preceding line, for example, to align to an opening brace.

Note: As with all style guidelines, code should match style already in use in an existing file.

```
while (really_long_variable_name_1 == really_long_variable_name_2 &&
    var3 == var4){  /* confusing to read as */
    x = y + z;      /* control stmt body lines up with second line of */
    a = b + c;      /* control statement itself if single indent used */
}

if (really_long_variable_name_1 == really_long_variable_name_2 &&
        var3 == var4){  /* two tabs used */
    x = y + z;          /* statement body no longer lines up */
    a = b + c;
}

z = a + really + long + statement + that + needs +
        two + lines + gets + indented + on + the +
        second + and + subsequent + lines;
```

- Do not add whitespace at the end of a line.
- Do not add whitespace or a blank line at the end of a file.

Control Statements and Loops

- Include a space after keywords (if, while, for, return, switch).
- Do not use braces ({ and }) for control statements with zero or just a single statement, unless that statement is more than a single line in which case the braces are permitted.

```
for (p = buf; *p != '\0'; ++p)
        ;       /* nothing */
for (;;)
        stmt;
for (;;) {
        z = a + really + long + statement + that + needs +
                two + lines + gets + indented + on + the +
                second + and + subsequent + lines;
}
for (;;) {
        if (cond)
                stmt;
}
if (val != NULL)
        val = realloc(val, newsize);
```

- Parts of a for loop may be left empty.

```
for (; cnt < 15; cnt++) {
        stmt1;
        stmt2;
}
```

- Closing and opening braces go on the same line as the else keyword.
- Braces that are not necessary should be left out.

```
if (test)
        stmt;
else if (bar) {
        stmt;
        stmt;
} else
        stmt;
```

Function Calls

- Do not use spaces after function names.
- Commas should have a space after them.
- No spaces after `(` or `[` or preceding the `]` or `)` characters.

```
error = function(a1, a2);
if (error != 0)
        exit(error);
```

Operators

- Unary operators do not require spaces, binary operators do.
- Do not use parentheses unless they are required for precedence or unless the statement is confusing without them. However, remember that other people may be more easily confused than you.

Exit

Exits should be 0 on success, or 1 on failure.

```
        exit(0);        /*
                         * Avoid obvious comments such as
                         * "Exit 0 on success."
                         */
}
```

Local Variables

- Variables should be declared at the start of a block of code rather than in the middle. The exception to this is when the variable is `const` in which case the declaration must be at the point of first use/assignment.
- When declaring variables in functions, multiple variables per line are OK. However, if multiple declarations would cause the line to exceed a reasonable line length, begin a new set of declarations on the next line rather than using a line continuation.
- Be careful to not obfuscate the code by initializing variables in the declarations, only the last variable on a line should be initialized. If multiple variables are to be initialized when defined, put one per line.
- Do not use function calls in initializers, except for `const` variables.

```
int i = 0, j = 0, k = 0;  /* bad, too many initializer */

char a = 0;         /* OK, one variable per line with initializer */
char b = 0;

float x, y = 0.0;  /* OK, only last variable has initializer */
```

Casts and sizeof

- Casts and sizeof statements are not followed by a space.
- Always write sizeof statements with parenthesis. The redundant parenthesis rules do not apply to sizeof(var) instances.

12.1.7 C Function Definition, Declaration and Use

Prototypes

- It is recommended (and generally required by the compiler) that all non-static functions are prototyped somewhere.
- Functions local to one source module should be declared static, and should not be prototyped unless absolutely necessary.
- Functions used from other parts of code (external API) must be prototyped in the relevant include file.
- Function prototypes should be listed in a logical order, preferably alphabetical unless there is a compelling reason to use a different ordering.
- Functions that are used locally in more than one module go into a separate header file, for example, “extern.h”.
- Do not use the `__P` macro.

- Functions that are part of an external API should be documented using Doxygen-like comments above declarations. See *Doxygen Guidelines* for details.
- Functions that are part of the external API must have an `rte_` prefix on the function name.
- Do not use uppercase letters - either in the form of ALL_UPPERCASE, or CamelCase - in function names. Lower-case letters and underscores only.
- When prototyping functions, associate names with parameter types, for example:

```
void function1(int fd); /* good */
void function2(int);    /* bad */
```

- Short function prototypes should be contained on a single line. Longer prototypes, e.g. those with many parameters, can be split across multiple lines. The second and subsequent lines should be further indented as for line statement continuations as described in the previous section.

```
static char *function1(int _arg, const char *_arg2,
       struct foo *_arg3,
       struct bar *_arg4,
       struct baz *_arg5);
static void usage(void);
```

Note: Unlike function definitions, the function prototypes do not need to place the function return type on a separate line.

Definitions

- The function type should be on a line by itself preceding the function.
- The opening brace of the function body should be on a line by itself.

```
static char *
function(int a1, int a2, float fl, int a4)
{
```

- Do not declare functions inside other functions. ANSI C states that such declarations have file scope regardless of the nesting of the declaration. Hiding file declarations in what appears to be a local scope is undesirable and will elicit complaints from a good compiler.
- Old-style (K&R) function declaration should not be used, use ANSI function declarations instead as shown below.
- Long argument lists should be wrapped as described above in the function prototypes section.

```
/*
 * All major routines should have a comment briefly describing what
 * they do. The comment before the "main" routine should describe
 * what the program does.
 */
int
main(int argc, char *argv[])
{
        char *ep;
        long num;
        int ch;
```

12.1.8 C Statement Style and Conventions

NULL Pointers

- NULL is the preferred null pointer constant. Use NULL instead of `(type *)0` or `(type *)NULL`, except where the compiler does not know the destination type e.g. for variadic args to a function.
- Test pointers against NULL, for example, use:

```
if (p == NULL) /* Good, compare pointer to NULL */

if (!p) /* Bad, using ! on pointer */
```

- Do not use ! for tests unless it is a boolean, for example, use:

```
if (*p == '\0') /* check character against (char)0 */
```

Return Value

- Functions which create objects, or allocate memory, should return pointer types, and NULL on error. The error type should be indicated may setting the variable `rte_errno` appropriately.
- Functions which work on bursts of packets, such as RX-like or TX-like functions, should return the number of packets handled.
- Other functions returning int should generally behave like system calls: returning 0 on success and -1 on error, setting `rte_errno` to indicate the specific type of error.
- Where already standard in a given library, the alternative error approach may be used where the negative value is not -1 but is instead `-errno` if relevant, for example, `-EINVAL`. Note, however, to allow consistency across functions returning integer or pointer types, the previous approach is preferred for any new libraries.
- For functions where no error is possible, the function type should be `void` not `int`.
- Routines returning `void *` should not have their return values cast to any pointer type. (Typecasting can prevent the compiler from warning about missing prototypes as any implicit definition of a function returns int, which, unlike `void *`, needs a typecast to assign to a pointer variable.)

Note: The above rule about not typecasting `void *` applies to malloc, as well as to DPDK functions.

- Values in return statements should not be enclosed in parentheses.

Logging and Errors

In the DPDK environment, use the logging interface provided:

```
/* register log types for this application */
int my_logtype1 = rte_log_register("myapp.log1");
int my_logtype2 = rte_log_register("myapp.log2");

/* set global log level to INFO */
rte_log_set_global_level(RTE_LOG_INFO);

/* only display messages higher than NOTICE for log2 (default
 * is DEBUG) */
```

```
rte_log_set_level(my_logtype2, RTE_LOG_NOTICE);

/* enable all PMD logs (whose identifier string starts with "pmd") */
rte_log_set_level_regexp("pmd.*", RTE_LOG_DEBUG);

/* log in debug level */
rte_log_set_global_level(RTE_LOG_DEBUG);
RTE_LOG(DEBUG, my_logtype1, "this is is a debug level message\n");
RTE_LOG(INFO, my_logtype1, "this is is a info level message\n");
RTE_LOG(WARNING, my_logtype1, "this is is a warning level message\n");
RTE_LOG(WARNING, my_logtype2, "this is is a debug level message (not displayed)\n");

/* log in info level */
rte_log_set_global_level(RTE_LOG_INFO);
RTE_LOG(DEBUG, my_logtype1, "debug level message (not displayed)\n");
```

Branch Prediction

- When a test is done in a critical zone (called often or in a data path) the code can use the `likely()` and `unlikely()` macros to indicate the expected, or preferred fast path. They are expanded as a compiler builtin and allow the developer to indicate if the branch is likely to be taken or not. Example:

```
#include <rte_branch_prediction.h>
if (likely(x > 1))
  do_stuff();
```

Note: The use of `likely()` and `unlikely()` should only be done in performance critical paths, and only when there is a clearly preferred path, or a measured performance increase gained from doing so. These macros should be avoided in non-performance-critical code.

Static Variables and Functions

- All functions and variables that are local to a file must be declared as `static` because it can often help the compiler to do some optimizations (such as, inlining the code).
- Functions that should be inlined should to be declared as `static inline` and can be defined in a .c or a .h file.

Note: Static functions defined in a header file must be declared as `static inline` in order to prevent compiler warnings about the function being unused.

Const Attribute

The `const` attribute should be used as often as possible when a variable is read-only.

Inline ASM in C code

The `asm` and `volatile` keywords do not have underscores. The AT&T syntax should be used. Input and output operands should be named to avoid confusion, as shown in the following example:

```
asm volatile("outb %[val], %[port]"
        : :
        [port] "dN" (port),
        [val] "a" (val));
```

Control Statements

- Forever loops are done with for statements, not while statements.
- Elements in a switch statement that cascade should have a FALLTHROUGH comment. For example:

```
switch (ch) {          /* Indent the switch. */
case 'a':              /* Don't indent the case. */
        aflag = 1;     /* Indent case body one tab. */
        /* FALLTHROUGH */
case 'b':
        bflag = 1;
        break;
case '?':
default:
        usage();
        /* NOTREACHED */
}
```

12.1.9 Python Code

All Python code should work with Python 2.7+ and 3.2+ and be compliant with PEP8 (Style Guide for Python Code).

The `pep8` tool can be used for testing compliance with the guidelines.

12.2 Design

12.2.1 Environment or Architecture-specific Sources

In DPDK and DPDK applications, some code is specific to an architecture (i686, x86_64) or to an executive environment (bsdapp or linuxapp) and so on. As far as is possible, all such instances of architecture or env-specific code should be provided via standard APIs in the EAL.

By convention, a file is common if it is not located in a directory indicating that it is specific. For instance, a file located in a subdir of "x86_64" directory is specific to this architecture. A file located in a subdir of "linuxapp" is specific to this execution environment.

Note: Code in DPDK libraries and applications should be generic. The correct location for architecture or executive environment specific code is in the EAL.

When absolutely necessary, there are several ways to handle specific code:

- Use a `#ifdef` with the CONFIG option in the C code. This can be done when the differences are small and they can be embedded in the same C file:

```
#ifdef RTE_ARCH_I686
toto();
#else
titi();
#endif
```

- Use the CONFIG option in the Makefile. This is done when the differences are more significant. In this case, the code is split into two separate files that are architecture or environment specific. This should only apply inside the EAL library.

Note: As in the linux kernel, the CONFIG_ prefix is not used in C code. This is only needed in Makefiles or shell scripts.

Per Architecture Sources

The following config options can be used:

- CONFIG_RTE_ARCH is a string that contains the name of the architecture.
- CONFIG_RTE_ARCH_I686, CONFIG_RTE_ARCH_X86_64, CONFIG_RTE_ARCH_X86_64_32 or CONFIG_RTE_ARCH_PPC_64 are defined only if we are building for those architectures.

Per Execution Environment Sources

The following config options can be used:

- CONFIG_RTE_EXEC_ENV is a string that contains the name of the executive environment.
- CONFIG_RTE_EXEC_ENV_BSDAPP or CONFIG_RTE_EXEC_ENV_LINUXAPP are defined only if we are building for this execution environment.

12.2.2 Library Statistics

Description

This document describes the guidelines for DPDK library-level statistics counter support. This includes guidelines for turning library statistics on and off and requirements for preventing ABI changes when implementing statistics.

Mechanism to allow the application to turn library statistics on and off

Each library that maintains statistics counters should provide a single build time flag that decides whether the statistics counter collection is enabled or not. This flag should be exposed as a variable within the DPDK configuration file. When this flag is set, all the counters supported by current library are collected for all the instances of every object type provided by the library. When this flag is cleared, none of the counters supported by the current library are collected for any instance of any object type provided by the library:

```
# DPDK file config/common_linuxapp, config/common_bsdapp, etc.
CONFIG_RTE_<LIBRARY_NAME>_STATS_COLLECT=y/n
```

The default value for this DPDK configuration file variable (either “yes” or “no”) is decided by each library.

Prevention of ABI changes due to library statistics support

The layout of data structures and prototype of functions that are part of the library API should not be affected by whether the collection of statistics counters is turned on or off for the current library. In practical terms, this means that space should always be allocated in the API data structures for statistics counters and the statistics related API functions are always built into the code, regardless of whether the statistics counter collection is turned on or off for the current library.

When the collection of statistics counters for the current library is turned off, the counters retrieved through the statistics related API functions should have a default value of zero.

Motivation to allow the application to turn library statistics on and off

It is highly recommended that each library provides statistics counters to allow an application to monitor the library-level run-time events. Typical counters are: number of packets received/dropped/transmitted, number of buffers allocated/freed, number of occurrences for specific events, etc.

However, the resources consumed for library-level statistics counter collection have to be spent out of the application budget and the counters collected by some libraries might not be relevant to the current application. In order to avoid any unwanted waste of resources and/or performance impacts, the application should decide at build time whether the collection of library-level statistics counters should be turned on or off for each library individually.

Library-level statistics counters can be relevant or not for specific applications:

- For Application A, counters maintained by Library X are always relevant and the application needs to use them to implement certain features, such as traffic accounting, logging, application-level statistics, etc. In this case, the application requires that collection of statistics counters for Library X is always turned on.
- For Application B, counters maintained by Library X are only useful during the application debug stage and are not relevant once debug phase is over. In this case, the application may decide to turn on the collection of Library X statistics counters during the debug phase and at a later stage turn them off.
- For Application C, counters maintained by Library X are not relevant at all. It might be that the application maintains its own set of statistics counters that monitor a different set of run-time events (e.g. number of connection requests, number of active users, etc). It might also be that the application uses multiple libraries (Library X, Library Y, etc) and it is interested in the statistics counters of Library Y, but not in those of Library X. In this case, the application may decide to turn the collection of statistics counters off for Library X and on for Library Y.

The statistics collection consumes a certain amount of CPU resources (cycles, cache bandwidth, memory bandwidth, etc) that depends on:

- Number of libraries used by the current application that have statistics counters collection turned on.
- Number of statistics counters maintained by each library per object type instance (e.g. per port, table, pipeline, thread, etc).
- Number of instances created for each object type supported by each library.
- Complexity of the statistics logic collection for each counter: when only some occurrences of a specific event are valid, additional logic is typically needed to decide whether the current occurrence of the event should be counted or not. For example, in the event of packet reception, when only TCP packets with destination port within a certain range should be recorded, conditional branches are usually required. When processing a burst of packets that have been validated for header integrity, counting the number of bits set in a bitmask might be needed.

12.2.3 PF and VF Considerations

The primary goal of DPDK is to provide a userspace dataplane. Managing VFs from a PF driver is a control plane feature and developers should generally rely on the Linux Kernel for that.

Developers should work with the Linux Kernel community to get the required functionality upstream. PF functionality should only be added to DPDK for testing and prototyping purposes while the kernel work is ongoing. It should also be marked with an "EXPERIMENTAL" tag. If the functionality isn't upstreamable then a case can be made to maintain the PF functionality in DPDK without the EXPERIMENTAL tag.

12.3 Managing ABI updates

12.3.1 Description

This document details some methods for handling ABI management in the DPDK. Note this document is not exhaustive, in that C library versioning is flexible allowing multiple methods to achieve various goals, but it will provide the user with some introductory methods

12.3.2 General Guidelines

1. Whenever possible, ABI should be preserved
2. The libraries marked in experimental state may change without constraint.
3. The addition of symbols is generally not problematic
4. The modification of symbols can generally be managed with versioning
5. The removal of symbols generally is an ABI break and requires bumping of the LIBABIVER macro

12.3.3 What is an ABI

An ABI (Application Binary Interface) is the set of runtime interfaces exposed by a library. It is similar to an API (Application Programming Interface) but is the result of compilation. It is also effectively cloned when applications link to dynamic libraries. That is to say when an application is compiled to link against dynamic libraries, it is assumed that the ABI remains constant between the time the application is compiled/linked, and the time that it runs. Therefore, in the case of dynamic linking, it is critical that an ABI is preserved, or (when modified), done in such a way that the application is unable to behave improperly or in an unexpected fashion.

12.3.4 The DPDK ABI policy

ABI versions are set at the time of major release labeling, and the ABI may change multiple times, without warning, between the last release label and the HEAD label of the git tree.

ABI versions, once released, are available until such time as their deprecation has been noted in the Release Notes for at least one major release cycle. For example consider the case where the ABI for DPDK 2.0 has been shipped and then a decision is made to modify it during the development of DPDK 2.1. The decision will be recorded in the Release Notes for the DPDK 2.1 release and the modification will be made available in the DPDK 2.2 release.

ABI versions may be deprecated in whole or in part as needed by a given update.

Some ABI changes may be too significant to reasonably maintain multiple versions. In those cases ABI's may be updated without backward compatibility being provided. The requirements for doing so are:

1. At least 3 acknowledgments of the need to do so must be made on the dpdk.org mailing list.
2. The changes (including an alternative map file) must be gated with the `RTE_NEXT_ABI` option, and provided with a deprecation notice at the same time. It will become the default ABI in the next release.
3. A full deprecation cycle, as explained above, must be made to offer downstream consumers sufficient warning of the change.
4. At the beginning of the next release cycle, every `RTE_NEXT_ABI` conditions will be removed, the `LIBABIVER` variable in the makefile(s) where the ABI is changed will be incremented, and the map files will be updated.

Note that the above process for ABI deprecation should not be undertaken lightly. ABI stability is extremely important for downstream consumers of the DPDK, especially when distributed in shared object form. Every effort should be made to preserve the ABI whenever possible. The ABI should only be changed for significant reasons, such as performance enhancements. ABI breakage due to changes such as reorganizing public structure fields for aesthetic or readability purposes should be avoided.

12.3.5 Examples of Deprecation Notices

The following are some examples of ABI deprecation notices which would be added to the Release Notes:

- The Macro `#RTE_FOO` is deprecated and will be removed with version 2.0, to be replaced with the inline function `rte_foo()`.
- The function `rte_mbuf_grok()` has been updated to include a new parameter in version 2.0. Backwards compatibility will be maintained for this function until the release of version 2.1
- The members of `struct rte_foo` have been reorganized in release 2.0 for performance reasons. Existing binary applications will have backwards compatibility in release 2.0, while newly built binaries will need to reference the new structure variant `struct rte_foo2`. Compatibility will be removed in release 2.2, and all applications will require updating and rebuilding to the new structure at that time, which will be renamed to the original `struct rte_foo`.
- Significant ABI changes are planned for the `librte_dostuff` library. The upcoming release 2.0 will not contain these changes, but release 2.1 will, and no backwards compatibility is planned due to the extensive nature of these changes. Binaries using this library built prior to version 2.1 will require updating and recompilation.

12.3.6 Versioning Macros

When a symbol is exported from a library to provide an API, it also provides a calling convention (ABI) that is embodied in its name, return type and arguments. Occasionally that function may need to change to accommodate new functionality or behavior. When that occurs, it is desirable to allow for backward compatibility for a time with older binaries that are dynamically linked to the DPDK.

To support backward compatibility the `lib/librte_compat/rte_compat.h` header file provides macros to use when updating exported functions. These macros are used in conjunction with the `rte_<library>_version.map` file for a given library to allow multiple versions of a symbol to exist in a shared library so that older binaries need not be immediately recompiled.

The macros exported are:

- `VERSION_SYMBOL(b, e, n)`: Creates a symbol version table entry binding versioned symbol `b@DPDK_n` to the internal function `b_e`.
- `BIND_DEFAULT_SYMBOL(b, e, n)`: Creates a symbol version entry instructing the linker to bind references to symbol `b` to the internal symbol `b_e`.
- `MAP_STATIC_SYMBOL(f, p)`: Declare the prototype `f`, and map it to the fully qualified function `p`, so that if a symbol becomes versioned, it can still be mapped back to the public symbol name.

12.3.7 Setting a Major ABI version

Downstreams might want to provide different DPDK releases at the same time to support multiple consumers of DPDK linked against older and newer sonames.

Also due to the interdependencies that DPDK libraries can have applications might end up with an executable space in which multiple versions of a library are mapped by ld.so.

Think of LibA that got an ABI bump and LibB that did not get an ABI bump but is depending on LibA.

Note: Application -> LibA.old -> LibB.new -> LibA.new

That is a conflict which can be avoided by setting `CONFIG_RTE_MAJOR_ABI`. If set, the value of `CONFIG_RTE_MAJOR_ABI` overwrites all - otherwise per library - versions defined in the libraries `LIBABIVER`. An example might be `CONFIG_RTE_MAJOR_ABI=16.11` which will make all libraries `librte<?>.so.16.11` instead of `librte<?>.so.<LIBABIVER>`.

12.3.8 Examples of ABI Macro use

Updating a public API

Assume we have a function as follows

```
/*
 * Create an acl context object for apps to
 * manipulate
 */
struct rte_acl_ctx *
rte_acl_create(const struct rte_acl_param *param)
{
        ...
}
```

Assume that struct rte_acl_ctx is a private structure, and that a developer wishes to enhance the acl api so that a debugging flag can be enabled on a per-context basis. This requires an addition to the structure (which, being private, is safe), but it also requires modifying the code as follows

```
/*
 * Create an acl context object for apps to
 * manipulate
 */
struct rte_acl_ctx *
rte_acl_create(const struct rte_acl_param *param, int debug)
{
        ...
}
```

Note also that, being a public function, the header file prototype must also be changed, as must all the call sites, to reflect the new ABI footprint. We will maintain previous ABI versions that are accessible only to previously compiled binaries

The addition of a parameter to the function is ABI breaking as the function is public, and existing application may use it in its current form. However, the compatibility macros in DPDK allow a developer to use symbol versioning so that multiple functions can be mapped to the same public symbol based on when an application was linked to it. To see

how this is done, we start with the requisite libraries version map file. Initially the version map file for the acl library looks like this

```
DPDK_2.0 {
     global:

     rte_acl_add_rules;
     rte_acl_build;
     rte_acl_classify;
     rte_acl_classify_alg;
     rte_acl_classify_scalar;
     rte_acl_create;
     rte_acl_dump;
     rte_acl_find_existing;
     rte_acl_free;
     rte_acl_ipv4vlan_add_rules;
     rte_acl_ipv4vlan_build;
     rte_acl_list_dump;
     rte_acl_reset;
     rte_acl_reset_rules;
     rte_acl_set_ctx_classify;

     local: *;
};
```

This file needs to be modified as follows

```
DPDK_2.0 {
     global:

     rte_acl_add_rules;
     rte_acl_build;
     rte_acl_classify;
     rte_acl_classify_alg;
     rte_acl_classify_scalar;
     rte_acl_create;
     rte_acl_dump;
     rte_acl_find_existing;
     rte_acl_free;
     rte_acl_ipv4vlan_add_rules;
     rte_acl_ipv4vlan_build;
     rte_acl_list_dump;
     rte_acl_reset;
     rte_acl_reset_rules;
     rte_acl_set_ctx_classify;

     local: *;
};

DPDK_2.1 {
     global:
     rte_acl_create;

} DPDK_2.0;
```

The addition of the new block tells the linker that a new version node is available (DPDK_2.1), which contains the symbol rte_acl_create, and inherits the symbols from the DPDK_2.0 node. This list is directly translated into a list of exported symbols when DPDK is compiled as a shared library

Next, we need to specify in the code which function map to the rte_acl_create symbol at which versions. First, at the site of the initial symbol definition, we need to update the function so that it is uniquely named, and not in conflict with the public symbol name

```
 struct rte_acl_ctx *
-rte_acl_create(const struct rte_acl_param *param)
+rte_acl_create_v20(const struct rte_acl_param *param)
{
       size_t sz;
       struct rte_acl_ctx *ctx;
       ...
```

Note that the base name of the symbol was kept intact, as this is conducive to the macros used for versioning symbols. That is our next step, mapping this new symbol name to the initial symbol name at version node 2.0. Immediately after the function, we add this line of code

```
VERSION_SYMBOL(rte_acl_create, _v20, 2.0);
```

Remembering to also add the rte_compat.h header to the requisite c file where these changes are being made. The above macro instructs the linker to create a new symbol `rte_acl_create@DPDK_2.0`, which matches the symbol created in older builds, but now points to the above newly named function. We have now mapped the original rte_acl_create symbol to the original function (but with a new name)

Next, we need to create the 2.1 version of the symbol. We create a new function name, with a different suffix, and implement it appropriately

```
struct rte_acl_ctx *
rte_acl_create_v21(const struct rte_acl_param *param, int debug);
{
     struct rte_acl_ctx *ctx = rte_acl_create_v20(param);

     ctx->debug = debug;

     return ctx;
}
```

This code serves as our new API call. Its the same as our old call, but adds the new parameter in place. Next we need to map this function to the symbol `rte_acl_create@DPDK_2.1`. To do this, we modify the public prototype of the call in the header file, adding the macro there to inform all including applications, that on re-link, the default rte_acl_create symbol should point to this function. Note that we could do this by simply naming the function above rte_acl_create, and the linker would chose the most recent version tag to apply in the version script, but we can also do this in the header file

```
struct rte_acl_ctx *
-rte_acl_create(const struct rte_acl_param *param);
+rte_acl_create(const struct rte_acl_param *param, int debug);
+BIND_DEFAULT_SYMBOL(rte_acl_create, _v21, 2.1);
```

The BIND_DEFAULT_SYMBOL macro explicitly tells applications that include this header, to link to the rte_acl_create_v21 function and apply the DPDK_2.1 version node to it. This method is more explicit and flexible than just re-implementing the exact symbol name, and allows for other features (such as linking to the old symbol version by default, when the new ABI is to be opt-in for a period.

One last thing we need to do. Note that we've taken what was a public symbol, and duplicated it into two uniquely and differently named symbols. We've then mapped each of those back to the public symbol `rte_acl_create` with different version tags. This only applies to dynamic linking, as static linking has no notion of versioning. That leaves this code in a position of no longer having a symbol simply named `rte_acl_create` and a static build will fail on that missing symbol.

To correct this, we can simply map a function of our choosing back to the public symbol in the static build with the `MAP_STATIC_SYMBOL` macro. Generally the assumption is that the most recent version of the symbol is the one you want to map. So, back in the C file where, immediately after `rte_acl_create_v21` is defined, we add this

```
struct rte_acl_ctx *
rte_acl_create_v21(const struct rte_acl_param *param, int debug)
{
     ...
}
MAP_STATIC_SYMBOL(struct rte_acl_ctx *rte_acl_create(const struct rte_acl_param␣
↪*param, int debug), rte_acl_create_v21);
```

That tells the compiler that, when building a static library, any calls to the symbol `rte_acl_create` should be linked to `rte_acl_create_v21`

That's it, on the next shared library rebuild, there will be two versions of rte_acl_create, an old DPDK_2.0 version, used by previously built applications, and a new DPDK_2.1 version, used by future built applications.

Deprecating part of a public API

Lets assume that you've done the above update, and after a few releases have passed you decide you would like to retire the old version of the function. After having gone through the ABI deprecation announcement process, removal is easy. Start by removing the symbol from the requisite version map file:

```
  DPDK_2.0 {
       global:

       rte_acl_add_rules;
       rte_acl_build;
       rte_acl_classify;
       rte_acl_classify_alg;
       rte_acl_classify_scalar;
       rte_acl_dump;
-      rte_acl_create
       rte_acl_find_existing;
       rte_acl_free;
       rte_acl_ipv4vlan_add_rules;
       rte_acl_ipv4vlan_build;
       rte_acl_list_dump;
       rte_acl_reset;
       rte_acl_reset_rules;
       rte_acl_set_ctx_classify;

       local: *;
  };

  DPDK_2.1 {
       global:
       rte_acl_create;
  } DPDK_2.0;
```

Next remove the corresponding versioned export.

```
-VERSION_SYMBOL(rte_acl_create, _v20, 2.0);
```

Note that the internal function definition could also be removed, but its used in our example by the newer version _v21, so we leave it in place. This is a coding style choice.

Lastly, we need to bump the LIBABIVER number for this library in the Makefile to indicate to applications doing dynamic linking that this is a later, and possibly incompatible library version:

```
-LIBABIVER := 1
+LIBABIVER := 2
```

Deprecating an entire ABI version

While removing a symbol from and ABI may be useful, it is often more practical to remove an entire version node at once. If a version node completely specifies an API, then removing part of it, typically makes it incomplete. In those cases it is better to remove the entire node

To do this, start by modifying the version map file, such that all symbols from the node to be removed are merged into the next node in the map

In the case of our map above, it would transform to look as follows

```
  DPDK_2.1 {
       global:

       rte_acl_add_rules;
       rte_acl_build;
       rte_acl_classify;
       rte_acl_classify_alg;
       rte_acl_classify_scalar;
       rte_acl_dump;
       rte_acl_create
       rte_acl_find_existing;
       rte_acl_free;
       rte_acl_ipv4vlan_add_rules;
       rte_acl_ipv4vlan_build;
       rte_acl_list_dump;
       rte_acl_reset;
       rte_acl_reset_rules;
       rte_acl_set_ctx_classify;

       local: *;
};
```

Then any uses of BIND_DEFAULT_SYMBOL that pointed to the old node should be updated to point to the new version node in any header files for all affected symbols.

```
-BIND_DEFAULT_SYMBOL(rte_acl_create, _v20, 2.0);
+BIND_DEFAULT_SYMBOL(rte_acl_create, _v21, 2.1);
```

Lastly, any VERSION_SYMBOL macros that point to the old version node should be removed, taking care to keep, where need old code in place to support newer versions of the symbol.

12.3.9 Running the ABI Validator

The `devtools` directory in the DPDK source tree contains a utility program, `validate-abi.sh`, for validating the DPDK ABI based on the Linux ABI Compliance Checker.

This has a dependency on the `abi-compliance-checker` and `and abi-dumper` utilities which can be installed via a package manager. For example:

```
sudo yum install abi-compliance-checker
sudo yum install abi-dumper
```

The syntax of the `validate-abi.sh` utility is:

```
./devtools/validate-abi.sh <REV1> <REV2> <TARGET>
```

Where `REV1` and `REV2` are valid gitrevisions(7) https://www.kernel.org/pub/software/scm/git/docs/gitrevisions.html on the local repo and target is the usual DPDK compilation target.

For example:

```
# Check between the previous and latest commit:
./devtools/validate-abi.sh HEAD~1 HEAD x86_64-native-linuxapp-gcc

# Check between two tags:
./devtools/validate-abi.sh v2.0.0 v2.1.0 x86_64-native-linuxapp-gcc

# Check between git master and local topic-branch "vhost-hacking":
./devtools/validate-abi.sh master vhost-hacking x86_64-native-linuxapp-gcc
```

After the validation script completes (it can take a while since it need to compile both tags) it will create compatibility reports in the `./compat_report` directory. Listed incompatibilities can be found as follows:

```
grep -lr Incompatible compat_reports/
```

12.4 DPDK Documentation Guidelines

This document outlines the guidelines for writing the DPDK Guides and API documentation in RST and Doxygen format.

It also explains the structure of the DPDK documentation and shows how to build the Html and PDF versions of the documents.

12.4.1 Structure of the Documentation

The DPDK source code repository contains input files to build the API documentation and User Guides.

The main directories that contain files related to documentation are shown below:

```
lib
|-- librte_acl
|-- librte_cfgfile
|-- librte_cmdline
|-- librte_compat
|-- librte_eal
|   |-- ...
...
doc
|-- api
+-- guides
    |-- freebsd_gsg
    |-- linux_gsg
    |-- prog_guide
    |-- sample_app_ug
```

```
    |-- guidelines
    |-- testpmd_app_ug
    |-- rel_notes
    |-- nics
    |-- xen
    |-- ...
```

The API documentation is built from Doxygen comments in the header files. These files are mainly in the `lib/librte_*` directories although some of the Poll Mode Drivers in `drivers/net` are also documented with Doxygen.

The configuration files that are used to control the Doxygen output are in the `doc/api` directory.

The user guides such as *The Programmers Guide* and the *FreeBSD* and *Linux Getting Started* Guides are generated from RST markup text files using the Sphinx Documentation Generator.

These files are included in the `doc/guides/` directory. The output is controlled by the `doc/guides/conf.py` file.

12.4.2 Role of the Documentation

The following items outline the roles of the different parts of the documentation and when they need to be updated or added to by the developer.

- **Release Notes**

  The Release Notes document which features have been added in the current and previous releases of DPDK and highlight any known issues. The Releases Notes also contain notifications of features that will change ABI compatibility in the next major release.

  Developers should include updates to the Release Notes with patch sets that relate to any of the following sections:

  - New Features
  - Resolved Issues (see below)
  - Known Issues
  - API Changes
  - ABI Changes
  - Shared Library Versions

  Resolved Issues should only include issues from previous releases that have been resolved in the current release. Issues that are introduced and then fixed within a release cycle do not have to be included here.

  Refer to the Release Notes from the previous DPDK release for the correct format of each section.

- **API documentation**

  The API documentation explains how to use the public DPDK functions. The API index page shows the generated API documentation with related groups of functions.

  The API documentation should be updated via Doxygen comments when new functions are added.

- **Getting Started Guides**

  The Getting Started Guides show how to install and configure DPDK and how to run DPDK based applications on different OSes.

  A Getting Started Guide should be added when DPDK is ported to a new OS.

- **The Programmers Guide**

  The Programmers Guide explains how the API components of DPDK such as the EAL, Memzone, Rings and the Hash Library work. It also explains how some higher level functionality such as Packet Distributor, Packet Framework and KNI work. It also shows the build system and explains how to add applications.

  The Programmers Guide should be expanded when new functionality is added to DPDK.

- **App Guides**

  The app guides document the DPDK applications in the `app` directory such as `testpmd`.

  The app guides should be updated if functionality is changed or added.

- **Sample App Guides**

  The sample app guides document the DPDK example applications in the examples directory. Generally they demonstrate a major feature such as L2 or L3 Forwarding, Multi Process or Power Management. They explain the purpose of the sample application, how to run it and step through some of the code to explain the major functionality.

  A new sample application should be accompanied by a new sample app guide. The guide for the Skeleton Forwarding app is a good starting reference.

- **Network Interface Controller Drivers**

  The NIC Drivers document explains the features of the individual Poll Mode Drivers, such as software requirements, configuration and initialization.

  New documentation should be added for new Poll Mode Drivers.

- **Guidelines**

  The guideline documents record community process, expectations and design directions.

  They can be extended, amended or discussed by submitting a patch and getting community approval.

12.4.3 Building the Documentation

Dependencies

The following dependencies must be installed to build the documentation:

- Doxygen.
- Sphinx (also called python-sphinx).
- TexLive (at least TexLive-core and the extra Latex support).
- Inkscape.

Doxygen generates documentation from commented source code. It can be installed as follows:

```
# Ubuntu/Debian.
sudo apt-get -y install doxygen

# Red Hat/Fedora.
sudo dnf     -y install doxygen
```

Sphinx is a Python documentation tool for converting RST files to Html or to PDF (via LaTeX). For full support with figure and table captioning the latest version of Sphinx can be installed as follows:

```
# Ubuntu/Debian.
sudo apt-get -y install python-pip
sudo pip install --upgrade sphinx
sudo pip install --upgrade sphinx_rtd_theme

# Red Hat/Fedora.
sudo dnf     -y install python-pip
sudo pip install --upgrade sphinx
sudo pip install --upgrade sphinx_rtd_theme
```

For further information on getting started with Sphinx see the Sphinx Tutorial.

Note: To get full support for Figure and Table numbering it is best to install Sphinx 1.3.1 or later.

Inkscape is a vector based graphics program which is used to create SVG images and also to convert SVG images to PDF images. It can be installed as follows:

```
# Ubuntu/Debian.
sudo apt-get -y install inkscape

# Red Hat/Fedora.
sudo dnf     -y install inkscape
```

TexLive is an installation package for Tex/LaTeX. It is used to generate the PDF versions of the documentation. The main required packages can be installed as follows:

```
# Ubuntu/Debian.
sudo apt-get -y install texlive-latex-extra

# Red Hat/Fedora, selective install.
sudo dnf     -y install texlive-collection-latexextra
```

Build commands

The documentation is built using the standard DPDK build system. Some examples are shown below:

- Generate all the documentation targets:

  ```
  make doc
  ```

- Generate the Doxygen API documentation in Html:

  ```
  make doc-api-html
  ```

- Generate the guides documentation in Html:

  ```
  make doc-guides-html
  ```

- Generate the guides documentation in Pdf:

  ```
  make doc-guides-pdf
  ```

The output of these commands is generated in the `build` directory:

```
build/doc
      |-- html
      |   |-- api
      |   +-- guides
      |
      +-- pdf
          +-- guides
```

Note: Make sure to fix any Sphinx or Doxygen warnings when adding or updating documentation.

The documentation output files can be removed as follows:

```
make doc-clean
```

12.4.4 Document Guidelines

Here are some guidelines in relation to the style of the documentation:

- Document the obvious as well as the obscure since it won't always be obvious to the reader. For example an instruction like "Set up 64 2MB Hugepages" is better when followed by a sample commandline or a link to the appropriate section of the documentation.
- Use American English spellings throughout. This can be checked using the `aspell` utility:

```
aspell --lang=en_US --check doc/guides/sample_app_ug/mydoc.rst
```

12.4.5 RST Guidelines

The RST (reStructuredText) format is a plain text markup format that can be converted to Html, PDF or other formats. It is most closely associated with Python but it can be used to document any language. It is used in DPDK to document everything apart from the API.

The Sphinx documentation contains a very useful RST Primer which is a good place to learn the minimal set of syntax required to format a document.

The official reStructuredText website contains the specification for the RST format and also examples of how to use it. However, for most developers the RST Primer is a better resource.

The most common guidelines for writing RST text are detailed in the Documenting Python guidelines. The additional guidelines below reiterate or expand upon those guidelines.

Line Length

- The recommended style for the DPDK documentation is to put sentences on separate lines. This allows for easier reviewing of patches. Multiple sentences which are not separated by a blank line are joined automatically into paragraphs, for example:

```
Here is an example sentence.
Long sentences over the limit shown below can be wrapped onto
a new line.
These three sentences will be joined into the same paragraph.

```

```
This is a new paragraph, since it is separated from the
previous paragraph by a blank line.
```

This would be rendered as follows:

> *Here is an example sentence. Long sentences over the limit shown below can be wrapped onto a new line. These three sentences will be joined into the same paragraph.*
>
> *This is a new paragraph, since it is separated from the previous paragraph by a blank line.*

- Long sentences should be wrapped at 120 characters +/- 10 characters. They should be wrapped at words.
- Lines in literal blocks must by less than 80 characters since they aren't wrapped by the document formatters and can exceed the page width in PDF documents.

Whitespace

- Standard RST indentation is 3 spaces. Code can be indented 4 spaces, especially if it is copied from source files.
- No tabs. Convert tabs in embedded code to 4 or 8 spaces.
- No trailing whitespace.
- Add 2 blank lines before each section header.
- Add 1 blank line after each section header.
- Add 1 blank line between each line of a list.

Section Headers

- Section headers should use the following underline formats:

```
Level 1 Heading
===============


Level 2 Heading
---------------


Level 3 Heading
~~~~~~~~~~~~~~~


Level 4 Heading
^^^^^^^^^^^^^^^
```

- Level 4 headings should be used sparingly.
- The underlines should match the length of the text.
- In general, the heading should be less than 80 characters, for conciseness.
- As noted above:
  - Add 2 blank lines before each section header.
  - Add 1 blank line after each section header.

Lists

- Bullet lists should be formatted with a leading * as follows:

```
* Item one.

* Item two is a long line that is wrapped and then indented to match
  the start of the previous line.

* One space character between the bullet and the text is preferred.
```

- Numbered lists can be formatted with a leading number but the preference is to use `#.` which will give automatic numbering. This is more convenient when adding or removing items:

```
#. Item one.

#. Item two is a long line that is wrapped and then indented to match
   the start of the previous line.

#. Item three.
```

- Definition lists can be written with or without a bullet:

```
* Item one.

  Some text about item one.

* Item two.

  Some text about item two.
```

- All lists, and sub-lists, must be separated from the preceding text by a blank line. This is a syntax requirement.
- All list items should be separated by a blank line for readability.

Code and Literal block sections

- Inline text that is required to be rendered with a fixed width font should be enclosed in backquotes like this: ``text``, so that it appears like this: `text`.
- Fixed width, literal blocks of texts should be indented at least 3 spaces and prefixed with `::` like this:

```
Here is some fixed width text::

   0x0001 0x0001 0x00FF 0x00FF
```

- It is also possible to specify an encoding for a literal block using the `.. code-block::` directive so that syntax highlighting can be applied. Examples of supported highlighting are:

```
.. code-block:: console
.. code-block:: c
.. code-block:: python
.. code-block:: diff
.. code-block:: none
```

That can be applied as follows:

```
.. code-block:: c

   #include<stdio.h>

   int main() {

      printf("Hello World\n");

      return 0;
   }
```

Which would be rendered as:

```
#include<stdio.h>

int main() {

   printf("Hello World\n");

   return 0;
}
```

- The default encoding for a literal block using the simplified `::` directive is `none`.
- Lines in literal blocks must be less than 80 characters since they can exceed the page width when converted to PDF documentation. For long literal lines that exceed that limit try to wrap the text at sensible locations. For example a long command line could be documented like this and still work if copied directly from the docs:

```
build/app/testpmd -l 0-2 -n3 --vdev=net_pcap0,iface=eth0     \
                          --vdev=net_pcap1,iface=eth1     \
                          -- -i --nb-cores=2 --nb-ports=2 \
                             --total-num-mbufs=2048
```

- Long lines that cannot be wrapped, such as application output, should be truncated to be less than 80 characters.

Images

- All images should be in SVG scalar graphics format. They should be true SVG XML files and should not include binary formats embedded in a SVG wrapper.
- The DPDK documentation contains some legacy images in PNG format. These will be converted to SVG in time.
- Inkscape is the recommended graphics editor for creating the images. Use some of the older images in `doc/guides/prog_guide/img/` as a template, for example `mbuf1.svg` or `ring-enqueue1.svg`.
- The SVG images should include a copyright notice, as an XML comment.
- Images in the documentation should be formatted as follows:
  - The image should be preceded by a label in the format `.. _figure_XXXX:` with a leading underscore and where `XXXX` is a unique descriptive name.
  - Images should be included using the `.. figure::` directive and the file type should be set to `*` (not `.svg`). This allows the format of the image to be changed if required, without updating the documentation.
  - Images must have a caption as part of the `.. figure::` directive.
- Here is an example of the previous three guidelines:

```
.. _figure_mempool:

.. figure:: img/mempool.*

   A mempool in memory with its associated ring.
```

- Images can then be linked to using the `:numref:` directive:

```
The mempool layout is shown in :numref:`figure_mempool`.
```

This would be rendered as: *The mempool layout is shown in Fig 6.3*.

Note: The `:numref:` directive requires Sphinx 1.3.1 or later. With earlier versions it will still be rendered as a link but won't have an automatically generated number.

- The caption of the image can be generated, with a link, using the `:ref:` directive:

```
:ref:`figure_mempool`
```

This would be rendered as: *A mempool in memory with its associated ring.*

Tables

- RST tables should be used sparingly. They are hard to format and to edit, they are often rendered incorrectly in PDF format, and the same information can usually be shown just as clearly with a definition or bullet list.
- Tables in the documentation should be formatted as follows:
  - The table should be preceded by a label in the format `.. _table_XXXX:` with a leading underscore and where `XXXX` is a unique descriptive name.
  - Tables should be included using the `.. table::` directive and must have a caption.
- Here is an example of the previous two guidelines:

```
.. _table_qos_pipes:

.. table:: Sample configuration for QOS pipes.

   +----------+----------+----------+
   | Header 1 | Header 2 | Header 3 |
   |          |          |          |
   +==========+==========+==========+
   | Text     | Text     | Text     |
   +----------+----------+----------+
   | ...      | ...      | ...      |
   +----------+----------+----------+
```

- Tables can be linked to using the `:numref:` and `:ref:` directives, as shown in the previous section for images. For example:

```
The QOS configuration is shown in :numref:`table_qos_pipes`.
```

- Tables should not include merged cells since they are not supported by the PDF renderer.

Hyperlinks

- Links to external websites can be plain URLs. The following is rendered as http://dpdk.org:

```
http://dpdk.org
```

- They can contain alternative text. The following is rendered as Check out DPDK:

```
`Check out DPDK <http://dpdk.org>`_
```

- An internal link can be generated by placing labels in the document with the format `.. _label_name`.
- The following links to the top of this section: *Hyperlinks*:

```
.. _links:

Hyperlinks
~~~~~~~~~~

* The following links to the top of this section: :ref:`links`:
```

Note: The label must have a leading underscore but the reference to it must omit it. This is a frequent cause of errors and warnings.

- The use of a label is preferred since it works across files and will still work if the header text changes.

12.4.6 Doxygen Guidelines

The DPDK API is documented using Doxygen comment annotations in the header files. Doxygen is a very powerful tool, it is extremely configurable and with a little effort can be used to create expressive documents. See the Doxygen website for full details on how to use it.

The following are some guidelines for use of Doxygen in the DPDK API documentation:

- New libraries that are documented with Doxygen should be added to the Doxygen configuration file: `doc/api/doxy-api.conf`. It is only required to add the directory that contains the files. It isn't necessary to explicitly name each file since the configuration matches all `rte_*.h` files in the directory.
- Use proper capitalization and punctuation in the Doxygen comments since they will become sentences in the documentation. This in particular applies to single line comments, which is the case the is most often forgotten.
- Use `@` style Doxygen commands instead of `\` style commands.
- Add a general description of each library at the head of the main header files:

```
/**
 * @file
 * RTE Mempool.
 *
 * A memory pool is an allocator of fixed-size object. It is
 * identified by its name, and uses a ring to store free objects.
 * ...
 */
```

- Document the purpose of a function, the parameters used and the return value:

```
/**
 * Attach a new Ethernet device specified by arguments.
 *
 * @param devargs
```

```
 *  A pointer to a strings array describing the new device
 *  to be attached. The strings should be a pci address like
 *  `0000:01:00.0` or **virtual** device name like `net_pcap0`.
 * @param port_id
 *  A pointer to a port identifier actually attached.
 *
 * @return
 *  0 on success and port_id is filled, negative on error.
 */
int rte_eth_dev_attach(const char *devargs, uint8_t *port_id);
```

- Doxygen supports Markdown style syntax such as bold, italics, fixed width text and lists. For example the second line in the `devargs` parameter in the previous example will be rendered as:

  > The strings should be a pci address like `0000:01:00.0` or **virtual** device name like `net_pcap0`.

- Use `-` instead of `*` for lists within the Doxygen comment since the latter can get confused with the comment delimiter.
- Add an empty line between the function description, the `@params` and `@return` for readability.
- Place the `@params` description on separate line and indent it by 2 spaces. (It would be better to use no indentation since this is more common and also because checkpatch complains about leading whitespace in comments. However this is the convention used in the existing DPDK code.)
- Documented functions can be linked to simply by adding `()` to the function name:

```
/**
 * The functions exported by the application Ethernet API to setup
 * a device designated by its port identifier must be invoked in
 * the following order:
 *     - rte_eth_dev_configure()
 *     - rte_eth_tx_queue_setup()
 *     - rte_eth_rx_queue_setup()
 *     - rte_eth_dev_start()
 */
```

  In the API documentation the functions will be rendered as links, see the online section of the rte_ethdev.h docs that contains the above text.

- The `@see` keyword can be used to create a *see also* link to another file or library. This directive should be placed on one line at the bottom of the documentation section.

```
/**
 * ...
 *
 * Some text that references mempools.
 *
 * @see eal_memzone.c
 */
```

- Doxygen supports two types of comments for documenting variables, constants and members: prefix and postfix:

```
/** This is a prefix comment. */
#define RTE_FOO_ERROR  0x023.

#define RTE_BAR_ERROR  0x024. /**< This is a postfix comment. */
```

- Postfix comments are preferred for struct members and constants if they can be documented in the same way:

```
struct rte_eth_stats {
    uint64_t ipackets; /**< Total number of received packets. */
    uint64_t opackets; /**< Total number of transmitted packets.*/
    uint64_t ibytes;   /**< Total number of received bytes. */
    uint64_t obytes;   /**< Total number of transmitted bytes. */
    uint64_t imissed;  /**< Total of RX missed packets. */
    uint64_t ibadcrc;  /**< Total of RX packets with CRC error. */
    uint64_t ibadlen;  /**< Total of RX packets with bad length. */
}
```

Note: postfix comments should be aligned with spaces not tabs in accordance with the *DPDK Coding Style*.

- If a single comment type can't be used, due to line length limitations then prefix comments should be preferred. For example this section of the code contains prefix comments, postfix comments on the same line and postfix comments on a separate line:

```
/** Number of elements in the elt_pa array. */
uint32_t    pg_num __rte_cache_aligned;
uint32_t    pg_shift;     /**< LOG2 of the physical pages. */
uintptr_t   pg_mask;      /**< Physical page mask value. */
uintptr_t   elt_va_start;
/**< Virtual address of the first mempool object. */
uintptr_t   elt_va_end;
/**< Virtual address of the <size + 1> mempool object. */
phys_addr_t elt_pa[MEMPOOL_PG_NUM_DEFAULT];
/**< Array of physical page addresses for the mempool buffer. */
```

This doesn't have an effect on the rendered documentation but it is confusing for the developer reading the code. It this case it would be clearer to use prefix comments throughout:

```
/** Number of elements in the elt_pa array. */
uint32_t    pg_num __rte_cache_aligned;
/** LOG2 of the physical pages. */
uint32_t    pg_shift;
/** Physical page mask value. */
uintptr_t   pg_mask;
/** Virtual address of the first mempool object. */
uintptr_t   elt_va_start;
/** Virtual address of the <size + 1> mempool object. */
uintptr_t   elt_va_end;
/** Array of physical page addresses for the mempool buffer. */
phys_addr_t elt_pa[MEMPOOL_PG_NUM_DEFAULT];
```

- Check for Doxygen warnings in new code by checking the API documentation build:

```
make doc-api-html >/dev/null
```

- Read the rendered section of the documentation that you have added for correctness, clarity and consistency with the surrounding text.

12.5 Contributing Code to DPDK

This document outlines the guidelines for submitting code to DPDK.

The DPDK development process is modelled (loosely) on the Linux Kernel development model so it is worth reading the Linux kernel guide on submitting patches: How to Get Your Change Into the Linux Kernel. The rationale for many

of the DPDK guidelines is explained in greater detail in the kernel guidelines.

12.5.1 The DPDK Development Process

The DPDK development process has the following features:

- The code is hosted in a public git repository.
- There is a mailing list where developers submit patches.
- There are maintainers for hierarchical components.
- Patches are reviewed publicly on the mailing list.
- Successfully reviewed patches are merged to the repository.
- Patches should be sent to the target repository or sub-tree, see below.
- All sub-repositories are merged into main repository for `-rc1` and `-rc2` versions of the release.
- After the `-rc2` release all patches should target the main repository.

The mailing list for DPDK development is dev@dpdk.org. Contributors will need to register for the mailing list in order to submit patches. It is also worth registering for the DPDK Patchwork

The development process requires some familiarity with the `git` version control system. Refer to the Pro Git Book for further information.

12.5.2 Maintainers and Sub-trees

The DPDK maintenance hierarchy is divided into a main repository `dpdk` and sub-repositories `dpdk-next-*`.

There are maintainers for the trees and for components within the tree.

Trees and maintainers are listed in the `MAINTAINERS` file. For example:

```
Crypto Drivers
--------------
M: Some Name <some.name@email.com>
B: Another Name <another.name@email.com>
T: git://dpdk.org/next/dpdk-next-crypto

Intel AES-NI GCM PMD
M: Some One <some.one@email.com>
F: drivers/crypto/aesni_gcm/
F: doc/guides/cryptodevs/aesni_gcm.rst
```

Where:

- `M` is a tree or component maintainer.
- `B` is a tree backup maintainer.
- `T` is a repository tree.
- `F` is a maintained file or directory.

Additional details are given in the `MAINTAINERS` file.

The role of the component maintainers is to:

- Review patches for the component or delegate the review. The review should be done, ideally, within 1 week of submission to the mailing list.

- Add an `acked-by` to patches, or patchsets, that are ready for committing to a tree.
- Reply to questions asked about the component.

Component maintainers can be added or removed by submitting a patch to the `MAINTAINERS` file. Maintainers should have demonstrated a reasonable level of contributions or reviews to the component area. The maintainer should be confirmed by an `ack` from an established contributor. There can be more than one component maintainer if desired.

The role of the tree maintainers is to:

- Maintain the overall quality of their tree. This can entail additional review, compilation checks or other tests deemed necessary by the maintainer.
- Commit patches that have been reviewed by component maintainers and/or other contributors. The tree maintainer should determine if patches have been reviewed sufficiently.
- Ensure that patches are reviewed in a timely manner.
- Prepare the tree for integration.
- Ensure that there is a designated back-up maintainer and coordinate a handover for periods where the tree maintainer can't perform their role.

Tree maintainers can be added or removed by submitting a patch to the `MAINTAINERS` file. The proposer should justify the need for a new sub-tree and should have demonstrated a sufficient level of contributions in the area or to a similar area. The maintainer should be confirmed by an `ack` from an existing tree maintainer. Disagreements on trees or maintainers can be brought to the Technical Board.

The backup maintainer for the master tree should be selected from the existing sub-tree maintainers from the project. The backup maintainer for a sub-tree should be selected from among the component maintainers within that sub-tree.

12.5.3 Getting the Source Code

The source code can be cloned using either of the following:

main repository:

```
git clone git://dpdk.org/dpdk
git clone http://dpdk.org/git/dpdk
```

sub-repositories (list):

```
git clone git://dpdk.org/next/dpdk-next-*
git clone http://dpdk.org/git/next/dpdk-next-*
```

12.5.4 Make your Changes

Make your planned changes in the cloned `dpdk` repo. Here are some guidelines and requirements:

- Follow the *DPDK Coding Style* guidelines.
- If you add new files or directories you should add your name to the `MAINTAINERS` file.
- New external functions should be added to the local `version.map` file. See the *Guidelines for ABI policy and versioning*. New external functions should also be added in alphabetical order.
- Important changes will require an addition to the release notes in `doc/guides/rel_notes/`. See the *Release Notes section of the Documentation Guidelines* for details.
- Test the compilation works with different targets, compilers and options, see *Checking Compilation*.

- Don't break compilation between commits with forward dependencies in a patchset. Each commit should compile on its own to allow for `git bisect` and continuous integration testing.
- Add tests to the the `app/test` unit test framework where possible.
- Add documentation, if relevant, in the form of Doxygen comments or a User Guide in RST format. See the *Documentation Guidelines*.

Once the changes have been made you should commit them to your local repo.

For small changes, that do not require specific explanations, it is better to keep things together in the same patch. Larger changes that require different explanations should be separated into logical patches in a patchset. A good way of thinking about whether a patch should be split is to consider whether the change could be applied without dependencies as a backport.

As a guide to how patches should be structured run `git log` on similar files.

12.5.5 Commit Messages: Subject Line

The first, summary, line of the git commit message becomes the subject line of the patch email. Here are some guidelines for the summary line:

- The summary line must capture the area and the impact of the change.
- The summary line should be around 50 characters.
- The summary line should be lowercase apart from acronyms.
- It should be prefixed with the component name (use git log to check existing components). For example:

```
ixgbe: fix offload config option name

config: increase max queues per port
```

- Use the imperative of the verb (like instructions to the code base).
- Don't add a period/full stop to the subject line or you will end up two in the patch name: `dpdk_description..patch`.

The actual email subject line should be prefixed by `[PATCH]` and the version, if greater than v1, for example: `PATCH v2`. The is generally added by `git send-email` or `git format-patch`, see below.

If you are submitting an RFC draft of a feature you can use `[RFC]` instead of `[PATCH]`. An RFC patch doesn't have to be complete. It is intended as a way of getting early feedback.

12.5.6 Commit Messages: Body

Here are some guidelines for the body of a commit message:

- The body of the message should describe the issue being fixed or the feature being added. It is important to provide enough information to allow a reviewer to understand the purpose of the patch.
- When the change is obvious the body can be blank, apart from the signoff.
- The commit message must end with a `Signed-off-by:` line which is added using:

```
git commit --signoff # or -s
```

The purpose of the signoff is explained in the Developer's Certificate of Origin section of the Linux kernel guidelines.

Note: All developers must ensure that they have read and understood the Developer's Certificate of Origin section of the documentation prior to applying the signoff and submitting a patch.

- The signoff must be a real name and not an alias or nickname. More than one signoff is allowed.
- The text of the commit message should be wrapped at 72 characters.
- When fixing a regression, it is a good idea to reference the id of the commit which introduced the bug. You can generate the required text using the following git alias:

```
git config alias.fixline "log -1 --abbrev=12 --format='Fixes: %h (\"%s\")'"
```

The `Fixes:` line can then be added to the commit message:

```
doc: fix vhost sample parameter

Update the docs to reflect removed dev-index.

Fixes: 17b8320a3e11 ("vhost: remove index parameter")

Signed-off-by: Alex Smith <alex.smith@example.com>
```

- When fixing an error or warning it is useful to add the error message and instructions on how to reproduce it.
- Use correct capitalization, punctuation and spelling.

In addition to the `Signed-off-by:` name the commit messages can also have tags for who reported, suggested, tested and reviewed the patch being posted. Please refer to the *Tested, Acked and Reviewed by* section.

12.5.7 Creating Patches

It is possible to send patches directly from git but for new contributors it is recommended to generate the patches with `git format-patch` and then when everything looks okay, and the patches have been checked, to send them with `git send-email`.

Here are some examples of using `git format-patch` to generate patches:

```
# Generate a patch from the last commit.
git format-patch -1

# Generate a patch from the last 3 commits.
git format-patch -3

# Generate the patches in a directory.
git format-patch -3 -o ~/patch/

# Add a cover letter to explain a patchset.
git format-patch -3 -o ~/patch/ --cover-letter

# Add a prefix with a version number.
git format-patch -3 -o ~/patch/ -v 2
```

Cover letters are useful for explaining a patchset and help to generate a logical threading to the patches. Smaller notes can be put inline in the patch after the `---` separator, for example:

```
Subject: [PATCH] fm10k/base: add FM10420 device ids

Add the device ID for Boulder Rapids and Atwood Channel to enable
drivers to support those devices.

Signed-off-by: Alex Smith <alex.smith@example.com>
---

ADD NOTES HERE.

 drivers/net/fm10k/base/fm10k_api.c  | 6 ++++++
 drivers/net/fm10k/base/fm10k_type.h | 6 ++++++
 2 files changed, 12 insertions(+)
...
```

Version 2 and later of a patchset should also include a short log of the changes so the reviewer knows what has changed. This can be added to the cover letter or the annotations. For example:

```
---
v3:
* Fixed issued with version.map.

v2:
* Added i40e support.
* Renamed ethdev functions from rte_eth_ieee15888_*() to rte_eth_timesync_*()
  since 802.1AS can be supported through the same interfaces.
```

12.5.8 Checking the Patches

Patches should be checked for formatting and syntax issues using the `checkpatches.sh` script in the `devtools` directory of the DPDK repo. This uses the Linux kernel development tool `checkpatch.pl` which can be obtained by cloning, and periodically, updating the Linux kernel sources.

The path to the original Linux script must be set in the environment variable `DPDK_CHECKPATCH_PATH`. This, and any other configuration variables required by the development tools, are loaded from the following files, in order of preference:

```
.develconfig
~/.config/dpdk/devel.config
/etc/dpdk/devel.config.
```

Once the environment variable the script can be run as follows:

```
devtools/checkpatches.sh ~/patch/
```

The script usage is:

```
checkpatches.sh [-h] [-q] [-v] [patch1 [patch2] ...]]"
```

Where:

- `-h`: help, usage.
- `-q`: quiet. Don't output anything for files without issues.
- `-v`: verbose.
- `patchX`: path to one or more patches.

Then the git logs should be checked using the `check-git-log.sh` script.

The script usage is:

```
check-git-log.sh [range]
```

Where the range is a `git log` option.

12.5.9 Checking Compilation

Compilation of patches and changes should be tested using the the `test-build.sh` script in the `devtools` directory of the DPDK repo:

```
devtools/test-build.sh x86_64-native-linuxapp-gcc+next+shared
```

The script usage is:

```
test-build.sh [-h] [-jX] [-s] [config1 [config2] ...]]
```

Where:

- `-h`: help, usage.
- `-jX`: use X parallel jobs in "make".
- `-s`: short test with only first config and without examples/doc.
- `config`: default config name plus config switches delimited with a + sign.

Examples of configs are:

```
x86_64-native-linuxapp-gcc
x86_64-native-linuxapp-gcc+next+shared
x86_64-native-linuxapp-clang+shared
```

The builds can be modifies via the following environmental variables:

- `DPDK_BUILD_TEST_CONFIGS` (target1+option1+option2 target2)
- `DPDK_DEP_CFLAGS`
- `DPDK_DEP_LDFLAGS`
- `DPDK_DEP_MOFED` (y/[n])
- `DPDK_DEP_PCAP` (y/[n])
- `DPDK_NOTIFY` (notify-send)

These can be set from the command line or in the config files shown above in the *Checking the Patches*.

The recommended configurations and options to test compilation prior to submitting patches are:

```
x86_64-native-linuxapp-gcc+shared+next
x86_64-native-linuxapp-clang+shared
i686-native-linuxapp-gcc

export DPDK_DEP_ZLIB=y
export DPDK_DEP_PCAP=y
export DPDK_DEP_SSL=y
```

12.5.10 Sending Patches

Patches should be sent to the mailing list using `git send-email`. You can configure an external SMTP with something like the following:

```
[sendemail]
    smtpuser = name@domain.com
    smtpserver = smtp.domain.com
    smtpserverport = 465
    smtpencryption = ssl
```

See the Git send-email documentation for more details.

The patches should be sent to `dev@dpdk.org`. If the patches are a change to existing files then you should send them TO the maintainer(s) and CC `dev@dpdk.org`. The appropriate maintainer can be found in the `MAINTAINERS` file:

```
git send-email --to maintainer@some.org --cc dev@dpdk.org 000*.patch
```

New additions can be sent without a maintainer:

```
git send-email --to dev@dpdk.org 000*.patch
```

You can test the emails by sending it to yourself or with the `--dry-run` option.

If the patch is in relation to a previous email thread you can add it to the same thread using the Message ID:

```
git send-email --to dev@dpdk.org --in-reply-to <1234-foo@bar.com> 000*.patch
```

The Message ID can be found in the raw text of emails or at the top of each Patchwork patch, for example. Shallow threading (`--thread --no-chain-reply-to`) is preferred for a patch series.

Once submitted your patches will appear on the mailing list and in Patchwork.

Experienced committers may send patches directly with `git send-email` without the `git format-patch` step. The options `--annotate` and `confirm = always` are recommended for checking patches before sending.

12.5.11 The Review Process

Patches are reviewed by the community, relying on the experience and collaboration of the members to double-check each other's work. There are a number of ways to indicate that you have checked a patch on the mailing list.

Tested, Acked and Reviewed by

To indicate that you have interacted with a patch on the mailing list you should respond to the patch in an email with one of the following tags:

- Reviewed-by:
- Acked-by:
- Tested-by:
- Reported-by:
- Suggested-by:

The tag should be on a separate line as follows:

```
tag-here: Name Surname <email@address.com>
```

Each of these tags has a specific meaning. In general, the DPDK community follows the kernel usage of the tags. A short summary of the meanings of each tag is given here for reference:

`Reviewed-by:` is a strong statement that the patch is an appropriate state for merging without any remaining serious technical issues. Reviews from community members who are known to understand the subject area and to perform thorough reviews will increase the likelihood of the patch getting merged.

`Acked-by:` is a record that the person named was not directly involved in the preparation of the patch but wishes to signify and record their acceptance and approval of it.

`Tested-by:` indicates that the patch has been successfully tested (in some environment) by the person named.

`Reported-by:` is used to acknowledge person who found or reported the bug.

`Suggested-by:` indicates that the patch idea was suggested by the named person.

Steps to getting your patch merged

The more work you put into the previous steps the easier it will be to get a patch accepted. The general cycle for patch review and acceptance is:

1. Submit the patch.
2. Check the automatic test reports in the coming hours.
3. Wait for review comments. While you are waiting review some other patches.
4. Fix the review comments and submit a `v n+1` patchset:

   ```
   git format-patch -3 -v 2
   ```

5. Update Patchwork to mark your previous patches as "Superseded".
6. If the patch is deemed suitable for merging by the relevant maintainer(s) or other developers they will `ack` the patch with an email that includes something like:

   ```
   Acked-by: Alex Smith <alex.smith@example.com>
   ```

   Note: When acking patches please remove as much of the text of the patch email as possible. It is generally best to delete everything after the `Signed-off-by:` line.
7. Having the patch `Reviewed-by:` and/or `Tested-by:` will also help the patch to be accepted.
8. If the patch isn't deemed suitable based on being out of scope or conflicting with existing functionality it may receive a `nack`. In this case you will need to make a more convincing technical argument in favor of your patches.
9. In addition a patch will not be accepted if it doesn't address comments from a previous version with fixes or valid arguments.
10. It is the responsibility of a maintainer to ensure that patches are reviewed and to provide an `ack` or `nack` of those patches as appropriate.
11. Once a patch has been acked by the relevant maintainer, reviewers may still comment on it for a further two weeks. After that time, the patch should be merged into the relevant git tree for the next release. Additional notes and restrictions:
    - Patches should be acked by a maintainer at least two days before the release merge deadline, in order to make that release.

- For patches acked with less than two weeks to go to the merge deadline, all additional comments should be made no later than two days before the merge deadline.
- After the appropriate time for additional feedback has passed, if the patch has not yet been merged to the relevant tree by the committer, it should be treated as though it had, in that any additional changes needed to it must be addressed by a follow-on patch, rather than rework of the original.
- Trivial patches may be merged sooner than described above at the tree committer's discretion.

12.6 DPDK Stable Releases and Long Term Support

This section sets out the guidelines for the DPDK Stable Releases and the DPDK Long Term Support releases (LTS).

12.6.1 Introduction

The purpose of the DPDK Stable Releases is to maintain releases of DPDK with backported fixes over an extended period of time. This provides downstream consumers of DPDK with a stable target on which to base applications or packages.

The Long Term Support release (LTS) is a designation applied to a Stable Release to indicate longer term support.

12.6.2 Stable Releases

Any major release of DPDK can be designated as a Stable Release if a maintainer volunteers to maintain it.

A Stable Release is used to backport fixes from an `N` release back to an `N-1` release, for example, from 16.11 to 16.07.

The duration of a stable is one complete release cycle (3 months). It can be longer, up to 1 year, if a maintainer continues to support the stable branch, or if users supply backported fixes, however the explicit commitment should be for one release cycle.

The release cadence is determined by the maintainer based on the number of bugfixes and the criticality of the bugs. Releases should be coordinated with the validation engineers to ensure that a tagged release has been tested.

12.6.3 LTS Release

A stable release can be designated as an LTS release based on community agreement and a commitment from a maintainer. An LTS release will have a maintenance duration of 2 years.

The current DPDK LTS release is 16.11.

It is anticipated that there will be at least 4 releases per year of the LTS or approximately 1 every 3 months. However, the cadence can be shorter or longer depending on the number and criticality of the backported fixes. Releases should be coordinated with the validation engineers to ensure that a tagged release has been tested.

12.6.4 What changes should be backported

Backporting should be limited to bug fixes.

Features should not be backported to stable releases. It may be acceptable, in limited cases, to back port features for the LTS release where:

- There is a justifiable use case (for example a new PMD).

- The change is non-invasive.
- The work of preparing the backport is done by the proposer.
- There is support within the community.

12.6.5 The Stable Mailing List

The Stable and LTS release are coordinated on the stable@dpdk.org mailing list.

All fix patches to the master branch that are candidates for backporting should also be CCed to the stable@dpdk.org mailing list.

12.6.6 Releasing

A Stable Release will be released by:

- Tagging the release with YY.MM.n (year, month, number).
- Uploading a tarball of the release to dpdk.org.
- Sending an announcement to the announce@dpdk.org list.

Stable releases are available on the dpdk.org download page.

12.6.7 ABI

The Stable Release should not be seen as a way of breaking or circumventing the DPDK ABI policy.

12.7 Patch Cheatsheet

Fig. 12.1: Cheat sheet for submitting patches to dev@dpdk.org

CHAPTER 13

Release Notes

13.1 Description of Release

This document contains the release notes for Data Plane Development Kit (DPDK) release version 0.11 and previous releases.

It lists new features, fixed bugs, API and ABI changes and known issues.

For instructions on compiling and running the release, see the *DPDK Getting Started Guide*.

13.2 DPDK Release 17.05

13.2.1 New Features

- **Reorganized the mbuf structure.**
  - Align fields to facilitate the writing of `data_off`, `refcnt`, and `nb_segs` in one operation.
  - Use 2 bytes for port and number of segments.
  - Move the sequence number in the second cache line.
  - Add a timestamp field.
  - Set default value for `refcnt`, `next` and `nb_segs` at mbuf free.
- **Added mbuf raw free API**

  Moved `rte_mbuf_raw_free()` and `rte_pktmbuf_prefree_seg()` functions to the public API.
- **Added free Tx mbuf on demand API.**

  Added a new function `rte_eth_tx_done_cleanup()` which allows an application to request the driver to release mbufs from their Tx ring that are no longer in use, independent of whether or not the `tx_rs_thresh` has been crossed.

- **Added EAL dynamic log framework.**

  Added new APIs to dynamically register named log types, and control the level of each type independently.

- **Added descriptor status ethdev API.**

  Added a new API to get the status of a descriptor.

  For Rx, it is almost similar to the `rx_descriptor_done` API, except it differentiates descriptors which are hold by the driver and not returned to the hardware. For Tx, it is a new API.

- **Increased number of next hops for LPM IPv6 to 2^21.**

  The next_hop field is extended from 8 bits to 21 bits for IPv6.

- **Added VFIO hotplug support.**

  How hotplug supported with UIO and VFIO drivers.

- **Added powerpc support in pci probing for vfio-pci devices.**

  sPAPR IOMMU based pci probing enabled for vfio-pci devices.

- **Kept consistent PMD batching behaviour.**

  Removed the limit of fm10k/i40e/ixgbe TX burst size and vhost RX/TX burst size in order to support the same policy of "make an best effort to RX/TX pkts" for PMDs.

- **Updated the ixgbe base driver.**

  Updated the ixgbe base driver, including the following changes:

  - Add link block check for KR.
  - Complete HW initialization even if SFP is not present.
  - Add VF xcast promiscuous mode.

- **Added powerpc support for i40e and its vector PMD .**

  i40e PMD and its vector PMD enabled by default in powerpc.

- **Added VF max bandwidth setting on i40e.**

  i40e HW supports to set the max bandwidth for a VF. Enable this capability.

- **Added VF TC min bandwidth setting on i40e.**

  i40e HW supports to set the allocated bandwidth for a TC on a VF. Enable this capability.

- **Added VF TC max bandwidth setting on i40e.**

  i40e HW supports to set the max bandwidth for a TC on a VF. Enable this capability.

- **Added TC strict priority mode setting on i40e.**

  There're 2 TX scheduling modes supported for TCs by i40e HW, round ribon mode and strict priority mode. By default it's round robin mode. Enable the capability to change the TX scheduling mode for a TC. It's a global setting on a physical port.

- **Added i40e dynamic device personalization support.**

  - Added dynamic device personalization processing to i40e FW.

- **Added TSO support for tunneled and non-tunneled packets on mlx5 driver.**

  Added support for Hardware TSO for tunneled and non-tunneled packets. Tunneling protocols supported are GRE and VXLAN.

- **Added support for Rx interrupts on mlx5 driver.**

  Rx queues can be armed with an interrupt which will trigger on the next packet arrival.

- **Updated the sfc_efx driver.**
  - Generic flow API support for Ethernet, VLAN, IPv4, IPv6, UDP and TCP pattern items with QUEUE action for ingress traffic.
  - Support virtual functions (VFs)

- **Added LiquidIO network PMD.**

  Added poll mode driver support for Cavium LiquidIO II server adapter VFs.

- **Added support for the Wind River Systems AVP PMD.**

  Added a new networking driver for the AVP device type. Theses devices are specific to the Wind River Systems virtualization platforms.

- **Added vmxnet3 version 3 support.**

  Added support for vmxnet3 version 3 which includes several performance enhancements viz. configurable TX data ring, Receive Data Ring, ability to register memory regions.

- **Updated the tap driver.**
  - Support MTU modification.
  - Support packet type for Rx.
  - Support segmented packets on Rx and Tx.
  - Speed up Rx on tap when no packets are available.
  - Support capturing traffic from another netdevice.
  - Dynamically change link status when the underlying interface state changes.
  - Generic flow API support for Ethernet, VLAN, IPv4, IPv6, UDP and TCP pattern items with DROP, QUEUE and PASSTHRU actions for ingress traffic.

- **Added MTU feature support to Virtio and Vhost.**

  Implemented new Virtio MTU feature into Vhost and Virtio:
  - Add `rte_vhost_mtu_get()` API to Vhost library.
  - Enable Vhost PMD's MTU get feature.
  - Get max MTU value from host in Virtio PMD

- **Added interrupt mode support for virtio-user.**

  Implemented Rxq interrupt mode and LSC support for virtio-user as a virtual device. Supported cases:
  - Rxq interrupt for virtio-user + vhost-user as the backend.
  - Rxq interrupt for virtio-user + vhost-kernel as the backend.
  - LSC interrupt for virtio-user + vhost-user as the backend.

- **Added event driven programming model library (rte_eventdev).**

  This API introduces event driven programming model.

  In a polling model, lcores poll ethdev ports and associated rx queues directly to look for packet. In an event driven model, by contrast, lcores call the scheduler that selects packets for them based on programmer-specified

criteria. Eventdev library added support for event driven programming model, which offer applications automatic multicore scaling, dynamic load balancing, pipelining, packet ingress order maintenance and synchronization services to simplify application packet processing.

By introducing event driven programming model, DPDK can support both polling and event driven programming models for packet processing, and applications are free to choose whatever model (or combination of the two) that best suits their needs.

- **Added Software Eventdev PMD.**

  Added support for the software eventdev PMD. The software eventdev is a software based scheduler device that implements the eventdev API. This PMD allows an application to configure a pipeline using the eventdev library, and run the scheduling workload on a CPU core.

- **Added Cavium OCTEONTX Eventdev PMD.**

  Added the new octeontx ssovf eventdev driver for OCTEONTX devices. See the "Event Device Drivers" document for more details on this new driver.

- **Added information metric library.**

  A library that allows information metrics to be added and updated by producers, typically other libraries, for later retrieval by consumers such as applications. It is intended to provide a reporting mechanism that is independent of other libraries such as ethdev.

- **Added bit-rate calculation library.**

  A library that can be used to calculate device bit-rates. Calculated bitrates are reported using the metrics library.

- **Added latency stats library.**

  A library that measures packet latency. The collected statistics are jitter and latency. For latency the minimum, average, and maximum is measured.

- **Updated the Cryptodev Scheduler PMD.**

  - Added packet-size based distribution mode, which distributes the enqueued crypto operations among two slaves, based on their data lengths.
  - Added fail-over scheduling mode, which enqueues crypto operations to a primary slave first. Then, any operation that cannot be enqueued is enqueued to a secondary slave.

- **Updated the QAT PMD.**

  The QAT PMD has been updated with additional support for:

  - AES DOCSIS BPI algorithm.
  - DES DOCSIS BPI algorithm.
  - ZUC EEA3/EIA3 algorithms.

- **Updated the AESNI MB PMD.**

  The AESNI MB PMD has been updated with additional support for:

  - AES DOCSIS BPI algorithm.

- **Updated the OpenSSL PMD.**

  The OpenSSL PMD has been updated with additional support for:

  - DES DOCSIS BPI algorithm.

13.2.2 Resolved Issues

EAL

Drivers

Libraries

Examples

Other

13.2.3 Known Issues

- **LSC interrupt cannot work for virtio-user + vhost-kernel.**

  LSC interrupt cannot be detected when setting the backend, tap device, up/down as we fail to find a way to monitor such event.

13.2.4 API Changes

- The LPM `next_hop` field is extended from 8 bits to 21 bits for IPv6 while keeping ABI compatibility.
- **Reworked rte_ring library**

  The rte_ring library has been reworked and updated. The following changes have been made to it:

  - removed the build-time setting `CONFIG_RTE_RING_SPLIT_PROD_CONS`
  - removed the build-time setting `CONFIG_RTE_LIBRTE_RING_DEBUG`
  - removed the build-time setting `CONFIG_RTE_RING_PAUSE_REP_COUNT`
  - removed the function `rte_ring_set_water_mark` as part of a general removal of watermarks support in the library.
  - added an extra parameter to the burst/bulk enqueue functions to return the number of free spaces in the ring after enqueue. This can be used by an application to implement its own watermark functionality.
  - added an extra parameter to the burst/bulk dequeue functions to return the number elements remaining in the ring after dequeue.
  - changed the return value of the enqueue and dequeue bulk functions to match that of the burst equivalents. In all cases, ring functions which operate on multiple packets now return the number of elements enqueued or dequeued, as appropriate. The updated functions are:

    * `rte_ring_mp_enqueue_bulk`
    * `rte_ring_sp_enqueue_bulk`
    * `rte_ring_enqueue_bulk`
    * `rte_ring_mc_dequeue_bulk`
    * `rte_ring_sc_dequeue_bulk`
    * `rte_ring_dequeue_bulk`

NOTE: the above functions all have different parameters as well as different return values, due to the other listed changes above. This means that all instances of the functions in existing code will be flagged by the compiler. The return value usage should be checked while fixing the compiler error due to the extra parameter.

- **Reworked rte_vhost library**

  The rte_vhost library has been reworked to make it generic enough so that user could build other vhost-user drivers on top of it. To achieve that, following changes have been made:

  - The following vhost-pmd APIs are removed:

    * `rte_eth_vhost_feature_disable`
    * `rte_eth_vhost_feature_enable`
    * `rte_eth_vhost_feature_get`

  - The vhost API `rte_vhost_driver_callback_register(ops)` is reworked to be per vhost-user socket file. Thus, it takes one more argument: `rte_vhost_driver_callback_register(path, ops)`.
  - The vhost API `rte_vhost_get_queue_num` is deprecated, instead, `rte_vhost_get_vring_num` should be used.
  - Following macros are removed in `rte_virtio_net.h`

    * `VIRTIO_RXQ`
    * `VIRTIO_TXQ`
    * `VIRTIO_QNUM`

  - Following net specific header files are removed in `rte_virtio_net.h`

    * `linux/virtio_net.h`
    * `sys/socket.h`
    * `linux/if.h`
    * `rte_ether.h`

  - The vhost struct `virtio_net_device_ops` is renamed to `vhost_device_ops`
  - The vhost API `rte_vhost_driver_session_start` is removed. Instead, `rte_vhost_driver_start` should be used, and no need to create a thread to call it.
  - The vhost public header file `rte_virtio_net.h` is renamed to `rte_vhost.h`

13.2.5 ABI Changes

- **Reorganized the mbuf structure.**

  The order and size of the fields in the `mbuf` structure changed, as described in the *New Features* section.

- The `rte_cryptodev_info.sym` structure has new field `max_nb_sessions_per_qp` to support drivers which may support limited number of sessions per queue_pair.

13.2.6 Removed Items

- KNI vhost support removed.
- dpdk_qat sample application removed.

13.2.7 Shared Library Versions

The libraries prepended with a plus sign were incremented in this version.

```
  librte_acl.so.2
+ librte_bitratestats.so.1
  librte_cfgfile.so.2
  librte_cmdline.so.2
  librte_cryptodev.so.2
  librte_distributor.so.1
+ librte_eal.so.4
  librte_ethdev.so.6
  librte_hash.so.2
  librte_ip_frag.so.1
  librte_jobstats.so.1
  librte_kni.so.2
  librte_kvargs.so.1
+ librte_latencystats.so.1
  librte_lpm.so.2
+ librte_mbuf.so.3
  librte_mempool.so.2
  librte_meter.so.1
+ librte_metrics.so.1
  librte_net.so.1
  librte_pdump.so.1
  librte_pipeline.so.3
  librte_pmd_bond.so.1
  librte_pmd_ring.so.2
  librte_port.so.3
  librte_power.so.1
  librte_reorder.so.1
  librte_ring.so.1
  librte_sched.so.1
  librte_table.so.2
  librte_timer.so.1
  librte_vhost.so.3
```

13.2.8 Tested Platforms

13.3 DPDK Release 17.02

13.3.1 New Features

- **Added support for representing buses in EAL**

  The `rte_bus` structure was introduced into the EAL. This allows for devices to be represented by buses they are connected to. A new bus can be added to DPDK by extending the `rte_bus` structure and implementing the scan and probe functions. Once a new bus is registered using the provided APIs, new devices can be detected and initialized using bus scan and probe callbacks.

  With this change, devices other than PCI or VDEV type can be represented in the DPDK framework.

- **Added generic EAL API for I/O device memory read/write operations.**

  This API introduces 8 bit, 16 bit, 32 bit and 64 bit I/O device memory read/write operations along with "relaxed" versions.

Weakly-ordered architectures like ARM need an additional I/O barrier for device memory read/write access over PCI bus. By introducing the EAL abstraction for I/O device memory read/write access, the drivers can access I/O device memory in an architecture-agnostic manner. The relaxed version does not have an additional I/O memory barrier, which is useful in accessing the device registers of integrated controllers which is implicitly strongly ordered with respect to memory access.

- **Added generic flow API (rte_flow).**

  This API provides a generic means to configure hardware to match specific ingress or egress traffic, alter its behavior and query related counters according to any number of user-defined rules.

  In order to expose a single interface with an unambiguous behavior that is common to all poll-mode drivers (PMDs) the `rte_flow` API is slightly higher-level than the legacy filtering framework, which it encompasses and supersedes (including all functions and filter types) .

  See the *Generic flow API* documentation for more information.

- **Added firmware version get API.**

  Added a new function `rte_eth_dev_fw_version_get()` to fetch the firmware version for a given device.

- **Added APIs for MACsec offload support to the ixgbe PMD.**

  Six new APIs have been added to the ixgbe PMD for MACsec offload support. The declarations for the APIs can be found in `rte_pmd_ixgbe.h`.

- **Added I219 NICs support.**

  Added support for I219 Intel 1GbE NICs.

- **Added VF Daemon (VFD) for i40e. - EXPERIMENTAL**

  This is an EXPERIMENTAL feature to enhance the capability of the DPDK PF as many VF management features are not currently supported by the kernel PF driver. Some new private APIs are implemented directly in the PMD without an abstraction layer. They can be used directly by some users who have the need.

  The new APIs to control VFs directly from PF include:

  - Set VF MAC anti-spoofing.
  - Set VF VLAN anti-spoofing.
  - Set TX loopback.
  - Set VF unicast promiscuous mode.
  - Set VF multicast promiscuous mode.
  - Set VF MTU.
  - Get/reset VF stats.
  - Set VF MAC address.
  - Set VF VLAN stripping.
  - Vf VLAN insertion.
  - Set VF broadcast mode.
  - Set VF VLAN tag.
  - Set VF VLAN filter.

VFD also includes VF to PF mailbox message management from an application. When the PF receives mailbox messages from the VF the PF should call the callback provided by the application to know if they're permitted to be processed.

As an EXPERIMENTAL feature, please be aware it can be changed or even removed without prior notice.

- **Updated the i40e base driver.**

  Updated the i40e base driver, including the following changes:

  - Replace existing legacy `memcpy()` calls with `i40e_memcpy()` calls.
  - Use `BIT()` macro instead of bit fields.
  - Add clear all WoL filters implementation.
  - Add broadcast promiscuous control per VLAN.
  - Remove unused `X722_SUPPORT` and `I40E_NDIS_SUPPORT` macros.

- **Updated the enic driver.**
  - Set new Rx checksum flags in mbufs to indicate unknown, good or bad checksums.
  - Fix set/remove of MAC addresses. Allow up to 64 addresses per device.
  - Enable TSO on outer headers.

- **Added Solarflare libefx-based network PMD.**

  Added a new network PMD which supports Solarflare SFN7xxx and SFN8xxx family of 10/40 Gbps adapters.

- **Updated the mlx4 driver.**
  - Addressed a few bugs.

- **Added support for Mellanox ConnectX-5 adapters (mlx5).**

  Added support for Mellanox ConnectX-5 family of 10/25/40/50/100 Gbps adapters to the existing mlx5 PMD.

- **Updated the mlx5 driver.**
  - Improve Tx performance by using vector logic.
  - Improve RSS balancing when number of queues is not a power of two.
  - Generic flow API support for Ethernet, IPv4, IPv4, UDP, TCP, VLAN and VXLAN pattern items with DROP and QUEUE actions.
  - Support for extended statistics.
  - Addressed several data path bugs.
  - As of MLNX_OFED 4.0-1.0.1.0, the Toeplitz RSS hash function is not symmetric anymore for consistency with other PMDs.

- **virtio-user with vhost-kernel as another exceptional path.**

  Previously, we upstreamed a virtual device, virtio-user with vhost-user as the backend as a way of enabling IPC (Inter-Process Communication) and user space container networking.

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

  - Maintenance: vhost and vhost-net (kernel) is an upstreamed and extensively used kernel module.
  - Features: vhost-net is designed to be a networking solution, which has lots of networking related features, like multi-queue, TSO, multi-seg mbuf, etc.

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

- **Added virtio Rx interrupt support.**

  Added a feature to enable Rx interrupt mode for virtio pci net devices as bound to VFIO (noiommu mode) and driven by virtio PMD.

  With this feature, the virtio PMD can switch between polling mode and interrupt mode, to achieve best performance, and at the same time save power. It can work on both legacy and modern virtio devices. In this mode, each `rxq` is mapped with an excluded MSIx interrupt.

  See the *Virtio Interrupt Mode* documentation for more information.

- **Added ARMv8 crypto PMD.**

  A new crypto PMD has been added, which provides combined mode cryptographic operations optimized for ARMv8 processors. The driver can be used to enhance performance in processing chained operations such as cipher + HMAC.

- **Updated the QAT PMD.**

  The QAT PMD has been updated with additional support for:

  - DES algorithm.
  - Scatter-gather list (SGL) support.

- **Updated the AESNI MB PMD.**

  - The Intel(R) Multi Buffer Crypto for IPsec library used in AESNI MB PMD has been moved to a new repository, in GitHub.
  - Support has been added for single operations (cipher only and authentication only).

- **Updated the AES-NI GCM PMD.**

  The AES-NI GCM PMD was migrated from the Multi Buffer library to the ISA-L library. The migration entailed adding additional support for:

  - GMAC algorithm.
  - 256-bit cipher key.
  - Session-less mode.
  - Out-of place processing
  - Scatter-gather support for chained mbufs (only out-of place and destination mbuf must be contiguous)

- **Added crypto performance test application.**

  Added a new performance test application for measuring performance parameters of PMDs available in the crypto tree.

- **Added Elastic Flow Distributor library (rte_efd).**

  Added a new library which uses perfect hashing to determine a target/value for a given incoming flow key.

  The library does not store the key itself for lookup operations, and therefore, lookup performance is not dependent on the key size. Also, the target/value can be any arbitrary value (8 bits by default). Finally, the storage requirement is much smaller than a hash-based flow table and therefore, it can better fit in CPU cache and scale to millions of flow keys.

  See the *Elastic Flow Distributor Library* documentation in the Programmers Guide document, for more information.

13.3.2 Resolved Issues

Drivers

- **net/virtio: Fixed multiple process support.**

  Fixed a few regressions introduced in recent releases that break the virtio multiple process support.

Examples

- **examples/ethtool: Fixed crash with non-PCI devices.**

  Fixed issue where querying a non-PCI device was dereferencing non-existent PCI data resulting in a segmentation fault.

13.3.3 API Changes

- **Moved five APIs for VF management from the ethdev to the ixgbe PMD.**

  The following five APIs for VF management from the PF have been removed from the ethdev, renamed, and added to the ixgbe PMD:

```
rte_eth_dev_set_vf_rate_limit()
rte_eth_dev_set_vf_rx()
rte_eth_dev_set_vf_rxmode()
rte_eth_dev_set_vf_tx()
rte_eth_dev_set_vf_vlan_filter()
```

  The API's have been renamed to the following:

```
rte_pmd_ixgbe_set_vf_rate_limit()
rte_pmd_ixgbe_set_vf_rx()
rte_pmd_ixgbe_set_vf_rxmode()
rte_pmd_ixgbe_set_vf_tx()
rte_pmd_ixgbe_set_vf_vlan_filter()
```

  The declarations for the API's can be found in `rte_pmd_ixgbe.h`.

13.3.4 ABI Changes

13.3.5 Shared Library Versions

The libraries prepended with a plus sign were incremented in this version.

```
  librte_acl.so.2
  librte_cfgfile.so.2
  librte_cmdline.so.2
  librte_cryptodev.so.2
  librte_distributor.so.1
  librte_eal.so.3
+ librte_ethdev.so.6
  librte_hash.so.2
  librte_ip_frag.so.1
  librte_jobstats.so.1
  librte_kni.so.2
```

```
librte_kvargs.so.1
librte_lpm.so.2
librte_mbuf.so.2
librte_mempool.so.2
librte_meter.so.1
librte_net.so.1
librte_pdump.so.1
librte_pipeline.so.3
librte_pmd_bond.so.1
librte_pmd_ring.so.2
librte_port.so.3
librte_power.so.1
librte_reorder.so.1
librte_ring.so.1
librte_sched.so.1
librte_table.so.2
librte_timer.so.1
librte_vhost.so.3
```

13.3.6 Tested Platforms

This release has been tested with the below list of CPU/device/firmware/OS. Each section describes a different set of combinations.

- Intel(R) platforms with Mellanox(R) NICs combinations
  - Platform details
    - Intel(R) Xeon(R) CPU E5-2697 v2 @ 2.70GHz
    - Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
    - Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz
  - OS:
    - CentOS 7.0
    - Fedora 23
    - Fedora 24
    - FreeBSD 10.3
    - Red Hat Enterprise Linux 7.2
    - SUSE Enterprise Linux 12
    - Ubuntu 14.04 LTS
    - Ubuntu 15.10
    - Ubuntu 16.04 LTS
    - Wind River Linux 8
  - MLNX_OFED: 4.0-1.0.1.0
  - NICs:
    - Mellanox(R) ConnectX(R)-3 Pro 40G MCX354A-FCC_Ax (2x40G)
      - Host interface: PCI Express 3.0 x8

      - Device ID: 15b3:1007
      - Firmware version: 2.40.5030
  - Mellanox(R) ConnectX(R)-4 10G MCX4111A-XCAT (1x10G)
      - Host interface: PCI Express 3.0 x8
      - Device ID: 15b3:1013
      - Firmware version: 12.18.1000
  - Mellanox(R) ConnectX(R)-4 10G MCX4121A-XCAT (2x10G)
      - Host interface: PCI Express 3.0 x8
      - Device ID: 15b3:1013
      - Firmware version: 12.18.1000
  - Mellanox(R) ConnectX(R)-4 25G MCX4111A-ACAT (1x25G)
      - Host interface: PCI Express 3.0 x8
      - Device ID: 15b3:1013
      - Firmware version: 12.18.1000
  - Mellanox(R) ConnectX(R)-4 25G MCX4121A-ACAT (2x25G)
      - Host interface: PCI Express 3.0 x8
      - Device ID: 15b3:1013
      - Firmware version: 12.18.1000
  - Mellanox(R) ConnectX(R)-4 40G MCX4131A-BCAT/MCX413A-BCAT (1x40G)
      - Host interface: PCI Express 3.0 x8
      - Device ID: 15b3:1013
      - Firmware version: 12.18.1000
  - Mellanox(R) ConnectX(R)-4 40G MCX415A-BCAT (1x40G)
      - Host interface: PCI Express 3.0 x16
      - Device ID: 15b3:1013
      - Firmware version: 12.18.1000
  - Mellanox(R) ConnectX(R)-4 50G MCX4131A-GCAT/MCX413A-GCAT (1x50G)
      - Host interface: PCI Express 3.0 x8
      - Device ID: 15b3:1013
      - Firmware version: 12.18.1000
  - Mellanox(R) ConnectX(R)-4 50G MCX414A-BCAT (2x50G)
      - Host interface: PCI Express 3.0 x8
      - Device ID: 15b3:1013
      - Firmware version: 12.18.1000
  - Mellanox(R) ConnectX(R)-4 50G MCX415A-GCAT/MCX416A-BCAT/MCX416A-GCAT (2x50G)
      - Host interface: PCI Express 3.0 x16

            - Device ID: 15b3:1013
            - Firmware version: 12.18.1000
        * Mellanox(R) ConnectX(R)-4 50G MCX415A-CCAT (1x100G)
            - Host interface: PCI Express 3.0 x16
            - Device ID: 15b3:1013
            - Firmware version: 12.18.1000
        * Mellanox(R) ConnectX(R)-4 100G MCX416A-CCAT (2x100G)
            - Host interface: PCI Express 3.0 x16
            - Device ID: 15b3:1013
            - Firmware version: 12.18.1000
        * Mellanox(R) ConnectX(R)-4 Lx 10G MCX4121A-XCAT (2x10G)
            - Host interface: PCI Express 3.0 x8
            - Device ID: 15b3:1015
            - Firmware version: 14.18.1000
        * Mellanox(R) ConnectX(R)-4 Lx 25G MCX4121A-ACAT (2x25G)
            - Host interface: PCI Express 3.0 x8
            - Device ID: 15b3:1015
            - Firmware version: 14.18.1000
        * Mellanox(R) ConnectX(R)-5 100G MCX556A-ECAT (2x100G)
            - Host interface: PCI Express 3.0 x16
            - Device ID: 15b3:1017
            - Firmware version: 16.18.1000
        * Mellanox(R) ConnectX-5 Ex EN 100G MCX516A-CDAT (2x100G)
            - Host interface: PCI Express 4.0 x16
            - Device ID: 15b3:1019
            - Firmware version: 16.18.1000

- IBM(R) Power8(R) with Mellanox(R) NICs combinations
    - Machine:
        * Processor: POWER8E (raw), AltiVec supported
            - type-model: 8247-22L
            - Firmware FW810.21 (SV810_108)
    - OS: Ubuntu 16.04 LTS PPC le
    - MLNX_OFED: 4.0-1.0.1.0
    - NICs:
        * Mellanox(R) ConnectX(R)-4 10G MCX4111A-XCAT (1x10G)
            - Host interface: PCI Express 3.0 x8

    - Device ID: 15b3:1013
    - Firmware version: 12.18.1000
* Mellanox(R) ConnectX(R)-4 10G MCX4121A-XCAT (2x10G)
    - Host interface: PCI Express 3.0 x8
    - Device ID: 15b3:1013
    - Firmware version: 12.18.1000
* Mellanox(R) ConnectX(R)-4 25G MCX4111A-ACAT (1x25G)
    - Host interface: PCI Express 3.0 x8
    - Device ID: 15b3:1013
    - Firmware version: 12.18.1000
* Mellanox(R) ConnectX(R)-4 25G MCX4121A-ACAT (2x25G)
    - Host interface: PCI Express 3.0 x8
    - Device ID: 15b3:1013
    - Firmware version: 12.18.1000
* Mellanox(R) ConnectX(R)-4 40G MCX4131A-BCAT/MCX413A-BCAT (1x40G)
    - Host interface: PCI Express 3.0 x8
    - Device ID: 15b3:1013
    - Firmware version: 12.18.1000
* Mellanox(R) ConnectX(R)-4 40G MCX415A-BCAT (1x40G)
    - Host interface: PCI Express 3.0 x16
    - Device ID: 15b3:1013
    - Firmware version: 12.18.1000
* Mellanox(R) ConnectX(R)-4 50G MCX4131A-GCAT/MCX413A-GCAT (1x50G)
    - Host interface: PCI Express 3.0 x8
    - Device ID: 15b3:1013
    - Firmware version: 12.18.1000
* Mellanox(R) ConnectX(R)-4 50G MCX414A-BCAT (2x50G)
    - Host interface: PCI Express 3.0 x8
    - Device ID: 15b3:1013
    - Firmware version: 12.18.1000
* Mellanox(R) ConnectX(R)-4 50G MCX415A-GCAT/MCX416A-BCAT/MCX416A-GCAT (2x50G)
    - Host interface: PCI Express 3.0 x16
    - Device ID: 15b3:1013
    - Firmware version: 12.18.1000
* Mellanox(R) ConnectX(R)-4 50G MCX415A-CCAT (1x100G)
    - Host interface: PCI Express 3.0 x16

            - Device ID: 15b3:1013
            - Firmware version: 12.18.1000
        * Mellanox(R) ConnectX(R)-4 100G MCX416A-CCAT (2x100G)
            - Host interface: PCI Express 3.0 x16
            - Device ID: 15b3:1013
            - Firmware version: 12.18.1000
        * Mellanox(R) ConnectX(R)-4 Lx 10G MCX4121A-XCAT (2x10G)
            - Host interface: PCI Express 3.0 x8
            - Device ID: 15b3:1015
            - Firmware version: 14.18.1000
        * Mellanox(R) ConnectX(R)-4 Lx 25G MCX4121A-ACAT (2x25G)
            - Host interface: PCI Express 3.0 x8
            - Device ID: 15b3:1015
            - Firmware version: 14.18.1000
        * Mellanox(R) ConnectX(R)-5 100G MCX556A-ECAT (2x100G)
            - Host interface: PCI Express 3.0 x16
            - Device ID: 15b3:1017
            - Firmware version: 16.18.1000

- Intel(R) platforms with Intel(R) NICs combinations
    - Platform details
        * Intel(R) Atom(TM) CPU C2758 @ 2.40GHz
        * Intel(R) Xeon(R) CPU D-1540 @ 2.00GHz
        * Intel(R) Xeon(R) CPU E5-4667 v3 @ 2.00GHz
        * Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
        * Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz
        * Intel(R) Xeon(R) CPU E5-2695 v4 @ 2.10GHz
        * Intel(R) Xeon(R) CPU E5-2658 v2 @ 2.40GHz
    - OS:
        * CentOS 7.2
        * Fedora 25
        * FreeBSD 11
        * Red Hat Enterprise Linux Server release 7.3
        * SUSE Enterprise Linux 12
        * Wind River Linux 8
        * Ubuntu 16.04
        * Ubuntu 16.10

  - NICs:
    * Intel(R) 82599ES 10 Gigabit Ethernet Controller
      · Firmware version: 0x61bf0001
      · Device id (pf/vf): 8086:10fb / 8086:10ed
      · Driver version: 4.0.1-k (ixgbe)
    * Intel(R) Corporation Ethernet Connection X552/X557-AT 10GBASE-T
      · Firmware version: 0x800001cf
      · Device id (pf/vf): 8086:15ad / 8086:15a8
      · Driver version: 4.2.5 (ixgbe)
    * Intel(R) Ethernet Converged Network Adapter X710-DA4 (4x10G)
      · Firmware version: 5.05
      · Device id (pf/vf): 8086:1572 / 8086:154c
      · Driver version: 1.5.23 (i40e)
    * Intel(R) Ethernet Converged Network Adapter X710-DA2 (2x10G)
      · Firmware version: 5.05
      · Device id (pf/vf): 8086:1572 / 8086:154c
      · Driver version: 1.5.23 (i40e)
    * Intel(R) Ethernet Converged Network Adapter XL710-QDA1 (1x40G)
      · Firmware version: 5.05
      · Device id (pf/vf): 8086:1584 / 8086:154c
      · Driver version: 1.5.23 (i40e)
    * Intel(R) Ethernet Converged Network Adapter XL710-QDA2 (2X40G)
      · Firmware version: 5.05
      · Device id (pf/vf): 8086:1583 / 8086:154c
      · Driver version: 1.5.23 (i40e)
    * Intel(R) Corporation I350 Gigabit Network Connection
      · Firmware version: 1.48, 0x800006e7
      · Device id (pf/vf): 8086:1521 / 8086:1520
      · Driver version: 5.2.13-k (igb)

13.4 DPDK Release 16.11

13.4.1 New Features

- **Added software parser for packet type.**
  - Added a new function `rte_pktmbuf_read()` to read the packet data from an mbuf chain, linearizing if required.

  - Added a new function `rte_net_get_ptype()` to parse an Ethernet packet in an mbuf chain and retrieve its packet type from software.
  - Added new functions `rte_get_ptype_*()` to dump a packet type as a string.

- **Improved offloads support in mbuf.**
  - Added a new function `rte_raw_cksum_mbuf()` to process the checksum of data embedded in an mbuf chain.
  - Added new Rx checksum flags in mbufs to describe more states: unknown, good, bad, or not present (useful for virtual drivers). This modification was done for IP and L4.
  - Added a new Rx LRO mbuf flag, used when packets are coalesced. This flag indicates that the segment size of original packets is known.

- **Added vhost-user dequeue zero copy support.**

  The copy in the dequeue path is avoided in order to improve the performance. In the VM2VM case, the boost is quite impressive. The bigger the packet size, the bigger performance boost you may get. However, for the VM2NIC case, there are some limitations, so the boost is not as impressive as the VM2VM case. It may even drop quite a bit for small packets.

  For that reason, this feature is disabled by default. It can be enabled when the `RTE_VHOST_USER_DEQUEUE_ZERO_COPY` flag is set. Check the VHost section of the Programming Guide for more information.

- **Added vhost-user indirect descriptors support.**

  If the indirect descriptor feature is enabled, each packet sent by the guest will take exactly one slot in the enqueue virtqueue. Without this feature, as in the current version, even 64 bytes packets take two slots with Virtio PMD on guest side.

  The main impact is better performance for 0% packet loss use-cases, as it behaves as if the virtqueue size was enlarged, so more packets can be buffered in the case of system perturbations. On the downside, small performance degradations were measured when running micro-benchmarks.

- **Added vhost PMD xstats.**

  Added extended statistics to vhost PMD from a per port perspective.

- **Supported offloads with virtio.**

  Added support for the following offloads in virtio:

  - Rx/Tx checksums.
  - LRO.
  - TSO.

- **Added virtio NEON support for ARM.**

  Added NEON support for ARM based virtio.

- **Updated the ixgbe base driver.**

  Updated the ixgbe base driver, including the following changes:

  - Added X550em_a 10G PHY support.
  - Added support for flow control auto negotiation for X550em_a 1G PHY.
  - Added X550em_a FW ALEF support.
  - Increased mailbox version to `ixgbe_mbox_api_13`.

  - Added two MAC operations for Hyper-V support.

- **Added APIs for VF management to the ixgbe PMD.**

  Eight new APIs have been added to the ixgbe PMD for VF management from the PF. The declarations for the API's can be found in `rte_pmd_ixgbe.h`.

- **Updated the enic driver.**
  - Added update to use interrupt for link status checking instead of polling.
  - Added more flow director modes on UCS Blade with firmware version >= 2.0(13e).
  - Added full support for MTU update.
  - Added support for the `rte_eth_rx_queue_count` function.

- **Updated the mlx5 driver.**
  - Added support for RSS hash results.
  - Added several performance improvements.
  - Added several bug fixes.

- **Updated the QAT PMD.**

  The QAT PMD was updated with additional support for:
  - MD5_HMAC algorithm.
  - SHA224-HMAC algorithm.
  - SHA384-HMAC algorithm.
  - GMAC algorithm.
  - KASUMI (F8 and F9) algorithm.
  - 3DES algorithm.
  - NULL algorithm.
  - C3XXX device.
  - C62XX device.

- **Added openssl PMD.**

  A new crypto PMD has been added, which provides several ciphering and hashing algorithms. All cryptography operations use the Openssl library crypto API.

- **Updated the IPsec example.**

  Updated the IPsec example with the following support:
  - Configuration file support.
  - AES CBC IV generation with cipher forward function.
  - AES GCM/CTR mode.

- **Added support for new gcc -march option.**

  The GCC 4.9 `-march` option supports the Intel processor code names. The config option `RTE_MACHINE` can be used to pass code names to the compiler via the `-march` flag.

13.4.2 Resolved Issues

Drivers

- **enic: Fixed several flow director issues.**
- **enic: Fixed inadvertent setting of L4 checksum ptype on ICMP packets.**
- **enic: Fixed high driver overhead when servicing Rx queues beyond the first.**

13.4.3 Known Issues

- **L3fwd-power app does not work properly when Rx vector is enabled.**

  The L3fwd-power app doesn't work properly with some drivers in vector mode since the queue monitoring works differently between scalar and vector modes leading to incorrect frequency scaling. In addition, L3fwd-power application requires the mbuf to have correct packet type set but in some drivers the vector mode must be disabled for this.

  Therefore, in order to use L3fwd-power, vector mode should be disabled via the config file.

- **Digest address must be supplied for crypto auth operation on QAT PMD.**

  The cryptodev API specifies that if the rte_crypto_sym_op.digest.data field, and by inference the digest.phys_addr field which points to the same location, is not set for an auth operation the driver is to understand that the digest result is located immediately following the region over which the digest is computed. The QAT PMD doesn't correctly handle this case and reads and writes to an incorrect location.

  Callers can workaround this by always supplying the digest virtual and physical address fields in the rte_crypto_sym_op for an auth operation.

13.4.4 API Changes

- The driver naming convention has been changed to make them more consistent. It especially impacts `--vdev` arguments. For example `eth_pcap` becomes `net_pcap` and `cryptodev_aesni_mb_pmd` becomes `crypto_aesni_mb`.

  For backward compatibility an alias feature has been enabled to support the original names.

- The log history has been removed.
- The `rte_ivshmem` feature (including library and EAL code) has been removed in 16.11 because it had some design issues which were not planned to be fixed.
- The `file_name` data type of `struct rte_port_source_params` and `struct rte_port_sink_params` is changed from `char *` to `const char *`.
- **Improved device/driver hierarchy and generalized hotplugging.**

  The device and driver relationship has been restructured by introducing generic classes. This paves the way for having PCI, VDEV and other device types as instantiated objects rather than classes in themselves. Hotplugging has also been generalized into EAL so that Ethernet or crypto devices can use the common infrastructure.

  - Removed `pmd_type` as a way of segregation of devices.
  - Moved `numa_node` and `devargs` into `rte_driver` from `rte_pci_driver`. These can now be used by any instantiated object of `rte_driver`.
  - Added `rte_device` class and all PCI and VDEV devices inherit from it

- Renamed devinit/devuninit handlers to probe/remove to make it more semantically correct with respect to the device <=> driver relationship.
- Moved hotplugging support to EAL. Hereafter, PCI and vdev can use the APIs `rte_eal_dev_attach` and `rte_eal_dev_detach`.
- Renamed helpers and support macros to make them more synonymous with their device types (e.g. `PMD_REGISTER_DRIVER` => `RTE_PMD_REGISTER_PCI`).
- Device naming functions have been generalized from ethdev and cryptodev to EAL. `rte_eal_pci_device_name` has been introduced for obtaining unique device name from PCI Domain-BDF description.
- Virtual device registration APIs have been added: `rte_eal_vdrv_register` and `rte_eal_vdrv_unregister`.

13.4.5 ABI Changes

13.4.6 Shared Library Versions

The libraries prepended with a plus sign were incremented in this version.

```
  librte_acl.so.2
  librte_cfgfile.so.2
  librte_cmdline.so.2
+ librte_cryptodev.so.2
  librte_distributor.so.1
+ librte_eal.so.3
+ librte_ethdev.so.5
  librte_hash.so.2
  librte_ip_frag.so.1
  librte_jobstats.so.1
  librte_kni.so.2
  librte_kvargs.so.1
  librte_lpm.so.2
  librte_mbuf.so.2
  librte_mempool.so.2
  librte_meter.so.1
  librte_net.so.1
  librte_pdump.so.1
  librte_pipeline.so.3
  librte_pmd_bond.so.1
  librte_pmd_ring.so.2
  librte_port.so.3
  librte_power.so.1
  librte_reorder.so.1
  librte_ring.so.1
  librte_sched.so.1
  librte_table.so.2
  librte_timer.so.1
  librte_vhost.so.3
```

13.4.7 Tested Platforms

1. SuperMicro 1U
   - BIOS: 1.0c

- Processor: Intel(R) Atom(TM) CPU C2758 @ 2.40GHz

2. SuperMicro 1U
   - BIOS: 1.0a
   - Processor: Intel(R) Xeon(R) CPU D-1540 @ 2.00GHz
   - Onboard NIC: Intel(R) X552/X557-AT (2x10G)
     - Firmware-version: 0x800001cf
     - Device ID (PF/VF): 8086:15ad /8086:15a8
   - kernel driver version: 4.2.5 (ixgbe)
3. SuperMicro 2U
   - BIOS: 1.0a
   - Processor: Intel(R) Xeon(R) CPU E5-4667 v3 @ 2.00GHz
4. Intel(R) Server board S2600GZ
   - BIOS: SE5C600.86B.02.02.0002.122320131210
   - Processor: Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
5. Intel(R) Server board W2600CR
   - BIOS: SE5C600.86B.02.01.0002.082220131453
   - Processor: Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
6. Intel(R) Server board S2600CWT
   - BIOS: SE5C610.86B.01.01.0009.060120151350
   - Processor: Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz
7. Intel(R) Server board S2600WTT
   - BIOS: SE5C610.86B.01.01.0005.101720141054
   - Processor: Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz
8. Intel(R) Server board S2600WTT
   - BIOS: SE5C610.86B.11.01.0044.090120151156
   - Processor: Intel(R) Xeon(R) CPU E5-2695 v4 @ 2.10GHz
9. Intel(R) Server board S2600WTT
   - Processor: Intel(R) Xeon(R) CPU E5-2697 v2 @ 2.70GHz
10. Intel(R) Server
    - Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz
11. IBM(R) Power8(R)
    - Machine type-model: 8247-22L
    - Firmware FW810.21 (SV810_108)
    - Processor: POWER8E (raw), AltiVec supported

13.4.8 Tested NICs

1. Intel(R) Ethernet Controller X540-AT2
   - Firmware version: 0x80000389
   - Device id (pf): 8086:1528
   - Driver version: 3.23.2 (ixgbe)
2. Intel(R) 82599ES 10 Gigabit Ethernet Controller
   - Firmware version: 0x61bf0001
   - Device id (pf/vf): 8086:10fb / 8086:10ed
   - Driver version: 4.0.1-k (ixgbe)
3. Intel(R) Corporation Ethernet Connection X552/X557-AT 10GBASE-T
   - Firmware version: 0x800001cf
   - Device id (pf/vf): 8086:15ad / 8086:15a8
   - Driver version: 4.2.5 (ixgbe)
4. Intel(R) Ethernet Converged Network Adapter X710-DA4 (4x10G)
   - Firmware version: 5.05
   - Device id (pf/vf): 8086:1572 / 8086:154c
   - Driver version: 1.5.23 (i40e)
5. Intel(R) Ethernet Converged Network Adapter X710-DA2 (2x10G)
   - Firmware version: 5.05
   - Device id (pf/vf): 8086:1572 / 8086:154c
   - Driver version: 1.5.23 (i40e)
6. Intel(R) Ethernet Converged Network Adapter XL710-QDA1 (1x40G)
   - Firmware version: 5.05
   - Device id (pf/vf): 8086:1584 / 8086:154c
   - Driver version: 1.5.23 (i40e)
7. Intel(R) Ethernet Converged Network Adapter XL710-QDA2 (2X40G)
   - Firmware version: 5.05
   - Device id (pf/vf): 8086:1583 / 8086:154c
   - Driver version: 1.5.23 (i40e)
8. Intel(R) Corporation I350 Gigabit Network Connection
   - Firmware version: 1.48, 0x800006e7
   - Device id (pf/vf): 8086:1521 / 8086:1520
   - Driver version: 5.2.13-k (igb)
9. Intel(R) Ethernet Multi-host Controller FM10000
   - Firmware version: N/A
   - Device id (pf/vf): 8086:15d0

- Driver version: 0.17.0.9 (fm10k)

10. Mellanox(R) ConnectX(R)-4 10G MCX4111A-XCAT (1x10G)
    - Host interface: PCI Express 3.0 x8
    - Device ID: 15b3:1013
    - MLNX_OFED: 3.4-1.0.0.0
    - Firmware version: 12.17.1010
11. Mellanox(R) ConnectX(R)-4 10G MCX4121A-XCAT (2x10G)
    - Host interface: PCI Express 3.0 x8
    - Device ID: 15b3:1013
    - MLNX_OFED: 3.4-1.0.0.0
    - Firmware version: 12.17.1010
12. Mellanox(R) ConnectX(R)-4 25G MCX4111A-ACAT (1x25G)
    - Host interface: PCI Express 3.0 x8
    - Device ID: 15b3:1013
    - MLNX_OFED: 3.4-1.0.0.0
    - Firmware version: 12.17.1010
13. Mellanox(R) ConnectX(R)-4 25G MCX4121A-ACAT (2x25G)
    - Host interface: PCI Express 3.0 x8
    - Device ID: 15b3:1013
    - MLNX_OFED: 3.4-1.0.0.0
    - Firmware version: 12.17.1010
14. Mellanox(R) ConnectX(R)-4 40G MCX4131A-BCAT/MCX413A-BCAT (1x40G)
    - Host interface: PCI Express 3.0 x8
    - Device ID: 15b3:1013
    - MLNX_OFED: 3.4-1.0.0.0
    - Firmware version: 12.17.1010
15. Mellanox(R) ConnectX(R)-4 40G MCX415A-BCAT (1x40G)
    - Host interface: PCI Express 3.0 x16
    - Device ID: 15b3:1013
    - MLNX_OFED: 3.4-1.0.0.0
    - Firmware version: 12.17.1010
16. Mellanox(R) ConnectX(R)-4 50G MCX4131A-GCAT/MCX413A-GCAT (1x50G)
    - Host interface: PCI Express 3.0 x8
    - Device ID: 15b3:1013
    - MLNX_OFED: 3.4-1.0.0.0
    - Firmware version: 12.17.1010

17. Mellanox(R) ConnectX(R)-4 50G MCX414A-BCAT (2x50G)
    - Host interface: PCI Express 3.0 x8
    - Device ID: 15b3:1013
    - MLNX_OFED: 3.4-1.0.0.0
    - Firmware version: 12.17.1010
18. Mellanox(R) ConnectX(R)-4 50G MCX415A-GCAT/MCX416A-BCAT/MCX416A-GCAT (2x50G)
    - Host interface: PCI Express 3.0 x16
    - Device ID: 15b3:1013
    - MLNX_OFED: 3.4-1.0.0.0
    - Firmware version: 12.17.1010
19. Mellanox(R) ConnectX(R)-4 50G MCX415A-CCAT (1x100G)
    - Host interface: PCI Express 3.0 x16
    - Device ID: 15b3:1013
    - MLNX_OFED: 3.4-1.0.0.0
    - Firmware version: 12.17.1010
20. Mellanox(R) ConnectX(R)-4 100G MCX416A-CCAT (2x100G)
    - Host interface: PCI Express 3.0 x16
    - Device ID: 15b3:1013
    - MLNX_OFED: 3.4-1.0.0.0
    - Firmware version: 12.17.1010
21. Mellanox(R) ConnectX(R)-4 Lx 10G MCX4121A-XCAT (2x10G)
    - Host interface: PCI Express 3.0 x8
    - Device ID: 15b3:1015
    - MLNX_OFED: 3.4-1.0.0.0
    - Firmware version: 14.17.1010
22. Mellanox(R) ConnectX(R)-4 Lx 25G MCX4121A-ACAT (2x25G)
    - Host interface: PCI Express 3.0 x8
    - Device ID: 15b3:1015
    - MLNX_OFED: 3.4-1.0.0.0
    - Firmware version: 14.17.1010

13.4.9 Tested OSes

- CentOS 7.2
- Fedora 23
- Fedora 24
- FreeBSD 10.3

- FreeBSD 11
- Red Hat Enterprise Linux Server release 6.7 (Santiago)
- Red Hat Enterprise Linux Server release 7.0 (Maipo)
- Red Hat Enterprise Linux Server release 7.2 (Maipo)
- SUSE Enterprise Linux 12
- Wind River Linux 6.0.0.26
- Wind River Linux 8
- Ubuntu 14.04
- Ubuntu 15.04
- Ubuntu 16.04

13.5 DPDK Release 16.07

13.5.1 New Features

- **Removed the mempool cache memory if caching is not being used.**

  The size of the mempool structure is reduced if the per-lcore cache is disabled.

- **Added mempool external cache for non-EAL thread.**

  Added new functions to create, free or flush a user-owned mempool cache for non-EAL threads. Previously the caching was always disabled on these threads.

- **Changed the memory allocation scheme in the mempool library.**
  - Added the ability to allocate a large mempool in fragmented virtual memory.
  - Added new APIs to populate a mempool with memory.
  - Added an API to free a mempool.
  - Modified the API of the `rte_mempool_obj_iter()` function.
  - Dropped the specific Xen Dom0 code.
  - Dropped the specific anonymous mempool code in testpmd.

- **Added a new driver for Broadcom NetXtreme-C devices.**

  Added the new bnxt driver for Broadcom NetXtreme-C devices. See the "Network Interface Controller Drivers" document for more details on this new driver.

- **Added a new driver for ThunderX nicvf devices.**

  Added the new thunderx net driver for ThunderX nicvf devices. See the "Network Interface Controller Drivers" document for more details on this new driver.

- **Added mailbox interrupt support for ixgbe and igb VFs.**

  When the physical NIC link comes up or down, the PF driver will send a mailbox message to notify each VF. To handle this link up/down event, support have been added for a mailbox interrupt to receive the message and allow the application to register a callback for it.

- **Updated the ixgbe base driver.**

  The ixgbe base driver was updated with changes including the following:

  - Added sgmii link for X550.
  - Added MAC link setup for X550a SFP and SFP+.
  - Added KR support for X550em_a.
  - Added new PHY definitions for M88E1500.
  - Added support for the VLVF to be bypassed when adding/removing a VFTA entry.
  - Added X550a flow control auto negotiation support.

- **Updated the i40e base driver.**

  Updated the i40e base driver including support for new devices IDs.

- **Updated the enic driver.**

  The enic driver was updated with changes including the following:

  - Optimized the Tx function.
  - Added Scattered Rx capability.
  - Improved packet type identification.
  - Added MTU update in non Scattered Rx mode and enabled MTU of up to 9208 with UCS Software release 2.2 on 1300 series VICs.

- **Updated the mlx5 driver.**

  The mlx5 driver was updated with changes including the following:

  - Data path was refactored to bypass Verbs to improve RX and TX performance.
  - Removed compilation parameters for inline send, `MLX5_MAX_INLINE`, and added command line parameter instead, `txq_inline`.
  - Improved TX scatter gather support: Removed compilation parameter `MLX5_PMD_SGE_WR_N`. Scatter-gather elements is set to the maximum value the NIC supports. Removed linearization logic, this decreases the memory consumption of the PMD.
  - Improved jumbo frames support, by dynamically setting RX scatter gather elements according to the MTU and mbuf size, no need for compilation parameter `MLX5_PMD_SGE_WR_N`

- **Added support for virtio on IBM POWER8.**

  The ioports are mapped in memory when using Linux UIO.

- **Added support for Virtio in containers.**

  Add a new virtual device, named virtio_user, to support virtio for containers.

  Known limitations:

  - Control queue and multi-queue are not supported yet.
  - Doesn't work with `--huge-unlink`.
  - Doesn't work with `--no-huge`.
  - Doesn't work when there are more than `VHOST_MEMORY_MAX_NREGIONS(8)` hugepages.
  - Root privilege is required for sorting hugepages by physical address.
  - Can only be used with the vhost user backend.

- **Added vhost-user client mode.**

  DPDK vhost-user now supports client mode as well as server mode. Client mode is enabled when the `RTE_VHOST_USER_CLIENT` flag is set while calling `rte_vhost_driver_register`.

  When DPDK vhost-user restarts from an normal or abnormal exit (such as a crash), the client mode allows DPDK to establish the connection again. Note that QEMU version v2.7 or above is required for this feature.

  DPDK vhost-user will also try to reconnect by default when:

  - The first connect fails (for example when QEMU is not started yet).
  - The connection is broken (for example when QEMU restarts).

  It can be turned off by setting the `RTE_VHOST_USER_NO_RECONNECT` flag.

- **Added NSH packet recognition in i40e.**
- **Added AES-CTR support to AESNI MB PMD.**

  Now AESNI MB PMD supports 128/192/256-bit counter mode AES encryption and decryption.

- **Added AES counter mode support for Intel QuickAssist devices.**

  Enabled support for the AES CTR algorithm for Intel QuickAssist devices. Provided support for algorithm-chaining operations.

- **Added KASUMI SW PMD.**

  A new Crypto PMD has been added, which provides KASUMI F8 (UEA1) ciphering and KASUMI F9 (UIA1) hashing.

- **Added multi-writer support for RTE Hash with Intel TSX.**

  The following features/modifications have been added to rte_hash library:

  - Enabled application developers to use an extra flag for `rte_hash` creation to specify default behavior (multi-thread safe/unsafe) with the `rte_hash_add_key` function.
  - Changed the Cuckoo Hash Search algorithm to breadth first search for multi-writer routines and split Cuckoo Hash Search and Move operations in order to reduce transactional code region and improve TSX performance.
  - Added a hash multi-writer test case to the test app.

- **Improved IP Pipeline Application.**

  The following features have been added to the ip_pipeline application:

  - Configure the MAC address in the routing pipeline and automatic route updates with change in link state.
  - Enable RSS per network interface through the configuration file.
  - Streamline the CLI code.

- **Added keepalive enhancements.**

  Added support for reporting of core states other than "dead" to monitoring applications, enabling the support of broader liveness reporting to external processes.

- **Added packet capture framework.**

  - A new library `librte_pdump` is added to provide a packet capture API.
  - A new `app/pdump` tool is added to demonstrate capture packets in DPDK.

- **Added floating VEB support for i40e PF driver.**

  A "floating VEB" is a special Virtual Ethernet Bridge (VEB) which does not have an upload port, but instead is used for switching traffic between virtual functions (VFs) on a port.

  For information on this feature, please see the "I40E Poll Mode Driver" section of the "Network Interface Controller Drivers" document.

- **Added support for live migration of a VM with SRIOV VF.**

  Live migration of a VM with Virtio and VF PMD's using the bonding PMD.

13.5.2 Resolved Issues

EAL

- **igb_uio: Fixed possible mmap failure for Linux >= 4.5.**

  The mmaping of the iomem range of the PCI device fails for kernels that enabled the `CONFIG_IO_STRICT_DEVMEM` option. The error seen by the user is as similar to the following:

```
EAL: pci_map_resource():

    cannot mmap(39, 0x7f1c51800000, 0x100000, 0x0):
    Invalid argument (0xffffffffffffffff)
```

  The `CONFIG_IO_STRICT_DEVMEM` kernel option was introduced in Linux v4.5.

  The issues was resolve by updating `igb_uio` to stop reserving PCI memory resources. From the kernel point of view the iomem region looks like idle and mmap works again. This matches the `uio_pci_generic` usage.

Drivers

- **i40e: Fixed vlan stripping from inner header.**

  Previously, for tunnel packets, such as VXLAN/NVGRE, the vlan tags of the inner header will be stripped without putting vlan info to descriptor. Now this issue is fixed by disabling vlan stripping from inner header.

- **i40e: Fixed the type issue of a single VLAN type.**

  Currently, if a single VLAN header is added in a packet, it's treated as inner VLAN. But generally, a single VLAN header is treated as the outer VLAN header. This issue is fixed by changing corresponding register for single VLAN.

- **enic: Fixed several issues when stopping then restarting ports and queues.**

  Fixed several crashes related to stopping then restarting ports and queues. Fixed possible crash when reconfiguring the number of Rx queue descriptors.

- **enic: Fixed Rx data mis-alignment if mbuf data offset modified.**

  Fixed possible Rx corruption when mbufs were returned to a pool with data offset other than RTE_PKTMBUF_HEADROOM.

- **enic: Fixed Tx IP/UDP/TCP checksum offload and VLAN insertion.**

- **enic: Fixed Rx error and missed counters.**

Libraries

- **mbuf: Fixed refcnt update when detaching.**

  Fix the `rte_pktmbuf_detach()` function to decrement the direct mbuf's reference counter. The previous behavior was not to affect the reference counter. This lead to a memory leak of the direct mbuf.

Examples

Other

13.5.3 Known Issues

13.5.4 API Changes

- The following counters are removed from the `rte_eth_stats` structure:
  - `ibadcrc`
  - `ibadlen`
  - `imcasts`
  - `fdirmatch`
  - `fdirmiss`
  - `tx_pause_xon`
  - `rx_pause_xon`
  - `tx_pause_xoff`
  - `rx_pause_xoff`
- The extended statistics are fetched by ids with `rte_eth_xstats_get` after a lookup by name `rte_eth_xstats_get_names`.
- The function `rte_eth_dev_info_get` fill the new fields `nb_rx_queues` and `nb_tx_queues` in the structure `rte_eth_dev_info`.
- The vhost function `rte_vring_available_entries` is renamed to `rte_vhost_avail_entries`.
- All existing vhost APIs and callbacks with `virtio_net` struct pointer as the parameter have been changed due to the ABI refactoring described below. It is replaced by `int vid`.
- The function `rte_vhost_enqueue_burst` no longer supports concurrent enqueuing packets to the same queue.
- The function `rte_eth_dev_set_mtu` adds a new return value `-EBUSY`, which indicates the operation is forbidden because the port is running.
- The script `dpdk_nic_bind.py` is renamed to `dpdk-devbind.py`. And the script `setup.sh` is renamed to `dpdk-setup.sh`.

13.5.5 ABI Changes

- The `rte_port_source_params` structure has new fields to support PCAP files. It was already in release 16.04 with `RTE_NEXT_ABI` flag.

- The `rte_eth_dev_info` structure has new fields `nb_rx_queues` and `nb_tx_queues` to support the number of queues configured by software.
- A Vhost ABI refactoring has been made: the `virtio_net` structure is no longer exported directly to the application. Instead, a handle, `vid`, has been used to represent this structure internally.

13.5.6 Shared Library Versions

The libraries prepended with a plus sign were incremented in this version.

```
+ libethdev.so.4
  librte_acl.so.2
  librte_cfgfile.so.2
  librte_cmdline.so.2
  librte_cryptodev.so.1
  librte_distributor.so.1
  librte_eal.so.2
  librte_hash.so.2
  librte_ip_frag.so.1
  librte_ivshmem.so.1
  librte_jobstats.so.1
  librte_kni.so.2
  librte_kvargs.so.1
  librte_lpm.so.2
  librte_mbuf.so.2
+ librte_mempool.so.2
  librte_meter.so.1
  librte_pdump.so.1
  librte_pipeline.so.3
  librte_pmd_bond.so.1
  librte_pmd_ring.so.2
+ librte_port.so.3
  librte_power.so.1
  librte_reorder.so.1
  librte_ring.so.1
  librte_sched.so.1
  librte_table.so.2
  librte_timer.so.1
+ librte_vhost.so.3
```

13.5.7 Tested Platforms

1. SuperMicro 1U
   - BIOS: 1.0c
   - Processor: Intel(R) Atom(TM) CPU C2758 @ 2.40GHz
2. SuperMicro 1U
   - BIOS: 1.0a
   - Processor: Intel(R) Xeon(R) CPU D-1540 @ 2.00GHz
   - Onboard NIC: Intel(R) X552/X557-AT (2x10G)
     - Firmware-version: 0x800001cf
     - Device ID (PF/VF): 8086:15ad /8086:15a8

- kernel driver version: 4.2.5 (ixgbe)

3. SuperMicro 2U
   - BIOS: 1.0a
   - Processor: Intel(R) Xeon(R) CPU E5-4667 v3 @ 2.00GHz
4. Intel(R) Server board S2600GZ
   - BIOS: SE5C600.86B.02.02.0002.122320131210
   - Processor: Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
5. Intel(R) Server board W2600CR
   - BIOS: SE5C600.86B.02.01.0002.082220131453
   - Processor: Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
6. Intel(R) Server board S2600CWT
   - BIOS: SE5C610.86B.01.01.0009.060120151350
   - Processor: Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz
7. Intel(R) Server board S2600WTT
   - BIOS: SE5C610.86B.01.01.0005.101720141054
   - Processor: Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz
8. Intel(R) Server board S2600WTT
   - BIOS: SE5C610.86B.11.01.0044.090120151156
   - Processor: Intel(R) Xeon(R) CPU E5-2695 v4 @ 2.10GHz

13.5.8 Tested NICs

1. Intel(R) Ethernet Controller X540-AT2
   - Firmware version: 0x80000389
   - Device id (pf): 8086:1528
   - Driver version: 3.23.2 (ixgbe)
2. Intel(R) 82599ES 10 Gigabit Ethernet Controller
   - Firmware version: 0x61bf0001
   - Device id (pf/vf): 8086:10fb / 8086:10ed
   - Driver version: 4.0.1-k (ixgbe)
3. Intel(R) Corporation Ethernet Connection X552/X557-AT 10GBASE-T
   - Firmware version: 0x800001cf
   - Device id (pf/vf): 8086:15ad / 8086:15a8
   - Driver version: 4.2.5 (ixgbe)
4. Intel(R) Ethernet Converged Network Adapter X710-DA4 (4x10G)
   - Firmware version: 5.04
   - Device id (pf/vf): 8086:1572 / 8086:154c

- Driver version: 1.4.26 (i40e)

5. Intel(R) Ethernet Converged Network Adapter X710-DA2 (2x10G)
   - Firmware version: 5.04
   - Device id (pf/vf): 8086:1572 / 8086:154c
   - Driver version: 1.4.25 (i40e)
6. Intel(R) Ethernet Converged Network Adapter XL710-QDA1 (1x40G)
   - Firmware version: 5.04
   - Device id (pf/vf): 8086:1584 / 8086:154c
   - Driver version: 1.4.25 (i40e)
7. Intel(R) Ethernet Converged Network Adapter XL710-QDA2 (2X40G)
   - Firmware version: 5.04
   - Device id (pf/vf): 8086:1583 / 8086:154c
   - Driver version: 1.4.25 (i40e)
8. Intel(R) Corporation I350 Gigabit Network Connection
   - Firmware version: 1.48, 0x800006e7
   - Device id (pf/vf): 8086:1521 / 8086:1520
   - Driver version: 5.2.13-k (igb)
9. Intel(R) Ethernet Multi-host Controller FM10000
   - Firmware version: N/A
   - Device id (pf/vf): 8086:15d0
   - Driver version: 0.17.0.9 (fm10k)

13.5.9 Tested OSes

- CentOS 7.0
- Fedora 23
- Fedora 24
- FreeBSD 10.3
- Red Hat Enterprise Linux 7.2
- SUSE Enterprise Linux 12
- Ubuntu 15.10
- Ubuntu 16.04 LTS
- Wind River Linux 8

13.6 DPDK Release 16.04

13.6.1 New Features

- **Added function to check primary process state.**

  A new function `rte_eal_primary_proc_alive()` has been added to allow the user to detect if a primary process is running. Use cases for this feature include fault detection, and monitoring using secondary processes.

- **Enabled bulk allocation of mbufs.**

  A new function `rte_pktmbuf_alloc_bulk()` has been added to allow the user to bulk allocate mbufs.

- **Added device link speed capabilities.**

  The structure `rte_eth_dev_info` now has a `speed_capa` bitmap, which allows the application to determine the supported speeds of each device.

- **Added bitmap of link speeds to advertise.**

  Added a feature to allow the definition of a set of advertised speeds for auto-negotiation, explicitly disabling link auto-negotiation (single speed) and full auto-negotiation.

- **Added new poll-mode driver for Amazon Elastic Network Adapters (ENA).**

  The driver operates for a variety of ENA adapters through feature negotiation with the adapter and upgradable commands set. The ENA driver handles PCI Physical and Virtual ENA functions.

- **Restored vmxnet3 TX data ring.**

  TX data ring has been shown to improve small packet forwarding performance on the vSphere environment.

- **Added vmxnet3 TX L4 checksum offload.**

  Added support for TCP/UDP checksum offload to vmxnet3.

- **Added vmxnet3 TSO support.**

  Added support for TSO to vmxnet3.

- **Added vmxnet3 support for jumbo frames.**

  Added support for linking multi-segment buffers together to handle Jumbo packets.

- **Enabled Virtio 1.0 support.**

  Enabled Virtio 1.0 support for Virtio pmd driver.

- **Supported Virtio for ARM.**

  Enabled Virtio support for ARMv7/v8. Tested for ARM64. Virtio for ARM supports VFIO-noiommu mode only. Virtio can work with other non-x86 architectures as well, like PowerPC.

- **Supported Virtio offload in vhost-user.**

  Added the offload and negotiation of checksum and TSO between vhost-user and vanilla Linux Virtio guest.

- **Added vhost-user live migration support.**

- **Added vhost driver.**

  Added a virtual PMD that wraps `librte_vhost`.

- **Added multicast promiscuous mode support on VF for ixgbe.**

  Added multicast promiscuous mode support for the ixgbe VF driver so all VFs can receive the multicast packets.

Please note if you want to use this promiscuous mode, you need both PF and VF driver to support it. The reason is that this VF feature is configured in the PF. If you use kernel PF driver and the dpdk VF driver, make sure the kernel PF driver supports VF multicast promiscuous mode. If you use dpdk PF and dpdk VF ensure the PF driver is the same version as the VF.

- **Added support for E-tag on X550.**

  E-tag is defined in 802.1BR - Bridge Port Extension.

  This feature is for the VF, but the settings are on the PF. It means the CLIs should be used on the PF, but some of their effects will be shown on the VF. The forwarding of E-tag packets based on GRP and E-CID_base will have an effect on the PF. Theoretically, the E-tag packets can be forwarded to any pool/queue but normally we'd like to forward the packets to the pools/queues belonging to the VFs. And E-tag insertion and stripping will have an effect on VFs. When a VF receives E-tag packets it should strip the E-tag. When the VF transmits packets, it should insert the E-tag. Both actions can be offloaded.

  When we want to use this E-tag support feature, the forwarding should be enabled to forward the packets received by the PF to the indicated VFs. And insertion and stripping should be enabled for VFs to offload the effort to hardware.

  Features added:

  - Support E-tag offloading of insertion and stripping.
  - Support Forwarding E-tag packets to pools based on GRP and E-CID_base.

- **Added support for VxLAN and NVGRE checksum off-load on X550.**

  - Added support for VxLAN and NVGRE RX/TX checksum off-load on X550. RX/TX checksum off-load is provided on both inner and outer IP header and TCP header.
  - Added functions to support VxLAN port configuration. The default VxLAN port number is 4789 but this can be updated programmatically.

- **Added support for new X550EM_a devices.**

  Added support for new X550EM_a devices and their MAC types, X550EM_a and X550EM_a_vf. Updated the relevant PMD to use the new devices and MAC types.

- **Added x550em_x V2 device support.**

  Added support for x550em_x V2 device. Only x550em_x V1 was supported before. A mask for V1 and V2 is defined and used to support both.

- **Supported link speed auto-negotiation on X550EM_X**

  Normally the auto-negotiation is supported by firmware and software doesn't care about it. But on x550em_x, firmware doesn't support auto-negotiation. As the ports of x550em_x are 10GbE, if we connect the port with a peer which is 1GbE, the link will always be down. We added the support for auto-negotiation by software to avoid this link down issue.

- **Added software-firmware sync on X550EM_a.**

  Added support for software-firmware sync for resource sharing. Use the PHY token, shared between software-firmware for PHY access on X550EM_a.

- **Updated the i40e base driver.**

  The i40e base driver was updated with changes including the following:

  - Use RX control AQ commands to read/write RX control registers.
  - Add new X722 device IDs, and removed X710 one was never used.
  - Expose registers for HASH/FD input set configuring.

- **Enabled PCI extended tag for i40e.**

  Enabled extended tag for i40e by checking and writing corresponding PCI config space bytes, to boost the performance. The legacy method of reading/writing sysfile supported by kernel module igb_uio is now deprecated.

- **Added i40e support for setting mac addresses.**
- **Added dump of i40e registers and EEPROM.**
- **Supported ether type setting of single and double VLAN for i40e**
- **Added VMDQ DCB mode in i40e.**

  Added support for DCB in VMDQ mode to i40e driver.

- **Added i40e VEB switching support.**
- **Added Flow director enhancements in i40e.**
- **Added PF reset event reporting in i40e VF driver.**
- **Added fm10k RX interrupt support.**
- **Optimized fm10k TX.**

  Optimized fm10k TX by freeing multiple mbufs at a time.

- **Handled error flags in fm10k vector RX.**

  Parse error flags in RX descriptor and set error bits in mbuf with vector instructions.

- **Added fm10k FTAG based forwarding support.**
- **Added mlx5 flow director support.**

  Added flow director support (`RTE_FDIR_MODE_PERFECT` and `RTE_FDIR_MODE_PERFECT_MAC_VLAN`).

  Only available with Mellanox OFED >= 3.2.

- **Added mlx5 RX VLAN stripping support.**

  Added support for RX VLAN stripping.

  Only available with Mellanox OFED >= 3.2.

- **Added mlx5 link up/down callbacks.**

  Implemented callbacks to bring link up and down.

- **Added mlx5 support for operation in secondary processes.**

  Implemented TX support in secondary processes (like mlx4).

- **Added mlx5 RX CRC stripping configuration.**

  Until now, CRC was always stripped. It can now be configured.

  Only available with Mellanox OFED >= 3.2.

- **Added mlx5 optional packet padding by HW.**

  Added an option to make PCI bus transactions rounded to a multiple of a cache line size for better alignment.

  Only available with Mellanox OFED >= 3.2.

- **Added mlx5 TX VLAN insertion support.**

  Added support for TX VLAN insertion.

  Only available with Mellanox OFED >= 3.2.

- **Changed szedata2 driver type from vdev to pdev.**

  Previously szedata2 device had to be added by `--vdev` option. Now szedata2 PMD recognizes the device automatically during EAL initialization.

- **Added szedata2 functions for setting link up/down.**
- **Added szedata2 promiscuous and allmulticast modes.**
- **Added af_packet dynamic removal function.**

  An af_packet device can now be detached using the API, like other PMD devices.

- **Increased number of next hops for LPM IPv4 to 2^24.**

  The `next_hop` field has been extended from 8 bits to 24 bits for IPv4.

- **Added support of SNOW 3G (UEA2 and UIA2) for Intel Quick Assist devices.**

  Enabled support for the SNOW 3G wireless algorithm for Intel Quick Assist devices. Support for cipher-only and hash-only is also provided along with algorithm-chaining operations.

- **Added SNOW3G SW PMD.**

  A new Crypto PMD has been added, which provides SNOW 3G UEA2 ciphering and SNOW3G UIA2 hashing.

- **Added AES GCM PMD.**

  Added new Crypto PMD to support AES-GCM authenticated encryption and authenticated decryption in software.

- **Added NULL Crypto PMD**

  Added new Crypto PMD to support null crypto operations in software.

- **Improved IP Pipeline Application.**

  The following features have been added to ip_pipeline application;

  - Added CPU utilization measurement and idle cycle rate computation.
  - Added link identification support through existing port-mask option or by specifying PCI device in every LINK section in the configuration file.
  - Added load balancing support in passthrough pipeline.

- **Added IPsec security gateway example.**

  Added a new application implementing an IPsec Security Gateway.

13.6.2 Resolved Issues

Drivers

- **ethdev: Fixed overflow for 100Gbps.**

  100Gbps in Mbps (100000) was exceeding the 16-bit max value of `link_speed` in `rte_eth_link`.

- **ethdev: Fixed byte order consistency between fdir flow and mask.**

  Fixed issue in ethdev library where the structure for setting fdir's mask and flow entry was not consistent in byte ordering.

- **cxgbe: Fixed crash due to incorrect size allocated for RSS table.**

  Fixed a segfault that occurs when accessing part of port 0's RSS table that gets overwritten by subsequent port 1's part of the RSS table due to incorrect size allocated for each entry in the table.

- **cxgbe: Fixed setting wrong device MTU.**

  Fixed an incorrect device MTU being set due to the Ethernet header and CRC lengths being added twice.

- **ixgbe: Fixed zeroed VF mac address.**

  Resolved an issue where the VF MAC address is zeroed out in cases where the VF driver is loaded while the PF interface is down. The solution is to only set it when we get an ACK from the PF.

- **ixgbe: Fixed setting flow director flag twice.**

  Resolved an issue where packets were being dropped when switching to perfect filters mode.

- **ixgbe: Set MDIO speed after MAC reset.**

  The MDIO clock speed must be reconfigured after the MAC reset. The MDIO clock speed becomes invalid, therefore the driver reads invalid PHY register values. The driver now set the MDIO clock speed prior to initializing PHY ops and again after the MAC reset.

- **ixgbe: Fixed maximum number of available TX queues.**

  In IXGBE, the maximum number of TX queues varies depending on the NIC operating mode. This was not being updated in the device information, providing an incorrect number in some cases.

- **i40e: Generated MAC address for each VFs.**

  It generates a MAC address for each VFs during PF host initialization, and keeps the VF MAC address the same among different VF launch.

- **i40e: Fixed failure of reading/writing RX control registers.**

  Fixed i40e issue of failing to read/write rx control registers when under stress with traffic, which might result in application launch failure.

- **i40e: Enabled vector driver by default.**

  Previously, vector driver was disabled by default as it couldn't fill packet type info for l3fwd to work well. Now there is an option for l3fwd to analyze the packet type so the vector driver is enabled by default.

- **i40e: Fixed link info of VF.**

  Previously, the VF's link speed stayed at 10GbE and status always was up. It did not change even when the physical link's status changed. Now this issue is fixed to make VF's link info consistent with physical link.

- **mlx5: Fixed possible crash during initialization.**

  A crash could occur when failing to allocate private device context.

- **mlx5: Added port type check.**

  Added port type check to prevent port initialization on non-Ethernet link layers and to report an error.

- **mlx5: Applied VLAN filtering to broadcast and IPv6 multicast flows.**

  Prevented reception of multicast frames outside of configured VLANs.

- **mlx5: Fixed RX checksum offload in non L3/L4 packets.**

  Fixed report of bad checksum for packets of unknown type.

- **aesni_mb: Fixed wrong return value when creating a device.**

  The `cryptodev_aesni_mb_init()` function was returning the device id of the device created, instead of 0 (on success) that `rte_eal_vdev_init()` expects. This made it impossible to create more than one aesni_mb device from the command line.

- **qat: Fixed AES GCM decryption.**

  Allowed AES GCM on the cryptodev API, but in some cases gave invalid results due to incorrect IV setting.

Libraries

- **hash: Fixed CRC32c hash computation for non multiple of 4 bytes sizes.**

  Fix crc32c hash functions to return a valid crc32c value for data lengths not a multiple of 4 bytes.

- **hash: Fixed hash library to support multi-process mode.**

  Fix hash library to support multi-process mode, using a jump table, instead of storing a function pointer to the key compare function. Multi-process mode only works with the built-in compare functions, however a custom compare function (not in the jump table) can only be used in single-process mode.

- **hash: Fixed return value when allocating an existing hash table.**

  Changed the `rte_hash*_create()` functions to return `NULL` and set `rte_errno` to `EEXIST` when the object name already exists. This is the behavior described in the API documentation in the header file. The previous behavior was to return a pointer to the existing object in that case, preventing the caller from knowing if the object had to be freed or not.

- **lpm: Fixed return value when allocating an existing object.**

  Changed the `rte_lpm*_create()` functions to return `NULL` and set `rte_errno` to `EEXIST` when the object name already exists. This is the behavior described in the API documentation in the header file. The previous behavior was to return a pointer to the existing object in that case, preventing the caller from knowing if the object had to be freed or not.

- **librte_port: Fixed segmentation fault for ring and ethdev writer nodrop.**

  Fixed core dump issue on txq and swq when dropless is set to yes.

Examples

- **l3fwd-power: Fixed memory leak for non-IP packet.**

  Fixed issue in l3fwd-power where, on receiving packets of types other than IPv4 or IPv6, the mbuf was not released, and caused a memory leak.

- **l3fwd: Fixed using packet type blindly.**

  l3fwd makes use of packet type information without querying if devices or PMDs really set it. For those devices that don't set ptypes, add an option to parse it.

- **examples/vhost: Fixed frequent mbuf allocation failure.**

  The vhost-switch often fails to allocate mbuf when dequeue from vring because it wrongly calculates the number of mbufs needed.

13.6.3 API Changes

- The ethdev statistics counter `imissed` is considered to be independent of `ierrors`. All drivers are now counting the missed packets only once, i.e. drivers will not increment ierrors anymore for missed packets.
- The ethdev structure `rte_eth_dev_info` was changed to support device speed capabilities.
- The ethdev structures `rte_eth_link` and `rte_eth_conf` were changed to support the new link API.
- The functions `rte_eth_dev_udp_tunnel_add` and `rte_eth_dev_udp_tunnel_delete` have been renamed into `rte_eth_dev_udp_tunnel_port_add` and `rte_eth_dev_udp_tunnel_port_delete`.
- The `outer_mac` and `inner_mac` fields in structure `rte_eth_tunnel_filter_conf` are changed from pointer to struct in order to keep code's readability.
- The fields in ethdev structure `rte_eth_fdir_masks` were changed to be in big endian.
- A parameter `vlan_type` has been added to the function `rte_eth_dev_set_vlan_ether_type`.
- The af_packet device init function is no longer public. The device should be attached via the API.
- The LPM `next_hop` field is extended from 8 bits to 24 bits for IPv4 while keeping ABI compatibility.
- A new `rte_lpm_config` structure is used so the LPM library will allocate exactly the amount of memory which is necessary to hold application's rules. The previous ABI is kept for compatibility.
- The prototype for the pipeline input port, output port and table action handlers are updated: the pipeline parameter is added, the packets mask parameter has been either removed or made input-only.

13.6.4 ABI Changes

- The RETA entry size in `rte_eth_rss_reta_entry64` has been increased from 8-bit to 16-bit.
- The ethdev flow director structure `rte_eth_fdir_flow` structure was changed. New fields were added to extend flow director's input set.
- The cmdline buffer size has been increase from 256 to 512.

13.6.5 Shared Library Versions

The libraries prepended with a plus sign were incremented in this version.

```
+ libethdev.so.3
  librte_acl.so.2
  librte_cfgfile.so.2
+ librte_cmdline.so.2
  librte_distributor.so.1
  librte_eal.so.2
  librte_hash.so.2
  librte_ip_frag.so.1
  librte_ivshmem.so.1
  librte_jobstats.so.1
  librte_kni.so.2
  librte_kvargs.so.1
  librte_lpm.so.2
  librte_mbuf.so.2
  librte_mempool.so.1
  librte_meter.so.1
+ librte_pipeline.so.3
```

```
librte_pmd_bond.so.1
librte_pmd_ring.so.2
librte_port.so.2
librte_power.so.1
librte_reorder.so.1
librte_ring.so.1
librte_sched.so.1
librte_table.so.2
librte_timer.so.1
librte_vhost.so.2
```

13.6.6 Tested Platforms

1. SuperMicro 1U
   - BIOS: 1.0c
   - Processor: Intel(R) Atom(TM) CPU C2758 @ 2.40GHz
2. SuperMicro 1U
   - BIOS: 1.0a
   - Processor: Intel(R) Xeon(R) CPU D-1540 @ 2.00GHz
   - Onboard NIC: Intel(R) X552/X557-AT (2x10G)
     - Firmware-version: 0x800001cf
     - Device ID (PF/VF): 8086:15ad /8086:15a8
   - kernel driver version: 4.2.5 (ixgbe)
3. SuperMicro 1U
   - BIOS: 1.0a
   - Processor: Intel(R) Xeon(R) CPU E5-4667 v3 @ 2.00GHz
4. Intel(R) Server board S2600GZ
   - BIOS: SE5C600.86B.02.02.0002.122320131210
   - Processor: Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
5. Intel(R) Server board W2600CR
   - BIOS: SE5C600.86B.02.01.0002.082220131453
   - Processor: Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
6. Intel(R) Server board S2600CWT
   - BIOS: SE5C610.86B.01.01.0009.060120151350
   - Processor: Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz
7. Intel(R) Server board S2600WTT
   - BIOS: SE5C610.86B.01.01.0005.101720141054
   - Processor: Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz
8. Intel(R) Server board S2600WTT
   - BIOS: SE5C610.86B.11.01.0044.090120151156

- Processor: Intel(R) Xeon(R) CPU E5-2695 v4 @ 2.10GHz

13.6.7 Tested NICs

1. Intel(R) Ethernet Controller X540-AT2
   - Firmware version: 0x80000389
   - Device id (pf): 8086:1528
   - Driver version: 3.23.2 (ixgbe)
2. Intel(R) 82599ES 10 Gigabit Ethernet Controller
   - Firmware version: 0x61bf0001
   - Device id (pf/vf): 8086:10fb / 8086:10ed
   - Driver version: 4.0.1-k (ixgbe)
3. Intel(R) Corporation Ethernet Connection X552/X557-AT 10GBASE-T
   - Firmware version: 0x800001cf
   - Device id (pf/vf): 8086:15ad / 8086:15a8
   - Driver version: 4.2.5 (ixgbe)
4. Intel(R) Ethernet Converged Network Adapter X710-DA4 (4x10G)
   - Firmware version: 5.02 0x80002284
   - Device id (pf/vf): 8086:1572 / 8086:154c
   - Driver version: 1.4.26 (i40e)
5. Intel(R) Ethernet Converged Network Adapter X710-DA2 (2x10G)
   - Firmware version: 5.02 0x80002282
   - Device id (pf/vf): 8086:1572 / 8086:154c
   - Driver version: 1.4.25 (i40e)
6. Intel(R) Ethernet Converged Network Adapter XL710-QDA1 (1x40G)
   - Firmware version: 5.02 0x80002281
   - Device id (pf/vf): 8086:1584 / 8086:154c
   - Driver version: 1.4.25 (i40e)
7. Intel(R) Ethernet Converged Network Adapter XL710-QDA2 (2X40G)
   - Firmware version: 5.02 0x80002285
   - Device id (pf/vf): 8086:1583 / 8086:154c
   - Driver version: 1.4.25 (i40e)
8. Intel(R) 82576EB Gigabit Ethernet Controller
   - Firmware version: 1.2.1
   - Device id (pf): 8086:1526
   - Driver version: 5.2.13-k (igb)
9. Intel(R) Ethernet Controller I210

- Firmware version: 3.16, 0x80000500, 1.304.0
- Device id (pf): 8086:1533
- Driver version: 5.2.13-k (igb)

10. Intel(R) Corporation I350 Gigabit Network Connection
    - Firmware version: 1.48, 0x800006e7
    - Device id (pf/vf): 8086:1521 / 8086:1520
    - Driver version: 5.2.13-k (igb)
11. Intel(R) Ethernet Multi-host Controller FM10000
    - Firmware version: N/A
    - Device id (pf/vf): 8086:15d0
    - Driver version: 0.17.0.9 (fm10k)

13.7 DPDK Release 2.2

13.7.1 New Features

- **Introduce ARMv7 and ARMv8 architectures.**
  - It is now possible to build DPDK for the ARMv7 and ARMv8 platforms.
  - ARMv7 can be tested with virtual PMD drivers.
  - ARMv8 can be tested with virtual and physical PMD drivers.
- **Enabled freeing of ring.**

  A new function `rte_ring_free()` has been added to allow the user to free a ring if it was created with `rte_ring_create()`.
- **Added keepalive support to EAL and example application.**
- **Added experimental cryptodev API**

  The cryptographic processing of packets is provided as a preview with two drivers for:
  - Intel QuickAssist devices
  - Intel AES-NI multi-buffer library

  Due to its experimental state, the API may change without prior notice.
- **Added ethdev APIs for additional IEEE1588 support.**

  Added functions to read, write and adjust system time in the NIC. Added client slave sample application to demonstrate the IEEE1588 functionality.
- **Extended Statistics.**

  Defined an extended statistics naming scheme to store metadata in the name string of each statistic. Refer to the Extended Statistics section of the Programmers Guide for more details.

  Implemented the extended statistics API for the following PMDs:
  - `igb`
  - `igbvf`

  - `i40e`
  - `i40evf`
  - `fm10k`
  - `virtio`

- **Added API in ethdev to retrieve RX/TX queue information.**
  - Added the ability for the upper layer to query RX/TX queue information.
  - Added new fields in `rte_eth_dev_info` to represent information about RX/TX descriptors min/max/align numbers, per queue, for the device.
- **Added RSS dynamic configuration to bonding.**
- **Updated the e1000 base driver.**

  The e1000 base driver was updated with several features including the following:
  - Added new i218 devices
  - Allowed both ULP and EEE in Sx state
  - Initialized 88E1543 (Marvell 1543) PHY
  - Added flags to set EEE advertisement modes
  - Supported inverted format ETrackId
  - Added bit to disable packetbuffer read
  - Added defaults for i210 RX/TX PBSIZE
  - Check more errors for ESB2 init and reset
  - Check more NVM read errors
  - Return code after setting receive address register
  - Removed all NAHUM6LP_HW tags
- **Added e1000 RX interrupt support.**
- **Added igb TSO support for both PF and VF.**
- **Added RSS enhancements to Intel x550 NIC.**
  - Added support for 512 entry RSS redirection table.
  - Added support for per VF RSS redirection table.
- **Added Flow director enhancements on Intel x550 NIC.**
  - Added 2 new flow director modes on x550. One is MAC VLAN mode, the other is tunnel mode.
- **Updated the i40e base driver.**

  The i40e base driver was updated with several changes including the following:
  - Added promiscuous on VLAN support
  - Added a workaround to drop all flow control frames
  - Added VF capabilities to virtual channel interface
  - Added TX Scheduling related AQ commands
  - Added additional PCTYPES supported for FortPark RSS

  - Added parsing for CEE DCBX TLVs
  - Added FortPark specific registers
  - Added AQ functions to handle RSS Key and LUT programming
  - Increased PF reset max loop limit
- **Added i40e vector RX/TX.**
- **Added i40e RX interrupt support.**
- **Added i40e flow control support.**
- **Added DCB support to i40e PF driver.**
- **Added RSS/FD input set granularity on Intel X710/XL710.**
- **Added different GRE key length for input set on Intel X710/XL710.**
- **Added flow director support in i40e VF.**
- **Added i40e support of early X722 series.**

  Added early X722 support, for evaluation only, as the hardware is alpha.
- **Added fm10k vector RX/TX.**
- **Added fm10k TSO support for both PF and VF.**
- **Added fm10k VMDQ support.**
- **New NIC Boulder Rapid support.**

  Added support for the Boulder Rapid variant of Intel's fm10k NIC family.
- **Enhanced support for the Chelsio CXGBE driver.**
  - Added support for Jumbo Frames.
  - Optimized forwarding performance for Chelsio T5 40GbE cards.
- **Improved enic TX packet rate.**

  Reduced frequency of TX tail pointer updates to the NIC.
- **Added support for link status interrupts in mlx4.**
- **Added partial support (TX only) for secondary processes in mlx4.**
- **Added support for Mellanox ConnectX-4 adapters (mlx5).**

  The mlx5 poll-mode driver implements support for Mellanox ConnectX-4 EN and Mellanox ConnectX-4 Lx EN families of 10/25/40/50/100 Gb/s adapters.

  Like mlx4, this PMD is only available for Linux and is disabled by default due to external dependencies (libibverbs and libmlx5).
- **Added driver for Netronome nfp-6xxx card.**

  Support for using Netronome nfp-6xxx with PCI VFs.
- **Added virtual szedata2 driver for COMBO cards.**

  Added virtual PMD for COMBO-100G and COMBO-80G cards. PMD is disabled in default configuration.
- **Enhanced support for virtio driver.**
  - Virtio ring layout optimization (fixed avail ring)
  - Vector RX

  - Simple TX

- **Added vhost-user multiple queue support.**
- **Added port hotplug support to vmxnet3.**
- **Added port hotplug support to xenvirt.**
- **Added ethtool shim and sample application.**
- **Added experimental performance thread example application.**

  The new sample application demonstrates L3 forwarding with different threading models: pthreads, cgroups, or lightweight threads. The example includes a simple cooperative scheduler.

  Due to its experimental state this application may change without notice. The application is supported only for Linux x86_64.

- **Enhancements to the IP pipeline application.**

  The following features have been added to the `ip_pipeline` application;

  - Added Multiple Producers/Multiple Consumers (MPSC) and fragmentation/reassembly support to software rings.
  - Added a dynamic pipeline reconfiguration feature that allows binding a pipeline to other threads at runtime using CLI commands.
  - Added enable/disable of `promisc` mode from `ip_pipeline` configuration file.
  - Added check on RX queues and TX queues of each link whether they are used correctly in the `ip_pipeline` configuration file.
  - Added flow id parameters to the flow-classification table entries.
  - Added more functions to the routing pipeline: ARP table enable/disable, Q-in-Q and MPLS encapsulation, add color (traffic-class for QoS) to the MPLS tag.
  - Added flow-actions pipeline for traffic metering/marking (for e.g. Two Rate Three Color Marker (trTCM)), policer etc.
  - Modified the pass-through pipeline's actions-handler to implement a generic approach to extract fields from the packet's header and copy them to packet metadata.

13.7.2 Resolved Issues

EAL

- **eal/linux: Fixed epoll timeout.**

  Fixed issue where the `rte_epoll_wait()` function didn't return when the underlying call to `epoll_wait()` timed out.

Drivers

- **e1000/base: Synchronize PHY interface on non-ME systems.**

  On power up, the MAC - PHY interface needs to be set to PCIe, even if the cable is disconnected. In ME systems, the ME handles this on exit from the Sx (Sticky mode) state. In non-ME, the driver handles it. Added a check for non-ME system to the driver code that handles it.

- **e1000/base: Increased timeout of reset check.**

  Previously, in `check_reset_block` RSPCIPHY was polled for 100 ms before determining that the ME veto was set. This was not enough and it was increased to 300 ms.

- **e1000/base: Disabled IPv6 extension header parsing on 82575.**

  Disabled IPv6 options as per hardware limitation.

- **e1000/base: Prevent ULP flow if cable connected.**

  Enabling ULP on link down when the cable is connected caused an infinite loop of link up/down indications in the NDIS driver. The driver now enables ULP only when the cable is disconnected.

- **e1000/base: Support different EEARBC for i210.**

  EEARBC has changed on i210. It means EEARBC has a different address on i210 than on other NICs. So, add a new entity named EEARBC_I210 to the register list and make sure the right one is being used on i210.

- **e1000/base: Fix K1 configuration.**

  Added fix for the following updates to the K1 configurations: TX idle period for entering K1 should be 128 ns. Minimum TX idle period in K1 should be 256 ns.

- **e1000/base: Fix link detect flow.**

  Fix link detect flow in case where auto-negotiate is not enabled, by calling `e1000_setup_copper_link_generic` instead of `e1000_phy_setup_autoneg`.

- **e1000/base: Fix link check for i354 M88E1112 PHY.**

  The `e1000_check_for_link_media_swap()` function is supposed to check PHY page 0 for copper and PHY page 1 for "other" (fiber) links. The driver switched back from page 1 to page 0 too soon, before `e1000_check_for_link_82575()` is executed and was never finding the link on the fiber (other).

  If the link is copper, as the M88E1112 page address is set to 1, it should be set back to 0 before checking this link.

- **e1000/base: Fix beacon duration for i217.**

  Fix for I217 Packet Loss issue - The Management Engine sets the FEXTNVM4 Beacon Duration incorrectly. This fix ensures that the correct value will always be set. Correct value for this field is 8 usec.

- **e1000/base: Fix TIPG for non 10 half duplex mode.**

  TIPG value is increased when setting speed to 10 half duplex to prevent packet loss. However, it was never decreased again when speed changed. This caused performance issues in the NDIS driver. Fix this to restore TIPG to default value on non 10 half duplex.

- **e1000/base: Fix reset of DH89XXCC SGMII.**

  For DH89XXCC_SGMII, a write flush leaves registers of this device trashed (0xFFFFFFFF). Add check for this device.

  Also, after both Port SW Reset and Device Reset case, the platform should wait at least 3ms before reading any registers. Remove this condition since waiting is conditionally executed only for Device Reset.

- **e1000/base: Fix redundant PHY power down for i210.**

  Bit 11 of PHYREG 0 is used to power down PHY. The use of PHYREG 16 is no longer necessary.

- **e1000/base: fix jumbo frame CRC failures.**

  Change the value of register 776.20[11:2] for jumbo mode from 0x1A to 0x1F. This is to enlarge the gap between read and write pointers in the TX FIFO.

- **e1000/base: Fix link flap on 82579.**

  Several customers have reported a link flap issue on 82579. The symptoms are random and intermittent link losses when 82579 is connected to specific switches. the Issue was root caused as an inter-operability problem between the NIC and at least some Broadcom PHYs in the Energy Efficient Ethernet wake mechanism.

  To fix the issue, we are disabling the Phase Locked Loop shutdown in 100M Low Power Idle. This solution will cause an increase of power in 100M EEE link. It may cost an additional 28mW in this specific mode.

- **igb: Fixed IEEE1588 frame identification in I210.**

  Fixed issue where the flag `PKT_RX_IEEE1588_PTP` was not being set in the Intel I210 NIC, as the EtherType in RX descriptor is in bits 8:10 of Packet Type and not in the default bits 0:2.

- **igb: Fixed VF start with PF stopped.**

  VF needs the PF interrupt support initialized even if not started.

- **igb: Fixed VF MAC address when using with DPDK PF.**

  Assign a random MAC address in VF when not assigned by PF.

- **igb: Removed CRC bytes from byte counter statistics.**

- **ixgbe: Fixed issue with X550 DCB.**

  Fixed a DCB issue with x550 where for 8 TCs (Traffic Classes), if a packet with user priority 6 or 7 was injected to the NIC, then the NIC would only put 3 packets into the queue. There was also a similar issue for 4 TCs.

- **ixgbe: Removed burst size restriction of vector RX.**

  Fixed issue where a burst size less than 32 didn't receive anything.

- **ixgbe: Fixed VF start with PF stopped.**

  VF needs the PF interrupt support initialized even if not started.

- **ixgbe: Fixed TX hang when RS distance exceeds HW limit.**

  Fixed an issue where the TX queue can hang when a lot of highly fragmented packets have to be sent. As part of that fix, `tx_rs_thresh` for ixgbe PMD is not allowed to be greater then to 32 to comply with HW restrictions.

- **ixgbe: Fixed rx error statistic counter.**

  Fixed an issue that the rx error counter of ixgbe was not accurate. The mac short packet discard count (mspdc) was added to the counter. Mac local faults and mac remote faults are removed as they do not count packets but errors, and jabber errors were removed as they are already accounted for by the CRC error counter. Finally the XEC (l3 / l4 checksum error) counter was removed due to errata, see commit 256ff05a9cae for details.

- **ixgbe: Removed CRC bytes from byte counter statistics.**

- **i40e: Fixed base driver allocation when not using first numa node.**

  Fixed i40e issue that occurred when a DPDK application didn't initialize ports if memory wasn't available on socket 0.

- **i40e: Fixed maximum of 64 queues per port.**

  Fixed an issue in i40e where it would not support more than 64 queues per port, even though the hardware actually supports it. The real number of queues may vary, as long as the total number of queues used in PF, VFs, VMDq and FD does not exceeds the hardware maximum.

- **i40e: Fixed statistics of packets.**

  Added discarding packets on VSI to the stats and rectify the old statistics.

- **i40e: Fixed issue of not freeing memzone.**

  Fixed an issue of not freeing a memzone in the call to free the memory for adminq DMA.

- **i40e: Removed CRC bytes from byte counter statistics.**
- **mlx: Fixed driver loading.**

  The mlx drivers were unable to load when built as a shared library, due to a missing symbol in the mempool library.

- **mlx4: Performance improvements.**

  Fixed bugs in TX and RX flows that improves mlx4 performance.

- **mlx4: Fixed TX loss after initialization.**
- **mlx4: Fixed scattered TX with too many segments.**
- **mlx4: Fixed memory registration for indirect mbuf data.**
- **vhost: Fixed Qemu shutdown.**

  Fixed issue with libvirt `virsh destroy` not killing the VM.

- **virtio: Fixed crash after changing link state.**

  Fixed IO permission in the interrupt handler.

- **virtio: Fixed crash when releasing queue.**

  Fixed issue when releasing null control queue.

Libraries

- **hash: Fixed memory allocation of Cuckoo Hash key table.**

  Fixed issue where an incorrect Cuckoo Hash key table size could be calculated limiting the size to 4GB.

- **hash: Fixed incorrect lookup if key is all zero.**

  Fixed issue in hash library that occurred if an all zero key was not added to the table and the key was looked up, resulting in an incorrect hit.

- **hash: Fixed thread scaling by reducing contention.**

  Fixed issue in the hash library where, using multiple cores with hardware transactional memory support, thread scaling did not work, due to the global ring that is shared by all cores.

Examples

- **l3fwd: Fixed crash with IPv6.**
- **vhost_xen: Fixed compile error.**

Other

- This release drops compatibility with Linux kernel 2.6.33. The minimum kernel requirement is now 2.6.34.

13.7.3 Known Issues

- Some drivers do not fill in the packet type when receiving. As the l3fwd example application requires this info, the i40e vector driver must be disabled to benefit of the packet type with i40e.
- Some (possibly all) VF drivers (e.g. i40evf) do not handle any PF reset events/requests in the VF driver. This means that the VF driver may not work after a PF reset in the host side. The workaround is to avoid triggering any PF reset events/requests on the host side.
- 100G link report support is missing.
- **Mellanox PMDs (mlx4 & mlx5):**
  - PMDs do not support CONFIG_RTE_BUILD_COMBINE_LIBS and CONFIG_RTE_BUILD_SHARED_LIB simultaneously.
  - There is performance degradation for small packets when the PMD is compiled with `SGE_WR_N = 4` compared to the performance when `SGE_WR_N = 1`. If scattered packets are not used it is recommended to compile the PMD with `SGE_WR_N = 1`.
  - When a Multicast or Broadcast packet is sent to the SR-IOV mlx4 VF, it is returned back to the port.
  - PMDs report "bad" L4 checksum when IP packet is received.
  - mlx5 PMD reports "bad" checksum although the packet has "good" checksum. Will be fixed in upcoming MLNX_OFED release.

13.7.4 API Changes

- The deprecated flow director API is removed. It was replaced by `rte_eth_dev_filter_ctrl()`.
- The `dcb_queue` is renamed to `dcb_tc` in following dcb configuration structures: `rte_eth_dcb_rx_conf`, `rte_eth_dcb_tx_conf`, `rte_eth_vmdq_dcb_conf`, `rte_eth_vmdq_dcb_tx_conf`.
- The `rte_eth_rx_queue_count()` function now returns "int" instead of "uint32_t" to allow the use of negative values as error codes on return.
- The function `rte_eal_pci_close_one()` is removed. It was replaced by `rte_eal_pci_detach()`.
- The deprecated ACL API `ipv4vlan` is removed.
- The deprecated hash function `rte_jhash2()` is removed. It was replaced by `rte_jhash_32b()`.
- The deprecated KNI functions are removed: `rte_kni_create()`, `rte_kni_get_port_id()` and `rte_kni_info_get()`.
- The deprecated ring PMD functions are removed: `rte_eth_ring_pair_create()` and `rte_eth_ring_pair_attach()`.
- The devargs union field `virtual` is renamed to `virt` for C++ compatibility.

13.7.5 ABI Changes

- The EAL and ethdev structures `rte_intr_handle` and `rte_eth_conf` were changed to support RX interrupt. This was already included in 2.1 under the `CONFIG_RTE_NEXT_ABI` #define.
- The ethdev flow director entries for SCTP were changed. This was already included in 2.1 under the `CONFIG_RTE_NEXT_ABI` #define.

- The ethdev flow director structure `rte_eth_fdir_flow_ext` structure was changed. New fields were added to support flow director filtering in VF.
- The size of the ethdev structure `rte_eth_hash_filter_info` is changed by adding a new element `rte_eth_input_set_conf` in a union.
- New fields `rx_desc_lim` and `tx_desc_lim` are added into `rte_eth_dev_info` structure.
- For debug builds, the functions `rte_eth_rx_burst()`, `rte_eth_tx_burst()` `rte_eth_rx_descriptor_done()` and `rte_eth_rx_queue_count()` will no longer be separate functions in the DPDK libraries. Instead, they will only be present in the `rte_ethdev.h` header file.
- The maximum number of queues per port `CONFIG_RTE_MAX_QUEUES_PER_PORT` is increased to 1024.
- The mbuf structure was changed to support the unified packet type. This was already included in 2.1 under the `CONFIG_RTE_NEXT_ABI` #define.
- The dummy malloc library is removed. The content was moved into EAL in 2.1.
- The LPM structure is changed. The deprecated field `mem_location` is removed.
- librte_table LPM: A new parameter to hold the table name will be added to the LPM table parameter structure.
- librte_table hash: The key mask parameter is added to the hash table parameter structure for 8-byte key and 16-byte key extendable bucket and LRU tables.
- librte_port: Macros to access the packet meta-data stored within the packet buffer has been adjusted to cover the packet mbuf structure.
- librte_cfgfile: Allow longer names and values by increasing the constants `CFG_NAME_LEN` and `CFG_VALUE_LEN` to 64 and 256 respectively.
- vhost: a new field enabled is added to the `vhost_virtqueue` structure.
- vhost: a new field `virt_qp_nb` is added to `virtio_net` structure, and the `virtqueue` field is moved to the end of virtio_net structure.
- vhost: a new operation `vring_state_changed` is added to `virtio_net_device_ops` structure.
- vhost: a few spaces are reserved both at `vhost_virtqueue` and `virtio_net` structure for future extension.

13.7.6 Shared Library Versions

The libraries prepended with a plus sign were incremented in this version.

```
+ libethdev.so.2
+ librte_acl.so.2
+ librte_cfgfile.so.2
  librte_cmdline.so.1
  librte_distributor.so.1
+ librte_eal.so.2
+ librte_hash.so.2
  librte_ip_frag.so.1
  librte_ivshmem.so.1
  librte_jobstats.so.1
+ librte_kni.so.2
  librte_kvargs.so.1
+ librte_lpm.so.2
+ librte_mbuf.so.2
  librte_mempool.so.1
  librte_meter.so.1
```

```
+ librte_pipeline.so.2
  librte_pmd_bond.so.1
+ librte_pmd_ring.so.2
+ librte_port.so.2
  librte_power.so.1
  librte_reorder.so.1
  librte_ring.so.1
  librte_sched.so.1
+ librte_table.so.2
  librte_timer.so.1
+ librte_vhost.so.2
```

13.8 DPDK Release 2.1

13.8.1 New Features

- **Enabled cloning of indirect mbufs.**

  This feature removes a limitation of `rte_pktmbuf_attach()` which generated the warning: "mbuf we're attaching to must be direct".

  Now, when attaching to an indirect mbuf it is possible to:

  - Copy all relevant fields (address, length, offload, ...) as before.
  - Get the pointer to the mbuf that embeds the data buffer (direct mbuf), and increase the reference counter.

  When detaching the mbuf, we can now retrieve this direct mbuf as the pointer is determined from the buffer address.

- **Extended packet type support.**

  In previous releases mbuf packet types were indicated by 6 bits in the `ol_flags`. This was not enough for some supported NICs. For example i40e hardware can recognize more than 150 packet types. Not being able to identify these additional packet types limits access to hardware offload capabilities

  So an extended "unified" packet type was added to support all possible PMDs. The 16 bit packet_type in the mbuf structure was changed to 32 bits and used for this purpose.

  To avoid breaking ABI compatibility, the code changes for this feature are enclosed in a `RTE_NEXT_ABI` ifdef. This is enabled by default but can be turned off for ABI compatibility with DPDK R2.0.

- **Reworked memzone to be allocated by malloc and also support freeing.**

  In the memory hierarchy, memsegs are groups of physically contiguous hugepages, memzones are slices of memsegs, and malloc slices memzones into smaller memory chunks.

  This feature modifies `malloc()` so it partitions memsegs instead of memzones. Now memzones allocate their memory from the malloc heap.

  Backward compatibility with API and ABI are maintained.

  This allow memzones, and any other structure based on memzones, for example mempools, to be freed. Currently only the API from freeing memzones is supported.

- **Interrupt mode PMD.**

  This feature introduces a low-latency one-shot RX interrupt into DPDK. It also adds a polling and interrupt mode switch control example.

DPDK userspace interrupt notification and handling mechanism is based on UIO/VFIO with the following limitations:

  - Per queue RX interrupt events are only allowed in VFIO which supports multiple MSI-X vectors.
  - In UIO, the RX interrupt shares the same vector with other interrupts. When the RX interrupt and LSC interrupt are both enabled, only the former is available.
  - RX interrupt is only implemented for the linuxapp target.
  - The feature is only currently enabled for tow PMDs: ixgbe and igb.

- **Packet Framework enhancements.**

  Several enhancements were made to the Packet Framework:

  - A new configuration file syntax has been introduced for IP pipeline applications. Parsing of the configuration file is changed.
  - Implementation of the IP pipeline application is modified to make it more structured and user friendly.
  - Implementation of the command line interface (CLI) for each pipeline type has been moved to the separate compilation unit. Syntax of pipeline CLI commands has been changed.
  - Initialization of IP pipeline is modified to match the new parameters structure.
  - New implementation of pass-through pipeline, firewall pipeline, routing pipeline, and flow classification has been added.
  - Master pipeline with CLI interface has been added.
  - Added extended documentation of the IP Pipeline.

- **Added API for IEEE1588 timestamping.**

  This feature adds an ethdev API to enable, disable and read IEEE1588/802.1AS PTP timestamps from devices that support it. The following functions were added:

  - `rte_eth_timesync_enable()`
  - `rte_eth_timesync_disable()`
  - `rte_eth_timesync_read_rx_timestamp()`
  - `rte_eth_timesync_read_tx_timestamp()`

  The "ieee1588" forwarding mode in testpmd was also refactored to demonstrate the new API.

- **Added multicast address filtering.**

  Added multicast address filtering via a new ethdev function `set_mc_addr_list()`.

  This overcomes a limitation in previous releases where the receipt of multicast packets on a given port could only be enabled by invoking the `rte_eth_allmulticast_enable()` function. This method did not work for VFs in SR-IOV architectures when the host PF driver does not allow these operation on VFs. In such cases, joined multicast addresses had to be added individually to the set of multicast addresses that are filtered by the [VF] port.

- **Added Flow Director extensions.**

  Several Flow Director extensions were added such as:

  - Support for RSS and Flow Director hashes in vector RX.
  - Added Flow Director for L2 payload.

- **Added RSS hash key size query per port.**

  This feature supports querying the RSS hash key size of each port. A new field `hash_key_size` has been added in the `rte_eth_dev_info` struct for storing hash key size in bytes.

- **Added userspace ethtool support.**

  Added userspace ethtool support to provide a familiar interface for applications that manage devices via kernel-space `ethtool_op` and `net_device_op`.

  The initial implementation focuses on operations that can be implemented through existing `netdev` APIs. More operations will be supported in later releases.

- **Updated the ixgbe base driver.**

  The ixgbe base driver was updated with several changes including the following:

  - Added a new 82599 device id.
  - Added new X550 PHY ids.
  - Added SFP+ dual-speed support.
  - Added wait helper for X550 IOSF accesses.
  - Added X550em features.
  - Added X557 PHY LEDs support.
  - Commands for flow director.
  - Issue firmware command when resetting X550em.

  See the git log for full details of the ixgbe/base changes.

- **Added additional hotplug support.**

  Port hotplug support was added to the following PMDs:

  - e1000/igb.
  - ixgbe.
  - i40e.
  - fm10k.
  - ring.
  - bonding.
  - virtio.

  Port hotplug support was added to BSD.

- **Added ixgbe LRO support.**

  Added LRO support for x540 and 82599 devices.

- **Added extended statistics for ixgbe.**

  Implemented `xstats_get()` and `xstats_reset()` in dev_ops for ixgbe to expose detailed error statistics to DPDK applications.

  These will be implemented for other PMDs in later releases.

- **Added proc_info application.**

  Created a new `proc_info` application, by refactoring the existing `dump_cfg` application, to demonstrate the usage of retrieving statistics, and the new extended statistics (see above), for DPDK interfaces.

- **Updated the i40e base driver.**

  The i40e base driver was updated with several changes including the following:

  - Support for building both PF and VF driver together.
  - Support for CEE DCBX on recent firmware versions.
  - Replacement of `i40e_debug_read_register()`.
  - Rework of `i40e_hmc_get_object_va`.
  - Update of shadow RAM read/write functions.
  - Enhancement of polling NVM semaphore.
  - Enhancements on adminq init and sending asq command.
  - Update of get/set LED functions.
  - Addition of AOC phy types to case statement in get_media_type.
  - Support for iSCSI capability.
  - Setting of FLAG_RD when sending driver version to FW.

  See the git log for full details of the i40e/base changes.

- **Added support for port mirroring in i40e.**

  Enabled mirror functionality in the i40e driver.

- **Added support for i40e double VLAN, QinQ, stripping and insertion.**

  Added support to the i40e driver for offloading double VLAN (QinQ) tags to the mbuf header, and inserting double vlan tags by hardware to the packets to be transmitted. Added a new field `vlan_tci_outer` in the `rte_mbuf` struct, and new flags in `ol_flags` to support this feature.

- **Added fm10k promiscuous mode support.**

  Added support for promiscuous/allmulticast enable and disable in the fm10k PF function. VF is not supported yet.

- **Added fm10k jumbo frame support.**

  Added support for jumbo frame less than 15K in both VF and PF functions in the fm10k pmd.

- **Added fm10k mac vlan filtering support.**

  Added support for the fm10k MAC filter, only available in PF. Updated the VLAN filter to add/delete one static entry in the MAC table for each combination of VLAN and MAC address.

- **Added support for the Broadcom bnx2x driver.**

  Added support for the Broadcom NetXtreme II bnx2x driver. It is supported only on Linux 64-bit and disabled by default.

- **Added support for the Chelsio CXGBE driver.**

  Added support for the CXGBE Poll Mode Driver for the Chelsio Terminator 5 series of 10G/40G adapters.

- **Enhanced support for Mellanox ConnectX-3 driver (mlx4).**

  - Support Mellanox OFED 3.0.
  - Improved performance for both RX and TX operations.
  - Better link status information.
  - Outer L3/L4 checksum offload support.

  - Inner L3/L4 checksum offload support for VXLAN.

- **Enabled VMXNET3 vlan filtering.**

  Added support for the VLAN filter functionality of the VMXNET3 interface.

- **Added support for vhost live migration.**

  Added support to allow live migration of vhost. Without this feature, qemu will report the following error: "migrate: Migration disabled: vhost lacks VHOST_F_LOG_ALL feature".

- **Added support for pcap jumbo frames.**

  Extended the PCAP PMD to support jumbo frames for RX and TX.

- **Added support for the TILE-Gx architecture.**

  Added support for the EZchip TILE-Gx family of SoCs.

- **Added hardware memory transactions/lock elision for x86.**

  Added the use of hardware memory transactions (HTM) on fast-path for rwlock and spinlock (a.k.a. lock elision). The methods are implemented for x86 using Restricted Transactional Memory instructions (Intel(r) Transactional Synchronization Extensions). The implementation fall-backs to the normal rwlock if HTM is not available or memory transactions fail. This is not a replacement for all rwlock usages since not all critical sections protected by locks are friendly to HTM. For example, an attempt to perform a HW I/O operation inside a hardware memory transaction always aborts the transaction since the CPU is not able to roll-back should the transaction fail. Therefore, hardware transactional locks are not advised to be used around `rte_eth_rx_burst()` and `rte_eth_tx_burst()` calls.

- **Updated Jenkins Hash function**

  Updated the version of the Jenkins Hash (jhash) function used in DPDK from the 1996 version to the 2006 version. This gives up to 35% better performance, compared to the original one.

  Note, the hashes generated by the updated version differ from the hashes generated by the previous version.

- **Added software implementation of the Toeplitz RSS hash**

  Added a software implementation of the Toeplitz hash function used by RSS. It can be used either for packet distribution on a single queue NIC or for simulating RSS computation on a specific NIC (for example after GRE header de-encapsulation).

- **Replaced the existing hash library with a Cuckoo hash implementation.**

  Replaced the existing hash library with another approach, using the Cuckoo Hash method to resolve collisions (open addressing). This method pushes items from a full bucket when a new entry must be added to it, storing the evicted entry in an alternative location, using a secondary hash function.

  This gives the user the ability to store more entries when a bucket is full, in comparison with the previous implementation.

  The API has not been changed, although new fields have been added in the `rte_hash` structure, which has been changed to internal use only.

  The main change when creating a new table is that the number of entries per bucket is now fixed, so its parameter is ignored now (it is still there to maintain the same parameters structure).

  Also, the maximum burst size in lookup_burst function hash been increased to 64, to improve performance.

- **Optimized KNI RX burst size computation.**

  Optimized KNI RX burst size computation by avoiding checking how many entries are in `kni->rx_q` prior to actually pulling them from the fifo.

- **Added KNI multicast.**

  Enabled adding multicast addresses to KNI interfaces by adding an empty callback for `set_rx_mode` (typically used for setting up hardware) so that the ioctl succeeds. This is the same thing as the Linux tap interface does.

- **Added cmdline polling mode.**

  Added the ability to process console input in the same thread as packet processing by using the `poll()` function.

- **Added VXLAN Tunnel End point sample application.**

  Added a Tunnel End point (TEP) sample application that simulates a VXLAN Tunnel Endpoint (VTEP) termination in DPDK. It is used to demonstrate the offload and filtering capabilities of Intel XL710 10/40 GbE NICsfor VXLAN packets.

- **Enabled combining of the ‘‘-m‘‘ and ‘‘–no-huge‘‘ EAL options.**

  Added option to allow combining of the `-m` and `--no-huge` EAL command line options.

  This allows user application to run as non-root but with higher memory allocations, and removes a constraint on `--no-huge` mode being limited to 64M.

13.8.2 Resolved Issues

- **acl: Fix ambiguity between test rules.**

  Some test rules had equal priority for the same category. That could cause an ambiguity in building the trie and test results.

- **acl: Fix invalid rule wildness calculation for bitmask field type.**

- **acl: Fix matching rule.**

- **acl: Fix unneeded trie splitting for subset of rules.**

  When rebuilding a trie for limited rule-set, don’t try to split the rule-set even further.

- **app/testpmd: Fix crash when port id out of bound.**

  Fixed issues in testpmd where using a port greater than 32 would cause a seg fault.

  Fixes: edab33b1c01d (“app/testpmd: support port hotplug”)

- **app/testpmd: Fix reply to a multicast ICMP request.**

  Set the IP source and destination addresses in the IP header of the ICMP reply.

- **app/testpmd: fix MAC address in ARP reply.**

  Fixed issue where in the `icmpecho` forwarding mode, ARP replies from testpmd contain invalid zero-filled MAC addresses.

  Fixes: 31db4d38de72 (“net: change arp header struct declaration”)

- **app/testpmd: fix default flow control values.**

  Fixes: 422a20a4e62d (“app/testpmd: fix uninitialized flow control variables”)

- **bonding: Fix crash when stopping inactive slave.**

- **bonding: Fix device initialization error handling.**

- **bonding: Fix initial link status of slave.**

  On Fortville NIC, link status change interrupt callback was not executed when slave in bonding was (re-)started.

- **bonding: Fix socket id for LACP slave.**

  Fixes: 46fb43683679 ("bond: add mode 4")

- **bonding: Fix device initialization error handling.**

- **cmdline: Fix small memory leak.**

  A function in `cmdline.c` had a return that did not free the buf properly.

- **config: Enable same drivers options for Linux and BSD.**

  Enabled vector ixgbe and i40e bulk alloc for BSD as it is already done for Linux.

  Fixes: 304caba12643 ("config: fix bsd options") Fixes: 0ff3324da2eb ("ixgbe: rework vector pmd following mbuf changes")

- **devargs: Fix crash on failure.**

  This problem occurred when passing an invalid PCI id to the blacklist API in devargs.

- **e1000/i40e: Fix descriptor done flag with odd address.**

- **e1000/igb: fix ieee1588 timestamping initialization.**

  Fixed issue with e1000 ieee1588 timestamp initialization. On initialization the IEEE1588 functions read the system time to set their timestamp. However, on some 1G NICs, for example, i350, system time is disabled by default and the IEEE1588 timestamp was always 0.

- **eal/bsd: Fix inappropriate header guards.**

- **eal/bsd: Fix virtio on FreeBSD.**

  Closing the `/dev/io` fd caused a SIGBUS in inb/outb instructions as the process lost the IOPL privileges once the fd is closed.

  Fixes: 8a312224bcde ("eal/bsd: fix fd leak")

- **eal/linux: Fix comments on vfio MSI.**

- **eal/linux: Fix irq handling with igb_uio.**

  Fixed an issue where the the introduction of `uio_pci_generic` broke interrupt handling with igb_uio.

  Fixes: c112df6875a5 ("eal/linux: toggle interrupt for uio_pci_generic")

- **eal/linux: Fix numa node detection.**

- **eal/linux: Fix socket value for undetermined numa node.**

  Sets zero as the default value of pci device numa_node if the socket could not be determined. This provides the same default value as FreeBSD which has no NUMA support, and makes the return value of `rte_eth_dev_socket_id()` be consistent with the API description.

- **eal/ppc: Fix cpu cycle count for little endian.**

  On IBM POWER8 PPC64 little endian architecture, the definition of tsc union will be different. This fix enables the right output from `rte_rdtsc()`.

- **ethdev: Fix check of threshold for TX freeing.**

  Fixed issue where the parameter to `tx_free_thresh` was not consistent between the drivers.

- **ethdev: Fix crash if malloc of user callback fails.**

  If `rte_zmalloc()` failed in `rte_eth_dev_callback_register` then the NULL pointer would be dereferenced.

- **ethdev: Fix illegal port access.**

  To obtain a detachable flag, `pci_drv` is accessed in `rte_eth_dev_is_detachable()`. However `pci_drv` is only valid if port is enabled. Fixed by checking `rte_eth_dev_is_valid_port()` first.

- **ethdev: Make tables const.**
- **ethdev: Rename and extend the mirror type.**
- **examples/distributor: Fix debug macro.**

  The macro to turn on additional debug output when the app was compiled with `-DDEBUG` was broken.

  Fixes: 07db4a975094 ("examples/distributor: new sample app")

- **examples/kni: Fix crash on exit.**
- **examples/vhost: Fix build with debug enabled.**

  Fixes: 72ec8d77ac68 ("examples/vhost: rework duplicated code")

- **fm10k: Fix RETA table initialization.**

  The fm10k driver has 128 RETA entries in 32 registers, but it only initialized the first 32 when doing multiple RX queue configurations. This fix initializes all 128 entries.

- **fm10k: Fix RX buffer size.**
- **fm10k: Fix TX multi-segment frame.**
- **fm10k: Fix TX queue cleaning after start error.**
- **fm10k: Fix Tx queue cleaning after start error.**
- **fm10k: Fix default mac/vlan in switch.**
- **fm10k: Fix interrupt fault handling.**
- **fm10k: Fix jumbo frame issue.**
- **fm10k: Fix mac/vlan filtering.**
- **fm10k: Fix maximum VF number.**
- **fm10k: Fix maximum queue number for VF.**

  Both PF and VF shared code in function `fm10k_stats_get()`. The function worked with PF, but had problems with VF since it has less queues than PF.

  Fixes: a6061d9e7075 ("fm10k: register PF driver")

- **fm10k: Fix queue disabling.**
- **fm10k: Fix switch synchronization.**
- **i40e/base: Fix error handling of NVM state update.**
- **i40e/base: Fix hardware port number for pass-through.**
- **i40e/base: Rework virtual address retrieval for lan queue.**
- **i40e/base: Update LED blinking.**
- **i40e/base: Workaround for PHY type with firmware < 4.4.**
- **i40e: Disable setting of PHY configuration.**
- **i40e: Fix SCTP flow director.**

- **i40e: Fix check of descriptor done flag.**

  Fixes: 4861cde46116 ("i40e: new poll mode driver") Fixes: 05999aab4ca6 ("i40e: add or delete flow director")
- **i40e: Fix condition to get VMDQ info.**
- **i40e: Fix registers access from big endian CPU.**
- **i40evf: Clear command when error occurs.**
- **i40evf: Fix RSS with less RX queues than TX queues.**
- **i40evf: Fix crash when setup TX queues.**
- **i40evf: Fix jumbo frame support.**
- **i40evf: Fix offload capability flags.**

  Added checksum offload capability flags which have already been supported for a long time.
- **ivshmem: Fix crash in corner case.**

  Fixed issues where depending on the configured segments it was possible to hit a segmentation fault as a result of decrementing an unsigned index with value 0.

  Fixes: 40b966a211ab ("ivshmem: library changes for mmaping using ivshmem")
- **ixgbe/base: Fix SFP probing.**
- **ixgbe/base: Fix TX pending clearing.**
- **ixgbe/base: Fix X550 CS4227 address.**
- **ixgbe/base: Fix X550 PCIe master disabling.**
- **ixgbe/base: Fix X550 check.**
- **ixgbe/base: Fix X550 init early return.**
- **ixgbe/base: Fix X550 link speed.**
- **ixgbe/base: Fix X550em CS4227 speed mode.**
- **ixgbe/base: Fix X550em SFP+ link stability.**
- **ixgbe/base: Fix X550em UniPHY link configuration.**
- **ixgbe/base: Fix X550em flow control for KR backplane.**
- **ixgbe/base: Fix X550em flow control to be KR only.**
- **ixgbe/base: Fix X550em link setup without SFP.**
- **ixgbe/base: Fix X550em mux after MAC reset.**

  Fixes: d2e72774e58c ("ixgbe/base: support X550")
- **ixgbe/base: Fix bus type overwrite.**
- **ixgbe/base: Fix init handling of X550em link down.**
- **ixgbe/base: Fix lan id before first i2c access.**
- **ixgbe/base: Fix mac type checks.**
- **ixgbe/base: Fix tunneled UDP and TCP frames in flow director.**

- **ixgbe: Check mbuf refcnt when clearing a ring.**

  The function to clear the TX ring when a port was being closed, e.g. on exit in testpmd, was not checking the mbuf refcnt before freeing it. Since the function in the vector driver to clear the ring after TX does not setting the pointer to NULL post-free, this caused crashes if mbuf debugging was turned on.

- **ixgbe: Fix RX with buffer address not word aligned.**

  Niantic HW expects the Header Buffer Address in the RXD must be word aligned.

- **ixgbe: Fix RX with buffer address not word aligned.**

- **ixgbe: Fix Rx queue reset.**

  Fix to reset vector related RX queue fields to their initial values.

  Fixes: c95584dc2b18 ("ixgbe: new vectorized functions for Rx/Tx")

- **ixgbe: Fix TSO in IPv6.**

  When TSO was used with IPv6, the generated frames were incorrect. The L4 frame was OK, but the length field of IPv6 header was not populated correctly.

- **ixgbe: Fix X550 flow director check.**

- **ixgbe: Fix check for split packets.**

  The check for split packets to be reassembled in the vector ixgbe PMD was incorrectly only checking the first 16 elements of the array instead of all 32.

  Fixes: cf4b4708a88a ("ixgbe: improve slow-path perf with vector scattered Rx")

- **ixgbe: Fix data access on big endian cpu.**

- **ixgbe: Fix flow director flexbytes offset.**

  Fixes: d54a9888267c ("ixgbe: support flexpayload configuration of flow director")

- **ixgbe: Fix number of segments with vector scattered Rx.**

  Fixes: cf4b4708a88a (ixgbe: improve slow-path perf with vector scattered Rx)

- **ixgbe: Fix offload config option name.**

  The RX_OLFLAGS option was renamed from DISABLE to ENABLE in the driver code and Linux config. It is now renamed also in the BSD config and documentation.

  Fixes: 359f106a69a9 ("ixgbe: prefer enabling olflags rather than not disabling")

- **ixgbe: Fix release queue mbufs.**

  The calculations of what mbufs were valid in the RX and TX queues were incorrect when freeing the mbufs for the vector PMD. This led to crashes due to invalid reference counts when mbuf debugging was turned on, and possibly other more subtle problems (such as mbufs being freed when in use) in other cases.

  Fixes: c95584dc2b18 ("ixgbe: new vectorized functions for Rx/Tx")

- **ixgbe: Move PMD specific fields out of base driver.**

  Move `rx_bulk_alloc_allowed` and `rx_vec_allowed` from `ixgbe_hw` to `ixgbe_adapter`.

  Fixes: 01fa1d6215fa ("ixgbe: unify Rx setup")

- **ixgbe: Rename TX queue release function.**

- **ixgbevf: Fix RX function selection.**

  The logic to select ixgbe the VF RX function is different than the PF.

- **ixgbevf: Fix link status for PF up/down events.**
- **kni: Fix RX loop limit.**

  Loop processing packets dequeued from rx_q was using the number of packets requested, not how many it actually received.

- **kni: Fix ioctl in containers, like Docker.**
- **kni: Fix multicast ioctl handling.**
- **log: Fix crash after log_history dump.**
- **lpm: Fix big endian support.**
- **lpm: Fix depth small entry add.**
- **mbuf: Fix cloning with private mbuf data.**

  Added a new `priv_size` field in mbuf structure that should be initialized at mbuf pool creation. This field contains the size of the application private data in mbufs.

  Introduced new static inline functions `rte_mbuf_from_indirect()` and `rte_mbuf_to_baddr()` to replace the existing macros, which take the private size into account when attaching and detaching mbufs.

- **mbuf: Fix data room size calculation in pool init.**

  Deduct the mbuf data room size from `mempool->elt_size` and `priv_size`, instead of using an hardcoded value that is not related to the real buffer size.

  To use `rte_pktmbuf_pool_init()`, the user can either:

  – Give a NULL parameter to rte_pktmbuf_pool_init(): in this case, the private size is assumed to be 0, and the room size is `mp->elt_size - sizeof(struct rte_mbuf)`.

  – Give the `rte_pktmbuf_pool_private` filled with appropriate data_room_size and priv_size values.

- **mbuf: Fix init when private size is not zero.**

  Allow the user to use the default `rte_pktmbuf_init()` function even if the mbuf private size is not 0.

- **mempool: Add structure for object headers.**

  Each object stored in mempools are prefixed by a header, allowing for instance to retrieve the mempool pointer from the object. When debug is enabled, a cookie is also added in this header that helps to detect corruptions and double-frees.

  Introduced a structure that materializes the content of this header, and will simplify future patches adding things in this header.

- **mempool: Fix pages computation to determine number of objects.**
- **mempool: Fix returned value after counting objects.**

  Fixes: 148f963fb532 (“xen: core library changes”)

- **mlx4: Avoid requesting TX completion events to improve performance.**

  Instead of requesting a completion event for each TX burst, request it on a fixed schedule once every MLX4_PMD_TX_PER_COMP_REQ (currently 64) packets to improve performance.

- **mlx4: Fix compilation as a shared library and on 32 bit platforms.**
- **mlx4: Fix possible crash on scattered mbuf allocation failure.**

  Fixes issue where failing to allocate a segment, `mlx4_rx_burst_sp()` could call `rte_pktmbuf_free()` on an incomplete scattered mbuf whose next pointer in the last segment is not set.

- **mlx4: Fix support for multiple vlan filters.**

  This fixes the "Multiple RX VLAN filters can be configured, but only the first one works" bug.

- **pcap: Fix storage of name and type in queues.**

  pcap_rx_queue/pcap_tx_queue should store it's own copy of name/type values, not the pointer to temporary allocated space.

- **pci: Fix memory leaks and needless increment of map address.**
- **pci: Fix uio mapping differences between linux and bsd.**
- **port: Fix unaligned access to metadata.**

  Fix RTE_MBUF_METADATA macros to allow for unaligned accesses to meta-data fields.

- **ring: Fix return of new port id on creation.**
- **timer: Fix race condition.**

  Eliminate problematic race condition in `rte_timer_manage()` that can lead to corruption of per-lcore pending-lists (implemented as skip-lists).

- **vfio: Fix overflow of BAR region offset and size.**

  Fixes: 90a1633b2347 ("eal/Linux: allow to map BARs with MSI-X tables")

- **vhost: Fix enqueue/dequeue to handle chained vring descriptors.**
- **vhost: Fix race for connection fd.**
- **vhost: Fix virtio freeze due to missed interrupt.**
- **virtio: Fix crash if CQ is not negotiated.**

  Fix NULL dereference if virtio control queue is not negotiated.

- **virtio: Fix ring size negotiation.**

  Negotiate the virtio ring size. The host may allow for very large rings but application may only want a smaller ring. Conversely, if the number of descriptors requested exceeds the virtio host queue size, then just silently use the smaller host size.

  This fixes issues with virtio in non-QEMU environments. For example Google Compute Engine allows up to 16K elements in ring.

- **vmxnet3: Fix link state handling.**

13.8.3 Known Issues

- When running the `vmdq` sample or `vhost` sample applications with the Intel(R) XL710 (i40e) NIC, the configuration option `CONFIG_RTE_MAX_QUEUES_PER_PORT` should be increased from 256 to 1024.
- VM power manager may not work on systems with more than 64 cores.

13.8.4 API Changes

- The order that user supplied RX and TX callbacks are called in has been changed to the order that they were added (fifo) in line with end-user expectations. The previous calling order was the reverse of this (lifo) and was counter intuitive for users. The actual API is unchanged.

13.8.5 ABI Changes

- The `rte_hash` structure has been changed to internal use only.

13.9 DPDK Release 2.0

13.9.1 New Features

- Poll-mode driver support for an early release of the PCIE host interface of the Intel(R) Ethernet Switch FM10000.
  - Basic Rx/Tx functions for PF/VF
  - Interrupt handling support for PF/VF
  - Per queue start/stop functions for PF/VF
  - Support Mailbox handling between PF/VF and PF/Switch Manager
  - Receive Side Scaling (RSS) for PF/VF
  - Scatter receive function for PF/VF
  - Reta update/query for PF/VF
  - VLAN filter set for PF
  - Link status query for PF/VF

Note: The software is intended to run on pre-release hardware and may contain unknown or unresolved defects or issues related to functionality and performance. The poll mode driver is also pre-release and will be updated to a released version post hardware and base driver release. Should the official hardware release be made between DPDK releases an updated poll-mode driver will be made available.

- Link Bonding
  - Support for adaptive load balancing (mode 6) to the link bonding library.
  - Support for registration of link status change callbacks with link bonding devices.
  - Support for slaves devices which do not support link status change interrupts in the link bonding library via a link status polling mechanism.
- PCI Hotplug with NULL PMD sample application
- ABI versioning
- x32 ABI
- Non-EAL Thread Support
- Multi-pthread Support
- Re-order Library
- ACL for AVX2
- Architecture Independent CRC Hash
- uio_pci_generic Support
- KNI Optimizations

- Vhost-user support
- Virtio (link, vlan, mac, port IO, perf)
- IXGBE-VF RSS
- RX/TX Callbacks
- Unified Flow Types
- Indirect Attached MBUF Flag
- Use default port configuration in TestPMD
- Tunnel offloading in TestPMD
- Poll Mode Driver - 40 GbE Controllers (librte_pmd_i40e)
  - Support for Flow Director
  - Support for ethertype filter
  - Support RSS in VF
  - Support configuring redirection table with different size from 1GbE and 10 GbE
  - 128/512 entries of 40GbE PF
  - 64 entries of 40GbE VF
  - Support configuring hash functions
  - Support for VXLAN packet on Intel® 40GbE Controllers
- Poll Mode Driver for Mellanox ConnectX-3 EN adapters (mlx4)

Note: This PMD is only available for Linux and is disabled by default due to external dependencies (libibverbs and libmlx4). Please refer to the NIC drivers guide for more information.

- Packet Distributor Sample Application
- Job Stats library and Sample Application.
- Enhanced Jenkins hash (jhash) library

Note: The hash values returned by the new jhash library are different from the ones returned by the previous library.

13.10 DPDK Release 1.8

13.10.1 New Features

- Link Bonding
  - Support for 802.3ad link aggregation (mode 4) and transmit load balancing (mode 5) to the link bonding library.
  - Support for registration of link status change callbacks with link bonding devices.
  - Support for slaves devices which do not support link status change interrupts in the link bonding library via a link status polling mechanism.

- Poll Mode Driver - 40 GbE Controllers (librte_pmd_i40e)
  - Support for Flow Director
  - Support for ethertype filter
  - Support RSS in VF
  - Support configuring redirection table with different size from 1GbE and 10 GbE
  - 128/512 entries of 40GbE PF
  - 64 entries of 40GbE VF
  - Support configuring hash functions
  - Support for VXLAN packet on Intel 40GbE Controllers
- Packet Distributor Sample Application

13.11 Supported Operating Systems

The following Linux distributions were successfully used to compiler or run DPDK.

- FreeBSD 10
- Fedora release 20
- Ubuntu 14.04 LTS
- Wind River Linux 6
- Red Hat Enterprise Linux 6.5
- SUSE Enterprise Linux 11 SP3

These distributions may need additional packages that are not installed by default, or a specific kernel. Refer to the *Linux guide* and *FreeBSD guide* for details.

13.12 Known Issues and Limitations in Legacy Releases

This section describes known issues with the DPDK software that aren't covered in the version specific release notes sections.

13.12.1 Unit Test for Link Bonding may fail at test_tlb_tx_burst()

Description: Unit tests will fail in `test_tlb_tx_burst()` function with error for uneven distribution of packets.

Implication: Unit test link_bonding_autotest will fail.

Resolution/Workaround: There is no workaround available.

Affected Environment/Platform: Fedora 20.

Driver/Module: Link Bonding.

13.12.2 Pause Frame Forwarding does not work properly on igb

Description: For igb devices rte_eth_flow_ctrl_set does not work as expected. Pause frames are always forwarded on igb, regardless of the `RFCE`, `MPMCF` and `DPF` registers.

Implication: Pause frames will never be rejected by the host on 1G NICs and they will always be forwarded.

Resolution/Workaround: There is no workaround available.

Affected Environment/Platform: All.

Driver/Module: Poll Mode Driver (PMD).

13.12.3 In packets provided by the PMD, some flags are missing

Description: In packets provided by the PMD, some flags are missing. The application does not have access to information provided by the hardware (packet is broadcast, packet is multicast, packet is IPv4 and so on).

Implication: The `ol_flags` field in the `rte_mbuf` structure is not correct and should not be used.

Resolution/Workaround: The application has to parse the Ethernet header itself to get the information, which is slower.

Affected Environment/Platform: All.

Driver/Module: Poll Mode Driver (PMD).

13.12.4 The rte_malloc library is not fully implemented

Description: The `rte_malloc` library is not fully implemented.

Implication: All debugging features of rte_malloc library described in architecture documentation are not yet implemented.

Resolution/Workaround: No workaround available.

Affected Environment/Platform: All.

Driver/Module: `rte_malloc`.

13.12.5 HPET reading is slow

Description: Reading the HPET chip is slow.

Implication: An application that calls `rte_get_hpet_cycles()` or `rte_timer_manage()` runs slower.

Resolution/Workaround: The application should not call these functions too often in the main loop. An alternative is to use the TSC register through `rte_rdtsc()` which is faster, but specific to an lcore and is a cycle reference, not a time reference.

Affected Environment/Platform: All.

Driver/Module: Environment Abstraction Layer (EAL).

13.12.6 HPET timers do not work on the Osage customer reference platform

Description: HPET timers do not work on the Osage customer reference platform which includes an Intel® Xeon® processor 5500 series processor) using the released BIOS from Intel.

Implication: On Osage boards, the implementation of the `rte_delay_us()` function must be changed to not use the HPET timer.

Resolution/Workaround: This can be addressed by building the system with the `CONFIG_RTE_LIBEAL_USE_HPET=n` configuration option or by using the `--no-hpet` EAL option.

Affected Environment/Platform: The Osage customer reference platform. Other vendor platforms with Intel® Xeon® processor 5500 series processors should work correctly, provided the BIOS supports HPET.

Driver/Module: `lib/librte_eal/common/include/rte_cycles.h`

13.12.7 Not all variants of supported NIC types have been used in testing

Description: The supported network interface cards can come in a number of variants with different device ID's. Not all of these variants have been tested with the DPDK.

The NIC device identifiers used during testing:

- Intel® Ethernet Controller XL710 for 40GbE QSFP+ [8086:1584]
- Intel® Ethernet Controller XL710 for 40GbE QSFP+ [8086:1583]
- Intel® Ethernet Controller X710 for 10GbE SFP+ [8086:1572]
- Intel® 82576 Gigabit Ethernet Controller [8086:10c9]
- Intel® 82576 Quad Copper Gigabit Ethernet Controller [8086:10e8]
- Intel® 82580 Dual Copper Gigabit Ethernet Controller [8086:150e]
- Intel® I350 Quad Copper Gigabit Ethernet Controller [8086:1521]
- Intel® 82599 Dual Fibre 10 Gigabit Ethernet Controller [8086:10fb]
- Intel® Ethernet Server Adapter X520-T2 [8086: 151c]
- Intel® Ethernet Controller X540-T2 [8086:1528]
- Intel® 82574L Gigabit Network Connection [8086:10d3]
- Emulated Intel® 82540EM Gigabit Ethernet Controller [8086:100e]
- Emulated Intel® 82545EM Gigabit Ethernet Controller [8086:100f]
- Intel® Ethernet Server Adapter X520-4 [8086:154a]
- Intel® Ethernet Controller I210 [8086:1533]

Implication: Risk of issues with untested variants.

Resolution/Workaround: Use tested NIC variants. For those supported Ethernet controllers, additional device IDs may be added to the software if required.

Affected Environment/Platform: All.

Driver/Module: Poll-mode drivers

13.12.8 Multi-process sample app requires exact memory mapping

Description: The multi-process example application assumes that it is possible to map the hugepage memory to the same virtual addresses in client and server applications. Occasionally, very rarely with 64-bit, this does not occur and a client application will fail on startup. The Linux "address-space layout randomization" security feature can sometimes cause this to occur.

Implication: A multi-process client application fails to initialize.

Resolution/Workaround: See the "Multi-process Limitations" section in the DPDK Programmer's Guide for more information.

Affected Environment/Platform: All.

Driver/Module: Multi-process example application

13.12.9 Packets are not sent by the 1 GbE/10 GbE SR-IOV driver when the source MAC is not the MAC assigned to the VF NIC

Description: The 1 GbE/10 GbE SR-IOV driver can only send packets when the Ethernet header's source MAC address is the same as that of the VF NIC. The reason for this is that the Linux `ixgbe` driver module in the host OS has its anti-spoofing feature enabled.

Implication: Packets sent using the 1 GbE/10 GbE SR-IOV driver must have the source MAC address correctly set to that of the VF NIC. Packets with other source address values are dropped by the NIC if the application attempts to transmit them.

Resolution/Workaround: Configure the Ethernet source address in each packet to match that of the VF NIC.

Affected Environment/Platform: All.

Driver/Module: 1 GbE/10 GbE VF Poll Mode Driver (PMD).

13.12.10 SR-IOV drivers do not fully implement the rte_ethdev API

Description: The SR-IOV drivers only supports the following rte_ethdev API functions:

- rte_eth_dev_configure()
- rte_eth_tx_queue_setup()
- rte_eth_rx_queue_setup()
- rte_eth_dev_info_get()
- rte_eth_dev_start()
- rte_eth_tx_burst()
- rte_eth_rx_burst()
- rte_eth_dev_stop()
- rte_eth_stats_get()
- rte_eth_stats_reset()
- rte_eth_link_get()
- rte_eth_link_get_no_wait()

Implication: Calling an unsupported function will result in an application error.

Resolution/Workaround: Do not use other rte_ethdev API functions in applications that use the SR-IOV drivers.

Affected Environment/Platform: All.

Driver/Module: VF Poll Mode Driver (PMD).

13.12.11 PMD does not work with –no-huge EAL command line parameter

Description: Currently, the DPDK does not store any information about memory allocated by `malloc()` (for example, NUMA node, physical address), hence PMD drivers do not work when the ``--no-huge` command line parameter is supplied to EAL.

Implication: Sending and receiving data with PMD will not work.

Resolution/Workaround: Use huge page memory or use VFIO to map devices.

Affected Environment/Platform: Systems running the DPDK on Linux

Driver/Module: Poll Mode Driver (PMD).

13.12.12 Some hardware off-load functions are not supported by the VF Driver

Description: Currently, configuration of the following items is not supported by the VF driver:

- IP/UDP/TCP checksum offload
- Jumbo Frame Receipt
- HW Strip CRC

Implication: Any configuration for these items in the VF register will be ignored. The behavior is dependent on the current PF setting.

Resolution/Workaround: For the PF (Physical Function) status on which the VF driver depends, there is an option item under PMD in the config file. For others, the VF will keep the same behavior as PF setting.

Affected Environment/Platform: All.

Driver/Module: VF (SR-IOV) Poll Mode Driver (PMD).

13.12.13 Kernel crash on IGB port unbinding

Description: Kernel crash may occur when unbinding 1G ports from the igb_uio driver, on 2.6.3x kernels such as shipped with Fedora 14.

Implication: Kernel crash occurs.

Resolution/Workaround: Use newer kernels or do not unbind ports.

Affected Environment/Platform: 2.6.3x kernels such as shipped with Fedora 14

Driver/Module: IGB Poll Mode Driver (PMD).

13.12.14 Twinpond and Ironpond NICs do not report link status correctly

Description: Twin Pond/Iron Pond NICs do not bring the physical link down when shutting down the port.

Implication: The link is reported as up even after issuing `shutdown` command unless the cable is physically disconnected.

Resolution/Workaround: None.

Affected Environment/Platform: Twin Pond and Iron Pond NICs

Driver/Module: Poll Mode Driver (PMD).

13.12.15 Discrepancies between statistics reported by different NICs

Description: Gigabit Ethernet devices from Intel include CRC bytes when calculating packet reception statistics regardless of hardware CRC stripping state, while 10-Gigabit Ethernet devices from Intel do so only when hardware CRC stripping is disabled.

Implication: There may be a discrepancy in how different NICs display packet reception statistics.

Resolution/Workaround: None

Affected Environment/Platform: All.

Driver/Module: Poll Mode Driver (PMD).

13.12.16 Error reported opening files on DPDK initialization

Description: On DPDK application startup, errors may be reported when opening files as part of the initialization process. This occurs if a large number, for example, 500 or more, or if hugepages are used, due to the per-process limit on the number of open files.

Implication: The DPDK application may fail to run.

Resolution/Workaround: If using 2 MB hugepages, consider switching to a fewer number of 1 GB pages. Alternatively, use the `ulimit` command to increase the number of files which can be opened by a process.

Affected Environment/Platform: All.

Driver/Module: Environment Abstraction Layer (EAL).

13.12.17 Intel® QuickAssist Technology sample application does not work on a 32-bit OS on Shumway

Description: The Intel® Communications Chipset 89xx Series device does not fully support NUMA on a 32-bit OS. Consequently, the sample application cannot work properly on Shumway, since it requires NUMA on both nodes.

Implication: The sample application cannot work in 32-bit mode with emulated NUMA, on multi-socket boards.

Resolution/Workaround: There is no workaround available.

Affected Environment/Platform: Shumway

Driver/Module: All.

13.12.18 Differences in how different Intel NICs handle maximum packet length for jumbo frame

Description: 10 Gigabit Ethernet devices from Intel do not take VLAN tags into account when calculating packet size while Gigabit Ethernet devices do so for jumbo frames.

Implication: When receiving packets with VLAN tags, the actual maximum size of useful payload that Intel Gigabit Ethernet devices are able to receive is 4 bytes (or 8 bytes in the case of packets with extended VLAN tags) less than that of Intel 10 Gigabit Ethernet devices.

Resolution/Workaround: Increase the configured maximum packet size when using Intel Gigabit Ethernet devices.

Affected Environment/Platform: All.

Driver/Module: Poll Mode Driver (PMD).

13.12.19 Binding PCI devices to igb_uio fails on Linux kernel 3.9 when more than one device is used

Description: A known bug in the uio driver included in Linux kernel version 3.9 prevents more than one PCI device to be bound to the igb_uio driver.

Implication: The Poll Mode Driver (PMD) will crash on initialization.

Resolution/Workaround: Use earlier or later kernel versions, or apply the following patch.

Affected Environment/Platform: Linux systems with kernel version 3.9

Driver/Module: igb_uio module

13.12.20 GCC might generate Intel® AVX instructions for processors without Intel® AVX support

Description: When compiling DPDK (and any DPDK app), gcc may generate Intel® AVX instructions, even when the processor does not support Intel® AVX.

Implication: Any DPDK app might crash while starting up.

Resolution/Workaround: Either compile using icc or set `EXTRA_CFLAGS='-O3'` prior to compilation.

Affected Environment/Platform: Platforms which processor does not support Intel® AVX.

Driver/Module: Environment Abstraction Layer (EAL).

13.12.21 Ethertype filter could receive other packets (non-assigned) in Niantic

Description: On Intel® Ethernet Controller 82599EB When Ethertype filter (priority enable) was set, unmatched packets also could be received on the assigned queue, such as ARP packets without 802.1q tags or with the user priority not equal to set value. Launch the testpmd by disabling RSS and with multiply queues, then add the ethertype filter like the following and then start forwarding:

```
add_ethertype_filter 0 ethertype 0x0806 priority enable 3 queue 2 index 1
```

When sending ARP packets without 802.1q tag and with user priority as non-3 by tester, all the ARP packets can be received on the assigned queue.

Implication: The user priority comparing in Ethertype filter cannot work probably. It is a NIC's issue due to the following: "In fact, ETQF.UP is not functional, and the information will be added in errata of 82599 and X540."

Resolution/Workaround: None

Affected Environment/Platform: All.

Driver/Module: Poll Mode Driver (PMD).

13.12.22 Cannot set link speed on Intel® 40G Ethernet controller

Description: On Intel® 40G Ethernet Controller you cannot set the link to specific speed.

Implication: The link speed cannot be changed forcibly, though it can be configured by application.

Resolution/Workaround: None

Affected Environment/Platform: All.

Driver/Module: Poll Mode Driver (PMD).

13.12.23 Devices bound to igb_uio with VT-d enabled do not work on Linux kernel 3.15-3.17

Description: When VT-d is enabled (`iommu=pt intel_iommu=on`), devices are 1:1 mapped. In the Linux kernel unbinding devices from drivers removes that mapping which result in IOMMU errors. Introduced in Linux kernel 3.15 commit, solved in Linux kernel 3.18 commit.

Implication: Devices will not be allowed to access memory, resulting in following kernel errors:

```
dmar: DRHD: handling fault status reg 2
dmar: DMAR:[DMA Read] Request device [02:00.0] fault addr a0c58000
DMAR:[fault reason 02] Present bit in context entry is clear
```

Resolution/Workaround: Use earlier or later kernel versions, or avoid driver binding on boot by blacklisting the driver modules. I.e., in the case of `ixgbe`, we can pass the kernel command line option: `modprobe.blacklist=ixgbe`. This way we do not need to unbind the device to bind it to igb_uio.

Affected Environment/Platform: Linux systems with kernel versions 3.15 to 3.17.

Driver/Module: `igb_uio` module.

13.12.24 VM power manager may not work on systems with more than 64 cores

Description: When using VM power manager on a system with more than 64 cores, VM(s) should not use cores 64 or higher.

Implication: VM power manager should not be used with VM(s) that are using cores 64 or above.

Resolution/Workaround: Do not use cores 64 or above.

Affected Environment/Platform: Platforms with more than 64 cores.

Driver/Module: VM power manager application.

13.12.25 DPDK may not build on some Intel CPUs using clang < 3.7.0

Description: When compiling DPDK with an earlier version than 3.7.0 of clang, CPU flags are not detected on some Intel platforms such as Intel Broadwell/Skylake (and possibly future CPUs), and therefore compilation fails due to missing intrinsics.

Implication: DPDK will not build when using a clang version < 3.7.0.

Resolution/Workaround: Use clang 3.7.0 or higher, or gcc.

Affected Environment/Platform: Platforms with Intel Broadwell/Skylake using an old clang version.

Driver/Module: Environment Abstraction Layer (EAL).

13.12.26 The last EAL argument is replaced by the program name in argv[]

Description: The last EAL argument is replaced by program name in `argv[]` after `eal_parse_args` is called. This is the intended behavior but it causes the pointer to the last EAL argument to be lost.

Implication: If the last EAL argument in `argv[]` is generated by a malloc function, changing it will cause memory issues when freeing the argument.

Resolution/Workaround: An application should not consider the value in `argv[]` as unchanged.

Affected Environment/Platform: ALL.

Driver/Module: Environment Abstraction Layer (EAL).

13.12.27 I40e VF may not receive packets in the promiscuous mode

Description: Promiscuous mode is not supported by the DPDK i40e VF driver when using the i40e Linux kernel driver as host driver.

Implication: The i40e VF does not receive packets when the destination MAC address is unknown.

Resolution/Workaround: Use a explicit destination MAC address that matches the VF.

Affected Environment/Platform: All.

Driver/Module: Poll Mode Driver (PMD).

13.12.28 uio pci generic module bind failed in X710/XL710/XXV710

Description: The `uio_pci_generic` module is not supported by XL710, since the errata of XL710 states that the Interrupt Status bit is not implemented. The errata is the item #71 from the xl710 controller spec. The hw limitation is the same as other X710/XXV710 NICs.

Implication: When use `--bind=uio_pci_generic`, the `uio_pci_generic` module probes device and check the Interrupt Status bit. Since it is not supported by X710/XL710/XXV710, it return a *failed* value. The statement that these products don't support INTx masking, is indicated in the related linux kernel commit.

Resolution/Workaround: Do not bind the `uio_pci_generic` module in X710/XL710/XXV710 NICs.

Affected Environment/Platform: All.

Driver/Module: Poll Mode Driver (PMD).

13.12.29 virtio tx_burst() function cannot do TSO on shared packets

Description: The standard TX function of virtio driver does not manage shared packets properly when doing TSO. These packets should be read-only but the driver modifies them.

When doing TSO, the virtio standard expects that the L4 checksum is set to the pseudo header checksum in the packet data, which is different than the DPDK API. The driver patches the L4 checksum to conform to the virtio standard, but this solution is invalid when dealing with shared packets (clones), because the packet data should not be modified.

Implication: In this situation, the shared data will be modified by the driver, potentially causing race conditions with the other users of the mbuf data.

Resolution/Workaround: The workaround in the application is to ensure that the network headers in the packet data are not shared.

Affected Environment/Platform: Virtual machines running a virtio driver.

Driver/Module: Poll Mode Driver (PMD).

13.13 ABI and API Deprecation

See the *guidelines document for details of the ABI policy*. API and ABI deprecation notices are to be posted here.

13.13.1 Deprecation Notices

- eal: the following functions are deprecated starting from 17.05 and will be removed in 17.08:
  - `rte_set_log_level`, replaced by `rte_log_set_global_level`
  - `rte_get_log_level`, replaced by `rte_log_get_global_level`
  - `rte_set_log_type`, replaced by `rte_log_set_level`
  - `rte_get_log_type`, replaced by `rte_log_get_level`
- igb_uio: iomem mapping and sysfs files created for iomem and ioport in igb_uio will be removed, because we are able to detect these from what Linux has exposed, like the way we have done with uio-pci-generic. This change targets release 17.05.
- vfio: Some functions are planned to be exported outside librte_eal in 17.05. VFIO APIs like `vfio_setup_device`, `vfio_get_group_fd` can be used by subsystem other than EAL/PCI. For that, these need to be exported symbols. Such APIs are planned to be renamed according to `rte_*` naming convention and exported from librte_eal.
- The VDEV subsystem will be converted as driver of the new bus model. It will imply some EAL API changes in 17.05.
- `eth_driver` is planned to be removed in 17.05. This currently serves as a placeholder for PMDs to register themselves. Changes for `rte_bus` will provide a way to handle device initialization currently being done in `eth_driver`. Similarly, `rte_pci_driver` is planned to be removed from `rte_cryptodev_driver` in 17.05.
- ethdev: An API change is planned for 17.05 for the function `_rte_eth_dev_callback_process`. In 17.05 the function will return an `int` instead of `void` and a fourth parameter `void *ret_param` will be added.
- ethdev: for 17.05 it is planned to deprecate the following nine rte_eth_dev_* functions and move them into the ixgbe PMD:

  `rte_eth_dev_bypass_init`, `rte_eth_dev_bypass_state_set`, `rte_eth_dev_bypass_state_show`, `rte_eth_dev_bypass_event_store`, `rte_eth_dev_bypass_event_show`, `rte_eth_dev_wd_timeout_store`, `rte_eth_dev_bypass_wd_timeout_show`, `rte_eth_dev_bypass_ver_show`, `rte_eth_dev_bypass_wd_reset`.

  The following fields will be removed from `struct eth_dev_ops`:

  `bypass_init_t`, `bypass_state_set_t`, `bypass_state_show_t`, `bypass_event_set_t`, `bypass_event_show_t`, `bypass_wd_timeout_set_t`, `bypass_wd_timeout_show_t`, `bypass_ver_show_t`, `bypass_wd_reset_t`.

  The functions will be renamed to the following, and moved to the `ixgbe` PMD:

`rte_pmd_ixgbe_bypass_init`, `rte_pmd_ixgbe_bypass_state_set`, `rte_pmd_ixgbe_bypass_state_show`, `rte_pmd_ixgbe_bypass_event_set`, `rte_pmd_ixgbe_bypass_event_show`, `rte_pmd_ixgbe_bypass_wd_timeout_set`, `rte_pmd_ixgbe_bypass_wd_timeout_show`, `rte_pmd_ixgbe_bypass_ver_show`, `rte_pmd_ixgbe_bypass_wd_reset`.

- The mbuf flags PKT_RX_VLAN_PKT and PKT_RX_QINQ_PKT are deprecated and are respectively replaced by PKT_RX_VLAN_STRIPPED and PKT_RX_QINQ_STRIPPED, that are better described. The old flags and their behavior will be kept until 17.02 and will be removed in 17.05.
- ethdev: the legacy filter API, including `rte_eth_dev_filter_supported()`, `rte_eth_dev_filter_ctrl()` as well as filter types MACVLAN, ETHERTYPE, FLEXIBLE, SYN, NTUPLE, TUNNEL, FDIR, HASH and L2_TUNNEL, is superseded by the generic flow API (rte_flow) in PMDs that implement the latter. Target release for removal of the legacy API will be defined once most PMDs have switched to rte_flow.
- crypto/scheduler: the following two functions are deprecated starting from 17.05 and will be removed in 17.08:
  - `rte_crpytodev_scheduler_mode_get`, replaced by `rte_cryptodev_scheduler_mode_get`
  - `rte_crpytodev_scheduler_mode_set`, replaced by `rte_cryptodev_scheduler_mode_set`

CHAPTER 14

FAQ

This document contains some Frequently Asked Questions that arise when working with DPDK.

14.1 What does "EAL: map_all_hugepages(): open failed: Permission denied Cannot init memory" mean?

This is most likely due to the test application not being run with sudo to promote the user to a superuser. Alternatively, applications can also be run as regular user. For more information, please refer to *DPDK Getting Started Guide*.

14.2 If I want to change the number of TLB Hugepages allocated, how do I remove the original pages allocated?

The number of pages allocated can be seen by executing the following command:

```
grep Huge /proc/meminfo
```

Once all the pages are mmapped by an application, they stay that way. If you start a test application with less than the maximum, then you have free pages. When you stop and restart the test application, it looks to see if the pages are available in the `/dev/huge` directory and mmaps them. If you look in the directory, you will see `n` number of 2M pages files. If you specified 1024, you will see 1024 page files. These are then placed in memory segments to get contiguous memory.

If you need to change the number of pages, it is easier to first remove the pages. The usertools/dpdk-setup.sh script provides an option to do this. See the "Quick Start Setup Script" section in the *DPDK Getting Started Guide* for more information.

14.3 If I execute "l2fwd -l 0-3 -m 64 -n 3 – -p 3", I get the following output, indicating that there are no socket 0 hugepages to allocate the mbuf and ring structures to?

I have set up a total of 1024 Hugepages (that is, allocated 512 2M pages to each NUMA node).

The -m command line parameter does not guarantee that huge pages will be reserved on specific sockets. Therefore, allocated huge pages may not be on socket 0. To request memory to be reserved on a specific socket, please use the –socket-mem command-line parameter instead of -m.

14.4 I am running a 32-bit DPDK application on a NUMA system, and sometimes the application initializes fine but cannot allocate memory. Why is that happening?

32-bit applications have limitations in terms of how much virtual memory is available, hence the number of hugepages they are able to allocate is also limited (1 GB per page size). If your system has a lot (>1 GB per page size) of hugepage memory, not all of it will be allocated. Due to hugepages typically being allocated on a local NUMA node, the hugepages allocation the application gets during the initialization depends on which NUMA node it is running on (the EAL does not affinitize cores until much later in the initialization process). Sometimes, the Linux OS runs the DPDK application on a core that is located on a different NUMA node from DPDK master core and therefore all the hugepages are allocated on the wrong socket.

To avoid this scenario, either lower the amount of hugepage memory available to 1 GB per page size (or less), or run the application with taskset affinitizing the application to a would-be master core.

For example, if your EAL coremask is 0xff0, the master core will usually be the first core in the coremask (0x10); this is what you have to supply to taskset:

```
taskset 0x10 ./l2fwd -l 4-11 -n 2
```

In this way, the hugepages have a greater chance of being allocated to the correct socket. Additionally, a `--socket-mem` option could be used to ensure the availability of memory for each socket, so that if hugepages were allocated on the wrong socket, the application simply will not start.

14.5 On application startup, there is a lot of EAL information printed. Is there any way to reduce this?

Yes, the option `--log-level=` accepts one of these numbers:

```
#define RTE_LOG_EMERG 1U    /* System is unusable. */
#define RTE_LOG_ALERT 2U    /* Action must be taken immediately. */
#define RTE_LOG_CRIT 3U     /* Critical conditions. */
#define RTE_LOG_ERR 4U      /* Error conditions. */
#define RTE_LOG_WARNING 5U  /* Warning conditions. */
#define RTE_LOG_NOTICE 6U   /* Normal but significant condition. */
#define RTE_LOG_INFO 7U     /* Informational. */
#define RTE_LOG_DEBUG 8U    /* Debug-level messages. */
```

It is also possible to change the default level at compile time with `CONFIG_RTE_LOG_LEVEL`.

14.6 How can I tune my network application to achieve lower latency?

Traditionally, there is a trade-off between throughput and latency. An application can be tuned to achieve a high throughput, but the end-to-end latency of an average packet typically increases as a result. Similarly, the application can be tuned to have, on average, a low end-to-end latency at the cost of lower throughput.

To achieve higher throughput, the DPDK attempts to aggregate the cost of processing each packet individually by processing packets in bursts. Using the testpmd application as an example, the "burst" size can be set on the command line to a value of 16 (also the default value). This allows the application to request 16 packets at a time from the PMD. The testpmd application then immediately attempts to transmit all the packets that were received, in this case, all 16 packets. The packets are not transmitted until the tail pointer is updated on the corresponding TX queue of the network port. This behavior is desirable when tuning for high throughput because the cost of tail pointer updates to both the RX and TX queues can be spread across 16 packets, effectively hiding the relatively slow MMIO cost of writing to the PCIe* device.

However, this is not very desirable when tuning for low latency, because the first packet that was received must also wait for the other 15 packets to be received. It cannot be transmitted until the other 15 packets have also been processed because the NIC will not know to transmit the packets until the TX tail pointer has been updated, which is not done until all 16 packets have been processed for transmission.

To consistently achieve low latency even under heavy system load, the application developer should avoid processing packets in bunches. The testpmd application can be configured from the command line to use a burst value of 1. This allows a single packet to be processed at a time, providing lower latency, but with the added cost of lower throughput.

14.7 Without NUMA enabled, my network throughput is low, why?

I have a dual Intel® Xeon® E5645 processors 2.40 GHz with four Intel® 82599 10 Gigabit Ethernet NICs. Using eight logical cores on each processor with RSS set to distribute network load from two 10 GbE interfaces to the cores on each processor.

Without NUMA enabled, memory is allocated from both sockets, since memory is interleaved. Therefore, each 64B chunk is interleaved across both memory domains.

The first 64B chunk is mapped to node 0, the second 64B chunk is mapped to node 1, the third to node 0, the fourth to node 1. If you allocated 256B, you would get memory that looks like this:

```
256B buffer
Offset 0x00 - Node 0
Offset 0x40 - Node 1
Offset 0x80 - Node 0
Offset 0xc0 - Node 1
```

Therefore, packet buffers and descriptor rings are allocated from both memory domains, thus incurring QPI bandwidth accessing the other memory and much higher latency. For best performance with NUMA disabled, only one socket should be populated.

14.8 I am getting errors about not being able to open files. Why?

As the DPDK operates, it opens a lot of files, which can result in reaching the open files limits, which is set using the ulimit command or in the limits.conf file. This is especially true when using a large number (>512) of 2 MB huge pages. Please increase the open file limit if your application is not able to open files. This can be done either by issuing a ulimit command or editing the limits.conf file. Please consult Linux manpages for usage information.

14.9 VF driver for IXGBE devices cannot be initialized

Some versions of Linux IXGBE driver do not assign a random MAC address to VF devices at initialization. In this case, this has to be done manually on the VM host, using the following command:

```
ip link set <interface> vf <VF function> mac <MAC address>
```

where <interface> being the interface providing the virtual functions for example, eth0, <VF function> being the virtual function number, for example 0, and <MAC address> being the desired MAC address.

14.10 Is it safe to add an entry to the hash table while running?

Currently the table implementation is not a thread safe implementation and assumes that locking between threads and processes is handled by the user's application. This is likely to be supported in future releases.

14.11 What is the purpose of setting iommu=pt?

DPDK uses a 1:1 mapping and does not support IOMMU. IOMMU allows for simpler VM physical address translation. The second role of IOMMU is to allow protection from unwanted memory access by an unsafe device that has DMA privileges. Unfortunately, the protection comes with an extremely high performance cost for high speed NICs.

Setting `iommu=pt` disables IOMMU support for the hypervisor.

14.12 When trying to send packets from an application to itself, meaning smac==dmac, using Intel(R) 82599 VF packets are lost.

Check on register `LLE(PFVMTXSSW[n])`, which allows an individual pool to send traffic and have it looped back to itself.

14.13 Can I split packet RX to use DPDK and have an application's higher order functions continue using Linux pthread?

The DPDK's lcore threads are Linux pthreads bound onto specific cores. Configure the DPDK to do work on the same cores and run the application's other work on other cores using the DPDK's "coremask" setting to specify which cores it should launch itself on.

14.14 Is it possible to exchange data between DPDK processes and regular userspace processes via some shared memory or IPC mechanism?

Yes - DPDK processes are regular Linux/BSD processes, and can use all OS provided IPC mechanisms.

14.15 Can the multiple queues in Intel(R) I350 be used with DPDK?

I350 has RSS support and 8 queue pairs can be used in RSS mode. It should work with multi-queue DPDK applications using RSS.

14.16 How can hugepage-backed memory be shared among multiple processes?

See the Primary and Secondary examples in the *multi-process sample application*.