Intel® Data Plane Development Kit - L3 Forwarding Sample Application

User Guide

April 2012

Intel Confidential

Reference Number: 482251-1.2

Contents

Revision History

Date	Revision	Description
March 2012	1.2	Updates for software release 1.2
November 2011	1.1	Update to cover the replacement of the -q command line option with the more versatile --config option as described in Section 4.0, “Running the Application” on page 5. Updates to Section 5.1, Section 5.2, Section 5.3, and Section 5.4 to align with sample code.
September 2011	1.0	Initial release

1.0 Introduction

The L3 Forwarding application is a simple example of packet processing using the Intel® DPDK. The application performs L3 forwarding.

1.1 Documentation Roadmap

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

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

  A glossary of terms is also provided.
- **API Reference**: Provides detailed information about Intel® DPDK functions, data structures and other programming constructs.
- **Sample Application User Guides**: A set of guides, each describing a sample application that showcases specific functionality, together with instructions on how to compile, run and use the sample application.

2.0 Overview

The application demonstrates the use of the `hash` and `LPM` libraries in the Intel® DPDK to implement packet forwarding. The initialization and run-time paths are very similar to those of the L2 forwarding application (see the *Intel® DPDK L2 Forwarding Sample Application User Guide* for more information). This guide highlights the differences between the two applications. The main difference from the L2 Forwarding sample application is that the forwarding decision is taken based on information read from the input packet.

The lookup method is either hash-based or LPM-based and is selected at compile time. When the selected lookup method is hash-based, a hash object is used to emulate the flow classification stage. The hash object is used in correlation with the flow table to map each input packet to its flow at runtime.

The hash lookup key is represented by the DiffServ 5-tuple composed of the flowing 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 the initialization time. When the selected lookup method is LPM based, an LPM object is used to emulate the forwarding stage for IPv4 packets. The LPM object is used as the routing table to identify the next hop for each input packet at runtime.

The LPM lookup key is represented by the Destination IP Address field read from the input packet. The ID of the output interface for the input packet is the next hop returned by the LPM lookup. The set of LPM rules used by the application is statically configured and loaded into the LPM object at the initialization time.

3.0 Compiling the Application

To compile the application:

1. Go to the sample application directory:

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}/examples/l3fwd
```

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

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

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

3. Build the application:

```
make
```

4.0 Running the Application

The application has a number of command line options:

```
./build/l3fwd [EAL options] -- -p PORTMASK [-P]
[--config(port,queue,lcore)[,(port,queue,lcore]]
```

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

For example, consider a dual processor socket platform where cores 0,2,4,6, 8, and 10 appear on socket 0, while cores 1,3,5,7,9, and 11 appear on socket 1. Let's say that the programmer wants to use memory from both NUMA nodes, the platform has only two ports and the programmer wants to use two cores from each processor socket to do the packet processing.

To enable L3 forwarding between two ports, using two cores from each processor, while also taking advantage of local memory accesses by optimizing around NUMA, the programmer must enable two queues from each port, pin to the appropriate cores and allocate memory from the appropriate NUMA node. This is achieved using the following command:

```
./build/l3fwd -c f -n 4 -- -p 0x3 --config="(0,0,0),(0,1,2),(1,0,1),(1,1,3)"
```

In this command:

- The `-c` option enables cores 0, 1, 2, 3
- The `-p` option enables ports 0 and 1
- The `--config` option enables two queues on each port and maps each (port,queue) pair to a specific core. Logic to enable multiple RX queues using RSS and to allocate memory from the correct NUMA nodes is included in the application and is done transparently. The following table shows the mapping in this example:

Port	Queue	lcore	Description
0	0	0	Map queue 0 from port 0 to lcore 0.
0	1	2	Map queue 1 from port 0 to lcore 2.
1	0	1	Map queue 0 from port 1 to lcore 1.
1	1	3	Map queue 1 from port 1 to lcore 3.

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

5.0 Explanation

The following sections provide some explanation of code. As metioned in the overveiw section, the initialization and run-time paths are very similar to those of the L2 forwarding application (see the *L2 Forwarding Sample Application User Guide* for more information). The following sections describe aspects that are specific to the L3 Forwarding sample application.

5.1 Hash Initialization

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

```
#if (APP_LOOKUP_METHOD == APP_LOOKUP_EXACT_MATCH)
static void
setup_hash(int socketid)
{
    unsigned i;
    int ret;
    char s[64];

    /* create  hashes */
    snprintf(s, sizeof(s), "l3fwd_hash_%d", socketid);
    l3fwd_hash_params.name = s;
    l3fwd_hash_params.socket_id = socketid;
    l3fwd_lookup_struct[socketid] = rte_hash_create(&l3fwd_hash_params);
    if (l3fwd_lookup_struct[socketid] == NULL)
        rte_panic("Unable to create the l3fwd hash on "
                "socket %d\n", socketid);

    /* populate the hash */
    for (i = 0; i < L3FWD_NUM_ROUTES; i++) {
        ret = rte_hash_add_key (l3fwd_lookup_struct[socketid],
                (void *) &l3fwd_route_array[i].key);
        if (ret < 0) {
            rte_panic("Unable to add entry %u to the"
                "l3fwd hash on socket %d\n", i, socketid);
        }
        l3fwd_out_if[ret] = l3fwd_route_array[i].if_out;
        printf("Hash: Adding key\n");
        print_key(l3fwd_route_array[i].key);
    }
}
#endif
```

5.2 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), "L3FWD_LPM_%d", socketid);
    l3fwd_lookup_struct[socketid] = rte_lpm_create(s, socketid,
                L3FWD_LPM_MAX_RULES, RTE_LPM_MEMZONE);
    if (l3fwd_lookup_struct[socketid] == NULL)
        rte_panic("Unable to create the l3fwd LPM table"
                " on socket %d\n", socketid);

    /* populate the LPM table */
    for (i = 0; i < L3FWD_NUM_ROUTES; i++) {
        ret = rte_lpm_add(l3fwd_lookup_struct[socketid],
            l3fwd_route_array[i].ip,
            l3fwd_route_array[i].depth,
            l3fwd_route_array[i].if_out);

        if (ret < 0) {
            rte_panic("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",
            l3fwd_route_array[i].ip,
            l3fwd_route_array[i].depth,
            l3fwd_route_array[i].if_out);
    }
}
#endif
```

5.3 Packet Forwarding for Hash-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 hash-based lookups is done by the `get_dst_port()` function below:

```
static inline uint8_t
get_dst_port(struct ipv4_hdr *ipv4_hdr,  uint8_t portid, lookup_struct_t *
l3fwd_lookup_struct)
{
    struct ipv4_5tuple key;
    struct tcp_hdr *tcp;
    struct udp_hdr *udp;
    int ret = 0;

    key.ip_dst = rte_be_to_cpu_32(ipv4_hdr->dst_addr);
    key.ip_src = rte_be_to_cpu_32(ipv4_hdr->src_addr);
    key.proto = ipv4_hdr->next_proto_id;

    switch (ipv4_hdr->next_proto_id) {
    case INET_IPPROTO_TCP:
        tcp = (struct tcp_hdr *)((unsigned char *) ipv4_hdr +
                    sizeof(struct ipv4_hdr));
        key.port_dst = rte_be_to_cpu_16(tcp->dst_port);
        key.port_src = rte_be_to_cpu_16(tcp->src_port);
        break;

    case INET_IPPROTO_UDP:
        udp = (struct udp_hdr *)((unsigned char *) ipv4_hdr +
                    sizeof(struct ipv4_hdr));
        key.port_dst = rte_be_to_cpu_16(udp->dst_port);
        key.port_src = rte_be_to_cpu_16(udp->src_port);
        break;

    default:
        key.port_dst = 0;
        key.port_src = 0;
    }

    /* Find destination port */
    ret = rte_hash_lookup(l3fwd_lookup_struct, (const void *)&key);
    return (uint8_t)((ret < 0)? portid : l3fwd_out_if[ret]);
}
```

5.4 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_dst_port()` function below:

```
static inline uint8_t
get_dst_port(struct ipv4_hdr *ipv4_hdr, uint8_t portid, lookup_struct_t *
l3fwd_lookup_struct)
{
    uint8_t next_hop;

    return (uint8_t) ((rte_lpm_lookup(l3fwd_lookup_struct,
            rte_be_to_cpu_32(ipv4_hdr->dst_addr), &next_hop) == 0)?
            next_hop : portid);
}
```

§ §