DPDK

DATA PLANE DEVELOPMENT KIT

Sample Applications User Guides

Release 20.02.1

May 18, 2020

CHAPTER

ONE

INTRODUCTION TO THE DPDK SAMPLE APPLICATIONS

The DPDK Sample Applications are small standalone applications which demonstrate various features of DPDK. They can be considered as a cookbook of DPDK features. Users interested in getting started with DPDK can take the applications, try out the features, and then extend them to fit their needs.

1.1 Running Sample Applications

Some sample applications may have their own command-line parameters described in their respective guides, however all of them also share the same EAL parameters. Please refer to EAL parameters (Linux) or EAL parameters (FreeBSD) for a list of available EAL command-line options.

1.2 The DPDK Sample Applications

There are many sample applications available in the examples directory of DPDK. These examples range from simple to reasonably complex but most are designed to demonstrate one particular feature of DPDK. Some of the more interesting examples are highlighted below.

- *Hello World*: As with most introductions to a programming framework a good place to start is with the Hello World application. The Hello World example sets up the DPDK Environment Abstraction Layer (EAL), and prints a simple "Hello World" message to each of the DPDK enabled cores. This application doesn't do any packet forwarding but it is a good way to test if the DPDK environment is compiled and set up properly.
- *Basic Forwarding/Skeleton Application*: The Basic Forwarding/Skeleton contains the minimum amount of code required to enable basic packet forwarding with DPDK. This allows you to test if your network interfaces are working with DPDK.
- *Network Layer 2 forwarding*: The Network Layer 2 forwarding, or `l2fwd` application does forwarding based on Ethernet MAC addresses like a simple switch.
- *Network Layer 2 forwarding*: The Network Layer 2 forwarding, or `l2fwd-event` application does forwarding based on Ethernet MAC addresses like a simple switch. It demonstrates usage of poll and event mode IO mechanism under a single application.
- *Network Layer 3 forwarding*: The Network Layer3 forwarding, or `l3fwd` application does forwarding based on Internet Protocol, IPv4 or IPv6 like a simple router.
- *Hardware packet copying*: The Hardware packet copying, or `ioatfwd` application demonstrates how to use IOAT rawdev driver for copying packets between two threads.

- *Packet Distributor*: The Packet Distributor demonstrates how to distribute packets arriving on an Rx port to different cores for processing and transmission.
- *Multi-Process Application*: The multi-process application shows how two DPDK processes can work together using queues and memory pools to share information.
- *RX/TX callbacks 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 calculates the latency of a packet between RX (packet arrival) and TX (packet transmission) by adding callbacks to the RX and TX packet processing functions.
- *IPsec Security Gateway*: The IPsec Security Gateway application is minimal example of something closer to a real world example. This is also a good example of an application using the DPDK Cryptodev framework.
- *Precision Time Protocol (PTP) client*: The PTP client is another minimal implementation of a real world application. In this case the application is a PTP client that communicates with a PTP master clock to synchronize time on a Network Interface Card (NIC) using the IEEE1588 protocol.
- *Quality of Service (QoS) Scheduler*: The QoS Scheduler application demonstrates the use of DPDK to provide QoS scheduling.

There are many more examples shown in the following chapters. Each of the documented sample applications show how to compile, configure and run the application as well as explaining the main functionality of the code.

CHAPTER

TWO

COMPILING THE SAMPLE APPLICATIONS

This section explains how to compile the DPDK sample applications.

2.1 To compile all the sample applications

Set the path to DPDK source code if its not set:

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

Go to DPDK source:

```
cd $RTE_SDK
```

Build DPDK:

```
make defconfig
make
```

Build the sample applications:

```
export RTE_TARGET=build
make -C examples
```

For other possible `RTE_TARGET` values and additional information on compiling see Compiling DPDK on Linux or Compiling DPDK on FreeBSD. Applications are output to: `$RTE_SDK/examples/app-dir/build` or `$RTE_SDK/examples/app-dir/$RTE_TARGET`.

In the example above the compiled application is written to the `build` subdirectory. To have the applications written to a different location, the `O=/path/to/build/directory` option may be specified in the make command.

```
make O=/tmp
```

To build the applications for debugging use the `DEBUG` option. This option adds some extra flags, disables compiler optimizations and sets verbose output.

```
make DEBUG=1
```

2.2 To compile a single application

Set the path to DPDK source code:

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

Go to DPDK source:

```
cd $RTE_SDK
```

Build DPDK:

```
make defconfig
make
```

Go to the sample application directory. Unless otherwise specified the sample applications are located in `$RTE_SDK/examples/`.

Build the application:

```
export RTE_TARGET=build
make
```

2.3 To cross compile the sample application(s)

For cross compiling the sample application(s), please append 'CROSS=$(CROSS_COMPILER_PREFIX)' to the 'make' command. In example of AARCH64 cross compiling:

```
export RTE_TARGET=build
export RTE_SDK=/path/to/rte_sdk
make -C examples CROSS=aarch64-linux-gnu-
       or
make CROSS=aarch64-linux-gnu-
```

CHAPTER

THREE

COMMAND LINE SAMPLE APPLICATION

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

3.1 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 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*

The application is located in the `cmd_line` sub-directory.

3.3 Running the Application

To run the application in linux 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.

3.4 Explanation

The following sections provide some explanation of the code.

3.4.1 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.

3.4.2 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
{
   /* ... */
}

cmdline_parse_token_string_t cmd_obj_action = TOKEN_STRING_INITIALIZER(struct cmd_obj_del_show_

parse_token_obj_list_t cmd_obj_obj = TOKEN_OBJ_LIST_INITIALIZER(struct cmd_obj_del_show_result,

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.

CHAPTER

FOUR

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.

4.1 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `ethtool` sub-directory.

4.2 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.

4.3 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 follows:

- `drvinfo`: Print driver info
- `eeprom`: Dump EEPROM to file
- `module-eeprom`: Dump plugin module 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

4.4 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.

4.4.1 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.

4.4.2 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.

4.5 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_module_info()`

- rte_ethtool_get_module_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()

CHAPTER

FIVE

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.

5.1 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `helloworld` sub-directory.

5.2 Running the Application

To run the example in a linux 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.

5.3 Explanation

The following sections provide some explanation of code.

5.3.1 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 Linux environment). The argc and argv arguments are provided to the rte_eal_init() function. The value returned is the number of parsed arguments.

5.3.2 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.

CHAPTER

SIX

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.

6.1 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `skeleton` sub-directory.

6.2 Running the Application

To run the example in a `linux` 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.

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*.

6.3.1 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:

```
RTE_ETH_FOREACH_DEV(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.

6.3.2 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(uint16_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 rte_ether_addr addr;
    int retval;
    uint16_t q;

    if (!rte_eth_dev_is_valid_port(port))
        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. */
        retval = rte_eth_promiscuous_enable(port);
        if (retval != 0)
            return retval;

        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 = RTE_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:

```
retval = rte_eth_promiscuous_enable(port);
```

6.3.3 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)
{
    uint16_t port;

    /*
     * Check that the port is on the same NUMA node as the polling thread
     * for best performance.
     */
    RTE_ETH_FOREACH_DEV(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.
         */
        RTE_ETH_FOREACH_DEV(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 (;;) {
    RTE_ETH_FOREACH_DEV(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`.

CHAPTER

SEVEN

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.

If hardware timestamping is supported by the NIC, the sample application will also display the average latency since the packet was timestamped in hardware, on top of the latency since the packet was received and processed by the RX callback.

7.1 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `rxtx_callbacks` sub-directory.

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
```

7.2 Running the Application

To run the example in a `linux` environment:

```
./build/rxtx_callbacks -l 1 -n 4 -- [-t]
```

Use -t to enable hardware timestamping. If not supported by the NIC, an error will be displayed.

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

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.

7.3.1 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:

7.3.2 The Port Initialization Function

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

```
static inline int
port_init(uint16_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 rte_ether_addr addr;
    int retval;
    uint16_t q;

    /* 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. */
    retval = rte_eth_promiscuous_enable(port);
    if (retval != 0)
        return retval;

    /* 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.

7.3.3 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(uint16_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).

7.3.4 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(uint16_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.

CHAPTER

EIGHT

FLOW CLASSIFY SAMPLE APPLICATION

The Flow Classify sample application is based on the simple *skeleton* example of a forwarding application.

It is intended as a demonstration of the basic components of a DPDK forwarding application which uses the Flow Classify library API's.

Please refer to the ../prog_guide/flow_classify_lib for more information.

8.1 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `flow_classify` sub-directory.

8.2 Running the Application

To run the example in a `linux` environment:

```
cd ~/dpdk/examples/flow_classify
./build/flow_classify -c 4 -n 4 -- --rule_ipv4="../ipv4_rules_file.txt"
```

Please refer to the *DPDK Getting Started Guide*, section ../linux_gsg/build_sample_apps for general information on running applications and the Environment Abstraction Layer (EAL) options.

8.3 Sample ipv4_rules_file.txt

```
#file format:
#src_ip/masklen dst_ip/masklen src_port : mask dst_port : mask proto/mask priority
#
2.2.2.3/24 2.2.2.7/24 32 : 0xffff 33 : 0xffff 17/0xff 0
9.9.9.3/24 9.9.9.7/24 32 : 0xffff 33 : 0xffff 17/0xff 1
9.9.9.3/24 9.9.9.7/24 32 : 0xffff 33 : 0xffff 6/0xff 2
9.9.8.3/24 9.9.8.7/24 32 : 0xffff 33 : 0xffff 6/0xff 3
6.7.8.9/24 2.3.4.5/24 32 : 0x0000 33 : 0x0000 132/0xff 4
```

8.4 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*.

8.4.1 ACL field definitions for the IPv4 5 tuple rule

The following field definitions are used when creating the ACL table during initialisation of the `Flow Classify` application..

```
 enum {
     PROTO_FIELD_IPV4,
     SRC_FIELD_IPV4,
     DST_FIELD_IPV4,
     SRCP_FIELD_IPV4,
     DSTP_FIELD_IPV4,
     NUM_FIELDS_IPV4
};

enum {
    PROTO_INPUT_IPV4,
    SRC_INPUT_IPV4,
    DST_INPUT_IPV4,
    SRCP_DESTP_INPUT_IPV4
};

static struct rte_acl_field_def ipv4_defs[NUM_FIELDS_IPV4] = {
    /* first input field - always one byte long. */
    {
        .type = RTE_ACL_FIELD_TYPE_BITMASK,
        .size = sizeof(uint8_t),
        .field_index = PROTO_FIELD_IPV4,
        .input_index = PROTO_INPUT_IPV4,
        .offset = sizeof(struct rte_ether_hdr) +
            offsetof(struct rte_ipv4_hdr, next_proto_id),
    },
    /* next input field (IPv4 source address) - 4 consecutive bytes. */
    {
        /* rte_flow uses a bit mask for IPv4 addresses */
        .type = RTE_ACL_FIELD_TYPE_BITMASK,
        .size = sizeof(uint32_t),
        .field_index = SRC_FIELD_IPV4,
        .input_index = SRC_INPUT_IPV4,
        .offset = sizeof(struct rte_ether_hdr) +
            offsetof(struct rte_ipv4_hdr, src_addr),
    },
    /* next input field (IPv4 destination address) - 4 consecutive bytes. */
    {
        /* rte_flow uses a bit mask for IPv4 addresses */
        .type = RTE_ACL_FIELD_TYPE_BITMASK,
        .size = sizeof(uint32_t),
        .field_index = DST_FIELD_IPV4,
        .input_index = DST_INPUT_IPV4,
        .offset = sizeof(struct rte_ether_hdr) +
            offsetof(struct rte_ipv4_hdr, dst_addr),
    },
    /*
     * Next 2 fields (src & dst ports) form 4 consecutive bytes.
     * They share the same input index.
     */
    {
        /* rte_flow uses a bit mask for protocol ports */
        .type = RTE_ACL_FIELD_TYPE_BITMASK,
```

```
        .size = sizeof(uint16_t),
        .field_index = SRCP_FIELD_IPV4,
        .input_index = SRCP_DESTP_INPUT_IPV4,
        .offset = sizeof(struct rte_ether_hdr) +
            sizeof(struct rte_ipv4_hdr) +
            offsetof(struct rte_tcp_hdr, src_port),
    },
    {
         /* rte_flow uses a bit mask for protocol ports */
         .type = RTE_ACL_FIELD_TYPE_BITMASK,
         .size = sizeof(uint16_t),
         .field_index = DSTP_FIELD_IPV4,
         .input_index = SRCP_DESTP_INPUT_IPV4,
         .offset = sizeof(struct rte_ether_hdr) +
             sizeof(struct rte_ipv4_hdr) +
             offsetof(struct rte_tcp_hdr, dst_port),
    },
};
```

8.4.2 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");
```

It then parses the flow_classify application arguments

```
ret = parse_args(argc, argv);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Invalid flow_classify parameters\n");
```

The `main()` function 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:

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

The `main()` function creates the `flow classifier object` and adds an `ACL table` to the flow classifier.

```
struct flow_classifier {
    struct rte_flow_classifier *cls;
};

struct flow_classifier_acl {
    struct flow_classifier cls;
} __rte_cache_aligned;

/* Memory allocation */
size = RTE_CACHE_LINE_ROUNDUP(sizeof(struct flow_classifier_acl));
cls_app = rte_zmalloc(NULL, size, RTE_CACHE_LINE_SIZE);
if (cls_app == NULL)
    rte_exit(EXIT_FAILURE, "Cannot allocate classifier memory\n");

cls_params.name = "flow_classifier";
cls_params.socket_id = socket_id;

cls_app->cls = rte_flow_classifier_create(&cls_params);
if (cls_app->cls == NULL) {
    rte_free(cls_app);
    rte_exit(EXIT_FAILURE, "Cannot create classifier\n");
}

/* initialise ACL table params */
table_acl_params.name = "table_acl_ipv4_5tuple";
table_acl_params.n_rule_fields = RTE_DIM(ipv4_defs);
table_acl_params.n_rules = FLOW_CLASSIFY_MAX_RULE_NUM;
memcpy(table_acl_params.field_format, ipv4_defs, sizeof(ipv4_defs));

/* initialise table create params */
cls_table_params.ops = &rte_table_acl_ops,
cls_table_params.arg_create = &table_acl_params,
cls_table_params.type = RTE_FLOW_CLASSIFY_TABLE_ACL_IP4_5TUPLE;

ret = rte_flow_classify_table_create(cls_app->cls, &cls_table_params);
if (ret) {
    rte_flow_classifier_free(cls_app->cls);
    rte_free(cls);
    rte_exit(EXIT_FAILURE, "Failed to create classifier table\n");
}
```

It then reads the ipv4_rules_file.txt file and initialises the parameters for the `rte_flow_classify_table_entry_add` API. This API adds a rule to the ACL table.

```
if (add_rules(parm_config.rule_ipv4_name)) {
    rte_flow_classifier_free(cls_app->cls);
    rte_free(cls_app);
    rte_exit(EXIT_FAILURE, "Failed to add rules\n");
}
```

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(cls_app);
```

The `lcore_main()` function is explained below.

8.4.3 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 rte_ether_addr addr;
    int retval;
    uint16_t q;

    /* 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;

    /* Display the port MAC address. */
    retval = rte_eth_macaddr_get(port, &addr);
    if (retval < 0)
        return retval;
    printf("Port %u MAC: %02" PRIx8 " %02" PRIx8 " %02" PRIx8
           " %02" PRIx8 " %02" PRIx8 " %02" PRIx8 "\n",
           port,
           addr.addr_bytes[0], addr.addr_bytes[1],
           addr.addr_bytes[2], addr.addr_bytes[3],
           addr.addr_bytes[4], addr.addr_bytes[5]);

    /* Enable RX in promiscuous mode for the Ethernet device. */
    retval = rte_eth_promiscuous_enable(port);
    if (retval != 0)
            return retval;

    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 = RTE_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:

```
retval = rte_eth_promiscuous_enable(port);
```

8.4.4 The Add Rules function

The `add_rules` function reads the `ipv4_rules_file.txt` file and calls the `add_classify_rule` function which calls the `rte_flow_classify_table_entry_add` API.

```
static int
add_rules(const char *rule_path)
{
    FILE *fh;
    char buff[LINE_MAX];
    unsigned int i = 0;
    unsigned int total_num = 0;
    struct rte_eth_ntuple_filter ntuple_filter;

    fh = fopen(rule_path, "rb");
    if (fh == NULL)
        rte_exit(EXIT_FAILURE, "%s: Open %s failed\n", __func__,
                 rule_path);

    fseek(fh, 0, SEEK_SET);

    i = 0;
    while (fgets(buff, LINE_MAX, fh) != NULL) {
        i++;

        if (is_bypass_line(buff))
            continue;

        if (total_num >= FLOW_CLASSIFY_MAX_RULE_NUM - 1) {
            printf("\nINFO: classify rule capacity %d reached\n",
                   total_num);
            break;
        }

        if (parse_ipv4_5tuple_rule(buff, &ntuple_filter) != 0)
            rte_exit(EXIT_FAILURE,
                     "%s Line %u: parse rules error\n",
                     rule_path, i);

        if (add_classify_rule(&ntuple_filter) != 0)
            rte_exit(EXIT_FAILURE, "add rule error\n");

        total_num++;
    }

    fclose(fh);
```

```
    return 0;
}
```

8.4.5 The Lcore Main function

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

```
/* flow classify data */
static int num_classify_rules;
static struct rte_flow_classify_rule *rules[MAX_NUM_CLASSIFY];
static struct rte_flow_classify_ipv4_5tuple_stats ntuple_stats;
static struct rte_flow_classify_stats classify_stats = {
        .stats = (void *)&ntuple_stats
};

static __attribute__((noreturn)) void
lcore_main(cls_app)
{
    uint16_t port;

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

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

    /* 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.
         */
        RTE_ETH_FOREACH_DEV(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;

            for (i = 0; i < MAX_NUM_CLASSIFY; i++) {
                if (rules[i]) {
                    ret = rte_flow_classifier_query(
                        cls_app->cls,
                        bufs, nb_rx, rules[i],
```

```
                        &classify_stats);
                    if (ret)
                        printf(
                            "rule [%d] query failed ret [%d]\n\n",
                            i, ret);
                    else {
                        printf(
                            "rule[%d] count=%"PRIu64"\n",
                            i, ntuple_stats.counter1);

                        printf("proto = %d\n",
                            ntuple_stats.ipv4_5tuple.proto);
                    }
                }
            }

            /* 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 (;;) {
    RTE_ETH_FOREACH_DEV(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`.

CHAPTER

NINE

BASIC RTE FLOW FILTERING SAMPLE APPLICATION

The Basic RTE flow filtering sample application is a simple example of a creating a RTE flow rule.

It is intended as a demonstration of the basic components RTE flow rules.

9.1 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/flow_filtering
```

Set the target, for example:

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

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

Build the application as follows:

```
make
```

9.2 Running the Application

To run the example in a `linux` environment:

```
./build/flow -l 1 -n 1
```

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

9.3 Explanation

The example is built from 2 files, `main.c` which holds the example logic and `flow_blocks.c` that holds the implementation for building the flow rule.

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*.

9.3.1 The Main Function

The `main()` function located in `main.c` file performs the initialization and runs the main loop function.

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_pktmbuf_pool_create("mbuf_pool", 4096, 128, 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 `init_port()` function which is explained in the next section:

```
init_port();
```

Once the initialization is complete, we set the flow rule using the following code:

```
/* create flow for send packet with */
flow = generate_ipv4_flow(port_id, selected_queue,
                          SRC_IP, EMPTY_MASK,
                          DEST_IP, FULL_MASK, &error);
if (!flow) {
       printf("Flow can't be created %d message: %s\n",
                    error.type,
                    error.message ? error.message : "(no stated reason)");
       rte_exit(EXIT_FAILURE, "error in creating flow");
}
```

In the last part the application is ready to launch the `main_loop()` function. Which is explained below.

```
main_loop();
```

9.3.2 The Port Initialization Function

The main functional part of the port initialization used in the flow filtering application is shown below:

```
init_port(void)
{
        int ret;
        uint16_t i;
        struct rte_eth_conf port_conf = {
                .rxmode = {
                        .split_hdr_size = 0,
                        },
                .txmode = {
                        .offloads =
                                DEV_TX_OFFLOAD_VLAN_INSERT |
                                DEV_TX_OFFLOAD_IPV4_CKSUM  |
                                DEV_TX_OFFLOAD_UDP_CKSUM   |
                                DEV_TX_OFFLOAD_TCP_CKSUM   |
                                DEV_TX_OFFLOAD_SCTP_CKSUM  |
```

```
                        DEV_TX_OFFLOAD_TCP_TSO,
        },
};
struct rte_eth_txconf txq_conf;
struct rte_eth_rxconf rxq_conf;
struct rte_eth_dev_info dev_info;

printf(":: initializing port: %d\n", port_id);
ret = rte_eth_dev_configure(port_id,
        nr_queues, nr_queues, &port_conf);
if (ret < 0) {
        rte_exit(EXIT_FAILURE,
                ":: cannot configure device: err=%d, port=%u\n",
                ret, port_id);
}

rte_eth_dev_info_get(port_id, &dev_info);
rxq_conf = dev_info.default_rxconf;
rxq_conf.offloads = port_conf.rxmode.offloads;
/* only set Rx queues: something we care only so far */
for (i = 0; i < nr_queues; i++) {
        ret = rte_eth_rx_queue_setup(port_id, i, 512,
                rte_eth_dev_socket_id(port_id),
                &rxq_conf,
                mbuf_pool);
        if (ret < 0) {
                rte_exit(EXIT_FAILURE,
                        ":: Rx queue setup failed: err=%d, port=%u\n",
                        ret, port_id);
        }
}

txq_conf = dev_info.default_txconf;
txq_conf.offloads = port_conf.txmode.offloads;

for (i = 0; i < nr_queues; i++) {
        ret = rte_eth_tx_queue_setup(port_id, i, 512,
                rte_eth_dev_socket_id(port_id),
                &txq_conf);
        if (ret < 0) {
                rte_exit(EXIT_FAILURE,
                        ":: Tx queue setup failed: err=%d, port=%u\n",
                        ret, port_id);
        }
}

ret = rte_eth_promiscuous_enable(port_id);
if (ret != 0) {
        rte_exit(EXIT_FAILURE,
                ":: cannot enable promiscuous mode: err=%d, port=%u\n",
                ret, port_id);
}

ret = rte_eth_dev_start(port_id);
if (ret < 0) {
        rte_exit(EXIT_FAILURE,
                "rte_eth_dev_start:err=%d, port=%u\n",
                ret, port_id);
}

assert_link_status();

printf(":: initializing port: %d done\n", port_id);
```

```
}
```

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

```
struct rte_eth_conf port_conf = {
        .rxmode = {
                .split_hdr_size = 0,
                },
        .txmode = {
                .offloads =
                        DEV_TX_OFFLOAD_VLAN_INSERT |
                        DEV_TX_OFFLOAD_IPV4_CKSUM  |
                        DEV_TX_OFFLOAD_UDP_CKSUM   |
                        DEV_TX_OFFLOAD_TCP_CKSUM   |
                        DEV_TX_OFFLOAD_SCTP_CKSUM  |
                        DEV_TX_OFFLOAD_TCP_TSO,
                },
        };

ret = rte_eth_dev_configure(port_id, nr_queues, nr_queues, &port_conf);
if (ret < 0) {
     rte_exit(EXIT_FAILURE,
              ":: cannot configure device: err=%d, port=%u\n",
              ret, port_id);
}
rte_eth_dev_info_get(port_id, &dev_info);
rxq_conf = dev_info.default_rxconf;
rxq_conf.offloads = port_conf.rxmode.offloads;
```

For this example we are configuring number of rx and tx queues that are connected to a single port.

```
for (i = 0; i < nr_queues; i++) {
       ret = rte_eth_rx_queue_setup(port_id, i, 512,
                                    rte_eth_dev_socket_id(port_id),
                                    &rxq_conf,
                                    mbuf_pool);
       if (ret < 0) {
               rte_exit(EXIT_FAILURE,
                        ":: Rx queue setup failed: err=%d, port=%u\n",
                        ret, port_id);
       }
}

for (i = 0; i < nr_queues; i++) {
       ret = rte_eth_tx_queue_setup(port_id, i, 512,
                                    rte_eth_dev_socket_id(port_id),
                                    &txq_conf);
       if (ret < 0) {
               rte_exit(EXIT_FAILURE,
                        ":: Tx queue setup failed: err=%d, port=%u\n",
                        ret, port_id);
       }
}
```

In the next step we create and apply the flow rule. which is to send packets with destination ip equals to 192.168.1.1 to queue number 1. The detail explanation of the `generate_ipv4_flow()` appears later in this document:

```
flow = generate_ipv4_flow(port_id, selected_queue,
                          SRC_IP, EMPTY_MASK,
                          DEST_IP, FULL_MASK, &error);
```

We are setting the RX port to promiscuous mode:

```
ret = rte_eth_promiscuous_enable(port_id);
if (ret != 0) {
     rte_exit(EXIT_FAILURE,
              ":: cannot enable promiscuous mode: err=%d, port=%u\n",
              ret, port_id);
}
```

The last step is to start the port.

```
ret = rte_eth_dev_start(port_id);
if (ret < 0)  {
     rte_exit(EXIT_FAILURE, "rte_eth_dev_start:err%d, port=%u\n",
                      ret, port_id);
}
```

9.3.3 The main_loop function

As we saw above the `main()` function calls an application function to handle the main loop. For the flow filtering application the main_loop function looks like the following:

```
static void
main_loop(void)
{
        struct rte_mbuf *mbufs[32];
        struct rte_ether_hdr *eth_hdr;
        uint16_t nb_rx;
        uint16_t i;
        uint16_t j;

        while (!force_quit) {
                for (i = 0; i < nr_queues; i++) {
                        nb_rx = rte_eth_rx_burst(port_id,
                                                 i, mbufs, 32);
                        if (nb_rx) {
                                for (j = 0; j < nb_rx; j++) {
                                        struct rte_mbuf *m = mbufs[j];

                                        eth_hdr = rte_pktmbuf_mtod(m,
                                                     struct rte_ether_hdr *);
                                        print_ether_addr("src=",
                                                     &eth_hdr->s_addr);
                                        print_ether_addr(" - dst=",
                                                     &eth_hdr->d_addr);
                                        printf(" - queue=0x%x",
                                                         (unsigned int)i);
                                        printf("\n");
                                        rte_pktmbuf_free(m);
                                }
                        }
                }
        }
        /* closing and releasing resources */
        rte_flow_flush(port_id, &error);
        rte_eth_dev_stop(port_id);
        rte_eth_dev_close(port_id);
}
```

The main work of the application is reading the packets from all queues and printing for each packet the destination queue:

```
while (!force_quit) {
    for (i = 0; i < nr_queues; i++) {
```

```
                nb_rx = rte_eth_rx_burst(port_id, i, mbufs, 32);
            if (nb_rx) {
                for (j = 0; j < nb_rx; j++) {
                    struct rte_mbuf *m = mbufs[j];
                    eth_hdr = rte_pktmbuf_mtod(m, struct rte_ether_hdr *);
                    print_ether_addr("src=", &eth_hdr->s_addr);
                    print_ether_addr(" - dst=", &eth_hdr->d_addr);
                    printf(" - queue=0x%x", (unsigned int)i);
                    printf("\n");
                    rte_pktmbuf_free(m);
                }
            }
        }
    }
```

The forwarding loop can be interrupted and the application closed using `Ctrl-C`. Which results in closing the port and the device using `rte_eth_dev_stop` and `rte_eth_dev_close`

9.3.4 The generate_ipv4_flow function

The generate_ipv4_flow function is responsible for creating the flow rule. This function is located in the `flow_blocks.c` file.

```
static struct rte_flow *
generate_ipv4_flow(uint8_t port_id, uint16_t rx_q,
                uint32_t src_ip, uint32_t src_mask,
                uint32_t dest_ip, uint32_t dest_mask,
                struct rte_flow_error *error)
{
        struct rte_flow_attr attr;
        struct rte_flow_item pattern[MAX_PATTERN_NUM];
        struct rte_flow_action action[MAX_ACTION_NUM];
        struct rte_flow *flow = NULL;
        struct rte_flow_action_queue queue = { .index = rx_q };
        struct rte_flow_item_ipv4 ip_spec;
        struct rte_flow_item_ipv4 ip_mask;

        memset(pattern, 0, sizeof(pattern));
        memset(action, 0, sizeof(action));

        /*
         * set the rule attribute.
         * in this case only ingress packets will be checked.
         */
        memset(&attr, 0, sizeof(struct rte_flow_attr));
        attr.ingress = 1;

        /*
         * create the action sequence.
         * one action only,  move packet to queue
         */
        action[0].type = RTE_FLOW_ACTION_TYPE_QUEUE;
        action[0].conf = &queue;
        action[1].type = RTE_FLOW_ACTION_TYPE_END;

        /*
         * set the first level of the pattern (ETH).
         * since in this example we just want to get the
         * ipv4 we set this level to allow all.
         */
        pattern[0].type = RTE_FLOW_ITEM_TYPE_ETH;
```

```
        /*
         * setting the second level of the pattern (IP).
         * in this example this is the level we care about
         * so we set it according to the parameters.
         */
        memset(&ip_spec, 0, sizeof(struct rte_flow_item_ipv4));
        memset(&ip_mask, 0, sizeof(struct rte_flow_item_ipv4));
        ip_spec.hdr.dst_addr = htonl(dest_ip);
        ip_mask.hdr.dst_addr = dest_mask;
        ip_spec.hdr.src_addr = htonl(src_ip);
        ip_mask.hdr.src_addr = src_mask;
        pattern[1].type = RTE_FLOW_ITEM_TYPE_IPV4;
        pattern[1].spec = &ip_spec;
        pattern[1].mask = &ip_mask;

        /* the final level must be always type end */
        pattern[2].type = RTE_FLOW_ITEM_TYPE_END;

        int res = rte_flow_validate(port_id, &attr, pattern, action, error);
        if(!res)
            flow = rte_flow_create(port_id, &attr, pattern, action, error);

        return flow;
}
```

The first part of the function is declaring the structures that will be used.

```
struct rte_flow_attr attr;
struct rte_flow_item pattern[MAX_PATTERN_NUM];
struct rte_flow_action action[MAX_ACTION_NUM];
struct rte_flow *flow;
struct rte_flow_error error;
struct rte_flow_action_queue queue = { .index = rx_q };
struct rte_flow_item_ipv4 ip_spec;
struct rte_flow_item_ipv4 ip_mask;
```

The following part create the flow attributes, in our case ingress.

```
memset(&attr, 0, sizeof(struct rte_flow_attr));
attr.ingress = 1;
```

The third part defines the action to be taken when a packet matches the rule. In this case send the packet to queue.

```
action[0].type = RTE_FLOW_ACTION_TYPE_QUEUE;
action[0].conf = &queue;
action[1].type = RTE_FLOW_ACTION_TYPE_END;
```

The fourth part is responsible for creating the pattern and is built from number of steps. In each step we build one level of the pattern starting with the lowest one.

Setting the first level of the pattern ETH:

```
pattern[0].type = RTE_FLOW_ITEM_TYPE_ETH;
```

Setting the second level of the pattern IP:

```
memset(&ip_spec, 0, sizeof(struct rte_flow_item_ipv4));
memset(&ip_mask, 0, sizeof(struct rte_flow_item_ipv4));
ip_spec.hdr.dst_addr = htonl(dest_ip);
ip_mask.hdr.dst_addr = dest_mask;
ip_spec.hdr.src_addr = htonl(src_ip);
ip_mask.hdr.src_addr = src_mask;
pattern[1].type = RTE_FLOW_ITEM_TYPE_IPV4;
```

```
pattern[1].spec = &ip_spec;
pattern[1].mask = &ip_mask;
```

Closing the pattern part.

```
pattern[2].type = RTE_FLOW_ITEM_TYPE_END;
```

The last part of the function is to validate the rule and create it.

```
int res = rte_flow_validate(port_id, &attr, pattern, action, &error);
if (!res)
     flow = rte_flow_create(port_id, &attr, pattern, action, &error);
```

CHAPTER

TEN

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.

10.1 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).

10.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `ip_fragmentation` sub-directory.

10.3 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 linux 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 linux 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[] = {
    {RTE_IPV4(100, 10, 0, 0), 16, 0},
    {RTE_IPV4(100, 20, 0, 0), 16, 1},
    {RTE_IPV4(100, 30, 0, 0), 16, 2},
    {RTE_IPV4(100, 40, 0, 0), 16, 3},
    {RTE_IPV4(100, 50, 0, 0), 16, 4},
    {RTE_IPV4(100, 60, 0, 0), 16, 5},
    {RTE_IPV4(100, 70, 0, 0), 16, 6},
    {RTE_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}, 48, 6},
        {{8, 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 Abstraction Layer (EAL) options.

CHAPTER

ELEVEN

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.

11.1 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.

Note: The max port mask supported in the given hash table is 0xf, so only first four ports can be supported. If using non-consecutive ports, use the destination IPv4 address accordingly.

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.

11.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `ipv4_multicast` sub-directory.

11.3 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 -l 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.

11.4 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.

11.4.1 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.

11.4.2 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
                    return -1;
        }
    }
    return 0;
}
```

11.4.3 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 rte_ipv4_hdr *)rte_pktmbuf_adj(m, sizeof(struct rte_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 (!RTE_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 rte_ether_addr *dest_addr, struc
{
    struct rte_ether_hdr *ethdr;
    uint16_t len;

    /* Construct Ethernet header. */

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

    RTE_ASSERT(ethdr != NULL);

    rte_ether_addr_copy(dest_addr, &ethdr->d_addr);
    rte_ether_addr_copy(&ports_eth_addr[port], &ethdr->s_addr);
    ethdr->ether_type = rte_be_to_cpu_16(RTE_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);
}
```

11.4.4 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 = 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;
    rte_mbuf_sanity_check(hdr, RTE_MBUF_PKT, 1);

    return hdr;
}
```

CHAPTER

TWELVE

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.

12.1 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.

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.

12.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `ip_reassembly` sub-directory.

12.3 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[(
```

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 linux 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 linux 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[] = {
    {RTE_IPV4(100, 10, 0, 0), 16, 0},
    {RTE_IPV4(100, 20, 0, 0), 16, 1},
    {RTE_IPV4(100, 30, 0, 0), 16, 2},
    {RTE_IPV4(100, 40, 0, 0), 16, 3},
    {RTE_IPV4(100, 50, 0, 0), 16, 4},
    {RTE_IPV4(100, 60, 0, 0), 16, 5},
    {RTE_IPV4(100, 70, 0, 0), 16, 6},
    {RTE_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.

12.4 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.

12.4.1 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
{
    RTE_LOG(ERR, IP_RSMBL, "ip_frag_tbl_create(%u) on " "lcore: %u for queue: %u failed\n",  ma
    return -1;
}
```

12.4.2 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_
    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;
}
```

12.4.3 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.

12.4.4 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.

CHAPTER

THIRTEEN

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 (tshark, rsync, 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, iproute, tshark) with the DPDK ports and also the exchange of packets between the DPDK application and the Linux* kernel.

The Kernel NIC Interface sample application requires that the KNI kernel module `rte_kni` be loaded into the kernel. See ../prog_guide/kernel_nic_interface for more information on loading the `rte_kni` kernel module.

13.1 Overview

The Kernel NIC Interface sample application `kni` allocates one or more KNI interfaces for each physical NIC port. For each physical NIC port, `kni` uses two DPDK threads in user space; one thread reads from the port and writes to the corresponding KNI interfaces and the other thread reads from the KNI interfaces and writes the data unmodified to the physical NIC port.

It is recommended to configure one KNI interface for each physical NIC port. The application can be configured with more than one KNI interface for each physical NIC port 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.

If link monitoring is enabled with the `-m` command line flag, one additional pthread is launched which will check the link status of each physical NIC port and will update the carrier status of the corresponding

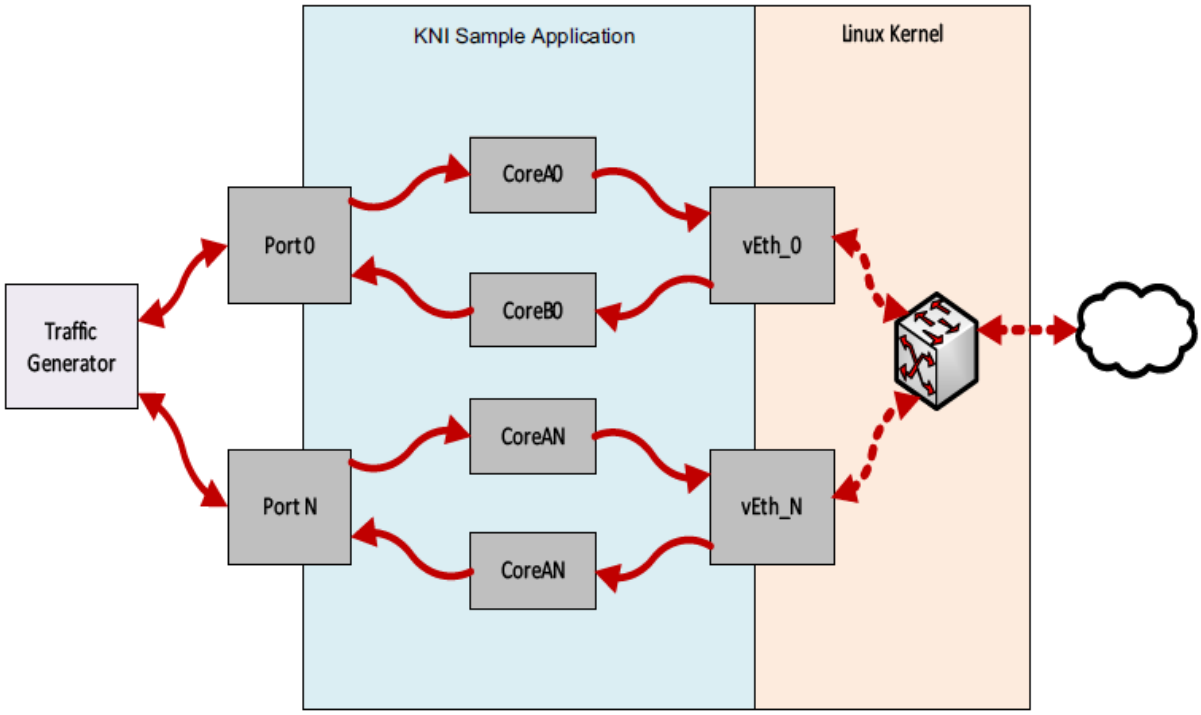


Fig. 13.1: Kernel NIC Application Packet Flow

KNI interface(s) to match the physical NIC port's state. This means that the KNI interface(s) will be disabled automatically when the Ethernet link goes down and enabled when the Ethernet link goes up.

If link monitoring is enabled, the `rte_kni` kernel module should be loaded such that the default carrier state is set to *off*. This ensures that the KNI interface is only enabled *after* the Ethernet link of the corresponding NIC port has reached the linkup state.

If link monitoring is not enabled, the `rte_kni` kernel module should be loaded with the default carrier state set to *on*. This sets the carrier state of the KNI interfaces to *on* when the KNI interfaces are enabled without regard to the actual link state of the corresponding NIC port. This is useful for testing in loopback mode where the NIC port may not be physically connected to anything.

13.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `examples/kni` sub-directory.

Note: This application is intended as a linux only.

13.3 Running the kni Example Application

The `kni` example application requires a number of command line options:

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

Where:

- `-p PORTMASK`:

  Hexadecimal bitmask of ports to configure.

- `--config="(port,lcore_rx,lcore_tx[,lcore_kthread,...])[,(port,lcore_rx,lcore`

  Determines which lcores the Rx and Tx DPDK tasks, and (optionally) the KNI kernel thread(s) are bound to for each physical port.

- `-P`:

  Optional flag to 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.

- `-m`:

  Optional flag to enable monitoring and updating of the Ethernet carrier state. With this option set, a thread will be started which will periodically check the Ethernet link status of the physical Ethernet ports and set the carrier state of the corresponding KNI network interface to match it. This means that the KNI interface will be disabled automatically when the Ethernet link goes down and enabled when the Ethernet link goes up.

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 must include the lcores specified by `lcore_rx` and `lcore_tx` for each port, but does not need to include lcores specified by `lcore_kthread` as those cores are used to pin the kernel threads in the `rte_kni` kernel module.

The `--config` parameter must include a set of `(port,lcore_rx,lcore_tx,[lcore_kthread,...])` values for each physical port specified in the `-p PORTMASK` parameter.

The optional `lcore_kthread` lcore ID parameter in `--config` can be specified zero, one or more times for each physical port.

If no lcore ID is specified for `lcore_kthread`, one KNI interface will be created for the physical port `port` and the KNI kernel thread(s) will have no specific core affinity.

If one or more lcore IDs are specified for `lcore_kthread`, a KNI interface will be created for each lcore ID specified, bound to the physical port `port`. If the `rte_kni` kernel module is loaded in multiple kernel thread mode, a kernel thread will be created for each KNI interface and bound to the specified core. If the `rte_kni` kernel module is loaded in single kernel thread mode, only one kernel thread is started for all KNI interfaces. The kernel thread will be bound to the first `lcore_kthread` lcore ID specified.

13.3.1 Example Configurations

The following commands will first load the `rte_kni` kernel module in multiple kernel thread mode. The `kni` application is then started using two ports; Port 0 uses lcore 4 for the Rx task, lcore 6 for the Tx task, and will create a single KNI interface `vEth0_0` with the kernel thread bound to lcore 8. Port 1 uses lcore 5 for the Rx task, lcore 7 for the Tx task, and will create a single KNI interface `vEth1_0` with the kernel thread bound to lcore 9.

```
# rmmod rte_kni
# insmod kmod/rte_kni.ko kthread_mode=multiple
# ./build/kni -l 4-7 -n 4 -- -P -p 0x3 -m --config="(0,4,6,8),(1,5,7,9)"
```

The following example is identical, except an additional `lcore_kthread` core is specified per physical port. In this case, `kni` will create four KNI interfaces: `vEth0_0`/`vEth0_1` bound to physical port 0 and `vEth1_0`/`vEth1_1` bound to physical port 1.

The kernel thread for each interface will be bound as follows:

- `vEth0_0` - bound to lcore 8.
- `vEth0_1` - bound to lcore 10.
- `vEth1_0` - bound to lcore 9.
- `vEth1_1` - bound to lcore 11

```
# rmmod rte_kni
# insmod kmod/rte_kni.ko kthread_mode=multiple
# ./build/kni -l 4-7 -n 4 -- -P -p 0x3 -m --config="(0,4,6,8,10),(1,5,7,9,11)"
```

The following example can be used to test the interface between the `kni` test application and the `rte_kni` kernel module. In this example, the `rte_kni` kernel module is loaded in single kernel thread mode, loopback mode enabled, and the default carrier state is set to *on* so that the corresponding physical NIC port does not have to be connected in order to use the KNI interface. One KNI interface `vEth0_0` is created for port 0 and one KNI interface `vEth1_0` is created for port 1. Since `rte_kni` is loaded in "single kernel thread" mode, the one kernel thread is bound to lcore 8.

Since the physical NIC ports are not being used, link monitoring can be disabled by **not** specifying the `-m` flag to `kni`:

```
# rmmod rte_kni
# insmod kmod/rte_kni.ko lo_mode=lo_mode_fifo carrier=on
# ./build/kni -l 4-7 -n 4 -- -P -p 0x3 --config="(0,4,6,8),(1,5,7,9)"
```

13.4 KNI Operations

Once the `kni` application is started, the user can use the normal Linux commands to manage the KNI interfaces as if they were any other Linux network interface.

Enable KNI interface and assign an IP address:

```
# ip addr add dev vEth0_0 192.168.0.1
```

Show KNI interface configuration and statistics:

```
# ip -s -d addr show vEth0_0
```

The user can also check and reset the packet statistics inside the `kni` application by sending the app the USR1 and USR2 signals:

```
# Print statistics
# pkill -USR1 kni

# Zero statistics
# pkill -USR2 kni
```

Dump network traffic:

```
# tshark -n -i vEth0_0
```

The normal Linux commands can also be used to change the MAC address and MTU size used by the physical NIC which corresponds to the KNI interface. However, if more than one KNI interface is configured for a physical port, these commands will only work on the first KNI interface for that port.

Change the MAC address:

```
# ip link set dev vEth0_0 lladdr 0C:01:02:03:04:08
```

Change the MTU size:

```
# ip link set dev vEth0_0 mtu 1450
```

Limited ethtool support:

```
# ethtool -i vEth0_0
```

When the `kni` application is closed, all the KNI interfaces are deleted from the Linux kernel.

13.5 Explanation

The following sections provide some explanation of code.

13.5.1 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 in the function `kni_alloc`.

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 in the function `parse_config`.

13.5.2 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 (`kni_ingress`), 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.

For the other case that reads from kernel NIC interfaces and writes to a physical NIC port (`kni_egress`), 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*).

CHAPTER

FOURTEEN

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.

14.1 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.

14.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `l2fwd_keep_alive` sub-directory.

14.3 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 linux 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.

14.4 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 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->live_data[rte_lcore_id()].core_state = RTE_KA_STATE_ALIVE;
}
```

CHAPTER

FIFTEEN

PACKET COPYING USING INTEL® QUICKDATA TECHNOLOGY

15.1 Overview

This sample is intended as a demonstration of the basic components of a DPDK forwarding application and example of how to use IOAT driver API to make packets copies.

Also while forwarding, 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 compare performance of using software packet copy with copy done using a DMA device for different sizes of packets. The example will print out statistics each second. The stats shows received/send packets and packets dropped or failed to copy.

15.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `ioat` sub-directory.

15.3 Running the Application

In order to run the hardware copy application, the copying device needs to be bound to user-space IO driver.

Refer to the "IOAT Rawdev Driver" chapter in the "Rawdev Drivers" document for information on using the driver.

The application requires a number of command line options:

```
./build/ioatfwd [EAL options] -- [-p MASK] [-q NQ] [-s RS] [-c <sw|hw>]
    [--[no-]mac-updating]
```

where,

- p MASK: A hexadecimal bitmask of the ports to configure (default is all)
- q NQ: Number of Rx queues used per port equivalent to CBDMA channels per port (default is 1)
- c CT: Performed packet copy type: software (sw) or hardware using DMA (hw) (default is hw)

- s RS: Size of IOAT rawdev ring for hardware copy mode or rte_ring for software copy mode (default is 2048)
- –[no-]mac-updating: Whether MAC address of packets should be changed or not (default is mac-updating)

The application can be launched in various configurations depending on provided parameters. The app can use up to 2 lcores: one of them receives incoming traffic and makes a copy of each packet. The second lcore then updates MAC address and sends the copy. If one lcore per port is used, both operations are done sequentially. For each configuration an additional lcore is needed since the master lcore does not handle traffic but is responsible for configuration, statistics printing and safe shutdown of all ports and devices.

The application can use a maximum of 8 ports.

To run the application in a Linux environment with 3 lcores (the master lcore, plus two forwarding cores), a single port (port 0), software copying and MAC updating issue the command:

```
$ ./build/ioatfwd -l 0-2 -n 2 -- -p 0x1 --mac-updating -c sw
```

To run the application in a Linux environment with 2 lcores (the master lcore, plus one forwarding core), 2 ports (ports 0 and 1), hardware copying and no MAC updating issue the command:

```
$ ./build/ioatfwd -l 0-1 -n 1 -- -p 0x3 --no-mac-updating -c hw
```

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

15.4 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*.

15.4.1 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:

```
/* init EAL */
ret = rte_eal_init(argc, argv);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Invalid EAL arguments\n");
```

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

```
nb_mbufs = RTE_MAX(rte_eth_dev_count_avail() * (nb_rxd + nb_txd
    + MAX_PKT_BURST + rte_lcore_count() * MEMPOOL_CACHE_SIZE),
    MIN_POOL_SIZE);

/* Create the mbuf pool */
ioat_pktmbuf_pool = rte_pktmbuf_pool_create("mbuf_pool", nb_mbufs,
    MEMPOOL_CACHE_SIZE, 0, RTE_MBUF_DEFAULT_BUF_SIZE,
    rte_socket_id());
if (ioat_pktmbuf_pool == NULL)
    rte_exit(EXIT_FAILURE, "Cannot init mbuf pool\n");
```

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 the ports:

```
/* Initialise each port */
RTE_ETH_FOREACH_DEV(portid) {
    port_init(portid, ioat_pktmbuf_pool);
}
```

Each port is configured using `port_init()` function. The Ethernet ports are configured with local settings using the `rte_eth_dev_configure()` function and the `port_conf` struct. The RSS is enabled so that multiple Rx queues could be used for packet receiving and copying by multiple CBDMA channels per port:

```
/* configuring port to use RSS for multiple RX queues */
static const struct rte_eth_conf port_conf = {
    .rxmode = {
        .mq_mode        = ETH_MQ_RX_RSS,
        .max_rx_pkt_len = RTE_ETHER_MAX_LEN
    },
    .rx_adv_conf = {
        .rss_conf = {
            .rss_key = NULL,
            .rss_hf = ETH_RSS_PROTO_MASK,
        }
    }
};
```

For this example the ports are set up with the number of Rx queues provided with -q option and 1 Tx queue using the `rte_eth_rx_queue_setup()` and `rte_eth_tx_queue_setup()` functions.

The Ethernet port is then started:

```
ret = rte_eth_dev_start(portid);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "rte_eth_dev_start:err=%d, port=%u\n",
        ret, portid);
```

Finally the Rx port is set in promiscuous mode:

```
rte_eth_promiscuous_enable(portid);
```

After that each port application assigns resources needed.

```
check_link_status(ioat_enabled_port_mask);

if (!cfg.nb_ports) {
    rte_exit(EXIT_FAILURE,
        "All available ports are disabled. Please set portmask.\n");
}

/* Check if there is enough lcores for all ports. */
cfg.nb_lcores = rte_lcore_count() - 1;
if (cfg.nb_lcores < 1)
    rte_exit(EXIT_FAILURE,
        "There should be at least one slave lcore.\n");

ret = 0;

if (copy_mode == COPY_MODE_IOAT_NUM) {
    assign_rawdevs();
} else /* copy_mode == COPY_MODE_SW_NUM */ {
    assign_rings();
}
```

Depending on mode set (whether copy should be done by software or by hardware) special structures are assigned to each port. If software copy was chosen, application have to assign ring structures for packet exchanging between lcores assigned to ports.

```
static void
assign_rings(void)
{
    uint32_t i;

    for (i = 0; i < cfg.nb_ports; i++) {
        char ring_name[20];

        snprintf(ring_name, 20, "rx_to_tx_ring_%u", i);
        /* Create ring for inter core communication */
        cfg.ports[i].rx_to_tx_ring = rte_ring_create(
                ring_name, ring_size,
                rte_socket_id(), RING_F_SP_ENQ);

        if (cfg.ports[i].rx_to_tx_ring == NULL)
            rte_exit(EXIT_FAILURE, "%s\n",
                    rte_strerror(rte_errno));
    }
}
```

When using hardware copy each Rx queue of the port is assigned an IOAT device (`assign_rawdevs()`) using IOAT Rawdev Driver API functions:

```
static void
assign_rawdevs(void)
{
    uint16_t nb_rawdev = 0, rdev_id = 0;
    uint32_t i, j;

    for (i = 0; i < cfg.nb_ports; i++) {
        for (j = 0; j < cfg.ports[i].nb_queues; j++) {
            struct rte_rawdev_info rdev_info = { 0 };

            do {
                if (rdev_id == rte_rawdev_count())
                    goto end;
                rte_rawdev_info_get(rdev_id++, &rdev_info);
            } while (strcmp(rdev_info.driver_name,
                IOAT_PMD_RAWDEV_NAME_STR) != 0);

            cfg.ports[i].ioat_ids[j] = rdev_id - 1;
            configure_rawdev_queue(cfg.ports[i].ioat_ids[j]);
            ++nb_rawdev;
        }
    }
end:
    if (nb_rawdev < cfg.nb_ports * cfg.ports[0].nb_queues)
        rte_exit(EXIT_FAILURE,
            "Not enough IOAT rawdevs (%u) for all queues (%u).\n",
            nb_rawdev, cfg.nb_ports * cfg.ports[0].nb_queues);
    RTE_LOG(INFO, IOAT, "Number of used rawdevs: %u.\n", nb_rawdev);
}
```

The initialization of hardware device is done by `rte_rawdev_configure()` function using `rte_rawdev_info` struct. After configuration the device is started using `rte_rawdev_start()` function. Each of the above operations is done in `configure_rawdev_queue()`.

```
static void
configure_rawdev_queue(uint32_t dev_id)
```

```
{
    struct rte_ioat_rawdev_config dev_config = { .ring_size = ring_size };
    struct rte_rawdev_info info = { .dev_private = &dev_config };

    if (rte_rawdev_configure(dev_id, &info) != 0) {
        rte_exit(EXIT_FAILURE,
            "Error with rte_rawdev_configure()\n");
    }
    if (rte_rawdev_start(dev_id) != 0) {
        rte_exit(EXIT_FAILURE,
            "Error with rte_rawdev_start()\n");
    }
}
```

If initialization is successful, memory for hardware device statistics is allocated.

Finally `main()` function starts all packet handling lcores and starts printing stats in a loop on the master lcore. The application can be interrupted and closed using `Ctrl-C`. The master lcore waits for all slave processes to finish, deallocates resources and exits.

The processing lcores launching function are described below.

15.4.2 The Lcores Launching Functions

As described above, `main()` function invokes `start_forwarding_cores()` function in order to start processing for each lcore:

```
static void start_forwarding_cores(void)
{
    uint32_t lcore_id = rte_lcore_id();

    RTE_LOG(INFO, IOAT, "Entering %s on lcore %u\n",
            __func__, rte_lcore_id());

    if (cfg.nb_lcores == 1) {
        lcore_id = rte_get_next_lcore(lcore_id, true, true);
        rte_eal_remote_launch((lcore_function_t *)rxtx_main_loop,
            NULL, lcore_id);
    } else if (cfg.nb_lcores > 1) {
        lcore_id = rte_get_next_lcore(lcore_id, true, true);
        rte_eal_remote_launch((lcore_function_t *)rx_main_loop,
            NULL, lcore_id);

        lcore_id = rte_get_next_lcore(lcore_id, true, true);
        rte_eal_remote_launch((lcore_function_t *)tx_main_loop, NULL,
            lcore_id);
    }
}
```

The function launches Rx/Tx processing functions on configured lcores using `rte_eal_remote_launch()`. The configured ports, their number and number of assigned lcores are stored in user-defined `rxtx_transmission_config` struct:

```
struct rxtx_transmission_config {
    struct rxtx_port_config ports[RTE_MAX_ETHPORTS];
    uint16_t nb_ports;
    uint16_t nb_lcores;
};
```

The structure is initialized in 'main()' function with the values corresponding to ports and lcores configuration provided by the user.

15.4.3 The Lcores Processing Functions

For receiving packets on each port, the `ioat_rx_port()` function is used. The function receives packets on each configured Rx queue. Depending on the mode the user chose, it will enqueue packets to IOAT rawdev channels and then invoke copy process (hardware copy), or perform software copy of each packet using `pktmbuf_sw_copy()` function and enqueue them to an rte_ring:

```
/* Receive packets on one port and enqueue to IOAT rawdev or rte_ring. */
static void
ioat_rx_port(struct rxtx_port_config *rx_config)
{
    uint32_t nb_rx, nb_enq, i, j;
    struct rte_mbuf *pkts_burst[MAX_PKT_BURST];
    for (i = 0; i < rx_config->nb_queues; i++) {

        nb_rx = rte_eth_rx_burst(rx_config->rxtx_port, i,
            pkts_burst, MAX_PKT_BURST);

        if (nb_rx == 0)
            continue;

        port_statistics.rx[rx_config->rxtx_port] += nb_rx;

        if (copy_mode == COPY_MODE_IOAT_NUM) {
            /* Perform packet hardware copy */
            nb_enq = ioat_enqueue_packets(pkts_burst,
                nb_rx, rx_config->ioat_ids[i]);
            if (nb_enq > 0)
                rte_ioat_do_copies(rx_config->ioat_ids[i]);
        } else {
            /* Perform packet software copy, free source packets */
            int ret;
            struct rte_mbuf *pkts_burst_copy[MAX_PKT_BURST];

            ret = rte_mempool_get_bulk(ioat_pktmbuf_pool,
                (void *)pkts_burst_copy, nb_rx);

            if (unlikely(ret < 0))
                rte_exit(EXIT_FAILURE,
                    "Unable to allocate memory.\n");

            for (j = 0; j < nb_rx; j++)
                pktmbuf_sw_copy(pkts_burst[j],
                    pkts_burst_copy[j]);

            rte_mempool_put_bulk(ioat_pktmbuf_pool,
                (void *)pkts_burst, nb_rx);

            nb_enq = rte_ring_enqueue_burst(
                rx_config->rx_to_tx_ring,
                (void *)pkts_burst_copy, nb_rx, NULL);

            /* Free any not enqueued packets. */
            rte_mempool_put_bulk(ioat_pktmbuf_pool,
                (void *)&pkts_burst_copy[nb_enq],
                nb_rx - nb_enq);
        }

        port_statistics.copy_dropped[rx_config->rxtx_port] +=
            (nb_rx - nb_enq);
    }
}
```

The packets are received in burst mode using `rte_eth_rx_burst()` function. When using hardware copy mode the packets are enqueued in copying device's buffer using `ioat_enqueue_packets()` which calls `rte_ioat_enqueue_copy()`. When all received packets are in the buffer the copy operations are started by calling `rte_ioat_do_copies()`. Function `rte_ioat_enqueue_copy()` operates on physical address of the packet. Structure `rte_mbuf` contains only physical address to start of the data buffer (`buf_iova`). Thus the address is adjusted by `addr_offset` value in order to get the address of `rearm_data` member of `rte_mbuf`. That way both the packet data and metadata can be copied in a single operation. This method can be used because the mbufs are direct mbufs allocated by the apps. If another app uses external buffers, or indirect mbufs, then multiple copy operations must be used.

```
static uint32_t
ioat_enqueue_packets(struct rte_mbuf **pkts,
    uint32_t nb_rx, uint16_t dev_id)
{
    int ret;
    uint32_t i;
    struct rte_mbuf *pkts_copy[MAX_PKT_BURST];

    const uint64_t addr_offset = RTE_PTR_DIFF(pkts[0]->buf_addr,
        &pkts[0]->rearm_data);

    ret = rte_mempool_get_bulk(ioat_pktmbuf_pool,
            (void *)pkts_copy, nb_rx);

    if (unlikely(ret < 0))
        rte_exit(EXIT_FAILURE, "Unable to allocate memory.\n");

    for (i = 0; i < nb_rx; i++) {
        /* Perform data copy */
        ret = rte_ioat_enqueue_copy(dev_id,
            pkts[i]->buf_iova
                - addr_offset,
            pkts_copy[i]->buf_iova
                - addr_offset,
            rte_pktmbuf_data_len(pkts[i])
                + addr_offset,
            (uintptr_t)pkts[i],
            (uintptr_t)pkts_copy[i],
            0 /* nofence */);

        if (ret != 1)
            break;
    }

    ret = i;
    /* Free any not enqueued packets. */
    rte_mempool_put_bulk(ioat_pktmbuf_pool, (void *)&pkts[i], nb_rx - i);
    rte_mempool_put_bulk(ioat_pktmbuf_pool, (void *)&pkts_copy[i],
        nb_rx - i);

    return ret;
}
```

All completed copies are processed by `ioat_tx_port()` function. When using hardware copy mode the function invokes `rte_ioat_completed_copies()` on each assigned IOAT channel to gather copied packets. If software copy mode is used the function dequeues copied packets from the rte_ring. Then each packet MAC address is changed if it was enabled. After that copies are sent in burst mode using '' rte_eth_tx_burst()''.

```
/* Transmit packets from IOAT rawdev/rte_ring for one port. */
static void
ioat_tx_port(struct rxtx_port_config *tx_config)
{
    uint32_t i, j, nb_dq = 0;
    struct rte_mbuf *mbufs_src[MAX_PKT_BURST];
    struct rte_mbuf *mbufs_dst[MAX_PKT_BURST];

    for (i = 0; i < tx_config->nb_queues; i++) {
        if (copy_mode == COPY_MODE_IOAT_NUM) {
            /* Deque the mbufs from IOAT device. */
            nb_dq = rte_ioat_completed_copies(
                tx_config->ioat_ids[i], MAX_PKT_BURST,
                (void *)mbufs_src, (void *)mbufs_dst);
        } else {
            /* Deque the mbufs from rx_to_tx_ring. */
            nb_dq = rte_ring_dequeue_burst(
                tx_config->rx_to_tx_ring, (void *)mbufs_dst,
                MAX_PKT_BURST, NULL);
        }

        if (nb_dq == 0)
            return;

        if (copy_mode == COPY_MODE_IOAT_NUM)
            rte_mempool_put_bulk(ioat_pktmbuf_pool,
                (void *)mbufs_src, nb_dq);

        /* Update macs if enabled */
        if (mac_updating) {
            for (j = 0; j < nb_dq; j++)
                update_mac_addrs(mbufs_dst[j],
                    tx_config->rxtx_port);
        }

        const uint16_t nb_tx = rte_eth_tx_burst(
            tx_config->rxtx_port, 0,
            (void *)mbufs_dst, nb_dq);

        port_statistics.tx[tx_config->rxtx_port] += nb_tx;

        /* Free any unsent packets. */
        if (unlikely(nb_tx < nb_dq))
            rte_mempool_put_bulk(ioat_pktmbuf_pool,
            (void *)&mbufs_dst[nb_tx],
                nb_dq - nb_tx);
    }
}
```

15.4.4 The Packet Copying Functions

In order to perform packet copy there is a user-defined function `pktmbuf_sw_copy()` used. It copies a whole packet by copying metadata from source packet to new mbuf, and then copying a data chunk of source packet. Both memory copies are done using `rte_memcpy()`:

```
static inline void
pktmbuf_sw_copy(struct rte_mbuf *src, struct rte_mbuf *dst)
{
    /* Copy packet metadata */
    rte_memcpy(&dst->rearm_data,
        &src->rearm_data,
```

```
            offsetof(struct rte_mbuf, cacheline1)
                - offsetof(struct rte_mbuf, rearm_data));

        /* Copy packet data */
        rte_memcpy(rte_pktmbuf_mtod(dst, char *),
            rte_pktmbuf_mtod(src, char *), src->data_len);
    }
```

The metadata in this example is copied from `rearm_data` member of `rte_mbuf` struct up to `cacheline1`.

In order to understand why software packet copying is done as shown above please refer to the "Mbuf Library" section of the *DPDK Programmer's Guide*.

CHAPTER

SIXTEEN

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.

16.1 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, 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

16.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `l2fwd-crypt` sub-directory.

16.3 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/AEAD] /
[--cipher_algo ALGO] [--cipher_op ENCRYPT/DECRYPT] [--cipher_key KEY] /
[--cipher_key_random_size SIZE] [--cipher_iv IV] [--cipher_iv_random_size SIZE] /
[--auth_algo ALGO] [--auth_op GENERATE/VERIFY] [--auth_key KEY] /
[--auth_key_random_size SIZE] [--auth_iv IV] [--auth_iv_random_size SIZE] /
[--aead_algo ALGO] [--aead_op ENCRYPT/DECRYPT] [--aead_key KEY] /
[--aead_key_random_size SIZE] [--aead_iv] [--aead_iv_random_size SIZE] /
[--aad AAD] [--aad_random_size SIZE] /
[--digest size SIZE] [--sessionless] [--cryptodev_mask MASK] /
[--mac-updating] [--no-mac-updating]
```

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)
  or AEAD (AEAD)
  (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.
- cipher_iv: set the cipher IV to be used. Bytes has to be separated with ":"
- cipher_iv_random_size: set the size of the cipher IV, which will be generated randomly.
  Note that if –cipher_iv is used, this will be ignored.
- auth_algo: select the authentication algorithm (default is sha1-hmac)
- auth_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.
- auth_iv: set the auth IV to be used. Bytes has to be separated with ":"
- auth_iv_random_size: set the size of the auth IV, which will be generated randomly.
  Note that if –auth_iv is used, this will be ignored.
- aead_algo: select the AEAD algorithm (default is aes-gcm)
- aead_op: select the AEAD operation to perform: ENCRYPT or DECRYPT
  (default is ENCRYPT)

- aead_key: set the AEAD key to be used. Bytes has to be separated with ":"
- aead_key_random_size: set the size of the AEAD key,

  which will be generated randomly.

  Note that if –aead_key is used, this will be ignored.
- aead_iv: set the AEAD IV to be used. Bytes has to be separated with ":"
- aead_iv_random_size: set the size of the AEAD IV, which will be generated randomly.

  Note that if –aead_iv 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.
- cryptodev_mask: A hexadecimal bitmask of the cryptodevs to be used by the application.

  (default is all cryptodevs).
- [no-]mac-updating: Enable or disable MAC addresses updating (enabled by default).

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 linux environment with 2 lcores, 2 ports and 2 crypto devices, issue the command:

```
$ ./build/l2fwd-crypto -l 0-1 -n 4 --vdev "crypto_aesni_mb0" \
--vdev "crypto_aesni_mb1" -- -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.

Note:

- The `l2fwd-crypto` sample application requires IPv4 packets for crypto operation.
- If multiple Ethernet ports is passed, then equal number of crypto devices are to be passed.
- All crypto devices shall use the same session.

16.4 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.

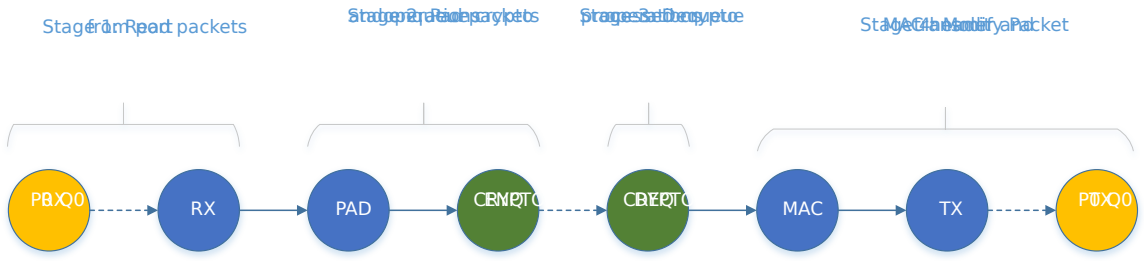


Fig. 16.1: Encryption flow Through the L2 Forwarding with Crypto Application

The following sections provide some explanation of the application.

16.4.1 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.

16.4.2 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.

16.4.3 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;
        struct rte_cryptodev_sym_session *session;
        uint8_t socket_id = rte_cryptodev_socket_id(cdev_id);
        struct rte_mempool *sess_mp = session_pool_socket[socket_id];


        if (options->xform_chain == L2FWD_CRYPTO_AEAD) {
                first_xform = &options->aead_xform;
        } else 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;
        }

        session = rte_cryptodev_sym_session_create(sess_mp);

        if (session == NULL)
                return NULL;

       if (rte_cryptodev_sym_session_init(cdev_id, session,
                           first_xform, sess_mp) < 0)
                return NULL;

       return session;
}

...

port_cparams[i].session = initialize_crypto_session(options,
                              port_cparams[i].dev_id);
```

16.4.4 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).

16.4.5 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);
```

CHAPTER

SEVENTEEN

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.

17.1 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. 17.1.

The application can also be used in a virtualized environment as shown in Fig. 17.2.

The L2 Forwarding application can also be used as a starting point for developing a new application based on the DPDK.

17.1.1 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
```

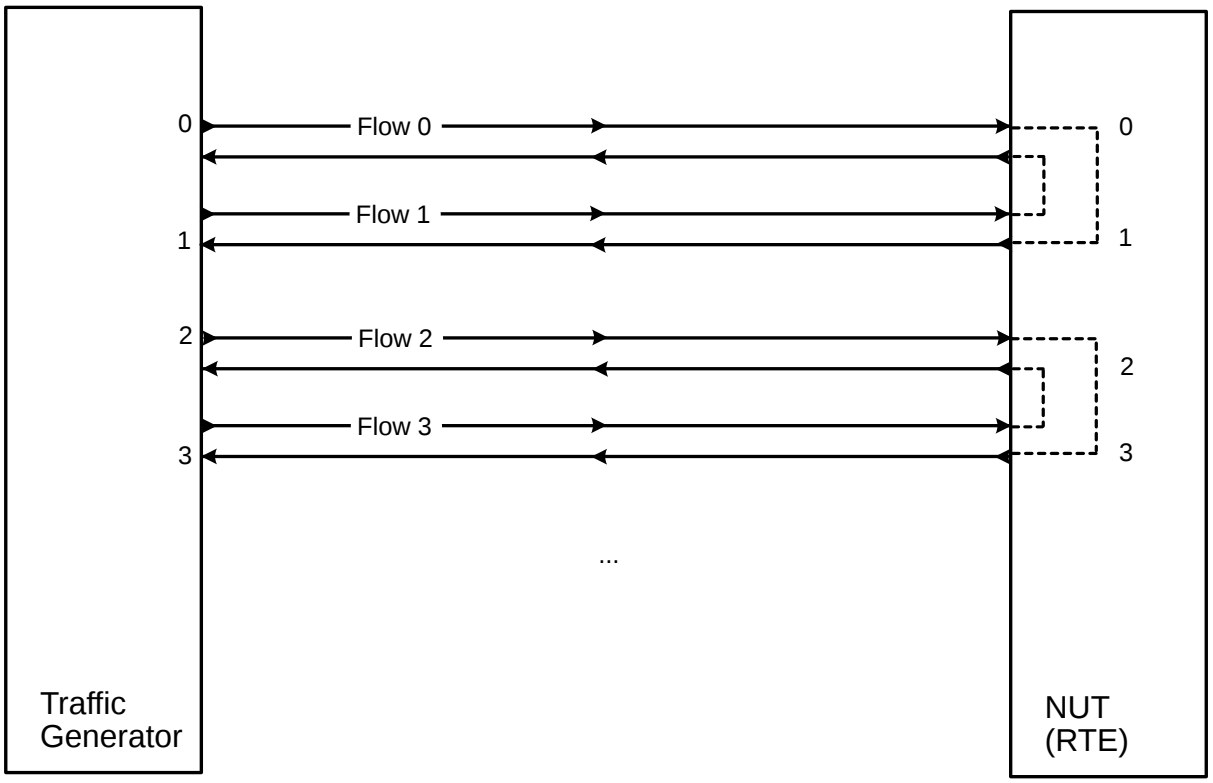


Fig. 17.1: Performance Benchmark Setup (Basic Environment)

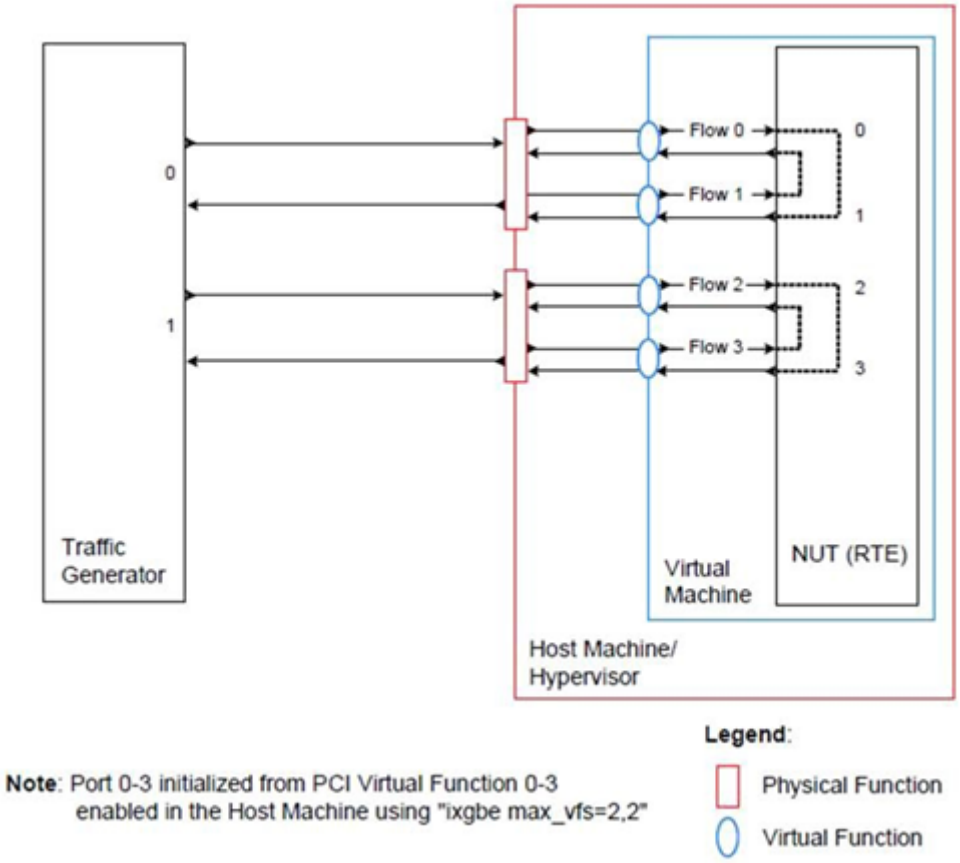


Fig. 17.2: Performance Benchmark Setup (Virtualized Environment)

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.

17.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `l2fwd-jobstats` sub-directory.

17.3 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 linux 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.

17.4 Explanation

The following sections provide some explanation of the code.

17.4.1 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");
```

17.4.2 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.

17.4.3 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*.

```
/* 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.
 */
RTE_ETH_FOREACH_DEV(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);
```

17.4.4 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.

17.4.5 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);
```

17.4.6 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)

17.4.7 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.

17.4.8 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 rte_ether_hdr *eth;
    void *tmp;
    unsigned dst_port;

    dst_port = l2fwd_dst_ports[portid];

    eth = rte_pktmbuf_mtod(m, struct rte_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 */

        rte_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, uint16_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;
    uint16_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);
    }
```

CHAPTER

EIGHTEEN

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.

18.1 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. 18.1, or in a virtualized environment as shown in Fig. 18.2.

This application may be used for basic VM to VM communication as shown in Fig. 18.3, 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.

18.1.1 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:

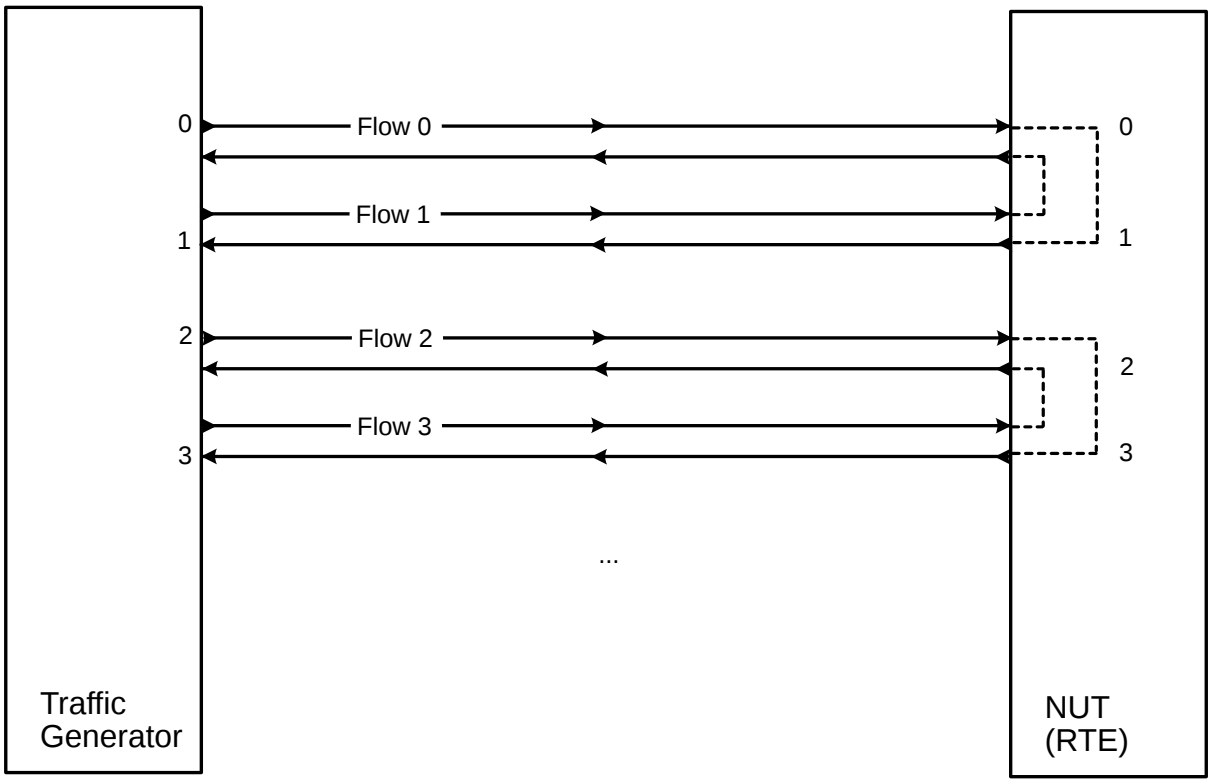


Fig. 18.1: Performance Benchmark Setup (Basic Environment)

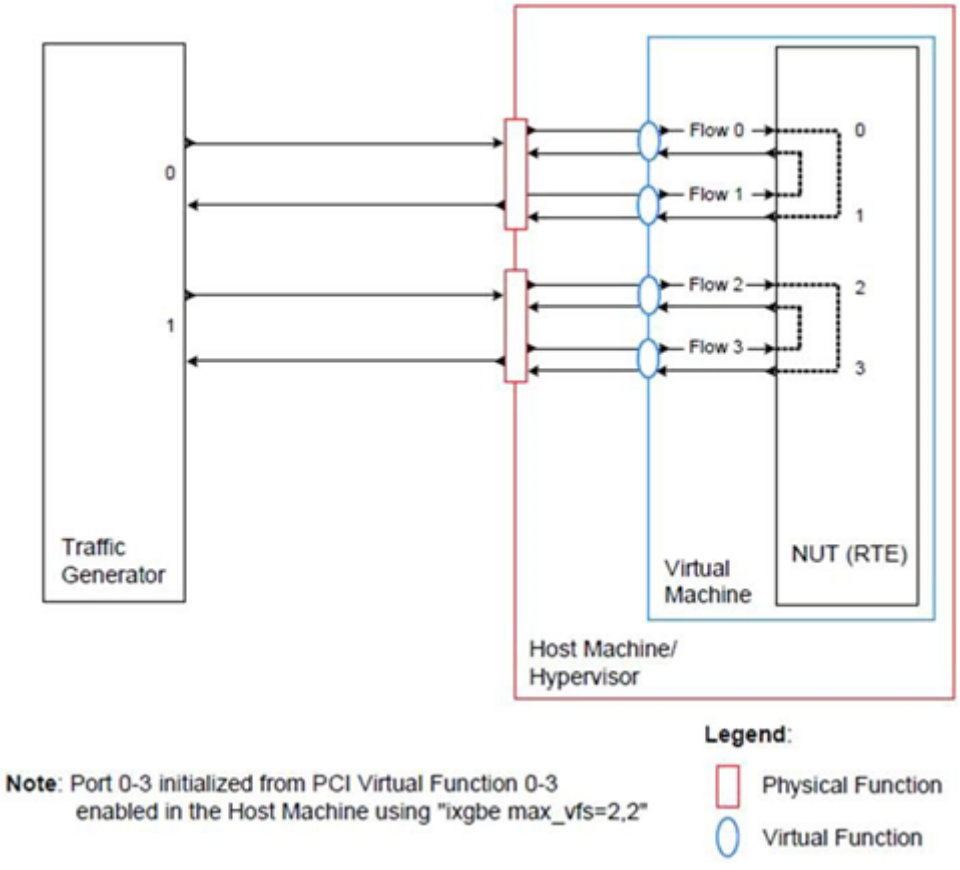


Fig. 18.2: Performance Benchmark Setup (Virtualized Environment)

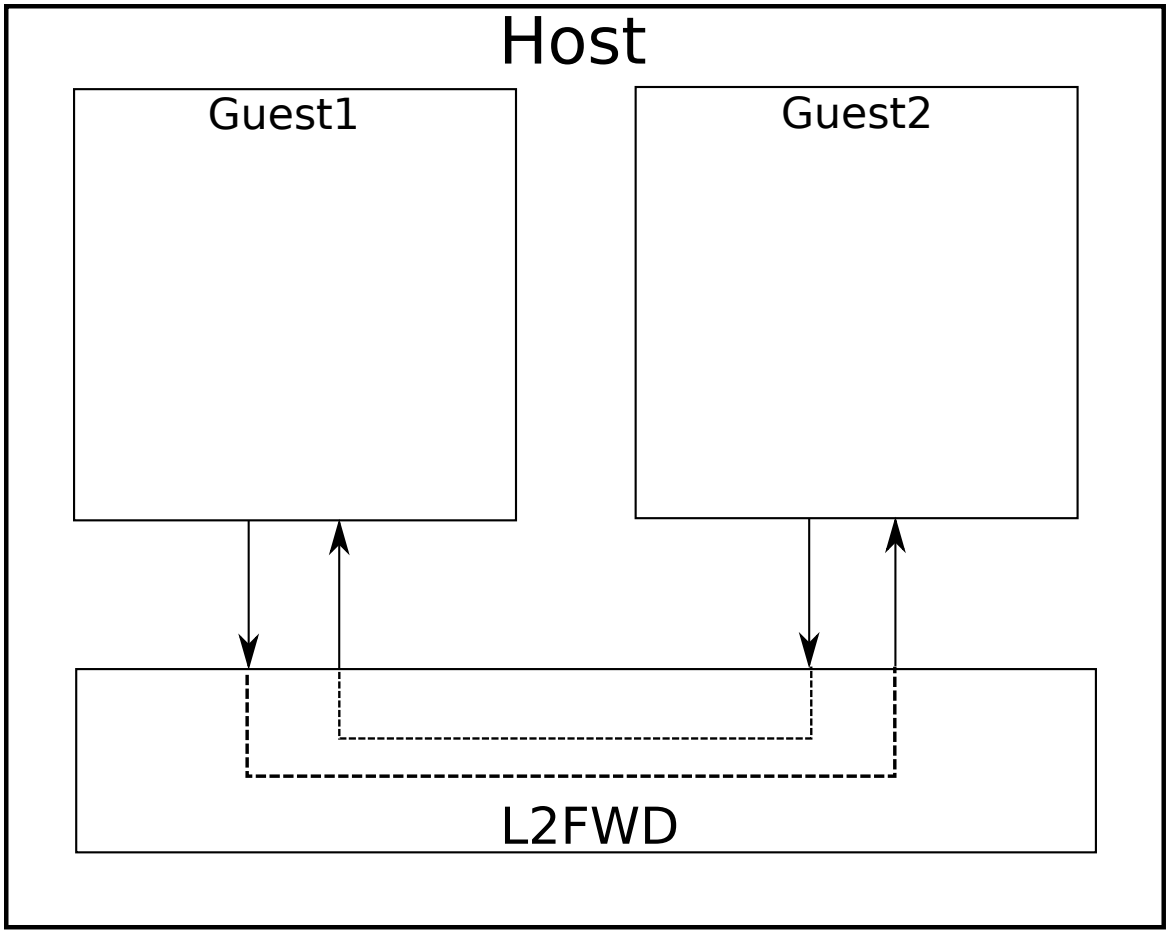


Fig. 18.3: Virtual Machine to Virtual Machine communication.

```
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.

18.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `l2fwd` sub-directory.

18.3 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 linux 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.

18.4 Explanation

The following sections provide some explanation of the code.

18.4.1 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");
```

18.4.2 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.

18.4.3 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_pci_probe() < 0)
    rte_exit(EXIT_FAILURE, "Cannot probe PCI\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.
 */
```

```
RTE_ETH_FOREACH_DEV(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_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);
```

18.4.4 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*).

18.4.5 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_co
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 */
};
```

18.4.6 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 rte_ether_hdr *eth;
    void *tmp;
    unsigned dst_port;

    dst_port = l2fwd_dst_ports[portid];

    eth = rte_pktmbuf_mtod(m, struct rte_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 */

    rte_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, uint16_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) po

        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;
}
```

CHAPTER

NINETEEN

L2 FORWARDING EVENTDEV SAMPLE APPLICATION

The L2 Forwarding eventdev sample application is a simple example of packet processing using the Data Plane Development Kit (DPDK) to demonstrate usage of poll and event mode packet I/O mechanism.

19.1 Overview

The L2 Forwarding eventdev sample application, 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=0x0f), 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

Application receives packets from RX_PORT using below mentioned methods:

- Poll mode
- Eventdev mode (default)

This application can be used to benchmark performance using a traffic-generator, as shown in the Fig. 19.1.

19.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `l2fwd-event` sub-directory.

19.3 Running the Application

The application requires a number of command line options:

```
./build/l2fwd-event [EAL options] -- -p PORTMASK [-q NQ] --[no-]mac-updating --mode=MODE --ever
```

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).

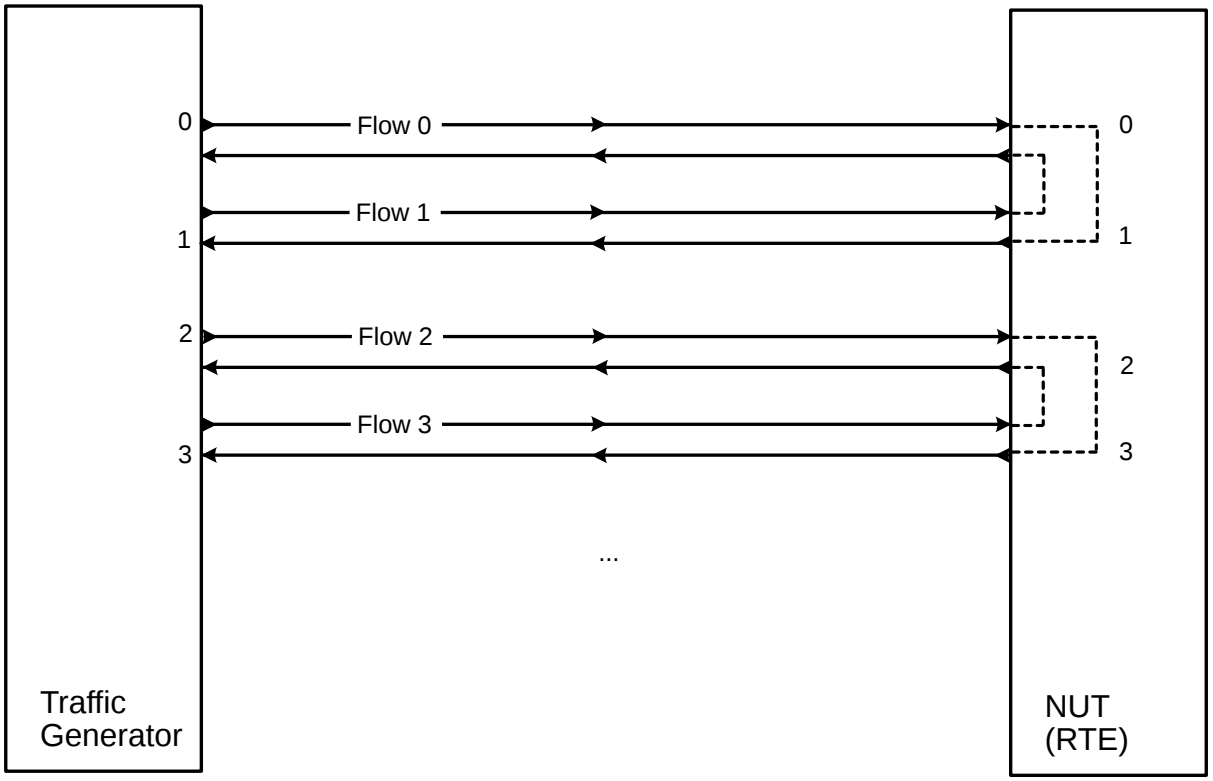


Fig. 19.1: Performance Benchmark Setup (Basic Environment)

- –mode=MODE: Packet transfer mode for I/O, poll or eventdev. Eventdev by default.
- –eventq-sched=SCHED_MODE: Event queue schedule mode, Ordered, Atomic or Parallel. Atomic by default.

Sample usage commands are given below to run the application into different mode:

Poll mode with 4 lcores, 16 ports and 8 RX queues per lcore and MAC address updating enabled, issue the command:

```
./build/l2fwd-event -l 0-3 -n 4 -- -q 8 -p ffff --mode=poll
```

Eventdev mode with 4 lcores, 16 ports , sched method ordered and MAC address updating enabled, issue the command:

```
./build/l2fwd-event -l 0-3 -n 4 -- -p ffff --eventq-sched=ordered
```

or

```
./build/l2fwd-event -l 0-3 -n 4 -- -q 8 -p ffff --mode=eventdev --eventq-sched=ordered
```

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

To run application with S/W scheduler, it uses following DPDK services:

- Software scheduler
- Rx adapter service function
- Tx adapter service function

Application needs service cores to run above mentioned services. Service cores must be provided as EAL parameters along with the –vdev=event_sw0 to enable S/W scheduler. Following is the sample

command:

```
./build/l2fwd-event -l 0-7 -s 0-3 -n 4 --vdev event_sw0 -- -q 8 -p ffff --mode=eventdev --event
```

19.4 Explanation

The following sections provide some explanation of the code.

19.4.1 Command Line Arguments

The L2 Forwarding eventdev 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 for non eventdev parameters and in **parse_eventdev_args()** for eventdev parameters. 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 and eventdev parameters are parsed in eventdev_resource_setup() function during eventdev setup:

```
/* init EAL */

ret = rte_eal_init(argc, argv);
if (ret < 0)
    rte_panic("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_panic("Invalid L2FWD arguments\n");
.
.
.

/* Parse eventdev command line options */
ret = parse_eventdev_args(argc, argv);
if (ret < 0)
    return ret;
```

19.4.2 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.

19.4.3 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 and Event mode Driver in the *DPDK Programmer's Guide* - Rel 1.4 EAR and the *DPDK API Reference*.

```
if (rte_pci_probe() < 0)
    rte_panic("Cannot probe PCI\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.
 */

RTE_ETH_FOREACH_DEV(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_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_panic("Cannot configure device: err=%d, port=%u\n",
                  ret, portid);
```

19.4.4 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_panic("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*).

19.4.5 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_panic("rte_eth_tx_queue_setup:err=%d, port=%u\n",
              ret, (unsigned) portid);
```

To configure eventdev support, application setups following components:

- Event dev
- Event queue
- Event Port
- Rx/Tx adapters
- Ethernet ports

19.4.6 Event device Initialization

Application can use either H/W or S/W based event device scheduler implementation and supports single instance of event device. It configures event device as per below configuration

```
struct rte_event_dev_config event_d_conf = {
     .nb_event_queues = ethdev_count, /* Dedicated to each Ethernet port */
     .nb_event_ports = num_workers, /* Dedicated to each lcore */
     .nb_events_limit  = 4096,
     .nb_event_queue_flows = 1024,
     .nb_event_port_dequeue_depth = 128,
     .nb_event_port_enqueue_depth = 128
};

ret = rte_event_dev_configure(event_d_id, &event_d_conf);
if (ret < 0)
     rte_panic("Error in configuring event device\n");
```

In case of S/W scheduler, application runs eventdev scheduler service on service core. Application retrieves service id and finds the best possible service core to run S/W scheduler.

```
rte_event_dev_info_get(evt_rsrc->event_d_id, &evdev_info);
if (evdev_info.event_dev_cap  & RTE_EVENT_DEV_CAP_DISTRIBUTED_SCHED) {
        ret = rte_event_dev_service_id_get(evt_rsrc->event_d_id,
                        &service_id);
        if (ret != -ESRCH && ret != 0)
                rte_panic("Error in starting eventdev service\n");
        l2fwd_event_service_enable(service_id);
}
```

19.4.7 Event queue Initialization

Each Ethernet device is assigned a dedicated event queue which will be linked to all available event ports i.e. each lcore can dequeue packets from any of the Ethernet ports.

```
struct rte_event_queue_conf event_q_conf = {
     .nb_atomic_flows = 1024,
     .nb_atomic_order_sequences = 1024,
     .event_queue_cfg = 0,
     .schedule_type = RTE_SCHED_TYPE_ATOMIC,
     .priority = RTE_EVENT_DEV_PRIORITY_HIGHEST
};

/* User requested sched mode */
event_q_conf.schedule_type = eventq_sched_mode;
for (event_q_id = 0; event_q_id < ethdev_count; event_q_id++) {
     ret = rte_event_queue_setup(event_d_id, event_q_id,
                                         &event_q_conf);
     if (ret < 0)
           rte_panic("Error in configuring event queue\n");
}
```

In case of S/W scheduler, an extra event queue is created which will be used for Tx adapter service function for enqueue operation.

19.4.8 Event port Initialization

Each worker thread is assigned a dedicated event port for enq/deq operations to/from an event device. All event ports are linked with all available event queues.

```
struct rte_event_port_conf event_p_conf = {
      .dequeue_depth = 32,
      .enqueue_depth = 32,
      .new_event_threshold = 4096
};

for (event_p_id = 0; event_p_id < num_workers; event_p_id++) {
      ret = rte_event_port_setup(event_d_id, event_p_id,
                                 &event_p_conf);
      if (ret < 0)
            rte_panic("Error in configuring event port %d\n", event_p_id);

      ret = rte_event_port_link(event_d_id, event_p_id, NULL,
                                NULL, 0);
      if (ret < 0)
            rte_panic("Error in linking event port %d to queue\n",
                      event_p_id);
}
```

In case of S/W scheduler, an extra event port is created by DPDK library which is retrieved by the application and same will be used by Tx adapter service.

```
ret = rte_event_eth_tx_adapter_event_port_get(tx_adptr_id, &tx_port_id);
if (ret)
        rte_panic("Failed to get Tx adapter port id: %d\n", ret);

ret = rte_event_port_link(event_d_id, tx_port_id,
                          &evt_rsrc.evq.event_q_id[
                                evt_rsrc.evq.nb_queues - 1],
                          NULL, 1);
if (ret != 1)
        rte_panic("Unable to link Tx adapter port to Tx queue:err=%d\n",
                  ret);
```

19.4.9 Rx/Tx adapter Initialization

Each Ethernet port is assigned a dedicated Rx/Tx adapter for H/W scheduler. Each Ethernet port's Rx queues are connected to its respective event queue at priority 0 via Rx adapter configuration and Ethernet port's tx queues are connected via Tx adapter.

```
RTE_ETH_FOREACH_DEV(port_id) {
        if ((rsrc->enabled_port_mask & (1 << port_id)) == 0)
                continue;
        ret = rte_event_eth_rx_adapter_create(adapter_id, event_d_id,
                                              &evt_rsrc->def_p_conf);
        if (ret)
                rte_panic("Failed to create rx adapter[%d]\n",
                          adapter_id);

        /* Configure user requested sched type*/
        eth_q_conf.ev.sched_type = rsrc->sched_type;
        eth_q_conf.ev.queue_id = evt_rsrc->evq.event_q_id[q_id];
        ret = rte_event_eth_rx_adapter_queue_add(adapter_id, port_id,
                                                 -1, &eth_q_conf);
        if (ret)
                rte_panic("Failed to add queues to Rx adapter\n");

        ret = rte_event_eth_rx_adapter_start(adapter_id);
        if (ret)
                rte_panic("Rx adapter[%d] start Failed\n", adapter_id);

        evt_rsrc->rx_adptr.rx_adptr[adapter_id] = adapter_id;
```

```
        adapter_id++;
        if (q_id < evt_rsrc->evq.nb_queues)
                q_id++;
}

adapter_id = 0;
RTE_ETH_FOREACH_DEV(port_id) {
        if ((rsrc->enabled_port_mask & (1 << port_id)) == 0)
                continue;
        ret = rte_event_eth_tx_adapter_create(adapter_id, event_d_id,
                                        &evt_rsrc->def_p_conf);
        if (ret)
                rte_panic("Failed to create tx adapter[%d]\n",
                          adapter_id);

        ret = rte_event_eth_tx_adapter_queue_add(adapter_id, port_id,
                                                 -1);
        if (ret)
                rte_panic("Failed to add queues to Tx adapter\n");

        ret = rte_event_eth_tx_adapter_start(adapter_id);
        if (ret)
                rte_panic("Tx adapter[%d] start Failed\n", adapter_id);

        evt_rsrc->tx_adptr.tx_adptr[adapter_id] = adapter_id;
        adapter_id++;
}
```

For S/W scheduler instead of dedicated adapters, common Rx/Tx adapters are configured which will be shared among all the Ethernet ports. Also DPDK library need service cores to run internal services for Rx/Tx adapters. Application gets service id for Rx/Tx adapters and after successful setup it runs the services on dedicated service cores.

```
for (i = 0; i < evt_rsrc->rx_adptr.nb_rx_adptr; i++) {
        ret = rte_event_eth_rx_adapter_caps_get(evt_rsrc->event_d_id,
                        evt_rsrc->rx_adptr.rx_adptr[i], &caps);
        if (ret < 0)
                rte_panic("Failed to get Rx adapter[%d] caps\n",
                          evt_rsrc->rx_adptr.rx_adptr[i]);
        ret = rte_event_eth_rx_adapter_service_id_get(
                                        evt_rsrc->event_d_id,
                                        &service_id);
        if (ret != -ESRCH && ret != 0)
                rte_panic("Error in starting Rx adapter[%d] service\n",
                          evt_rsrc->rx_adptr.rx_adptr[i]);
        l2fwd_event_service_enable(service_id);
}

for (i = 0; i < evt_rsrc->tx_adptr.nb_tx_adptr; i++) {
        ret = rte_event_eth_tx_adapter_caps_get(evt_rsrc->event_d_id,
                        evt_rsrc->tx_adptr.tx_adptr[i], &caps);
        if (ret < 0)
                rte_panic("Failed to get Rx adapter[%d] caps\n",
                          evt_rsrc->tx_adptr.tx_adptr[i]);
        ret = rte_event_eth_tx_adapter_service_id_get(
                        evt_rsrc->event_d_id,
                        &service_id);
        if (ret != -ESRCH && ret != 0)
                rte_panic("Error in starting Rx adapter[%d] service\n",
                          evt_rsrc->tx_adptr.tx_adptr[i]);
        l2fwd_event_service_enable(service_id);
}
```

19.4.10 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.

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. If number of ports are odd in portmask then packet from last port will be forwarded to first port i.e. if portmask=0x07, then forwarding will take place like p0—>p1, p1—>p2, p2—>p0.

Also to optimize enqueue operation, l2fwd_simple_forward() stores incoming mbufs up to MAX_PKT_BURST. Once it reaches up to limit, all packets are transmitted to destination ports.

```
static void
l2fwd_simple_forward(struct rte_mbuf *m, uint32_t portid)
{
    uint32_t dst_port;
    int32_t sent;
    struct rte_eth_dev_tx_buffer *buffer;

    dst_port = l2fwd_dst_ports[portid];

    if (mac_updating)
        l2fwd_mac_updating(m, dst_port);

    buffer = tx_buffer[dst_port];
    sent = rte_eth_tx_buffer(dst_port, 0, buffer, m);
    if (sent)
    port_statistics[dst_port].tx += sent;
}
```

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 rte_eth_tx_buffer() 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.

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 (i = 0; i < qconf->n_rx_port; i++) {
                portid = l2fwd_dst_ports[qconf->rx_port_list[i]];
                buffer = tx_buffer[portid];
                sent = rte_eth_tx_buffer_flush(portid, 0,
                                               buffer);
                if (sent)
                        port_statistics[portid].tx += sent;
        }

        /* 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 >= 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;
}
```

In the **l2fwd_event_loop()** function, the main task is to read ingress packets from the event ports. This is done using the following code:

```
/* Read packet from eventdev */
nb_rx = rte_event_dequeue_burst(event_d_id, event_p_id,
                                events, deq_len, 0);
if (nb_rx == 0) {
        rte_pause();
        continue;
}

for (i = 0; i < nb_rx; i++) {
        mbuf[i] = events[i].mbuf;
        rte_prefetch0(rte_pktmbuf_mtod(mbuf[i], void *));
}
```

Before reading packets, deq_len is fetched to ensure correct allowed deq length by the eventdev. The rte_event_dequeue_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_eventdev_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.

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. If number of ports are odd in portmask then packet from last port will be forwarded to first port i.e. if portmask=0x07, then forwarding will take place like p0—>p1, p1—>p2, p2—>p0.

l2fwd_eventdev_forward() does not stores incoming mbufs. Packet will forwarded be to destination ports via Tx adapter or generic event dev enqueue API depending H/W or S/W scheduler is used.

```
nb_tx = rte_event_eth_tx_adapter_enqueue(event_d_id, port_id, ev,
                                         nb_rx);
while (nb_tx < nb_rx && !rsrc->force_quit)
        nb_tx += rte_event_eth_tx_adapter_enqueue(
                        event_d_id, port_id,
                        ev + nb_tx, nb_rx - nb_tx);
```

CHAPTER

TWENTY

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

20.1 Compiling the Application

Note: 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

1. 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
```

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `l2fwd-cat` sub-directory.

20.2 Running the Application

To run the example in a `linux` 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.

20.3 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*.

20.3.1 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.

CHAPTER

TWENTYONE

L3 FORWARDING SAMPLE APPLICATION

The L3 Forwarding application is a simple example of packet processing using DPDK to demonstrate usage of poll and event mode packet I/O mechanism. The application performs L3 forwarding.

21.1 Overview

The application demonstrates the use of the hash and LPM libraries in the DPDK to implement packet forwarding using poll or event mode PMDs for packet I/O. The initialization and run-time paths are very similar to those of the *L2 Forwarding Sample Application (in Real and Virtualized Environments)* and *L2 Forwarding Eventdev Sample Application*. The main difference from the L2 Forwarding sample application is that optionally packet can be Rx/Tx from/to eventdev instead of port directly and forwarding decision is made based on information read from the input packet.

Eventdev can optionally use S/W or H/W (if supported by platform) scheduler implementation for packet I/O based on run time parameters.

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.

21.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `l3fwd` sub-directory.

21.3 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]
                         [--per-port-pool]
                         [--mode]
                         [--eventq-sched]
                         [--event-eth-rxqs]
```

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.
- `--per-port-pool`: Optional, set to use independent buffer pools per port. Without this option, single buffer pool is used for all ports.
- `--mode`: Optional, Packet transfer mode for I/O, poll or eventdev.
- `--eventq-sched`: Optional, Event queue synchronization method, Ordered, Atomic or Parallel. Only valid if –mode=eventdev.
- `--event-eth-rxqs`: Optional, Number of ethernet RX queues per device. Only valid if –mode=eventdev.

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.

To use eventdev mode with sync method **ordered** on above mentioned environment, Following is the sample command:

```
./build/l3fwd -l 0-3 -n 4 -w <event device> -- -p 0x3 --eventq-sched=ordered
```

or

```
./build/l3fwd -l 0-3 -n 4 -w <event device> -- -p 0x03 --mode=eventdev --eventq-sched=ordered
```

In this command:

- -w option whitelist the event device supported by platform. Way to pass this device may vary based on platform.
- The –mode option defines PMD to be used for packet I/O.
- The –eventq-sched option enables synchronization menthod of event queue so that packets will be scheduled accordingly.

If application uses S/W scheduler, it uses following DPDK services:

- Software scheduler
- Rx adapter service function
- Tx adapter service function

Application needs service cores to run above mentioned services. Service cores must be provided as EAL parameters along with the –vdev=event_sw0 to enable S/W scheduler. Following is the sample command:

```
./build/l3fwd -l 0-7 -s 0xf0000 -n 4 --vdev event_sw0 -- -p 0x3 --mode=eventdev --eventq-sched=
```

In case of eventdev mode, *–config* option is not used for ethernet port configuration. Instead each ethernet port will be configured with mentioned setup:

- Single Rx/Tx queue
- Each Rx queue will be connected to event queue via Rx adapter.
- Each Tx queue will be connected via Tx adapter.

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

21.4 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)* and *L2 Forwarding Eventdev Sample Application*. The following sections describe aspects that are specific to the L3 Forwarding sample application.

21.4.1 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_ent
            } else {
                /* populate the ipv6 hash */
                populate_ipv6_many_flow_into_table( ipv6_l3fwd_lookup_struct[socketid], hash_er
            }
        } 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
```

21.4.2 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,

    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_c

        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_l3fwc
    }
}
#endif
```

21.4.3 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, uint16_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 rte_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], uint16_t portid, struct lcore_conf *qconf)
{
    // ...

    data[0] = _mm_loadu_si128(( m128i*)(rte_pktmbuf_mtod(m[0], unsigned char *) + sizeof(struct
    data[1] = _mm_loadu_si128(( m128i*)(rte_pktmbuf_mtod(m[1], unsigned char *) + sizeof(struct
    data[2] = _mm_loadu_si128(( m128i*)(rte_pktmbuf_mtod(m[2], unsigned char *) + sizeof(struct
    data[3] = _mm_loadu_si128(( m128i*)(rte_pktmbuf_mtod(m[3], unsigned char *) + sizeof(struct

    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.

21.4.4 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 uint16_t
get_ipv4_dst_port(struct rte_ipv4_hdr *ipv4_hdr, uint16_t portid, lookup_struct_t *ipv4_l3fwd_l
```

```
{
    uint8_t next_hop;

    return ((rte_lpm_lookup(ipv4_l3fwd_lookup_struct, rte_be_to_cpu_32(ipv4_hdr->dst_addr), &ne
}
```

21.4.5 Eventdev Driver Initialization

Eventdev driver initialization is same as L2 forwarding eventdev application. Refer *L2 Forwarding Eventdev Sample Application* for more details.

CHAPTER

TWENTYTWO

L3 FORWARDING WITH POWER MANAGEMENT SAMPLE APPLICATION

22.1 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.

22.2 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.

22.3 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `l3fwd-power` sub-directory.

22.4 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,
```

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
- –empty-poll: Traffic Aware power management. See below for details
- –telemetry: Telemetry mode.

See *L3 Forwarding Sample Application* for details. The L3fwd-power example reuses the L3fwd command line options.

22.5 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.

22.5.1 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, portid;
    unsigned lcore_id;
    uint64_t hz;
    uint32_t n_tx_queue, nb_lcores;
    uint8_t 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,

        // ...
    }

    // ...
}
```

22.5.2 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_
            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_qu
                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;
            }
        }
    }
```

22.5.3 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.

22.5.4 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.

22.6 Empty Poll Mode

Additionally, there is a traffic aware mode of operation called "Empty Poll" where the number of empty polls can be monitored to keep track of how busy the application is. Empty poll mode can be enabled by the command line option –empty-poll.

See Power Management chapter in the DPDK Programmer's Guide for empty poll mode details.

```
./l3fwd-power -l xxx   -n 4   -w 0000:xx:00.0 -w 0000:xx:00.1 -- -p 0x3 -P --config="(0,0,xx),(
```

Where,

–empty-poll: Enable the empty poll mode instead of original algorithm

–empty-poll="training_flag, med_threshold, high_threshold"

- `training_flag` : optional, enable/disable training mode. Default value is 0. If the training_flag is set as 1(true), then the application will start in training mode and print out the trained threshold values. If the training_flag is set as 0(false), the application will start in normal mode, and will use either the default thresholds or those supplied on the command line. The trained threshold values are specific to the user's system, may give a better power profile when compared to the default threshold values.
- `med_threshold` : optional, sets the empty poll threshold of a modestly busy system state. If this is not supplied, the application will apply the default value of 350000.
- `high_threshold` : optional, sets the empty poll threshold of a busy system state. If this is not supplied, the application will apply the default value of 580000.
- -l : optional, set up the LOW power state frequency index
- -m : optional, set up the MED power state frequency index
- -h : optional, set up the HIGH power state frequency index

22.6.1 Empty Poll Mode Example Usage

To initially obtain the ideal thresholds for the system, the training mode should be run first. This is achieved by running the l3fwd-power app with the training flag set to "1", and the other parameters set to 0.

```
./examples/l3fwd-power/build/l3fwd-power -l 1-3 -- -p 0x0f --config="(0,0,2),(0,1,3)" --empty-p
```

This will run the training algorithm for x seconds on each core (cores 2 and 3), and then print out the recommended threshold values for those cores. The thresholds should be very similar for each core.

```
POWER: Bring up the Timer
POWER: set the power freq to MED
POWER: Low threshold is 230277
POWER: MED threshold is 335071
POWER: HIGH threshold is 523769
POWER: Training is Complete for 2
POWER: set the power freq to MED
POWER: Low threshold is 236814
POWER: MED threshold is 344567
POWER: HIGH threshold is 538580
POWER: Training is Complete for 3
```

Once the values have been measured for a particular system, the app can then be started without the training mode so traffic can start immediately.

```
./examples/l3fwd-power/build/l3fwd-power -l 1-3 -- -p 0x0f --config="(0,0,2),(0,1,3)" --empty-p
```

22.7 Telemetry Mode

The telemetry mode support for `l3fwd-power` is a standalone mode, in this mode `l3fwd-power` does simple l3fwding along with calculating empty polls, full polls, and busy percentage for each forwarding core. The aggregation of these values of all cores is reported as application level telemetry to metric library for every 500ms from the master core.

The busy percentage is calculated by recording the poll_count and when the count reaches a defined value the total cycles it took is measured and compared with minimum and maximum reference cycles and accordingly busy rate is set to either 0% or 50% or 100%.

Note:

- The CONFIG_RTE_LIBRTE_TELEMETRY should be set in order to get the stats in DPDK telemetry.

```
./examples/l3fwd-power/build/l3fwd-power --telemetry -l 1-3 -- -p 0x0f --config="(0,0,2),(0,1,3
```

The new stats `empty_poll` , `full_poll` and `busy_percent` can be viewed by running the script `/usertools/dpdk-telemetry-client.py` and selecting the menu option `Send for global Metrics`.

CHAPTER

TWENTYTHREE

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.

23.1 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.

23.1.1 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.

23.1.2 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.

23.1.3 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:

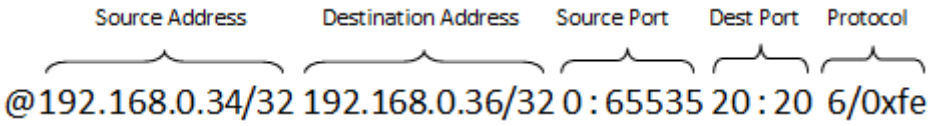


Fig. 23.1: A typical IPv4 ACL rule

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.

23.1.4 Rules File Example

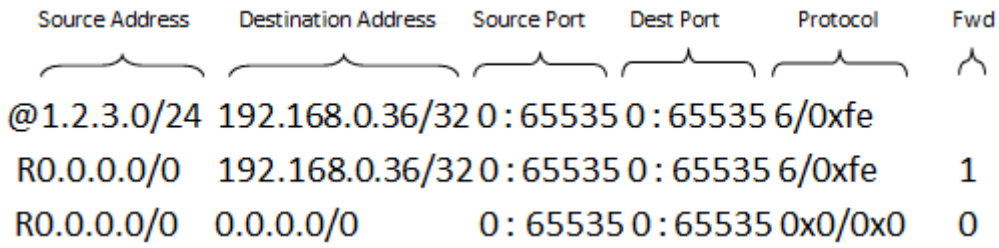


Fig. 23.2: Rules example

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/).

23.1.5 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.

23.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `l3fwd-acl` sub-directory.

23.3 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,lcc
```

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"
```

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.

23.4 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.

23.4.1 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.

23.4.2 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.

CHAPTER

TWENTYFOUR

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.

24.1 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.

24.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `link_status_interrupt` sub-directory.

24.3 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 linux 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.

24.4 Explanation

The following sections provide some explanation of the code.

24.4.1 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.

24.4.2 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.

24.4.3 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_pci_probe() < 0)
    rte_exit(EXIT_FAILURE, "Cannot probe PCI\n");

/*
 * Each logical core is assigned a dedicated TX queue on each port.
 */

RTE_ETH_FOREACH_DEV(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_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,
    },
    .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.

24.4.4 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(uint16_t port_id, enum rte_eth_event_type type, void *param)
{
    struct rte_eth_link link;
    int ret;

    RTE_SET_USED(param);

    printf("\n\nIn registered callback...\n");

    printf("Event type: %s\n", type == RTE_ETH_EVENT_INTR_LSC ? "LSC interrupt" : "unknown ever

    ret = rte_eth_link_get_nowait(port_id, &link);
    if (ret < 0) {
        printf("Failed to get port %d link status: %s\n\n",
               port_id, rte_strerror(-ret));
    } else 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.

24.4.5 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,
    },
};
```

24.4.6 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 */
};
```

24.4.7 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 rte_ether_hdr *eth;
    void *tmp;
    unsigned dst_port = lsi_dst_ports[portid];

    eth = rte_pktmbuf_mtod(m, struct rte_ether_hdr *);

    /* 02:00:00:00:00:xx */

    tmp = &eth->d_addr.addr_bytes[0];
```

```
    *((uint64_t *)tmp) = 0x000000000002 + (dst_port << 40);

    /* src addr */
    rte_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, uint16_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 ofte

    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;
    }
```

CHAPTER

TWENTYFIVE

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).

25.1 Overview

The architecture of the EFD flow-based load balancer sample application is presented in the following figure.

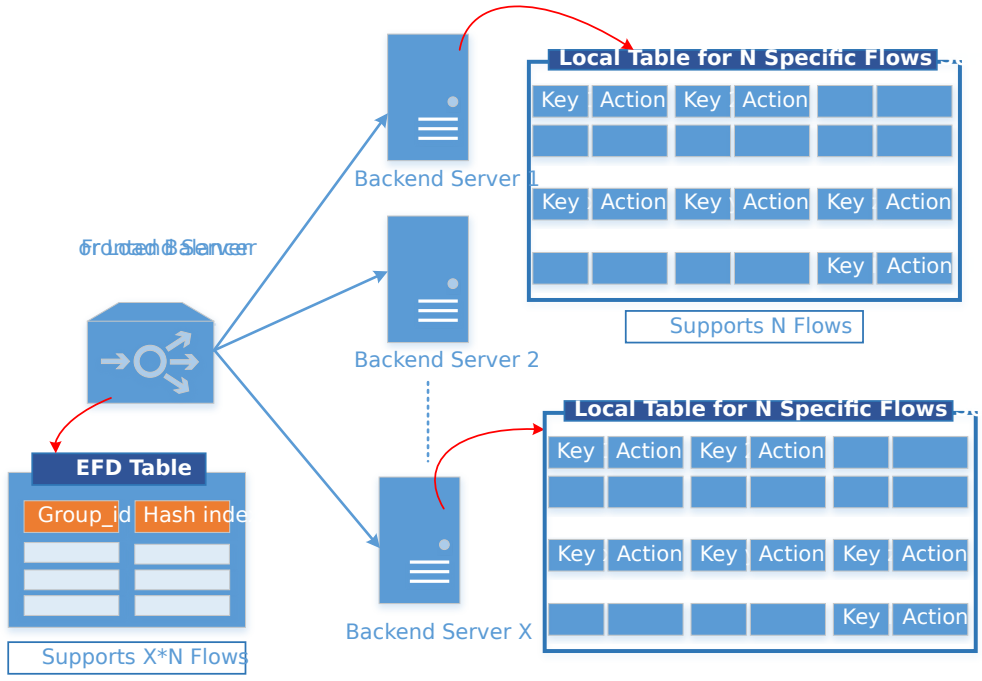


Fig. 25.1: Using EFD as a Flow-Level Load Balancer

As shown in Fig. 25.1, 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.

25.1.1 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).

25.1.2 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.

25.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `server_node_efd` sub-directory.

25.3 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.

25.4 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 rte_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 rte_ipv4_hdr *,
                sizeof(struct rte_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 rte_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 rte_ipv4_hdr *,
                sizeof(struct rte_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");
    }
```

CHAPTER

TWENTYSIX

SERVICE CORES SAMPLE APPLICATION

The service cores sample application demonstrates the service cores capabilities of DPDK. The service cores infrastructure is part of the DPDK EAL, and allows any DPDK component to register a service. A service is a work item or task, that requires CPU time to perform its duty.

This sample application registers 5 dummy services. These 5 services are used to show how the service_cores API can be used to orchestrate these services to run on different service lcores. This orchestration is done by calling the service cores APIs, however the sample application introduces a "profile" concept to contain the service mapping details. Note that the profile concept is application specific, and not a part of the service cores API.

26.1 Compiling the Application

1. Go to the example directory:

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

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

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

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

3. Build the application:

   ```
   make
   ```

26.2 Running the Application

To run the example, just execute the binary. Since the application dynamically adds service cores in the application code itself, there is no requirement to pass a service core-mask as an EAL argument at startup time.

```
$ ./build/service_cores
```

26.3 Explanation

The following sections provide some explanation of code focusing on registering applications from an applications point of view, and modifying the service core counts and mappings at runtime.

26.3.1 Registering a Service

The following code section shows how to register a service as an application. Note that the service component header must be included by the application in order to register services: `rte_service_component.h`, in addition to the ordinary service cores header `rte_service.h` which provides the runtime functions to add, remove and remap service cores.

```
struct rte_service_spec service = {
        .name = "service_name",
};
int ret = rte_service_component_register(services, &id);
if (ret)
        return -1;

/* set the service itself to be ready to run. In the case of
 * ethdev, eventdev etc PMDs, this will be set when the
 * appropriate configure or setup function is called.
 */
rte_service_component_runstate_set(id, 1);

/* Collect statistics for the service */
rte_service_set_stats_enable(id, 1);

/* The application sets the service to running state. Note that this
 * function enables the service to run - while the 'component' version
 * of this function (as above) marks the service itself as ready */
ret = rte_service_runstate_set(id, 1);
```

26.3.2 Controlling A Service Core

This section demonstrates how to add a service core. The `rte_service.h` header file provides the functions for dynamically adding and removing cores. The APIs to add and remove cores use lcore IDs similar to existing DPDK functions.

These are the functions to start a service core, and have it run a service:

```
/* the lcore ID to use as a service core */
uint32_t service_core_id = 7;
ret = rte_service_lcore_add(service_core_id);
if(ret)
        return -1;

/* service cores are in "stopped" state when added, so start it */
ret = rte_service_lcore_start(service_core_id);
if(ret)
        return -1;

/* map a service to the service core, causing it to run the service */
uint32_t service_id; /* ID of a registered service */
uint32_t enable = 1; /* 1 maps the service, 0 unmaps */
ret = rte_service_map_lcore_set(service_id, service_core_id, enable);
if(ret)
        return -1;
```

26.3.3 Removing A Service Core

To remove a service core, the steps are similar to adding but in reverse order. Note that it is not allowed to remove a service core if the service is running, and the service-core is the only core running that service (see documentation for `rte_service_lcore_stop` function for details).

26.3.4 Conclusion

The service cores infrastructure provides DPDK with two main features. The first is to abstract away hardware differences: the service core can CPU cycles to a software fallback implementation, allowing the application to be abstracted from the difference in HW / SW availability. The second feature is a flexible method of registering functions to be run, allowing the running of the functions to be scaled across multiple CPUs.

CHAPTER

TWENTYSEVEN

MULTI-PROCESS SAMPLE APPLICATION

This chapter describes the example applications for multi-processing that are included in the DPDK.

27.1 Example Applications

27.1.1 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 compile the sample application see *Compiling the Sample Applications*.

The applications are located in the `multi_process` sub-directory.

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.

27.1.2 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: Core 11 is ready (tid=84ffe700)
Starting core 11
simple_mp > send hello_secondary
simple_mp > core 11: Received 'hello_primary'
simple_mp > quit
```

```
EAL: Master core 8 is ready (tid=864a3820
EAL: Core 9 is ready (tid=85995700)
Starting core 9
simple_mp > core 9: Received 'hello_secon
simple_mp > send hello_primary
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_
} 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.

27.1.3 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.

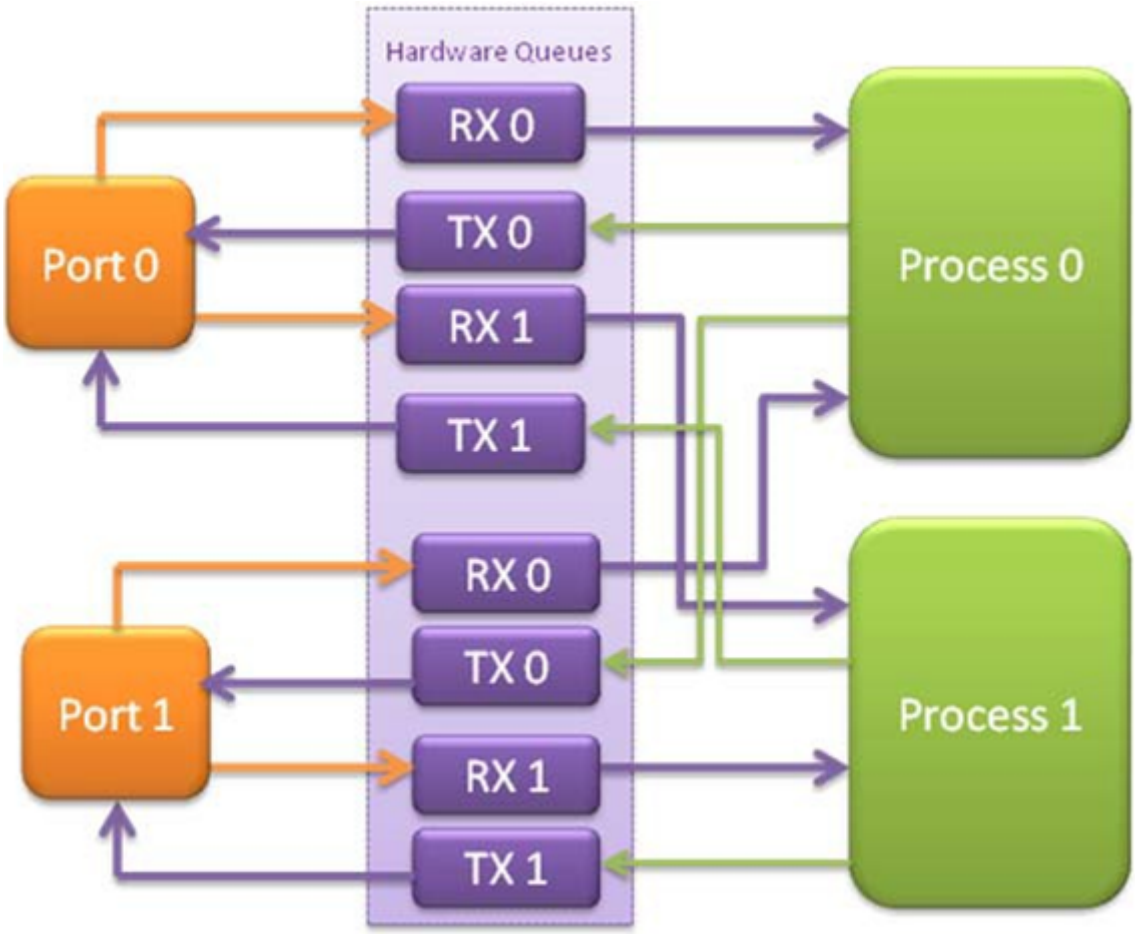


Fig. 27.1: Example Data Flow in a Symmetric Multi-process Application

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.

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_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_creat
```

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.

27.1.4 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.

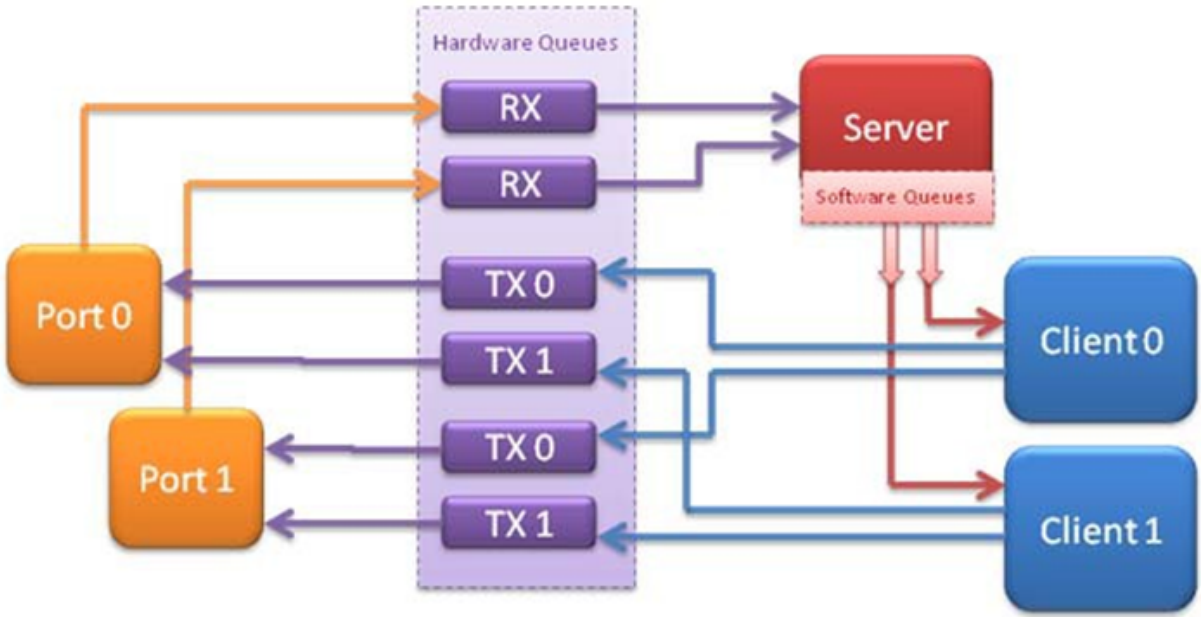


Fig. 27.2: Example Data Flow in a Client-Server Symmetric Multi-process Application

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 coremask/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.

CHAPTER

TWENTYEIGHT

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.

28.1 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.

28.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `qos_meter` sub-directory.

28.3 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.

28.4 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 28.1: 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.

Note:

- In color blind mode, first row GREEN color is only valid.
- To drop the packet, policer_table action has to be set to DROP.

CHAPTER

TWENTYNINE

QOS SCHEDULER SAMPLE APPLICATION

The QoS sample application demonstrates the use of the DPDK to provide QoS scheduling.

29.1 Overview

The architecture of the QoS scheduler application is shown in the following figure.

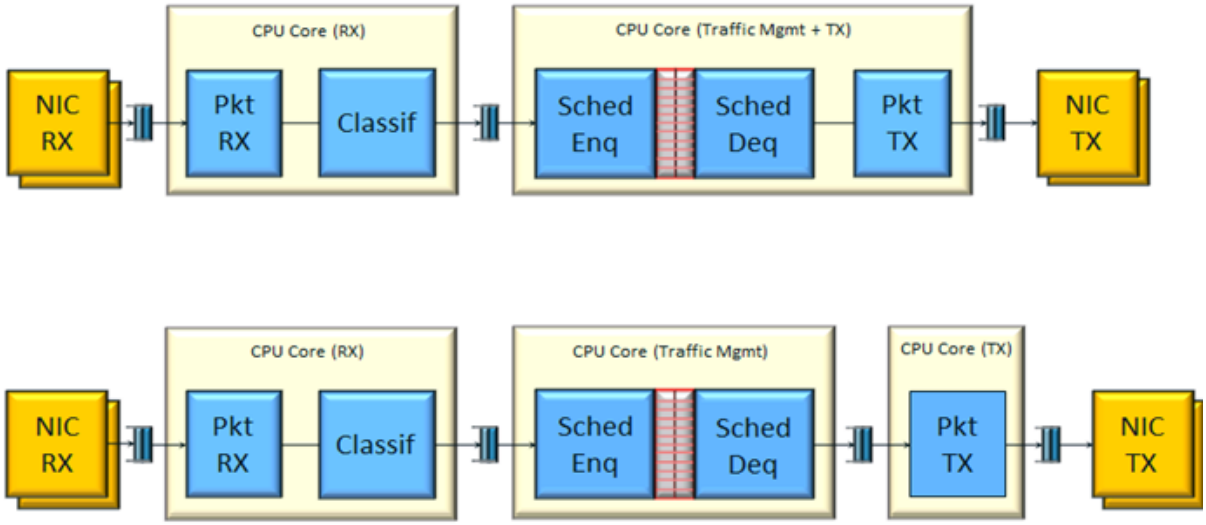


Fig. 29.1: QoS Scheduler Application Architecture

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 byte 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.

29.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `qos_sched` sub-directory.

Note: This application is intended as a linux only.

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.

29.3 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

; Subport configuration

[subport 0]
number of pipes per subport = 4096
queue sizes = 64 64 64 64 64 64 64 64 64 64 64 64 64
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 4 rate = 1250000000;     Bytes per second
tc 5 rate = 1250000000;     Bytes per second
tc 6 rate = 1250000000;     Bytes per second
tc 7 rate = 1250000000;     Bytes per second
tc 8 rate = 1250000000;     Bytes per second
tc 9 rate = 1250000000;     Bytes per second
tc 10 rate = 1250000000;     Bytes per second
tc 11 rate = 1250000000;     Bytes per second
tc 12 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 4 rate = 305175; Bytes per second
tc 5 rate = 305175; Bytes per second
tc 6 rate = 305175; Bytes per second
tc 7 rate = 305175; Bytes per second
tc 8 rate = 305175; Bytes per second
tc 9 rate = 305175; Bytes per second
tc 10 rate = 305175; Bytes per second
tc 11 rate = 305175; Bytes per second
tc 12 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 4 oversubscription weight = 1
tc 5 oversubscription weight = 1
tc 6 oversubscription weight = 1
tc 7 oversubscription weight = 1
tc 8 oversubscription weight = 1
tc 9 oversubscription weight = 1
tc 10 oversubscription weight = 1
tc 11 oversubscription weight = 1
tc 12 oversubscription weight = 1

tc 12 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

tc 4 wred min = 48 40 32
tc 4 wred max = 64 64 64
tc 4 wred inv prob = 10 10 10
tc 4 wred weight = 9 9 9

tc 5 wred min = 48 40 32
tc 5 wred max = 64 64 64
tc 5 wred inv prob = 10 10 10
tc 5 wred weight = 9 9 9

tc 6 wred min = 48 40 32
tc 6 wred max = 64 64 64
tc 6 wred inv prob = 10 10 10
```

```
tc 6 wred weight = 9 9 9

tc 7 wred min = 48 40 32
tc 7 wred max = 64 64 64
tc 7 wred inv prob = 10 10 10
tc 7 wred weight = 9 9 9

tc 8 wred min = 48 40 32
tc 8 wred max = 64 64 64
tc 8 wred inv prob = 10 10 10
tc 8 wred weight = 9 9 9

tc 9 wred min = 48 40 32
tc 9 wred max = 64 64 64
tc 9 wred inv prob = 10 10 10
tc 9 wred weight = 9 9 9

tc 10 wred min = 48 40 32
tc 10 wred max = 64 64 64
tc 10 wred inv prob = 10 10 10
tc 10 wred weight = 9 9 9

tc 11 wred min = 48 40 32
tc 11 wred max = 64 64 64
tc 11 wred inv prob = 10 10 10
tc 11 wred weight = 9 9 9

tc 12 wred min = 48 40 32
tc 12 wred max = 64 64 64
tc 12 wred inv prob = 10 10 10
tc 12 wred weight = 9 9 9
```

29.3.1 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.

29.3.2 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.

29.4 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 29.1: 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	13	TCs of the same pipe services in strict priority	Destination IP address (0.0.0.X)
Queue	High Priority TC: 1, Lowest Priority TC: 4	Queue of lowest priority traffic class (Best effort) 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.

CHAPTER

THIRTY

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.

30.1 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `timer` sub-directory.

30.2 Running the Application

To run the example in linux 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.

30.3 Explanation

The following sections provide some explanation of the code.

30.3.1 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.

30.3.2 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);
}
```

CHAPTER

THIRTYONE

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.

31.1 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.

31.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `packet_ordering` sub-directory.

31.3 Running the Application

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

31.3.1 Application Command Line

The application execution command line is:

```
./packet_ordering [EAL options] -- -p PORTMASK [--disable-reorder] [--insight-worker]
```

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.

The insight-worker long option enables output the packet statistics of each worker thread.

CHAPTER

THIRTYTWO

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.

32.1 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. 32.1. 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.

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.

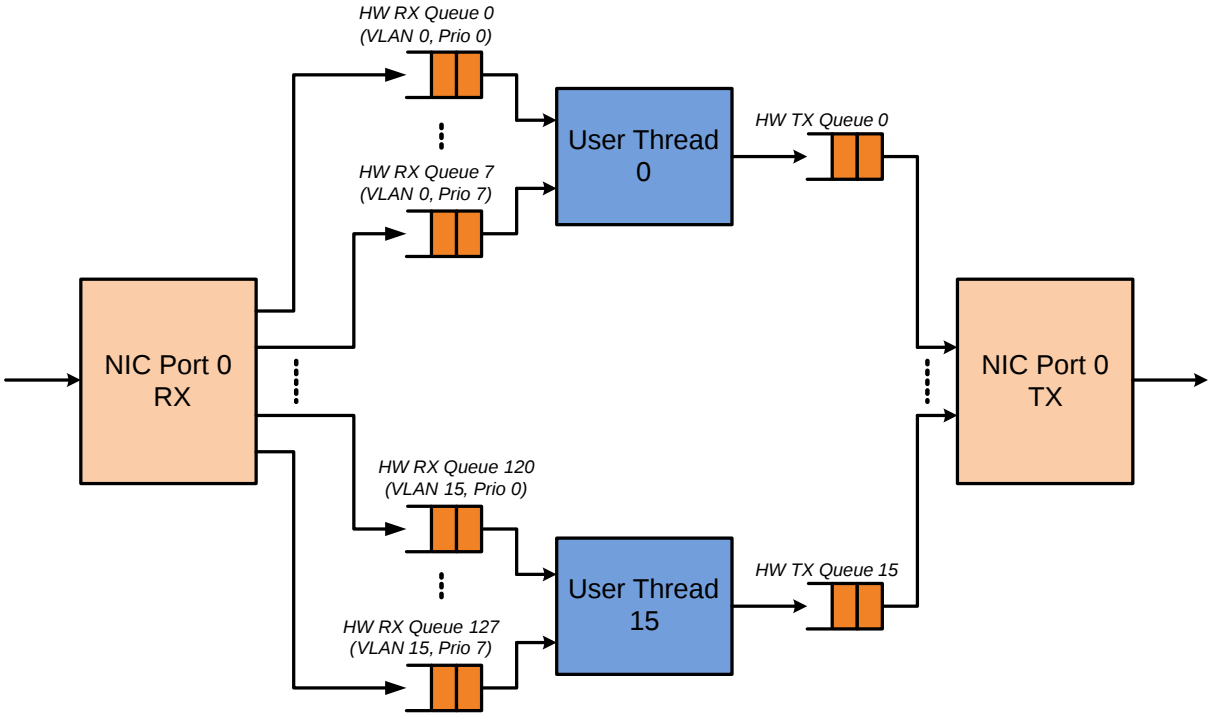


Fig. 32.1: Packet Flow Through the VMDQ and DCB Sample Application

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).

32.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `vmdq_dcb` sub-directory.

32.3 Running the Application

To run the example in a linux 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.

32.4 Explanation

The following sections provide some explanation of the code.

32.4.1 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,
    },
    .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 rte_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 rte_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.

32.4.2 Statistics Display

When run in a linux 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.

CHAPTER

THIRTYTHREE

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.

33.1 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.

33.1.1 Build

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `vhost` sub-directory.

Note: In this example, you need build DPDK both on the host and inside guest.

33.1.2 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.

33.1.3 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.

33.1.4 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
```

33.2 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
```

33.3 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, implying 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. it is worth to note that if NIC is bound to driver with iommu enabled, dequeue zero copy cannot work at VM2NIC mode (vm2vm=0) due to currently we don't setup iommu dma mapping for guest memory.

–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.

–builtin-net-driver A very simple vhost-user net driver which demonstrates how to use the generic vhost APIs will be used when this option is given. It is disabled by default.

33.4 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.

- Failed to build DPDK in VM

  Make sure "-cpu host" QEMU option is given.

- Device start fails if NIC's max queues > the default number of 128

  mbuf pool size is dependent on the MAX_QUEUES configuration, if NIC's max queue number is larger than 128, device start will fail due to insufficient mbuf.

  Change the default number to make it work as below, just set the number according to the NIC's property.

  ```
  make EXTRA_CFLAGS="-DMAX_QUEUES=320"
  ```

- Option "builtin-net-driver" is incompatible with QEMU

  QEMU vhost net device start will fail if protocol feature is not negotiated. DPDK virtio-user pmd can be the replacement of QEMU.

CHAPTER

THIRTYFOUR

VHOST_BLK SAMPLE APPLICATION

The vhost_blk sample application implemented a simple block device, which used as the backend of Qemu vhost-user-blk device. Users can extend the exist example to use other type of block device(e.g. AIO) besides memory based block device. Similar with vhost-user-net device, the sample application used domain socket to communicate with Qemu, and the virtio ring (split or packed format) was processed by vhost_blk sample application.

The sample application reuse lots codes from SPDK(Storage Performance Development Kit, https://github.com/spdk/spdk) vhost-user-blk target, for DPDK vhost library used in storage area, user can take SPDK as reference as well.

34.1 Testing steps

This section shows the steps how to start a VM with the block device as fast data path for critical application.

34.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `examples` sub-directory.

You will also need to build DPDK both on the host and inside the guest

34.2.1 Start the vhost_blk example

```
./vhost_blk -m 1024
```

34.2.2 Start the VM

```
qemu-system-x86_64 -machine accel=kvm \
    -m $mem -object memory-backend-file,id=mem,size=$mem,\
    mem-path=/dev/hugepages,share=on -numa node,memdev=mem \
    -drive file=os.img,if=none,id=disk \
    -device ide-hd,drive=disk,bootindex=0 \
    -chardev socket,id=char0,reconnect=1,path=/tmp/vhost.socket \
    -device vhost-user-blk-pci,ring_packed=1,chardev=char0,num-queues=1 \
    ...
```

Note: You must check whether your Qemu can support "vhost-user-blk" or not, Qemu v4.0 or newer version is required. reconnect=1 means live recovery support that qemu can reconnect vhost_blk after we restart vhost_blk example. ring_packed=1 means the device support packed ring but need the guest kernel version >= 5.0

CHAPTER

THIRTYFIVE

VHOST_CRYPTO SAMPLE APPLICATION

The vhost_crypto sample application implemented a simple Crypto device, which used as the backend of Qemu vhost-user-crypto device. Similar with vhost-user-net and vhost-user-scsi device, the sample application used domain socket to communicate with Qemu, and the virtio ring was processed by vhost_crypto sample application.

35.1 Testing steps

This section shows the steps how to start a VM with the crypto device as fast data path for critical application.

35.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `examples` sub-directory.

35.2.1 Start the vhost_crypto example

```
./vhost_crypto [EAL options] --
            --config (lcore,cdev-id,queue-id)[,(lcore,cdev-id,queue-id)]
            --socket-file lcore,PATH
            [--zero-copy]
            [--guest-polling]
```

where,

- config (lcore,cdev-id,queue-id): build the lcore-cryptodev id-queue id connection. Once specified, the specified lcore will only work with specified cryptodev's queue.
- socket-file lcore,PATH: the path of UNIX socket file to be created and the lcore id that will deal with the all workloads of the socket. Multiple instances of this config item is supported and one lcore supports processing multiple sockets.
- zero-copy: the presence of this item means the ZERO-COPY feature will be enabled. Otherwise it is disabled. PLEASE NOTE the ZERO-COPY feature is still in experimental stage and may cause the problem like segmentation fault. If the user wants to use LKCF in the guest, this feature shall be turned off.
- guest-polling: the presence of this item means the application assumes the guest works in polling mode, thus will NOT notify the guest completion of processing.

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).

35.2.2 Start the VM

```
qemu-system-x86_64 -machine accel=kvm \
    -m $mem -object memory-backend-file,id=mem,size=$mem,\
    mem-path=/dev/hugepages,share=on -numa node,memdev=mem \
    -drive file=os.img,if=none,id=disk \
    -device ide-hd,drive=disk,bootindex=0 \
    -chardev socket,id={chardev_id},path={PATH} \
    -object cryptodev-vhost-user,id={obj_id},chardev={chardev_id} \
    -device virtio-crypto-pci,id={dev_id},cryptodev={obj_id} \
    ...
```

Note: You must check whether your Qemu can support “vhost-user-crypto” or not.

CHAPTER

THIRTYSIX

VDPA SAMPLE APPLICATION

The vdpa sample application creates vhost-user sockets by using the vDPA backend. vDPA stands for vhost Data Path Acceleration which utilizes virtio ring compatible devices to serve virtio driver directly to enable datapath acceleration. As vDPA driver can help to set up vhost datapath, this application doesn't need to launch dedicated worker threads for vhost enqueue/dequeue operations.

36.1 Testing steps

This section shows the steps of how to start VMs with vDPA vhost-user backend and verify network connection & live migration.

36.1.1 Build

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `vdpa` sub-directory.

36.1.2 Start the vdpa example

```
./vdpa [EAL options]  -- [--client] [--interactive|-i] or [--iface SOCKET_PATH]
```

where

- –client means running vdpa app in client mode, in the client mode, QEMU needs to run as the server mode and take charge of socket file creation.
- –iface specifies the path prefix of the UNIX domain socket file, e.g. /tmp/vhost-user-, then the socket files will be named as /tmp/vhost-user-<n> (n starts from 0).
- –interactive means run the vdpa sample in interactive mode, currently 4 internal cmds are supported:
  1. help: show help message
  2. list: list all available vdpa devices
  3. create: create a new vdpa port with socket file and vdpa device address
  4. quit: unregister vhost driver and exit the application

Take IFCVF driver for example:

```
./vdpa -c 0x2 -n 4 --socket-mem 1024,1024 \
        -w 0000:06:00.3,vdpa=1 -w 0000:06:00.4,vdpa=1 \
        -- --interactive
```

Note: Here 0000:06:00.3 and 0000:06:00.4 refer to virtio ring compatible devices, and we need to bind vfio-pci to them before running vdpa sample.

- modprobe vfio-pci
- ./usertools/dpdk-devbind.py -b vfio-pci 06:00.3 06:00.4

Then we can create 2 vdpa ports in interactive cmdline.

```
vdpa> list
device id       device address  queue num       supported features
0               0000:06:00.3    1               0x14c238020
1               0000:06:00.4    1               0x14c238020
2               0000:06:00.5    1               0x14c238020

vdpa> create /tmp/vdpa-socket0 0000:06:00.3
vdpa> create /tmp/vdpa-socket1 0000:06:00.4
```

36.1.3 Start the VMs

```
qemu-system-x86_64 -cpu host -enable-kvm \
<snip>
-mem-prealloc \
-chardev socket,id=char0,path=<socket_file created in above steps> \
-netdev type=vhost-user,id=vdpa,chardev=char0 \
-device virtio-net-pci,netdev=vdpa,mac=00:aa:bb:cc:dd:ee,page-per-vq=on \
```

After the VMs launches, we can login the VMs and configure the ip, verify the network connection via ping or netperf.

Note: Suggest to use QEMU 3.0.0 which extends vhost-user for vDPA.

36.1.4 Live Migration

vDPA supports cross-backend live migration, user can migrate SW vhost backend VM to vDPA backend VM and vice versa. Here are the detailed steps. Assume A is the source host with SW vhost VM and B is the destination host with vDPA.

1. Start vdpa sample and launch a VM with exact same parameters as the VM on A, in migration-listen mode:

```
B: <qemu-command-line> -incoming tcp:0:4444 (or other PORT))
```

2. Start the migration (on source host):

```
A: (qemu) migrate -d tcp:<B ip>:4444 (or other PORT)
```

3. Check the status (on source host):

```
A: (qemu) info migrate
```

CHAPTER

THIRTYSEVEN

INTERNET PROTOCOL (IP) PIPELINE APPLICATION

37.1 Application overview

The *Internet Protocol (IP) Pipeline* application is intended to be a vehicle for rapid development of packet processing applications on multi-core CPUs.

Following OpenFlow and P4 design principles, the application can be used to create functional blocks called pipelines out of input/output ports, tables and actions in a modular way. Multiple pipelines can be inter-connected through packet queues to create complete applications (super-pipelines).

The pipelines are mapped to application threads, with each pipeline executed by a single thread and each thread able to run one or several pipelines. The possibilities of creating pipelines out of ports, tables and actions, connecting multiple pipelines together and mapping the pipelines to execution threads are endless, therefore this application can be seen as a true application generator.

Pipelines are created and managed through Command Line Interface (CLI):

- Any standard TCP client (e.g. telnet, netcat, custom script, etc) is typically able to connect to the application, send commands through the network and wait for the response before pushing the next command.
- **All the application objects are created and managed through CLI commands:**
  - 'Primitive' objects used to create pipeline ports: memory pools, links (i.e. network interfaces), SW queues, traffic managers, etc.
  - Action profiles: used to define the actions to be executed by pipeline input/output ports and tables.
  - Pipeline components: input/output ports, tables, pipelines, mapping of pipelines to execution threads.

37.2 Running the application

The application startup command line is:

```
ip_pipeline [EAL_ARGS] -- [-s SCRIPT_FILE] [-h HOST] [-p PORT]
```

The application startup arguments are:

`-s SCRIPT_FILE`

- Optional: Yes
- Default: Not present

- Argument: Path to the CLI script file to be run at application startup. No CLI script file will run at startup if this argument is not present.

`-h HOST`

- Optional: Yes
- Default: `0.0.0.0`
- Argument: IP Address of the host running ip pipeline application to be used by remote TCP based client (telnet, netcat, etc.) for connection.

`-p PORT`

- Optional: Yes
- Default: `8086`
- Argument: TCP port number at which the ip pipeline is running. This port number should be used by remote TCP client (such as telnet, netcat, etc.) to connect to host application.

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

The following is an example command to run ip pipeline application configured for layer 2 forwarding:

```
$ ./build/ip_pipeline -c 0x3 -- -s examples/route_ecmp.cli
```

The application should start successfully and display as follows:

```
EAL: Detected 40 lcore(s)
EAL: Detected 2 NUMA nodes
EAL: Multi-process socket /var/run/.rte_unix
EAL: Probing VFIO support...
EAL: PCI device 0000:02:00.0 on NUMA socket 0
EAL:   probe driver: 8086:10fb net_ixgbe
...
```

To run remote client (e.g. telnet) to communicate with the ip pipeline application:

```
$ telnet 127.0.0.1 8086
```

When running a telnet client as above, command prompt is displayed:

```
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.

Welcome to IP Pipeline!

pipeline>
```

Once application and telnet client start running, messages can be sent from client to application. At any stage, telnet client can be terminated using the quit command.

37.3 Application stages

37.3.1 Initialization

During this stage, EAL layer is initialised and application specific arguments are parsed. Furthermore, the data structures (i.e. linked lists) for application objects are initialized. In case of any initialization error, an error message is displayed and the application is terminated.

37.3.2 Run-time

The master thread is creating and managing all the application objects based on CLI input.

Each data plane thread runs one or several pipelines previously assigned to it in round-robin order. Each data plane thread executes two tasks in time-sharing mode:

1. *Packet processing task*: Process bursts of input packets read from the pipeline input ports.
2. *Message handling task*: Periodically, the data plane thread pauses the packet processing task and polls for request messages send by the master thread. Examples: add/remove pipeline to/from current data plane thread, add/delete rules to/from given table of a specific pipeline owned by the current data plane thread, read statistics, etc.

37.4 Examples

Table 37.1: Pipeline examples provided with the application

Name	Table(s)	Actions	Messages
L2fwd Note: Implemented using pipeline with a simple pass-through connection between input and output ports.	Stub	Forward	1. Mempool create 2. Link create 3. Pipeline create 4. Pipeline port in/out 5. Pipeline table 6. Pipeline port in table 7. Pipeline enable 8. Pipeline table rule add
Flow classification	Exact match • **Key = byte array** (16 bytes) • Offset = 278 • Table size = 64K	Forward	1. Mempool create 2. Link create 3. Pipeline create 4. Pipeline port in/out 5. Pipeline table 6. Pipeline port in table 7. Pipeline enable 8. Pipeline table rule add default 9. Pipeline table rule add
KNI	Stub	Forward	1. Mempool create 2. Link create 3. Pipeline create 4. Pipeline port in/out 5. Pipeline table 6. Pipeline port in table 7. Pipeline enable 8. Pipeline table rule add
Firewall	ACL • Key = n-tuple • Offset = 270 • Table size = 4K	Allow/Drop	1. Mempool create 2. Link create 3. Pipeline create 4. Pipeline port in/out 5. Pipeline table 6. Pipeline port in table 7. Pipeline enable 8. Pipeline table rule add default 9. Pipeline table rule add

37.5 Command Line Interface (CLI)

37.5.1 Link

Link configuration

```
link <link_name>
 dev <device_name>|port <port_id>
 rxq <n_queues> <queue_size> <mempool_name>
 txq <n_queues> <queue_size> promiscuous on | off
 [rss <qid_0> ... <qid_n>]
```

Note: The PCI device name must be specified in the Domain:Bus:Device.Function format.

37.5.2 Mempool

Mempool create

```
mempool <mempool_name> buffer <buffer_size>
pool <pool_size> cache <cache_size> cpu <cpu_id>
```

37.5.3 Software queue

Create software queue

```
swq <swq_name> size <size> cpu <cpu_id>
```

37.5.4 Traffic manager

Add traffic manager subport profile

```
tmgr subport profile
 <tb_rate> <tb_size>
 <tc0_rate> <tc1_rate> <tc2_rate> <tc3_rate> <tc4_rate>
 <tc5_rate> <tc6_rate> <tc7_rate> <tc8_rate>
 <tc9_rate> <tc10_rate> <tc11_rate> <tc12_rate>
 <tc_period>
 pps <n_pipes_per_subport>
 qsize <qsize_tc0> <qsize_tc1> <qsize_tc2>
 <qsize_tc3> <qsize_tc4> <qsize_tc5> <qsize_tc6>
 <qsize_tc7> <qsize_tc8> <qsize_tc9> <qsize_tc10>
 <qsize_tc11> <qsize_tc12>
```

Add traffic manager pipe profile

```
tmgr pipe profile
 <tb_rate> <tb_size>
 <tc0_rate> <tc1_rate> <tc2_rate> <tc3_rate> <tc4_rate>
 <tc5_rate> <tc6_rate> <tc7_rate> <tc8_rate>
 <tc9_rate> <tc10_rate> <tc11_rate> <tc12_rate>
 <tc_period>
 <tc_ov_weight>
 <wrr_weight0..3>
```

Create traffic manager port

```
tmgr <tmgr_name>
 rate <rate>
 spp <n_subports_per_port>
```

```
fo <frame_overhead>
mtu <mtu>
cpu <cpu_id>
```

Configure traffic manager subport

```
tmgr <tmgr_name>
 subport <subport_id>
 profile <subport_profile_id>
```

Configure traffic manager pipe

```
tmgr <tmgr_name>
 subport <subport_id>
 pipe from <pipe_id_first> to <pipe_id_last>
 profile <pipe_profile_id>
```

37.5.5 Tap

Create tap port

```
tap <name>
```

37.5.6 Kni

Create kni port

```
kni <kni_name>
 link <link_name>
 mempool <mempool_name>
 [thread <thread_id>]
```

37.5.7 Cryptodev

Create cryptodev port

```
cryptodev <cryptodev_name>
 dev <DPDK Cryptodev PMD name>
 queue <n_queues> <queue_size>
```

37.5.8 Action profile

Create action profile for pipeline input port

```
port in action profile <profile_name>
 [filter match | mismatch offset <key_offset> mask <key_mask> key <key_value> port <port_i
 [balance offset <key_offset> mask <key_mask> port <port_id0> ... <port_id15>]
```

Create action profile for the pipeline table

```
table action profile <profile_name>
 ipv4 | ipv6
 offset <ip_offset>
 fwd
 [balance offset <key_offset> mask <key_mask> outoffset <out_offset>]
 [meter srtcm | trtcm
     tc <n_tc>
     stats none | pkts | bytes | both]
 [tm spp <n_subports_per_port> pps <n_pipes_per_subport>]
```

```
        [encap ether | vlan | qinq | mpls | pppoe]
        [nat src | dst
            proto udp | tcp]
        [ttl drop | fwd
            stats none | pkts]
        [stats pkts | bytes | both]
        [sym_crypto cryptodev <cryptodev_name>
            mempool_create <mempool_name> mempool_init <mempool_name>]
        [time]
```

37.5.9 Pipeline

Create pipeline

```
pipeline <pipeline_name>
 period <timer_period_ms>
 offset_port_id <offset_port_id>
 cpu <cpu_id>
```

Create pipeline input port

```
pipeline <pipeline_name> port in
 bsz <burst_size>
 link <link_name> rxq <queue_id>
 | swq <swq_name>
 | tmgr <tmgr_name>
 | tap <tap_name> mempool <mempool_name> mtu <mtu>
 | kni <kni_name>
 | source mempool <mempool_name> file <file_name> bpp <n_bytes_per_pkt>
 [action <port_in_action_profile_name>]
 [disabled]
```

Create pipeline output port

```
pipeline <pipeline_name> port out
 bsz <burst_size>
 link <link_name> txq <txq_id>
 | swq <swq_name>
 | tmgr <tmgr_name>
 | tap <tap_name>
 | kni <kni_name>
 | sink [file <file_name> pkts <max_n_pkts>]
```

Create pipeline table

```
pipeline <pipeline_name> table
     match
     acl
         ipv4 | ipv6
         offset <ip_header_offset>
         size <n_rules>
     | array
         offset <key_offset>
         size <n_keys>
     | hash
         ext | lru
         key <key_size>
         mask <key_mask>
         offset <key_offset>
         buckets <n_buckets>
         size <n_keys>
     | lpm
         ipv4 | ipv6
```

```
          offset <ip_header_offset>
          size <n_rules>
      | stub
  [action <table_action_profile_name>]
```

Connect pipeline input port to table

```
pipeline <pipeline_name> port in <port_id> table <table_id>
```

Display statistics for specific pipeline input port, output port or table

```
pipeline <pipeline_name> port in <port_id> stats read [clear]
pipeline <pipeline_name> port out <port_id> stats read [clear]
pipeline <pipeline_name> table <table_id> stats read [clear]
```

Enable given input port for specific pipeline instance

```
pipeline <pipeline_name> port out <port_id> disable
```

Disable given input port for specific pipeline instance

```
pipeline <pipeline_name> port out <port_id> disable
```

Add default rule to table for specific pipeline instance

```
pipeline <pipeline_name> table <table_id> rule add
   match
      default
   action
      fwd
         drop
         | port <port_id>
         | meta
         | table <table_id>
```

Add rule to table for specific pipeline instance

```
pipeline <pipeline_name> table <table_id> rule add

match
   acl
      priority <priority>
      ipv4 | ipv6 <sa> <sa_depth> <da> <da_depth>
      <sp0> <sp1> <dp0> <dp1> <proto>
   | array <pos>
   | hash
      raw <key>
      | ipv4_5tuple <sa> <da> <sp> <dp> <proto>
      | ipv6_5tuple <sa> <da> <sp> <dp> <proto>
      | ipv4_addr <addr>
      | ipv6_addr <addr>
      | qinq <svlan> <cvlan>
   | lpm
      ipv4 | ipv6 <addr> <depth>

action
   fwd
      drop
      | port <port_id>
      | meta
      | table <table_id>
   [balance <out0> ... <out7>]
   [meter
      tc0 meter <meter_profile_id> policer g <pa> y <pa> r <pa>
      [tc1 meter <meter_profile_id> policer g <pa> y <pa> r <pa>
      tc2 meter <meter_profile_id> policer g <pa> y <pa> r <pa>
```

```
      tc3 meter <meter_profile_id> policer g <pa> y <pa> r <pa>]]
   [tm subport <subport_id> pipe <pipe_id>]
   [encap
      ether <da> <sa>
      | vlan <da> <sa> <pcp> <dei> <vid>
      | qinq <da> <sa> <pcp> <dei> <vid> <pcp> <dei> <vid>
      | mpls unicast | multicast
         <da> <sa>
         label0 <label> <tc> <ttl>
         [label1 <label> <tc> <ttl>
         [label2 <label> <tc> <ttl>
         [label3 <label> <tc> <ttl>]]]
      | pppoe <da> <sa> <session_id>]
   [nat ipv4 | ipv6 <addr> <port>]
   [ttl dec | keep]
   [stats]
   [time]
   [sym_crypto
      encrypt | decrypt
      type
      | cipher
         cipher_algo <algo> cipher_key <key> cipher_iv <iv>
      | cipher_auth
         cipher_algo <algo> cipher_key <key> cipher_iv <iv>
         auth_algo <algo> auth_key <key> digest_size <size>
      | aead
         aead_algo <algo> aead_key <key> aead_iv <iv> aead_aad <aad>
         digest_size <size>
      data_offset <data_offset>]

where:
   <pa> ::= g | y | r | drop
```

Add bulk rules to table for specific pipeline instance

```
pipeline <pipeline_name> table <table_id> rule add bulk <file_name> <n_rules>

Where:
- file_name = path to file
- File line format = match <match> action <action>
```

Delete table rule for specific pipeline instance

```
pipeline <pipeline_name> table <table_id> rule delete
   match <match>
```

Delete default table rule for specific pipeline instance

```
pipeline <pipeline_name> table <table_id> rule delete
   match
      default
```

Add meter profile to the table for specific pipeline instance

```
pipeline <pipeline_name> table <table_id> meter profile <meter_profile_id>
 add srtcm cir <cir> cbs <cbs> ebs <ebs>
 | trtcm cir <cir> pir <pir> cbs <cbs> pbs <pbs>
```

Delete meter profile from the table for specific pipeline instance

```
pipeline <pipeline_name> table <table_id>
 meter profile <meter_profile_id> delete
```

Update the dscp table for meter or traffic manager action for specific pipeline instance

```
pipeline <pipeline_name> table <table_id> dscp <file_name>

Where:
   - file_name = path to file
   - exactly 64 lines
   - File line format = <tc_id> <tc_queue_id> <color>, with <color> as: g | y | r
```

37.5.10 Pipeline enable/disable

Enable given pipeline instance for specific data plane thread

```
thread <thread_id> pipeline <pipeline_name> enable
```

Disable given pipeline instance for specific data plane thread

```
thread <thread_id> pipeline <pipeline_name> disable
```

CHAPTER

THIRTYEIGHT

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.

38.1 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.

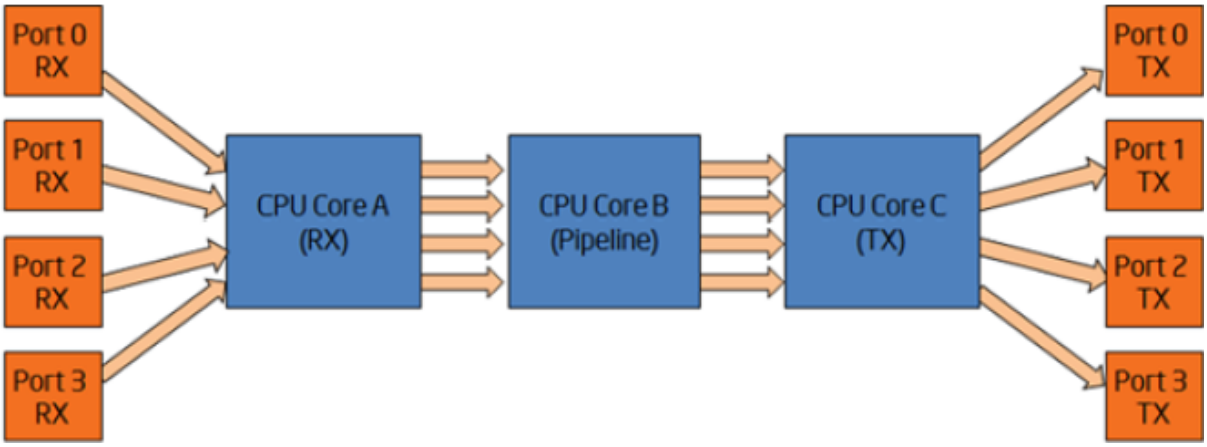


Fig. 38.1: Test Pipeline Application

38.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*

The application is located in the `$RTE_SDK/app/test-pipeline` directory.

38.3 Running the Application

38.3.1 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.

38.3.2 Table Types and Behavior

Table 38.1 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 38.1: 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]

38.3.3 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

CHAPTER

THIRTYNINE

EVENTDEV PIPELINE SAMPLE APPLICATION

The eventdev pipeline sample application is a sample app that demonstrates the usage of the eventdev API using the software PMD. It shows how an application can configure a pipeline and assign a set of worker cores to perform the processing required.

The application has a range of command line arguments allowing it to be configured for various numbers worker cores, stages,queue depths and cycles per stage of work. This is useful for performance testing as well as quickly testing a particular pipeline configuration.

39.1 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `examples` sub-directory.

39.2 Running the Application

The application has a lot of command line options. This allows specification of the eventdev PMD to use, and a number of attributes of the processing pipeline options.

An example eventdev pipeline running with the software eventdev PMD using these settings is shown below:

- `-r1`: core mask 0x1 for RX
- `-t1`: core mask 0x1 for TX
- `-e4`: core mask 0x4 for the software scheduler
- `-w FF00`: core mask for worker cores, 8 cores from 8th to 16th
- `-s4`: 4 atomic stages
- `-n0`: process infinite packets (run forever)
- `-c32`: worker dequeue depth of 32
- `-W1000`: do 1000 cycles of work per packet in each stage
- `-D`: dump statistics on exit

```
./build/eventdev_pipeline --vdev event_sw0 -- -r1 -t1 -e4 -w FF00 -s4 -n0 -c32 -W1000 -D
```

The application has some sanity checking built-in, so if there is a function (e.g.; the RX core) which doesn't have a cpu core mask assigned, the application will print an error message:

```
Core part of pipeline was not assigned any cores. This will stall the
pipeline, please check core masks (use -h for details on setting core masks):
        rx: 0
        tx: 1
```

Configuration of the eventdev is covered in detail in the programmers guide, see the Event Device Library section.

39.3 Observing the Application

At runtime the eventdev pipeline application prints out a summary of the configuration, and some runtime statistics like packets per second. On exit the worker statistics are printed, along with a full dump of the PMD statistics if required. The following sections show sample output for each of the output types.

39.3.1 Configuration

This provides an overview of the pipeline, scheduling type at each stage, and parameters to options such as how many flows to use and what eventdev PMD is in use. See the following sample output for details:

```
Config:
      ports: 2
      workers: 8
      packets: 0
      priorities: 1
      Queue-prio: 0
      qid0 type: atomic
      Cores available: 44
      Cores used: 10
      Eventdev 0: event_sw
Stages:
      Stage 0, Type Atomic    Priority = 128
      Stage 1, Type Atomic    Priority = 128
      Stage 2, Type Atomic    Priority = 128
      Stage 3, Type Atomic    Priority = 128
```

39.3.2 Runtime

At runtime, the statistics of the consumer are printed, stating the number of packets received, runtime in milliseconds, average mpps, and current mpps.

```
# consumer RX= xxxxxxx, time yyyy ms, avg z.zzz mpps [current w.www mpps]
```

39.3.3 Shutdown

At shutdown, the application prints the number of packets received and transmitted, and an overview of the distribution of work across worker cores.

```
Signal 2 received, preparing to exit...
  worker 12 thread done. RX=4966581 TX=4966581
  worker 13 thread done. RX=4963329 TX=4963329
  worker 14 thread done. RX=4953614 TX=4953614
  worker 0 thread done. RX=0 TX=0
  worker 11 thread done. RX=4970549 TX=4970549
  worker 10 thread done. RX=4986391 TX=4986391
  worker 9 thread done. RX=4970528 TX=4970528
```

```
  worker 15 thread done. RX=4974087 TX=4974087
  worker 8 thread done. RX=4979908 TX=4979908
  worker 2 thread done. RX=0 TX=0

Port Workload distribution:
worker 0 :      12.5 % (4979876 pkts)
worker 1 :      12.5 % (4970497 pkts)
worker 2 :      12.5 % (4986359 pkts)
worker 3 :      12.5 % (4970517 pkts)
worker 4 :      12.5 % (4966566 pkts)
worker 5 :      12.5 % (4963297 pkts)
worker 6 :      12.5 % (4953598 pkts)
worker 7 :      12.5 % (4974055 pkts)
```

To get a full dump of the state of the eventdev PMD, pass the `-D` flag to this application. When the app is terminated using `Ctrl+C`, the `rte_event_dev_dump()` function is called, resulting in a dump of the statistics that the PMD provides. The statistics provided depend on the PMD used, see the Event Device Drivers section for a list of eventdev PMDs.

CHAPTER

FORTY

DISTRIBUTOR SAMPLE APPLICATION

The distributor sample application is a simple example of packet distribution to cores using the Data Plane Development Kit (DPDK). It also makes use of Intel Speed Select Technology - Base Frequency (Intel SST-BF) to pin the distributor to the higher frequency core if available.

40.1 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.

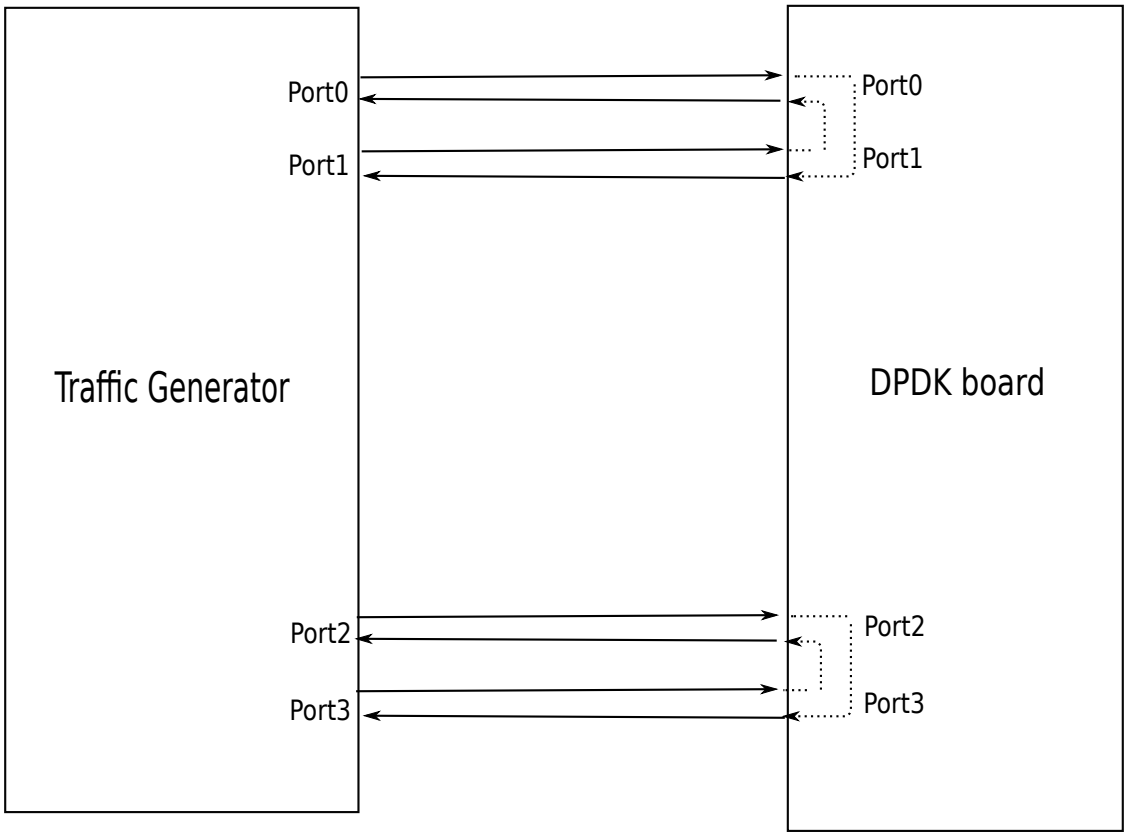


Fig. 40.1: Performance Benchmarking Setup (Basic Environment)

40.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `distributor` sub-directory.

40.3 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 linux 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.

40.4 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. 40.2 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.

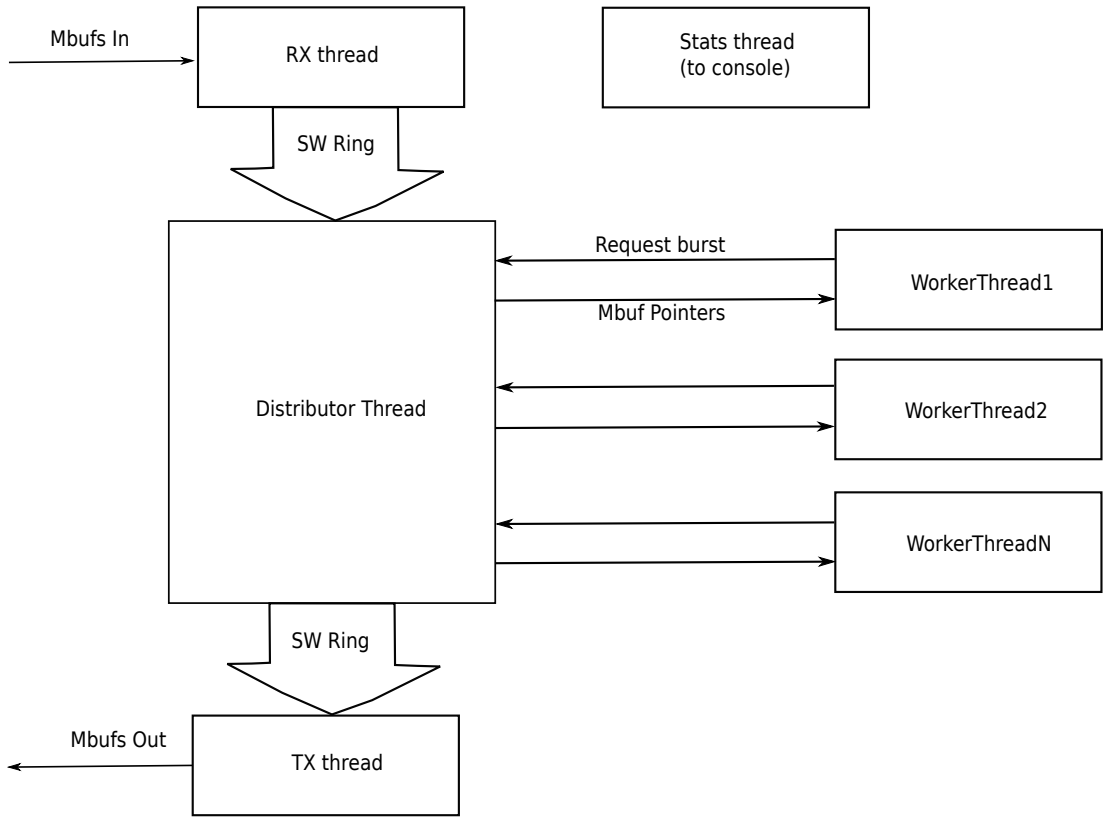


Fig. 40.2: Distributor Sample Application Layout

40.5 Intel SST-BF Support

In DPDK 19.05, support was added to the power management library for Intel-SST-BF, a technology that allows some cores to run at a higher frequency than others. An application note for Intel SST-BF is available, and is entitled Intel Speed Select Technology – Base Frequency - Enhancing Performance

The distributor application was also enhanced to be aware of these higher frequency SST-BF cores, and when starting the application, if high frequency SST-BF cores are present in the core mask, the application will identify these cores and pin the workloads appropriately. The distributor core is usually the bottleneck, so this is given first choice of the high frequency SST-BF cores, followed by the rx core and the tx core.

40.6 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.

40.7 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.

40.8 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*.

CHAPTER

FORTYONE

VIRTUAL MACHINE POWER MANAGEMENT APPLICATION

Applications running in virtual environments have an abstract view of the underlying hardware on the host. Specifically, applications cannot see the binding of virtual components to physical hardware. When looking at CPU resourcing, the pinning of Virtual CPUs (vCPUs) to Physical CPUs (pCPUs) on the host is not apparent to an application and this pinning may change over time. In addition, operating systems on Virtual Machines (VMs) 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 the operating systems running on the VMs.

The solution demonstrated in this sample application shows an example of how a DPDK application can indicate its processing requirements using VM-local only information (vCPU/lcore, and so on) to a host resident VM Power Manager. The VM Power Manager is responsible for:

- **Accepting requests for frequency changes for a vCPU**
- **Translating the vCPU to a pCPU using libvirt**
- **Performing the change in frequency**

This application demonstrates the following features:

- **The handling of VM application requests to change frequency.** VM applications can request frequency changes for a vCPU. The VM Power Management Application uses libvirt to translate that virtual CPU (vCPU) request to a physical CPU (pCPU) request and performs the frequency change.
- **The acceptance of power management policies from VM applications.** A VM application can send a policy to the host application. The policy contains rules that define the power management behaviour of the VM. The host application then applies the rules of the policy independent of the VM application. For example, the policy can contain time-of-day information for busy/quiet periods, and the host application can scale up/down the relevant cores when required. See *Command Line Options Available When Sending a Policy to the Host* for information on setting policy values.
- **Out-of-band monitoring of workloads using core hardware event counters.** The host application can manage power for an application by looking at the event counters of the cores and taking action based on the branch miss/hit ratio. See *Command Line Options for Enabling Out-of-band Branch Ratio Monitoring*.

  Note: This functionality also applies in non-virtualised environments.

In addition to the `librte_power` library used on the host, the application uses a special version of `librte_power` on each VM, which directs frequency changes and policies to the host monitor rather than the APCI `cpufreq` `sysfs` interface used on the host in non-virtualised environments.

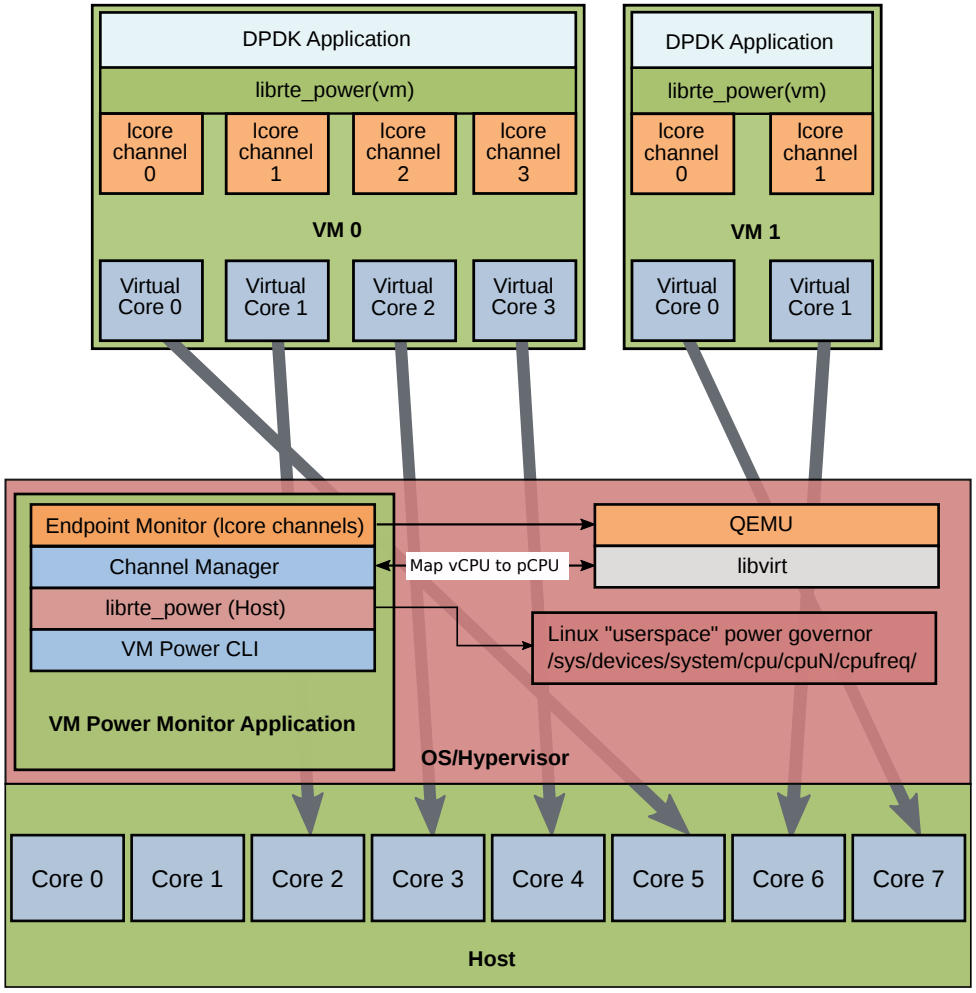


Fig. 41.1: Highlevel Solution

In the above diagram, the DPDK Applications are shown running in virtual machines, and the VM Power Monitor application is shown running in the host.

DPDK VM Application

- Reuse `librte_power` interface, but uses an implementation that forwards frequency requests to the host using a `virtio-serial` channel
- Each lcore has exclusive access to a single channel
- Sample application reuses `l3fwd_power`
- A CLI for changing frequency from within a VM is also included

VM Power Monitor

- Accepts VM commands over `virtio-serial` endpoints, monitored using `epoll`
- Commands include the virtual core to be modified, using `libvirt` to get the physical core mapping
- Uses `librte_power` to affect frequency changes using Linux userspace power governor (`acpi_cpufreq` OR `intel_pstate` driver)
- CLI: For adding VM channels to monitor, inspecting and changing channel state, manually altering CPU frequency. Also allows for the changings of vCPU to pCPU pinning

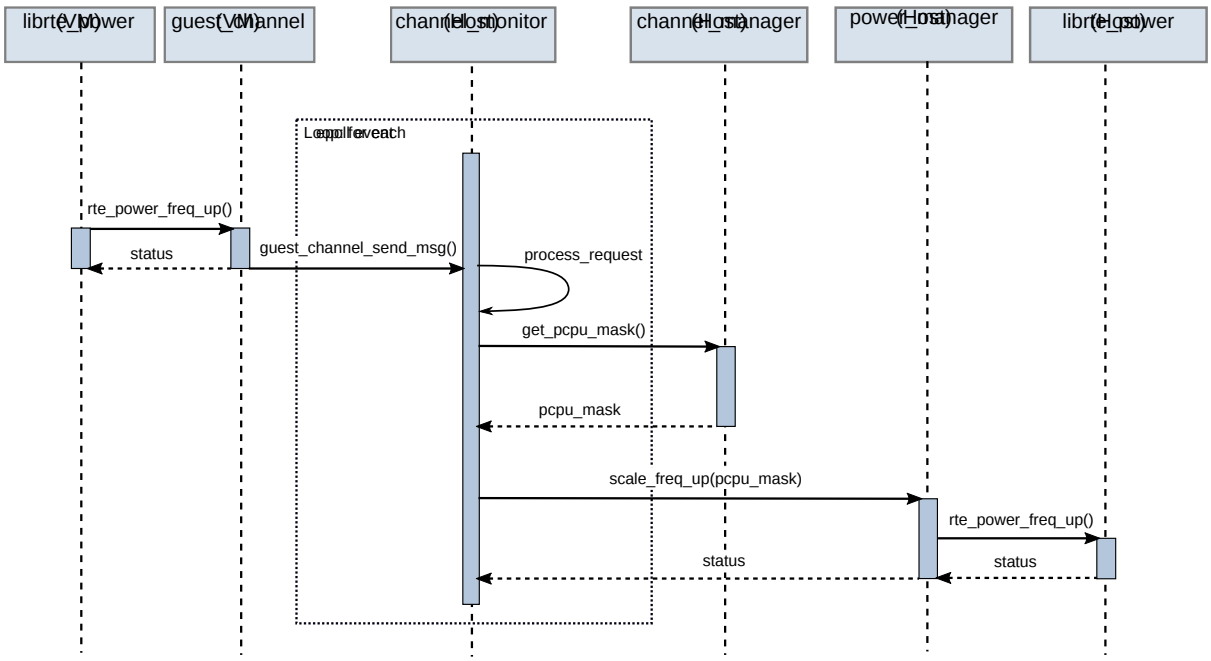
41.1 Sample Application Architecture Overview

The VM power management solution employs `qemu-kvm` to provide communications channels between the host and VMs in the form of a `virtio-serial` connection that appears as a para-virtualised serial device on a VM and can be configured to use various backends on the host. For this example, the configuration of each `virtio-serial` endpoint on the host as an `AF_UNIX` file socket, supporting poll/select and `epoll` for event notification. In this example, each channel endpoint on the host is monitored for `EPOLLIN` events using `epoll`. Each channel is specified as `qemu-kvm` arguments or as `libvirt` XML for each VM, where each VM can have several 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, the VM forwards a `librte_power` request over the `virtio-serial` channel to the host. Each request contains the vCPU and power command (scale up/down/min/max). The API for the host `librte_power` and guest `librte_power` is consistent across environments, with the selection of VM or host implementation determined automatically at runtime based on the environment. On receiving a request, the host translates the vCPU to a pCPU using the libvirt API before forwarding it to the host `librte_power`.

In addition to the ability to send power management requests to the host, a VM can send a power management policy to the host. In some cases, using a power management policy is a preferred option because it can eliminate possible latency issues that can occur when sending power management requests. Once the VM sends the policy to the host, the VM no longer needs to worry about power management, because the host now manages the power for the VM based on the policy. The policy can specify power behavior that is based on incoming traffic rates or time-of-day power adjustment (busy/quiet hour power adjustment for example). See *Command Line Options Available When Sending a Policy to the Host* for more information.

One method of power management is to sense how busy a core is when processing packets and adjusting power accordingly. One technique for doing this is to monitor the ratio of the branch miss to branch hits



counters and scale the core power accordingly. This technique is based on the premise that when a core is not processing packets, the ratio of branch misses to branch hits is very low, but when the core is processing packets, it is measurably higher. The implementation of this capability is as a policy of type `BRANCH_RATIO`. See *Command Line Options Available When Sending a Policy to the Host* for more information on using the BRANCH_RATIO policy option.

A JSON interface enables the specification of power management requests and policies in JSON format. The JSON interfaces provide a more convenient and more easily interpreted interface for the specification of requests and policies. See *JSON Interface for Power Management Requests and Policies* for more information.

41.1.1 Performance Considerations

While the Haswell microarchitecture allows for independent power control for each core, earlier microarchitectures do not offer such fine-grained control. When deploying on pre-Haswell platforms, greater care must be taken when selecting which cores are assigned to a VM, for example, a core does not scale down in frequency until all of its siblings are similarly scaled down.

41.2 Configuration

41.2.1 BIOS

To use the power management features of the DPDK, you must enable Enhanced Intel SpeedStep® Technology in the platform BIOS. Otherwise, the `sys` file folder `/sys/devices/system/cpu/cpu0/cpufreq` does not exist, and you cannot use CPU frequency-based power management. Refer to the relevant BIOS documentation to determine how to access these settings.

41.2.2 Host Operating System

The DPDK Power Management library can use either the `acpi_cpufreq` or the `intel_pstate` kernel driver for the management of core frequencies. In many cases, the `intel_pstate` driver is the default power management environment.

Should the `acpi-cpufreq driver` be required, the `intel_pstate` module must be disabled, and the `acpi-cpufreq` module loaded in its place.

To disable the `intel_pstate` driver, add the following to the `grub` Linux command line:

```
intel_pstate=disable
```

On reboot, load the `acpi_cpufreq` module:

```
modprobe acpi_cpufreq
```

41.2.3 Hypervisor Channel Configuration

Configure `virtio-serial` channels using `libvirt` XML. The XML structure is as follows:

```
<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, 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.

On the host, for each channel to appear in the path, ensure the creation of the `/tmp/powermonitor/` directory and the assignment of `qemu` permissions:

```
mkdir /tmp/powermonitor/
chown qemu:qemu /tmp/powermonitor
```

Note that files and directories in `/tmp` are generally removed when rebooting the host and you may need to perform the previous steps 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: `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 is present at:

`/dev/virtio-ports/virtio.serial.port.poweragent.{vm_channel_num}`

41.3 Compiling and Running the Host Application

41.3.1 Compiling the Host Application

For information on compiling the DPDK and sample applications, see see *Compiling the Sample Applications*.

The application is located in the `vm_power_manager` subdirectory.

To build just the `vm_power_manager` application using `make`:

```
export RTE_SDK=/path/to/rte_sdk
export RTE_TARGET=build
cd ${RTE_SDK}/examples/vm_power_manager/
make
```

The resulting binary is `${RTE_SDK}/build/examples/vm_power_manager`.

To build just the `vm_power_manager` application using `meson`/`ninja`:

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}
meson build
cd build
ninja
meson configure -Dexamples=vm_power_manager
ninja
```

The resulting binary is `${RTE_SDK}/build/examples/dpdk-vm_power_manager`.

41.3.2 Running the Host Application

The application does not have any specific command line options other than the EAL options:

```
./build/vm_power_mgr [EAL options]
```

The application requires exactly two cores to run. One core for the CLI and the other for the channel endpoint monitor. For example, to run on cores 0 and 1 on a system with four memory channels, issue the command:

```
./build/vm_power_mgr -l 0-1 -n 4
```

After successful initialization, the VM Power Manager CLI prompt appears:

```
vm_power>
```

Now, it is possible to add virtual machines 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 a 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 to add them.

It is possible to issue several commands from the CLI to manage VMs.

Remove the virtual machine identified by `{vm_name}` from the VM Power Manager using the command:

```
rm_vm {vm_name}
```

Add communication channels for the specified VM using the following command. 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. Specifying the keyword `all` attempts to add all channels for the VM:

```
 set_pcpu {vm_name} {vcpu} {pcpu}

Enable query of physical core information from a VM:

set_query {vm_name} enable|disable
```

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/ma

add_channels {vm_name} {list}|all
```

Enable or disable the communication channels in `{list}` (comma-separated) for the specified VM. Alternatively, replace `list` with the keyword `all`. Disabled channels receive packets on the host. However, the commands they specify are ignored. Set the status to enabled to begin processing requests again:

```
set_channel_status {vm_name} {list}|all enabled|disabled
```

Print to the CLI 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, and the status of each channel:

```
show_vm {vm_name}
```

Set the binding of a virtual CPU on a VM with name `{vm_name}` to the physical CPU mask:

```
set_pcpu_mask {vm_name} {vcpu} {pcpu}
```

Set the binding of the virtual CPU on the VM to the physical CPU:

```
      set_pcpu {vm_name} {vcpu} {pcpu}
```

It is also possible to perform manual control and inspection in relation to 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
```

41.3.3 Command Line Options for Enabling Out-of-band Branch Ratio Monitoring

There are a couple of command line parameters for enabling the out-of-band monitoring of branch ratios on cores doing busy polling using PMDs as described below:

--core-list {list of cores} Specify the list of cores to monitor the ratio of branch misses to branch hits. A tightly-polling PMD thread has a very low branch ratio, therefore the core frequency scales down to the minimum allowed value. On receiving packets, the code path changes, causing the branch ratio to increase. When the ratio goes above the ratio threshold, the core frequency scales up to the maximum allowed value.

--branch-ratio {ratio} Specify a floating-point number that identifies the threshold at which to scale up or down for the given workload. The default branch ratio is 0.01 and needs adjustment for different workloads.

41.4 Compiling and Running the Guest Applications

It is possible to use the `l3fwd-power` application (for example) with the `vm_power_manager`.

The distribution also provides a guest CLI for validating the setup.

For both `l3fwd-power` and the guest CLI, the host application must use the `add_channels` command to monitor the channels for the VM. To do this, issue the following commands in the host application:

```
vm_power> add_vm vmname
vm_power> add_channels vmname all
vm_power> set_channel_status vmname all enabled
vm_power> show_vm vmname
```

41.4.1 Compiling the Guest Application

For information on compiling DPDK and the sample applications in general, see *Compiling the Sample Applications*.

For compiling and running the `l3fwd-power` sample application, see *L3 Forwarding with Power Management Sample Application*.

The application is in the `guest_cli` subdirectory under `vm_power_manager`.

To build just the `guest_vm_power_manager` application using `make`, issue the following commands:

```
export RTE_SDK=/path/to/rte_sdk
export RTE_TARGET=build
cd ${RTE_SDK}/examples/vm_power_manager/guest_cli/
make
```

The resulting binary is `${RTE_SDK}/build/examples/guest_cli`.

Note: This sample application conditionally links in the Jansson JSON library. Consequently, if you are using a multilib or cross-compile environment, you may need to set the `PKG_CONFIG_LIBDIR` environmental variable to point to the relevant `pkgconfig` folder so that the correct library is linked in.

For example, if you are building for a 32-bit target, you could find the correct directory using the following find command:

```
# find /usr -type d -name pkgconfig
/usr/lib/i386-linux-gnu/pkgconfig
/usr/lib/x86_64-linux-gnu/pkgconfig
```

Then use:

```
export PKG_CONFIG_LIBDIR=/usr/lib/i386-linux-gnu/pkgconfig
```

You then use the `make` command as normal, which should find the 32-bit version of the library, if it installed. If not, the application builds without the JSON interface functionality.

To build just the `vm_power_manager` application using `meson`/`ninja`:

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}
meson build
cd build
ninja
meson configure -Dexamples=vm_power_manager/guest_cli
ninja
```

The resulting binary is `${RTE_SDK}/build/examples/guest_cli`.

41.4.2 Running the Guest Application

The standard EAL command line parameters are necessary:

```
./build/vm_power_mgr [EAL options] -- [guest options]
```

The guest example uses a channel for each lcore enabled. For example, to run on cores 0, 1, 2 and 3:

```
./build/guest_vm_power_mgr -l 0-3
```

41.4.3 Command Line Options Available When Sending a Policy to the Host

Optionally, there are several command line options for a user who needs to send a power policy to the host application:

`--vm-name {name of guest vm}` Allows the user to change the virtual machine name passed down to the host application using the power policy. The default is ubuntu2.

`--vcpu-list {list vm cores}` A comma-separated list of cores in the VM that the user wants the host application to monitor. The list of cores in any VM starts at zero, and the host application maps these to the physical cores once the policy passes down to the host. Valid syntax includes individual cores 2,3,4, a range of cores 2-4, or a combination of both 1,3,5-7.

`--busy-hours {list of busy hours}` A comma-separated list of hours in which to set the core frequency to the maximum. Valid syntax includes individual hours 2,3,4, a range of hours 2-4, or a combination of both 1,3,5-7. Valid hour values are 0 to 23.

`--quiet-hours {list of quiet hours}` A comma-separated list of hours in which to set the core frequency to minimum. Valid syntax includes individual hours 2,3,4, a range of hours 2-4, or a combination of both 1,3,5-7. Valid hour values are 0 to 23.

`--policy {policy type}` The type of policy. This can be one of the following values:

- TRAFFIC - Based on incoming traffic rates on the NIC.
- TIME - Uses a busy/quiet hours policy.
- BRANCH_RATIO - Uses branch ratio counters to determine core busyness.
- WORKLOAD - Sets the frequency to low, medium or high based on the received policy setting.

Note: Not all policy types need all parameters. For example, BRANCH_RATIO only needs the vcpu-list parameter.

After successful initialization, the VM Power Manager Guest CLI prompt appears:

```
vm_power(guest)>
```

To change the frequency of an lcore, use a `set_cpu_freq` command similar to the following:

```
set_cpu_freq {core_num} up|down|min|max
```

where, `{core_num}` is the lcore and channel to change frequency by scaling up/down/min/max.

To start an application, configure the power policy, and send it to the host, use a command like the following:

```
./build/guest_vm_power_mgr -l 0-3 -n 4 -- --vm-name=ubuntu --policy=BRANCH_RATIO --vcpu-list=2-
```

Once the VM Power Manager Guest CLI appears, issuing the 'send_policy now' command will send the policy to the host:

```
send_policy now
```

Once the policy is sent to the host, the host application takes over the power monitoring of the specified cores in the policy.

41.5 JSON Interface for Power Management Requests and Policies

In addition to the command line interface for the host command, and a `virtio-serial` interface for VM power policies, there is also a JSON interface through which power commands and policies can be sent.

Note: This functionality adds a dependency on the Jansson library. Install the Jansson development package on the system to avail of the JSON parsing functionality in the app. Issue the `apt-get install libjansson-dev` command to install the development package. The command and package name may be different depending on your operating system. It is worth noting that the app builds successfully if this package is not present, but a warning displays during compilation, and the JSON parsing functionality is not present in the app.

Send a request or policy to the VM Power Manager by simply opening a fifo file at `/tmp/powermonitor/fifo`, writing a JSON string to that file, and closing the file.

The JSON string can be a power management request or a policy, and takes the following format:

```
{"packet_type": {
"pair_1": value,
"pair_2": value
}}
```

The `packet_type` header can contain one of two values, depending on whether a power management request or policy is being sent. The two possible values are `instruction` and `policy` and the expected name-value pairs are different depending on which type is sent.

The pairs are in the format of standard JSON name-value pairs. The value type varies between the different name-value pairs, and may be integers, strings, arrays, and so on. See *JSON Interface Examples* for examples of policies and instructions and *JSON Name-value Pairs* for the supported names and value types.

41.5.1 JSON Interface Examples

The following is an example JSON string that creates a time-profile policy.

```
{"policy": {
"name": "ubuntu",
"command": "create",
"policy_type": "TIME",
"busy_hours":[ 17, 18, 19, 20, 21, 22, 23 ],
"quiet_hours":[ 2, 3, 4, 5, 6 ],
"core_list":[ 11 ]
}}
```

The following is an example JSON string that removes the named policy.

```
{"policy": {
"name": "ubuntu",
"command": "destroy",
}}
```

The following is an example JSON string for a power management request.

```
{"instruction": {
"name": "ubuntu",
"command": "power",
"unit": "SCALE_MAX",
"resource_id": 10
}}
```

To query the available frequences of an lcore, use the query_cpu_freq command. Where {core_num} is the lcore to query. Before using this command, please enable responses via the set_query command on the host.

```
query_cpu_freq {core_num}|all
```

To query the capabilities of an lcore, use the query_cpu_caps command. Where {core_num} is the lcore to query. Before using this command, please enable responses via the set_query command on the host.

```
query_cpu_caps {core_num}|all
```

To start the application and configure the power policy, and send it to the host:

```
./build/guest_vm_power_mgr -l 0-3 -n 4 -- --vm-name=ubuntu --policy=BRANCH_RATIO --vcpu-list=2-
```

Once the VM Power Manager Guest CLI appears, issuing the 'send_policy now' command will send the policy to the host:

```
send_policy now
```

Once the policy is sent to the host, the host application takes over the power monitoring of the specified cores in the policy.

41.5.2 JSON Name-value Pairs

The following are the name-value pairs supported by the JSON interface:

- *avg_packet_thresh*
- *busy_hours*
- *command*
- *core_list*

- *mac_list*
- *max_packet_thresh*
- *name*
- *policy_type*
- *quiet_hours*
- *resource_id*
- *unit*
- *workload*

avg_packet_thresh

Description The threshold below which the frequency is set to the minimum value for the TRAFFIC policy. If the traffic rate is above this value and below the maximum value, the frequency is set to medium.

Type integer

Values The number of packets below which the TRAFFIC policy applies the minimum frequency, or the medium frequency if between the average and maximum thresholds.

Required Yes

Example `"avg_packet_thresh": 100000`

busy_hours

Description The hours of the day in which we scale up the cores for busy times.

Type array of integers

Values An array with a list of hour values (0-23).

Required For the TIME policy only.

Example `"busy_hours":[ 17,18,19,20,21,22,23 ]`

command

Description The type of packet to send to the VM Power Manager. It is possible to create or destroy a policy or send a direct command to adjust the frequency of a core, as is possible on the command line interface.

Type string

Values Possible values are: - CREATE: Create a new policy. - DESTROY: Remove an existing policy. - POWER: Send an immediate command, max, min, and so on.

Required Yes

Example `"command": "CREATE"`

core_list

Description The cores to which to apply a policy.

Type array of integers

Values An array with a list of virtual CPUs.

Required For CREATE/DESTROY policy requests only.

Example `"core_list":[ 10,11 ]`

mac_list

Description When the policy is of type TRAFFIC, it is necessary to specify the MAC addresses that the host must monitor.

Type array of strings

Values An array with a list of MAC address strings.

Required For TRAFFIC policy types only.

Example `"mac_list":[ "de:ad:be:ef:01:01","de:ad:be:ef:01:02" ]`

max_packet_thresh

Description In a policy of type TRAFFIC, the threshold value above which the frequency is set to a maximum.

Type integer

Values The number of packets per interval above which the TRAFFIC policy applies the maximum frequency.

Required For the TRAFFIC policy only.

Example `"max_packet_thresh":  500000`

name

Description The name of the VM or host. Allows the parser to associate the policy with the relevant VM or host OS.

Type string

Values Any valid string.

Required Yes

Example `"name":  "ubuntu2"`

policy_type

Description The type of policy to apply. See the `--policy` option description for more information.

Type string

Values Possible values are:

- TIME: Time-of-day policy. Scale the frequencies of the relevant cores up/down depending on busy and quiet hours.
- TRAFFIC: Use statistics from the NIC and scale up and down accordingly.
- WORKLOAD: Determine how heavily loaded the cores are and scale up and down accordingly.
- BRANCH_RATIO: An out-of-band policy that looks at the ratio between branch hits and misses on a core and uses that information to determine how much packet processing a core is doing.

Required For `CREATE` and `DESTROY` policy requests only.

Example `"policy_type": "TIME"`

quiet_hours

Description The hours of the day to scale down the cores for quiet times.

Type array of integers

Values An array with a list of hour numbers with values in the range 0 to 23.

Required For the TIME policy only.

Example `"quiet_hours":[ 2,3,4,5,6 ]`

resource_id

Description The core to which to apply a power command.

Type integer

Values A valid core ID for the VM or host OS.

Required For the `POWER` instruction only.

Example `"resource_id": 10`

unit

Description The type of power operation to apply in the command.

Type string

Values

- SCALE_MAX: Scale the frequency of this core to the maximum.
- SCALE_MIN: Scale the frequency of this core to the minimum.
- SCALE_UP: Scale up the frequency of this core.
- SCALE_DOWN: Scale down the frequency of this core.
- ENABLE_TURBO: Enable Intel® Turbo Boost Technology for this core.

- DISABLE_TURBO: Disable Intel® Turbo Boost Technology for this core.

Required For the `POWER` instruction only.

Example `"unit": "SCALE_MAX"`

workload

Description In a policy of type WORKLOAD, it is necessary to specify how heavy the workload is.

Type string

Values

- HIGH: Scale the frequency of this core to maximum.
- MEDIUM: Scale the frequency of this core to minimum.
- LOW: Scale up the frequency of this core.

Required For the `WORKLOAD` policy only.

Example `"workload": "MEDIUM"`

CHAPTER

FORTYTWO

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.

42.1 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.

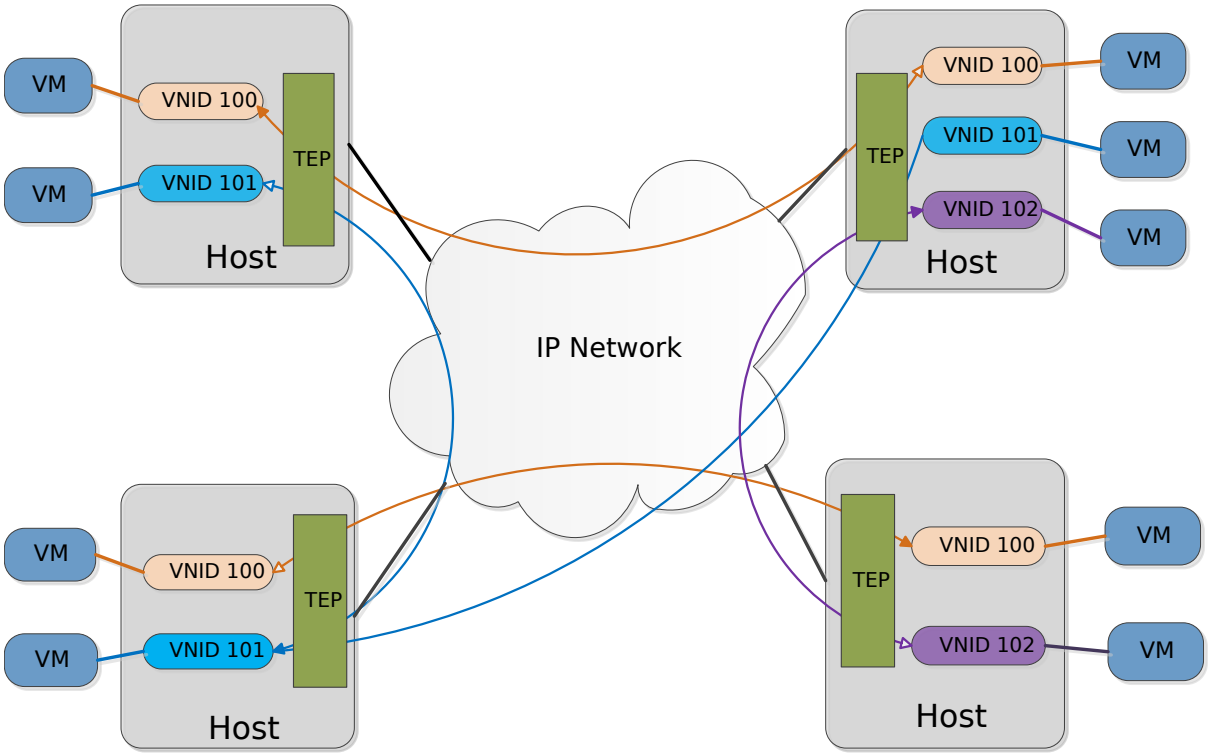


Fig. 42.1: Overlay Networking.

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.

42.2 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.

42.3 Supported Distributions

The example in this section have been validated with the following distributions:

- Fedora* 18
- Fedora* 19
- Fedora* 20

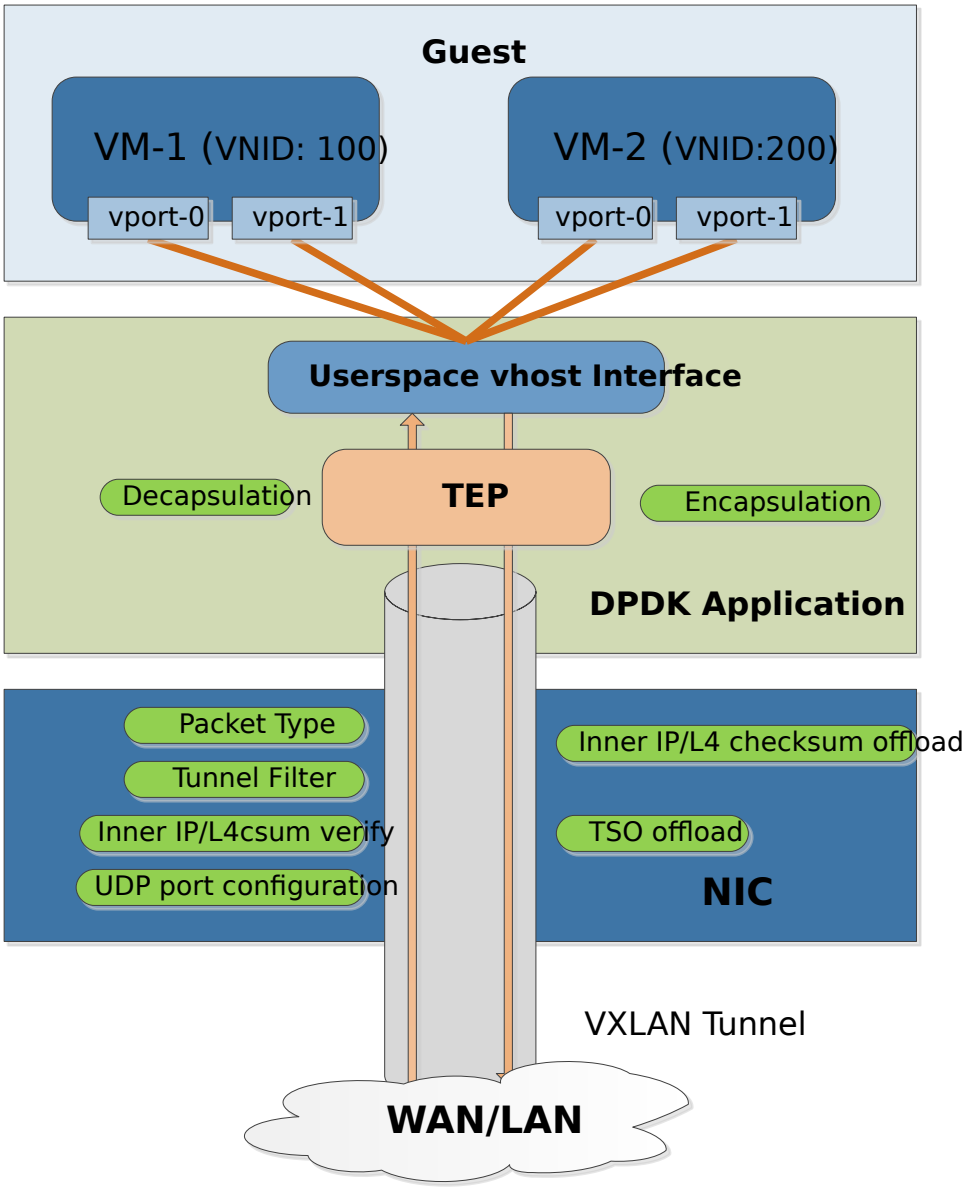


Fig. 42.2: TEP termination Framework Overview

42.4 Compiling the Sample Code

To enable vhost, turn on vhost library in the configure file `config/common_linux`.

```
CONFIG_RTE_LIBRTE_VHOST=y
```

Then following the to compile the sample application shown in *Compiling the Sample Applications*.

42.5 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.

42.5.1 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
```

42.6 Running the Virtual Machine (QEMU)

Refer to *Start the VM*.

42.7 Running DPDK in the Virtual Machine

Refer to *Run testpmd inside guest*.

42.8 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.

CHAPTER

FORTYTHREE

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.

43.1 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.

43.2 How the Application Works

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.

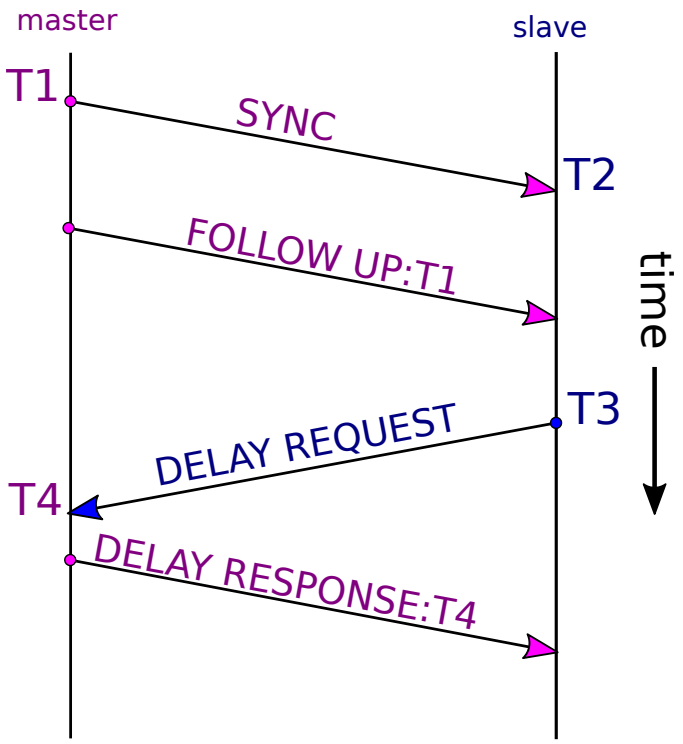


Fig. 43.1: PTP Synchronization Protocol

43.3 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `ptpclient` sub-directory.

Note: To compile the application edit the `config/common_linux` configuration file to enable IEEE1588 and then recompile DPDK:

```
CONFIG_RTE_LIBRTE_IEEE1588=y
```

43.4 Running the Application

To run the example in a `linux` 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.

43.5 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*.

43.5.1 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.

43.5.2 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`.

43.5.3 PTP parsing

The `parse_ptp_frames()` function processes PTP packets, implementing slave PTP IEEE1588 L2 functionality.

```
void
parse_ptp_frames(uint16_t portid, struct rte_mbuf *m) {
    struct ptp_header *ptp_hdr;
    struct rte_ether_hdr *eth_hdr;
    uint16_t eth_type;

    eth_hdr = rte_pktmbuf_mtod(m, struct rte_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 rte_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.

CHAPTER

FORTYFOUR

PERFORMANCE THREAD SAMPLE APPLICATION

The performance thread sample application is a derivative of the standard L3 forwarding application that demonstrates different threading models.

44.1 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.

44.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the *performance-thread/l3fwd-thread* sub-directory.

44.3 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.

44.3.1 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)"
```

44.3.2 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
```

44.3.3 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
```

44.4 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.

44.4.1 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.

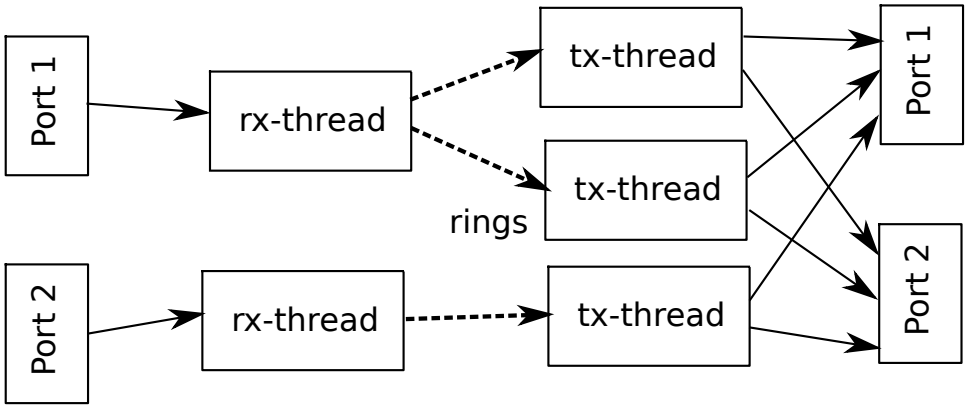
The TX threads poll software rings, perform the L3 forwarding hash/LPM match, and assemble packet bursts before performing burst transmit on the network interface.

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.

44.4.2 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 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 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.

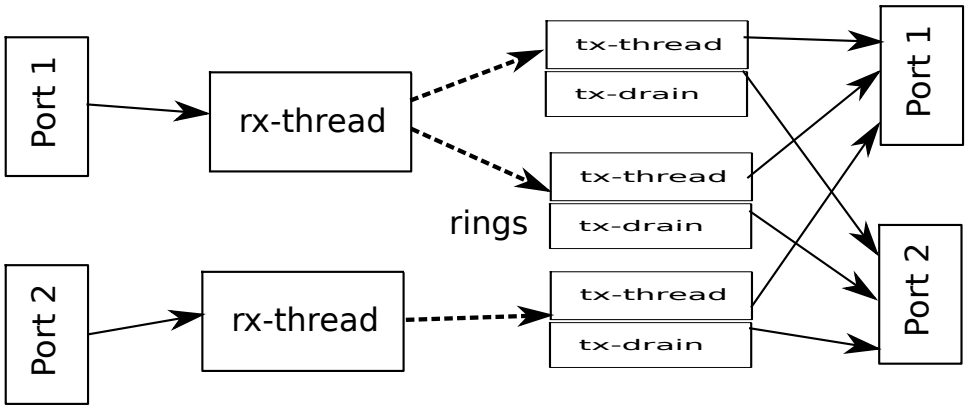
Additionally the TX L-thread spawns a worker L-thread to take care of polling the software rings, whilst it handles burst draining of the transmit buffer.

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).



44.4.3 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.

44.5 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.

44.5.1 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 modeled directly on the pthread get/set specific APIs, the second approach is modeled on the `RTE_PER_LCORE` macros, whereby `PER_LTHREAD` macros are introduced, in both cases the storage is local to the L-thread.

44.5.2 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 44.1: 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
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		

Continued on next page

Table 44.1 – continued from previous page

Pthread function	L-thread function	Notes
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.

44.5.3 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.

44.5.4 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-linux-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
```

44.5.5 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.

CHAPTER

FORTYFIVE

FEDERAL INFORMATION PROCESSING STANDARDS (FIPS) CRYPTODEV VALIDATION

45.1 Overview

Federal Information Processing Standards (FIPS) are publicly announced standards developed by the United States federal government for use in computer systems by non-military government agencies and government contractors.

This application is used to parse and perform symmetric cryptography computation to the NIST Cryptographic Algorithm Validation Program (CAVP) test vectors.

For an algorithm implementation to be listed on a cryptographic module validation certificate as an Approved security function, the algorithm implementation must meet all the requirements of FIPS 140-2 and must successfully complete the cryptographic algorithm validation process.

45.2 Limitations

- Only NIST CAVP request files are parsed by this application.
- The version of request file supported is `CAVS 21.0`
- If the header comment in a `.req` file does not contain a Algo tag i.e `AES,TDES,GCM` you need to manually add it into the header comment for example:

```
# VARIABLE KEY - KAT for CBC / # TDES VARIABLE KEY - KAT for CBC
```

- The application does not supply the test vectors. The user is expected to obtain the test vector files from NIST website. To obtain the `.req` files you need to email a person from the NIST website and pay for the `.req` files. The `.rsp` files from the site can be used to validate and compare with the `.rsp` files created by the FIPS application.
- **Supported test vectors**
  - AES-CBC (128,192,256) - GFSbox, KeySbox, MCT, MMT
  - AES-GCM (128,192,256) - EncryptExtIV, Decrypt
  - AES-CCM (128) - VADT, VNT, VPT, VTT, DVPT
  - AES-CMAC (128) - Generate, Verify
  - HMAC (SHA1, SHA224, SHA256, SHA384, SHA512)
  - TDES-CBC (1 Key, 2 Keys, 3 Keys) - MMT, Monte, Permop, Subkey, Varkey, VarText

45.3 Application Information

If a `.req` is used as the input file after the application is finished running it will generate a response file or `.rsp`. Differences between the two files are, the `.req` file has missing information for instance if doing encryption you will not have the cipher text and that will be generated in the response file. Also if doing decryption it will not have the plain text until it finished the work and in the response file it will be added onto the end of each operation.

The application can be run with a `.rsp` file and what the outcome of that will be is it will add a extra line in the generated `.rsp` which should be the same as the `.rsp` used to run the application, this is useful for validating if the application has done the operation correctly.

45.4 Compiling the Application

- Compile Application

```
make -C examples/fips_validation
```

- Run `dos2unix` on the request files

```
dos2unix AES/req/*
dos2unix AES_GCM/req/*
dos2unix CCM/req/*
dos2unix CMAC/req/*
dos2unix HMAC/req/*
dos2unix TDES/req/*
```

45.5 Running the Application

The application requires a number of command line options:

```
./fips_validation [EAL options]
-- --req-file FILE_PATH/FOLDER_PATH
--rsp-file FILE_PATH/FOLDER_PATH
[--cryptodev DEVICE_NAME] [--cryptodev-id ID] [--path-is-folder]
```

where,

- req-file: The path of the request file or folder, separated by `path-is-folder` option.
- rsp-file: The path that the response file or folder is stored. separated by `path-is-folder` option.
- cryptodev: The name of the target DPDK Crypto device to be validated.
- cryptodev-id: The id of the target DPDK Crypto device to be validated.
- path-is-folder: If presented the application expects req-file and rsp-file are folder paths.

To run the application in linux environment to test one AES FIPS test data file for crypto_aesni_mb PMD, issue the command:

```
$ ./fips_validation --vdev crypto_aesni_mb --
--req-file /PATH/TO/REQUEST/FILE.req --rsp-file ./PATH/TO/RESPONSE/FILE.rsp
--cryptodev crypto_aesni_mb
```

To run the application in linux environment to test all AES-GCM FIPS test data files in one folder for crypto_aesni_gcm PMD, issue the command:

```
$ ./fips_validation --vdev crypto_aesni_gcm0 --
--req-file /PATH/TO/REQUEST/FILE/FOLDER/
--rsp-file ./PATH/TO/RESPONSE/FILE/FOLDER/
--cryptodev-id 0 --path-is-folder
```

CHAPTER

FORTYSIX

IPSEC SECURITY GATEWAY SAMPLE APPLICATION

The IPsec Security Gateway application is an example of a "real world" application using DPDK cryptodev framework.

46.1 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 application also supports complete IPsec protocol offload to hardware (Look aside crypto accelerator or using ethernet device). It also support inline ipsec processing by the supported ethernet device during transmission. These modes can be selected during the SA creation configuration.

In case of complete protocol offload, the processing of headers(ESP and outer IP header) is done by the hardware and the application does not need to add/remove them during outbound/inbound processing.

For inline offloaded outbound traffic, the application will not do the LPM lookup for routing, as the port on which the packet has to be forwarded will be part of the SA. Security parameters will be configured on that port only, and sending the packet on other ports could result in unencrypted packets being sent out.

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 (Not needed in case of inline ipsec).
- Remove ESP and outer IP header (Not needed in case of protocol offload).
- 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 (Not needed in case protocol offload).
- Perform Encryption/Digest (Not needed in case of inline ipsec).
- Routing.
- Write packet to port.

46.2 Constraints

- No IPv6 options headers.
- No AH mode.
- Supported algorithms: AES-CBC, AES-CTR, AES-GCM, 3DES-CBC, HMAC-SHA1 and NULL.
- Each SA must be handle by a unique lcore (*1 RX queue per port*).

46.3 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `ipsec-secgw` sub-directory.

1. [Optional] Build the application for debugging: This option adds some extra flags, disables compiler optimizations and is verbose:

```
make DEBUG=1
```

46.4 Running the Application

The application has a number of command line options:

```
./build/ipsec-secgw [EAL options] --
                     -p PORTMASK -P -u PORTMASK -j FRAMESIZE
                     -l -w REPLAY_WINOW_SIZE -e -a
                     -c SAD_CACHE_SIZE
                     --config (port,queue,lcore)[,(port,queue,lcore]
                     --single-sa SAIDX
                     --rxoffload MASK
                     --txoffload MASK
                     --mtu MTU
                     --reassemble NUM
                     -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
- `-j FRAMESIZE`: *optional*. data buffer size (in bytes), in other words maximum data size for one segment. Packets with length bigger then FRAMESIZE still can be received, but will be segmented. Default value: RTE_MBUF_DEFAULT_BUF_SIZE (2176) Minimum value: RTE_MBUF_DEFAULT_BUF_SIZE (2176) Maximum value: UINT16_MAX (65535).
- `-l`: enables code-path that uses librte_ipsec.
- `-w REPLAY_WINOW_SIZE`: specifies the IPsec sequence number replay window size for each Security Association (available only with librte_ipsec code path).
- `-e`: enables Security Association extended sequence number processing (available only with librte_ipsec code path).
- `-a`: enables Security Association sequence number atomic behavior (available only with librte_ipsec code path).
- `-c`: specifies the SAD cache size. Stores the most recent SA in a per lcore cache. Cache represents flat array containing SA's indexed by SPI. Zero value disables cache. Default value: 128.
- `--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.
- `--rxoffload MASK`: RX HW offload capabilities to enable/use on this port (bitmask of DEV_RX_OFFLOAD_* values). It is an optional parameter and allows user to disable some of the RX HW offload capabilities. By default all HW RX offloads are enabled.
- `--txoffload MASK`: TX HW offload capabilities to enable/use on this port (bitmask of DEV_TX_OFFLOAD_* values). It is an optional parameter and allows user to disable some of the TX HW offload capabilities. By default all HW TX offloads are enabled.
- `--mtu MTU`: MTU value (in bytes) on all attached ethernet ports. Outgoing packets with length bigger then MTU will be fragmented. Incoming packets with length bigger then MTU will be discarded. Default value: 1500.
- `--frag-ttl FRAG_TTL_NS`: fragment lifetime (in nanoseconds). If packet is not reassembled within this time, received fragments will be discarded. Fragment lifetime should be decreased when there is a high fragmented traffic loss in high bandwidth networks. Should be lower for low number of reassembly buckets. Valid values: from 1 ns to 10 s. Default value: 10000000 (10 s).
- `--reassemble NUM`: max number of entries in reassemble fragment table. Zero value disables reassembly functionality. Default value: 0.
- `-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 "crypto_null" -- -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 "crypto_null"` 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 "crypto_aesni_mb" --vdev "crypto_null" \
       -- \
       -p 0xf -P -u 0x3 --config="(0,0,20),(1,0,20),(2,0,21),(3,0,21)" \
       -f sample.cfg
```

46.5 Configurations

The following sections provide the syntax of configurations to initialize your SP, SA, Routing and Neighbour 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.

46.5.1 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, Routing or Neighbour 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> <action_type> <port_id> <fallback>
```

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: Yes, unless <aead_algo> is not used
- Available options:
  - *null*: NULL algorithm
  - *aes-128-cbc*: AES-CBC 128-bit algorithm
  - *aes-256-cbc*: AES-CBC 256-bit algorithm
  - *aes-128-ctr*: AES-CTR 128-bit algorithm
  - *3des-cbc*: 3DES-CBC 192-bit algorithm
- Syntax: *cipher_algo <your algorithm>*

`<cipher_key>`

- Cipher key, NOT available when 'null' algorithm is used
- Optional: Yes, unless <aead_algo> is not used. Must be 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: Yes, unless <aead_algo> is not used
- Available options:
  - *null*: NULL algorithm
  - *sha1-hmac*: HMAC SHA1 algorithm

`<auth_key>`

- Authentication key, NOT available when 'null' or 'aes-128-gcm' algorithm is used.
- Optional: Yes, unless <aead_algo> is not used. Must be 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*

`<aead_algo>`

- AEAD algorithm
- Optional: Yes, unless <cipher_algo> and <auth_algo> are not used
- Available options:
  - *aes-128-gcm*: AES-GCM 128-bit algorithm
- Syntax: *cipher_algo <your algorithm>*

`<aead_key>`

- Cipher key, NOT available when 'null' algorithm is used
- Optional: Yes, unless <cipher_algo> and <auth_algo> are not used. Must be followed by <aead_algo> option
- Syntax: Hexadecimal bytes (0x0-0xFF) concatenate by colon symbol ':'. The number of bytes should be as same as the specified AEAD algorithm key size.

  For example: *aead_key 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

`<type>`

- Action type to specify the security action. This option specify the SA to be performed with look aside protocol offload to HW accelerator or protocol offload on ethernet device or inline crypto processing on the ethernet device during transmission.
- Optional: Yes, default type *no-offload*
- Available options:
  - *lookaside-protocol-offload*: look aside protocol offload to HW accelerator
  - *inline-protocol-offload*: inline protocol offload on ethernet device
  - *inline-crypto-offload*: inline crypto processing on ethernet device
  - *no-offload*: no offloading to hardware

`<port_id>`

- Port/device ID of the ethernet/crypto accelerator for which the SA is configured. For *inline-crypto-offload* and *inline-protocol-offload*, this port will be used for routing. The routing table will not be referred in this case.
- Optional: No, if *type* is not *no-offload*
- Syntax:
  - *port_id X* X is a valid device number in decimal

`<fallback>`

- Action type for ingress IPsec packets that inline processor failed to process. Only a combination of *inline-crypto-offload* as a primary session and *lookaside-none* as a fall-back session is supported at the moment.

If used in conjunction with IPsec window, its width needs be increased due to different processing times of inline and lookaside modes which results in packet reordering.

- Optional: Yes.
- Available options:
  - *lookaside-none*: use automatically chosen cryptodev to process packets
- Syntax:
  - *fallback lookaside-none*

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 aead_algo aes-128-gcm \
aead_key de:ad:be:ef:de:ad:be:ef:de:ad:be:ef:de:ad:be:ef \
mode ipv4-tunnel src 172.16.2.5 dst 172.16.1.5

sa out 5 cipher_algo aes-128-cbc cipher_key 0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0 \
auth_algo sha1-hmac auth_key 0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0 \
mode ipv4-tunnel src 172.16.1.5 dst 172.16.2.5 \
type lookaside-protocol-offload port_id 4

sa in 35 aead_algo aes-128-gcm \
aead_key de:ad:be:ef:de:ad:be:ef:de:ad:be:ef:de:ad:be:ef \
mode ipv4-tunnel src 172.16.2.5 dst 172.16.1.5 \
type inline-crypto-offload port_id 0
```

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
```

Neighbour rule syntax

The Neighbour rule syntax is shown as follows:

```
neigh <port> <dst_mac>
```

where each options means:

`<port>`

- The output port id
- Optional: No
- Syntax: *port X*

`<dst_mac>`

- The destination ethernet address to use for that port
- Optional: No
- Syntax:
  - XX:XX:XX:XX:XX:XX

Example Neighbour rules:

```
neigh port 0 DE:AD:BE:EF:01:02
```

46.6 Test directory

The test directory contains scripts for testing the various encryption algorithms.

The purpose of the scripts is to automate ipsec-secgw testing using another system running linux as a DUT.

The user must setup the following environment variables:

- `SGW_PATH`: path to the ipsec-secgw binary to test.
- `REMOTE_HOST`: IP address/hostname of the DUT.
- `REMOTE_IFACE`: interface name for the test-port on the DUT.
- `ETH_DEV`: ethernet device to be used on the SUT by DPDK ('-w <pci-id>')

Also the user can optionally setup:

- `SGW_LCORE`: lcore to run ipsec-secgw on (default value is 0)
- `CRYPTO_DEV`: crypto device to be used ('-w <pci-id>'). If none specified appropriate vdevs will be created by the script
- `MULTI_SEG_TEST`: ipsec-secgw option to enable reassembly support and specify size of reassembly table (e.g. `MULTI_SEG_TEST='--reassemble 128'`). This option must be set for fallback session tests.

Note that most of the tests require the appropriate crypto PMD/device to be available.

46.6.1 Server configuration

Two servers are required for the tests, SUT and DUT.

Make sure the user from the SUT can ssh to the DUT without entering the password. To enable this feature keys must be setup on the DUT.

`ssh-keygen` will make a private & public key pair on the SUT.

`ssh-copy-id` <user name>@<target host name> on the SUT will copy the public key to the DUT. It will ask for credentials so that it can upload the public key.

The SUT and DUT are connected through at least 2 NIC ports.

One NIC port is expected to be managed by linux on both machines and will be used as a control path.

The second NIC port (test-port) should be bound to DPDK on the SUT, and should be managed by linux on the DUT.

The script starts `ipsec-secgw` with 2 NIC devices: `test-port` and `tap vdev`.

It then configures the local tap interface and the remote interface and IPsec policies in the following way:

Traffic going over the test-port in both directions has to be protected by IPsec.

Traffic going over the TAP port in both directions does not have to be protected.

i.e:

DUT OS(NIC1)–(IPsec)–>(NIC1)ipsec-secgw(TAP)–(plain)–>(TAP)SUT OS

SUT OS(TAP)–(plain)–>(TAP)psec-secgw(NIC1)–(IPsec)–>(NIC1)DUT OS

It then tries to perform some data transfer using the scheme described above.

46.6.2 usage

In the ipsec-secgw/test directory

to run one test for IPv4 or IPv6

/bin/bash linux_test(4|6).sh <ipsec_mode>

to run all tests for IPv4 or IPv6

/bin/bash run_test.sh -4|-6

For the list of available modes please refer to run_test.sh.

CHAPTER

FORTYSEVEN

LOOP-BACK SAMPLE APPLICATION USING BASEBAND DEVICE (BBDEV)

The baseband sample application is a simple example of packet processing using the Data Plane Development Kit (DPDK) for baseband workloads using Wireless Device abstraction library.

47.1 Overview

The Baseband device sample application performs a loop-back operation using a baseband device capable of transceiving data packets. A packet is received on an ethernet port -> enqueued for downlink baseband operation -> dequeued from the downlink baseband device -> enqueued for uplink baseband operation -> dequeued from the baseband device -> then the received packet is compared with the baseband operations output. Then it's looped back to the ethernet port.

- The MAC header is preserved in the packet

47.2 Limitations

- Only one baseband device and one ethernet port can be used.

47.3 Compiling the Application

1. DPDK needs to be built with `baseband_turbo_sw` PMD driver enabled along with `FLEXRAN SDK` Libraries. Refer to *SW Turbo Poll Mode Driver* documentation for more details on this.

2. Go to the example directory:

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

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

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

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

4. Build the application:

```
make
```

47.4 Running the Application

The application accepts a number of command line options:

```
$ ./build/bbdev [EAL options] -- [-e ENCODING_CORES] [-d DECODING_CORES] /
[-p ETH_PORT_ID] [-b BBDEV_ID]
```

where:

- `e ENCODING_CORES`: hexmask for encoding lcores (default = 0x2)
- `d DECODING_CORES`: hexmask for decoding lcores (default = 0x4)
- `p ETH_PORT_ID`: ethernet port ID (default = 0)
- `b BBDEV_ID`: BBDev ID (default = 0)

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

To run the application in linux environment with the turbo_sw baseband device using the whitelisted port running on 1 encoding lcore and 1 decoding lcore issue the command:

```
$ ./build/bbdev --vdev='baseband_turbo_sw' -w <NIC0PCIADDR> -c 0x38 --socket-mem=2,2 \
--file-prefix=bbdev -- -e 0x10 -d 0x20
```

where, NIC0PCIADDR is the PCI address of the Rx port

This command creates one virtual bbdev devices `baseband_turbo_sw` where the device gets linked to a corresponding ethernet port as whitelisted by the parameter -w. 3 cores are allocated to the application, and assigned as:

- core 3 is the master and used to print the stats live on screen,
- core 4 is the encoding lcore performing Rx and Turbo Encode operations
- core 5 is the downlink lcore performing Turbo Decode, validation and Tx operations

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

47.5 Using Packet Generator with baseband device sample application

To allow the bbdev sample app to do the loopback, an influx of traffic is required. This can be done by using DPDK Pktgen to burst traffic on two ethernet ports, and it will print the transmitted along with the looped-back traffic on Rx ports. Executing the command below will generate traffic on the two whitelisted ethernet ports.

```
$ ./pktgen-3.4.0/app/x86_64-native-linux-gcc/pktgen -c 0x3 \
--socket-mem=1,1 --file-prefix=pg -w <NIC1PCIADDR> -- -m 1.0 -P
```

where:

- `-c COREMASK`: A hexadecimal bitmask of cores to run on
- `--socket-mem`: Memory to allocate on specific sockets (use comma separated values)
- `--file-prefix`: Prefix for hugepage filenames

- `-w <NIC1PCIADDR>`: Add a PCI device in white list. The argument format is <[domain:]bus:devid.func>.
- `-m <string>`: Matrix for mapping ports to logical cores.
- `-P`: PROMISCUOUS mode

Refer to *The Pktgen Application* documents for general information on running Pktgen with DPDK applications.

CHAPTER

FORTYEIGHT

NTB SAMPLE APPLICATION

The ntb sample application shows how to use ntb rawdev driver. This sample provides interactive mode to do packet based processing between two systems.

This sample supports 4 types of packet forwarding mode.

- `file-trans`: transmit files between two systems. The sample will be polling to receive files from the peer and save the file as `ntb_recv_file[N]`, [N] represents the number of received file.
- `rxonly`: NTB receives packets but doesn't transmit them.
- `txonly`: NTB generates and transmits packets without receiving any.
- `iofwd`: iofwd between NTB device and ethdev.

48.1 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `ntb` sub-directory.

48.2 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:

```
./build/ntb_fwd -c 0xf -n 6 -- -i
```

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

48.3 Command-line Options

The application supports the following command-line options.

- `--buf-size=N`

  Set the data size of the mbufs used to N bytes, where N < 65536. The default value is 2048.

- `--fwd-mode=mode`

  Set the packet forwarding mode as `file-trans`, `txonly`, `rxonly` or `iofwd`.

- `--nb-desc=N`

  Set number of descriptors of queue as N, namely queue size, where 64 <= N <= 1024. The default value is 1024.

- `--txfreet=N`

  Set the transmit free threshold of TX rings to N, where 0 <= N <= the value of `--nb-desc`. The default value is 256.

- `--burst=N`

  Set the number of packets per burst to N, where 1 <= N <= 32. The default value is 32.

- `--qp=N`

  Set the number of queues as N, where qp > 0. The default value is 1.

48.4 Using the application

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

```
ntb>
```

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

- `send [filepath]`: Send file to the peer host. Need to be in file-trans forwarding mode first.
- `start`: Start transmission.
- `stop`: Stop transmission.
- `show/clear port stats`: Show/Clear port stats and throughput.
- `set fwd file-trans/rxonly/txonly/iofwd`: Set packet forwarding mode.
- `quit`: Exit program.