Självständigt arbete på avancerad nivå

*Independent degree project – second cycle*

*Computer Engineering*

Evaluation and Implementation of Linux User-space Fast Path Technologies

Ahmed Khan
Abstract

The enormous increase in device connectivity for data and telecom devices places significant challenges on the packet processing techniques used in embedded systems such as IP stacks. Therefore, the traditional packet processing software cannot handle the line rate packet flow even for the most cutting edge devices. A solution to this problem is to allow applications to directly receive packets without passing through the normal kernel stack and drivers i.e. interface directly with the hardware. Two such open source libraries for Linux are PF_RING and Netmap. In addition Freescale has a similar technology called USDPAA.

In order to satisfy the first goal of this project, a detailed analysis and evaluation of PF_RING, Netmap and USDPAA has been conducted in order to determine how they compare in relation to a number of criteria such as functionality, support, performance, ease of use, software/hardware dependencies and project stability etc. Secondly based on the earlier work, a design is proposed that can be used to build and port an application to run on Freescale DPAA based hardware (P4080) on top of USDPAA.

Keywords: Embedded systems, IP stacks, PF_RING, Netmap, USDPAA, DPAA, and P4080.
Acknowledgements

First and foremost, I am grateful to Freescale Semiconductor and every member of the team for contributing to this project directly or indirectly and for making my stay such a wonderful experience.

I would like to thank Jonas for considering me for this thesis work and making me feel completely at ease with his warm welcoming nature and for his time and his belief in me. Once again thank you for playing a vital role by providing assistance and ideas to complete this thesis work.

I would also like to express my deep appreciation to my co-supervisors Ph.D candidate Wei Shen and Prof. Dr. Tingting Zhang for their support and help throughout my project.

Last but not the least; I am thankful to Mid Sweden University, the faculty and my colleagues for their encouragement throughout my studies.
# Table of Contents

Abstract .................................................................................................................. ii

Acknowledgements ................................................................................................. iii

Table of Contents .................................................................................................... iv

List of Figures ......................................................................................................... vi

Terminology ............................................................................................................. vii

1 Introduction .......................................................................................................... 9
   1.1 Background and problem motivation ......................................................... 10
   1.2 Overall aim and Scope ............................................................................... 10
   1.3 Concrete goals ............................................................................................. 11
   1.4 Outline ......................................................................................................... 11
   1.5 Intended Audience ...................................................................................... 11

2 Fast Path Technologies ....................................................................................... 12
   2.1 Traditional Linux Kernel ............................................................................ 12
   2.2 PF_RING ..................................................................................................... 13
      2.2.1 PF_RING Architecture ......................................................................... 14
      2.2.2 Socket Clustering ................................................................................ 15
      2.2.3 PF_RING from developers prospective .............................................. 16
      2.2.4 Circular buffer ..................................................................................... 17
      2.2.5 PF_RING API ....................................................................................... 18
   2.3 Netmap ......................................................................................................... 22
      2.3.1 Netmap Architecture .......................................................................... 22
      2.3.2 Netmap User API ................................................................................. 25
      2.3.3 Efficient Packet Processing ................................................................. 25
      2.3.4 Multiple Cores/NIC’s .......................................................................... 26
      2.3.5 Blocking I/O Primitives ....................................................................... 26
   2.4 USDPAA ....................................................................................................... 26
      2.4.1 USDPAA Components ....................................................................... 27
      2.4.2 Device Tree .......................................................................................... 27
      2.4.3 Queue Manager (QMan) Driver ......................................................... 28
      2.4.4 Buffer Manager (BMan) Driver ........................................................... 29
      2.4.5 DMA Memory ...................................................................................... 29
      2.4.6 Assumptions and Use cases ................................................................. 30
      2.4.7 Linux Support ....................................................................................... 31
   2.5 QorIQ P4080 Processor ............................................................................... 31
# Table of Contents

3 Methodology ......................................................................................33  
3.1 Software/ Hardware detail...............................................................33  
3.2 PF_RING .......................................................................................34  
3.3 Netmap .......................................................................................35  
3.4 USDPA .......................................................................................35  
3.4.1 hello_reflector .......................................................................36  

4 Design and Implementation .................................................................38  
4.1 API Design .................................................................................39  
4.2 Wrapper layer ............................................................................42  

5 Results ..............................................................................................45  
5.1 PF_RING test .............................................................................46  
5.2 USDPA test ...............................................................................49  

6 Conclusions ......................................................................................54  

7 Future Work .....................................................................................56  

References ............................................................................................57  

Appendix A ..........................................................................................59  
Core parts of the code ........................................................................59
# List of Figures

2.1 Compare traditional model with zero-copy model ........................................... 5  
2.2(a) PF_RING Architecture ................................................................................. 6  
2.2(b) Legacy device Architecture ......................................................................... 6  
2.3(a) Packet TX via PF_RING ................................................................................ 7  
2.3(b) Packet RX via PF_RING ................................................................................ 7  
2.4(a) Non cluster approach ................................................................................... 7  
2.4(b) PF_RING Socket cluster approach ................................................................ 7  
2.5 Combining Filtering with Balancing ................................................................. 8  
2.6 Netmap Architecture ....................................................................................... 15  
2.7 Shared memory region exported by Netmap ..................................................... 15  
2.8 USDPAA Software Portals ................................................................................ 18  
2.9 QMan and BMan access API in USDPAA ......................................................... 20  
2.10 QoriQ P4080 Processor .................................................................................. 24  
4.1 Wrapper API design ......................................................................................... 31  
4.2 User-space application interface concept ....................................................... 31  
4.3 User-space portable application ..................................................................... 32  
4.4 Execution method of Netmap_wrapper program ........................................... 35  
5.1 pf_send output for TX packets ........................................................................ 38  
5.2 Packets captured from pf_send (PF_RING) via Wireshark .............................. 39  
5.3 Physical setup of P4080 device ........................................................................ 40  
5.4 4x 1G connections on P4080 device .............................................................. 42  
5.5 P4080 Tera Term output results ..................................................................... 43
# Terminology

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP Stack</td>
<td>Default Linux Network stack</td>
</tr>
<tr>
<td>PF_RING</td>
<td>A novel framework for fast packet I/O</td>
</tr>
<tr>
<td>Netmap</td>
<td>A framework for fast packet processing</td>
</tr>
<tr>
<td>DPAA</td>
<td>Data Path Acceleration Architecture</td>
</tr>
<tr>
<td>USDPPAA</td>
<td>User-Space Data Path Acceleration Architecture</td>
</tr>
<tr>
<td>P4080</td>
<td>Freescale QorIQ P4080 communications processor</td>
</tr>
<tr>
<td>Libpcap</td>
<td>User-space packet capture library</td>
</tr>
<tr>
<td>NIC</td>
<td>Network Interface Card</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>NAPI</td>
<td>New Application Programming Interface</td>
</tr>
<tr>
<td>AKM</td>
<td>Accelerated Kernel Module</td>
</tr>
<tr>
<td>SDK</td>
<td>Software Development Kit</td>
</tr>
<tr>
<td>TX/RX</td>
<td>Transmitter/ Receiver</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In First Out</td>
</tr>
<tr>
<td>Eth (n)</td>
<td>Ethernet (port number)</td>
</tr>
<tr>
<td>Pkt_buf</td>
<td>Packet Buffers</td>
</tr>
<tr>
<td>IOCTL</td>
<td>Input/ Output Control system call</td>
</tr>
<tr>
<td>BMan</td>
<td>Buffer Manager</td>
</tr>
<tr>
<td>QMan</td>
<td>Queue Manager</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>-------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>FMan</td>
<td>Frame Manager</td>
</tr>
<tr>
<td>SoC</td>
<td>System on Chip</td>
</tr>
<tr>
<td>SMP</td>
<td>Symmetric Multi-Processing</td>
</tr>
<tr>
<td>UIO</td>
<td>User-space Input/Output</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access</td>
</tr>
<tr>
<td>MAC</td>
<td>Media Access Control</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>TFTP</td>
<td>Trivial File Transfer Protocol</td>
</tr>
<tr>
<td>GNU</td>
<td>General Public License</td>
</tr>
<tr>
<td>DIS</td>
<td>Distributed Interactive Simulation</td>
</tr>
<tr>
<td>ARP</td>
<td>Address Resolution Protocol</td>
</tr>
</tbody>
</table>
1 Introduction

Freescale Semiconductor is one of the best suppliers of devices for networking and telecom equipment and with customers all over the world. One of the many reasons why this organization dominates this industry is by providing highly optimized hardware solutions packaged with equally highly rated software solutions. With the passage of time and with the dramatic increases in relation to internet based services, advances in interconnection technologies are increasing the requirement for advanced and more efficient packet processing solutions. This huge increase in device connectivity for data and telecom systems, which is rapidly moving from the current 10Gbps Ethernet to 40Gbps and 100Gbps places significant challenges on the present packet processing techniques. For instance, a 10Gbps Ethernet interface can, in the worst case scenario, equal 15 million packets per second and in Linux, a normal kernel to user-space context switch alone consumes about 20-30k clock cycles, which would be necessary for a single or a small group of packets. Hence, a normal software deployment cannot handle the packet flow even using the most cutting edge devices.

Researchers have suggested changes to the software stack that is responsible for allowing the software applications to maintain the correct rate with the network hardware [16]. Optimization in the hardware for packet processing has been built to great effect but, even with the most optimized hardware, results will only improve if there is quality software that can leverage the acceleration in the hardware. Traditionally in Linux, the network stack is responsible for packet processing from the device driver to user-space applications. Although the network stack is a reliable mechanism, it also consumes a significant amount of time while processing these packets, providing different functionality at each layer in the architecture. However, this functionality is not required by some applications, for example, processing raw packets through the network stack will simply increase the processing overhead and will affect the network speed and the rate at which these packets are transmitted.

In traditional systems, the kernel is responsible for packet processing, which can cause there to be multiple data coping and a significant amount of content transformation. One suggestion is to move the kernel out of the critical path and thus moving some protocols into the user-
space from the kernel space and providing the user-space direct access to the device drivers [15]. Two such open libraries for Linux are PF_RING and Netmap. In addition Freescale has a similar technology called User-Space Data Path Acceleration Architecture (USDPAA) that leverages Freescale’s hardware technology for accelerated packet processing called Data Path Acceleration Architecture DPAA.

1.1 Background and problem motivation
In today’s network infrastructure, the ability of an application to deal with a huge amount of data in real time processing and its performance can have a critical effect on the overall experience of the end user. Traditionally, processing high speed network applications is an integral part of multi-core systems [1]. With the increase in data rates and an increase in the optimization within the hardware, an optimal software solution is necessary in order to keep pace with the hardware. A normal packet processing application will spend the majority of its time going through all the layers of a network stack in Linux, however, a better approach is felt to be to hand over control to the user-space for a limited set of functionality which has been carefully designed and to allow the applications to have direct access to the packets by completely passing the kernel network stack [15]. Two such approaches proposed by researchers are PF_RING and Netmap. In addition, Freescale also has a fast path technology called USDPAA.

This project evaluates the performance of these two open source libraries by implementing them on different networks and a design is proposed regarding how to use the Netmap application on top of USDPAA, which will leverage the DPAA hardware acceleration mechanism. Such solutions involving the use of applications that can be portable in relation to multiple platforms, can lead to a considerable reduction in both costs and overhead for the developers.

1.2 Overall aim and scope
The overall aim and scope of this dissertation is to study and analyze the different methods used to achieve fast packet processing in multi-core systems. The idea is to evaluate these methods by means of different parameters and to propose a design solution, which uses one of these open source libraries on top of the Freescale USDPAA framework. The main focus of this project is to conduct a research study in order to determine whether one of these libraries can be used with USDPAA and
what design mechanism can be used in order to achieve this goal. The scope of this project is to port an application from one of these libraries onto USDPAA and to determine its performance on the Freescale QorIQ P4080 processor.

1.3 **Concrete goals**

The goals set for this project work are stated below.

- Study generic concepts for embedded systems, multi-core, telecom and user-space fast path functionality in Linux.
- Analyse how modern telecom systems are designed using Freescale devices and how state-of-art SoC devices are designed to accelerate these applications.
- Study and evaluate PF_RING, Netmap and USDPAA including writing/ porting basic applications on top of these libraries to test performance, ease of use etc.
- Document basic functionality and characteristics of PF_RING, Netmap and USDPAA, comparing how they work.
- Design a solution regarding how to port PF_RING or Netmap into or onto USDPAA leveraging the DPAA hardware acceleration.
- Document the behaviour of such applications with Freescale hardware.

1.4 **Outline**

The outline of this report is organized into seven chapters. Chapter 1 provides a brief introduction of the concept. Chapter 2 describes the background and provides a detailed comparison (evaluation) of these technologies. Chapter 3 demonstrates the methodology adopted to test these libraries. Chapter 4 offers an insight into the solution designed to port a Netmap application on top of USDPAA. Chapter 5 discusses the result section of this report. Chapter 6 concludes the work performed during this project work and Chapter 7 is aimed at Future work.

1.5 **Intended Audience**

This report is presented for those readers who have sufficient knowledge relating to Multi-core systems and embedded systems and who are very familiar with C/C++ and Linux Kernel.
2 Fast Path Technologies

Processing and monitoring high speed networks is an integral part of Embedded/ Multi-core systems. The rate at which the packets are captured and handled explains the depth in the system [1]. To capture packets in low speed networks, of around 100Mbit/sec, it is possible to use a number of commodity tools and hardware based on a Library called ‘libpcap’. However, depending on the situation, there is sometimes more concern about the number of packets captured rather than the speed at which those packets are captured [5].

One means of improving packet capture performance in these situations and, also, in high speed networks is to use measurement boxes, custom capture cards etc, dedicated to monitor high speed networks. Another means is to make use of Fast packet I/O frameworks with common personal computers (PCI/ Express bus/ NIC card) [4]. Such frameworks are PF_RING, Netmap and USDPAA (Freescale product) which are discussed in detail in this chapter.

2.1 Traditional Linux Kernel

One significant advantage of using Linux is that it allows for the loading of new code into the kernel which enables a programmer to avoid dealing simultaneously with both user code and kernel code. This piece of code is known as ‘kernel module’. This mechanism allows programmers and developers to customize the handling of packets in a faster and more efficient manner. The packets can then be processed to the kernel stack. After the packet has been processed it can then be returned or discarded thus involving no additional time being spent on it. By using these kernel modules, packet processing can be performed immediately instead of cycles being spent in moving in and out of the user-space, completing the packet capture process and traversing the network stack [14].

However, it is still the case that, developing a kernel module is still more costly than for a normal module. The developers must be aware of all the internal data structures since, an error in the module can cause instability to the entire system. Thus using kernel module for packet processing is a risky scheme but, the performance improvements are
exciting. Figure 2.1 is a comparison of a traditional Linux kernel packet processing versus zero copy model packets processing [15].

Fig.2.1- Compare traditional model with zero-copy model. [15]

2.2 PF_RING

PF_RING is a kernel module, namely, a traffic analysis framework that enables efficient packet capturing and processing. It processes packets at high rates while providing a consistent API for packet processing. It significantly decreases the journey of a packet from wire to user-space of the kernel. PF_RING acts as a socket and makes use of device polling to enable packet capturing at high rates. It does so by creating a straight path from the network driver card to the user-space for incoming packets. It also provides a mechanism for preventing packets from being passed to the upper levels of the kernel, which can improve the processing of these packets at much higher rates. It can also be used for active traffic analysis and manipulating traffic. One of the key features of this module is that it does not depend on any device driver, although they can be optimized, for example an optimized network device driver can directly push the incoming packets into the PF_RING buffer by passing the standard kernel path [4]. Thus in order to achieve the maximum throughput it is very important to make use of these specialized drivers also known as PF_RING-aware drivers.
2.2.1 PF_RING Architecture

The PF_RING architecture consists of three main blocks; the accelerated kernel module (AKM), which facilitates the copying of low level packets from the network adapter into the PF_RING rings or sockets. The user-space RF_RING SDK that provides transparent PF_RING support to the user-space applications and finally, the PF_RING-aware drivers that can be used to improve the process of packet processing by efficiently copying packets from the network driver to the PF_RING without passing through the kernel data structures [3].

![PF_RING vs Legacy device Architecture](image)

Basically PF_RING is implemented as a socket type on which user-space applications can communicate with the kernel module. Applications can obtain a PF_RING handle and issue API calls. Such handles can be bound to an RX queue (on multiple queue network adapters) or to a physical network interface.

As mentioned above the PF_RING performs device polling and receives packets from NICs (Network Interface Card) with the assistance of Linux New API or NAPI. NAPI copies the packets from the NIC to the PF_RING buffer and then the application, which resides in user-land, reads the packets from the ring. Packets are read from a memory buffer allocated at the creation time. After a packet is read, it is discarded and the allocated memory is used to accommodate new packets.
Hence the PF_RING does not preserve packets and the kernel discards them after they have been read by the application [5]. The packet journey in the PF_RING before being queued into a PF_RING ring is shown in Figure 2.3;

![Fig.2.3(a)- Packet TX via PF_RING [3]](image)

![Fig.2.3(b)- Packet RX via PF_RING [3]](image)

### 2.2.2 Socket Clustering

The performance of the PF_RING can be enhanced in order to process packets more efficiently, by implementing two mechanisms named packet clustering and balancing. Ring/socket clustering is the ability to align PF_RING sockets to network interface bonding. Clustering works at the PF_RING socket level and it is basically a mechanism for balancing traffic across packet consumers. Its use can vary i.e it can be used to run several applications, each analyzing a portion of the overall network traffic. However it is possible for it to be used to create multi threaded applications that can each have a private per-thread socket. This means that different applications can bind themselves to a specific cluster id (using `pf_ring_set_cluster`) and for which each will analyze a portion of the packet [3].

![Fig.2.4(a)- Non cluster approach [17]](image)

![Fig.2.4(b)- PF_RING Socket cluster approach [17]](image)
PF_RING filtering rules combine the better of these two mechanisms by implementing traffic balancing for those packets that match a certain filter. The idea is to specify the same type of filter for various sockets that are grouped together. Those packets matching the filter criteria are then forwarded only to one of the sockets. A good flow chart representation of this concept is shown in Figure 2.5 [6].

![Flow chart](image)

Fig 2.5- Combining Filtering with Balancing [17]

### 2.2.3 PF_RING from developer's prospective

In order to optimize the process of packet capture and packet processing, a new type of socket, called PF_RING, is created which is based on the idea of a circular buffer where the arriving packets are stored.

The buffer is allocated when a socket is activated and deallocated when a socket is deactivated. Each socket will have its own buffer. When a packet arrives at the network card and is received by the device driver, the packet is copied to the pf_ring/socket. It might also be the case that the packet is discarded (if the sampling rate is not valid or the ring/buffer is full). As these packets are not directly forwarded to the upper layer but, are instead handled by the pf_ring, it increases the overall performance as the upper layer does not have to deal with the handling of the packets.
After the packets are copied in the ring, the socket ring/buffer is now exported to the user-space application through mmap\(^1\). In order for a user-space application to access the ring it is required to open the file and call mmap() syscall on it in order to obtain a pointer to the circular buffer. As in the case in relation to copying, the kernel copies the packets to the ring and moves the write pointer forward in a similarly manner to the user-space application reading a packet from the ring and moving the read pointer forward.

The new incoming packets overwrite the packers that have been read by the user-space application. It should be noted that the memory is not typically allocated/ deallocated by the packets but, is instead simply overwritten and, in the case of a full buffer, the packet is discarded. The length of the buffer and the size of the bucket can both be configured by the developer for all the sockets.

The use of this approach has several benefits: the incoming packets are not queued into the kernel network data structures but, are instead, directly copied into the socket (ring). Multiple user-space applications can open multiple PF_RING sockets simultaneously. The mmap primitive allows the user-space applications to access the circular buffer with no overhead due to the system call mechanism.

### 2.2.4 Circular buffer

A circular buffer or a ring buffer is a data structure that uses a single, fixed size buffer in the memory. This structure is used to buffer incoming packets. One key feature of a circular buffer is that it does not require shuffling its elements when one is consumed. In the case of a non circular buffer, it is necessary to shift all the elements after the consumption of one of them. The circular buffer is suitable as a FIFO.

A queue with a fixed maximum size is suitable for the implementation of a circular buffer; all the queue operations are at a constant time. However, expanding the buffer will increase the cost as the buffer will require a shifting memory. Generally the buffer works with four pointers; one pointer for the buffer in memory and another to the end of the buffer in memory, the third indicates the start of data and the fourth indicates the end of the data [5].

---

\(^1\) mmap is a system call and it creates a new mapping in the virtual address space
2.2.5 PF_RING API

The internal data structures of the PF_RING should be hidden from the user, who is only able to alter devices and packets by means of the API defined in the file pfring.h that comes with the PF_RING package. By default the library returns a negative value for exceptions and errors. A positive value means that the code has been executed successfully [3].

‘eth0’ to ‘eth5’ are the valid names that can be use for device naming. They are the same as in ifconfig or libpcap.

Socket initialization: pfring* pfring_open(char *device_name, u_int32_t caplen, u_int flags)

This line of code is used to initialize the socket and obtain a handle of type struct. It is possible to use physical devices e.g. ethX and virtual devices e.g. tapX to receive packets. In order to open a device it is necessary to possess the correct user rights.

The parameters of this function contain the device_name; name of the PF_RING device to be opened, Caplen; the maximum length of the capture packet, Flags; used for different type of control options. In relation to a successful execution, a handle is returned otherwise it returns a NULL value.

Device termination: void pfring_close(pfring *ring)

This call is used to close a PF_RING enabled open device. A device should always be closed before leaving an application.

The parameter is ring; the name of the PF_RING handle that an attempt is being made to close.

Reading packets: int pfring_recv(pfring *ring, u_char** buffer, u_int buffer_len, struct pfring_pkthdr *hdr, u_int8_t wait_for_incoming_packet)

This call will return an incoming packet if it is available.
The input parameters are; ring which is name of the PF_RING handle ready to be read buffer, which is the memory buffer reserved for the incoming packet (this parameter is a pointer to a pointer) and buffer_len which is the; variable representing the length of the memory area. In the case where the incoming packet is too long, it will be cut to fit the allocated space. Hdr; hd is also a memory space and the header of the packet will be copied here. wait_for_incoming_packet; if the value is 0, the availability of the packet will be checked and if it is not available then this call is blocked until a packet has arrived and is available. It will return 0 in the case where no packet is received or available, return 1 if successful and -1 if an error has occurred.

Parsing packets: int pfring_recv_parsed(pfring *ring, u_char** buffer, u_int buffer_len, struct pfring_pkthdr *hdr, u_int32_t wait_for_incoming_packet, u_int8_t level, u_int8_t add_timestamp, u_int8_t add_hash)
This call uses more functions or parameters with pfring_recv for reading packets. e.g parsing
The extra parameters are; level, which is a header level where to stop parsing add_timestamp, which add the time stamp add_hash, which computes the hash function.

Code example:

```c
struct pkt_parsing_info {
  /* Core fields (also used by NetFlow) */
  u_int16_t eth_type; /* Ethernet type */
  u_int16_t vlan_id;  /* VLAN Id */
  u_int8_t l3_proto, ipv4_tos; /* Layer 3 protocol/TOS */
  u_int32_t ipv4_src, ipv4_dst; /* IPv4 src/dst IP addresses */
  u_int16_t l4_src_port, l4_dst_port; /* Layer 4 src/dst ports */
  u_int8_t tcp_flags; /* TCP flags, 0 if not available */
  ....
};
```

int pfring_next_pkt_time(pfring *ring, struct timespec *ts)
This call returns the arrival time of the next packet.
Input parameters are; ring, the name of the PF_RING handle to check Ts, the structure where will the time will be stored.
It will return 0 in the case of success and a negative value in the case of failure.
Ring Clusters : int pfring_set_cluster(pfring *ring, u_int clusterId, cluster_type the_type)

This function call allows for a ring to be added to a cluster of rings. When sockets are clustered, they share the incoming packets. This method is very useful in multi-core systems for sharing packets in the same address space.

The input parameters are; ring, PF_RING handle to be cluster Clusterid, the id of the cluster to which the ring will be bound, the_type, type of the cluster (per flow, round robin).

It will return 0 on success and a negative value on failure.

Code example:

```c
if((pd = pfring_open(device, promisc, snaplen, 0)) == NULL) {
    printf("pfring_open error\n");
    return(-1);
} else {
    u_int32_t version;
    pfring_version(pd, &version);
    printf("Using PF_RING v.%d.%d.%d\n", 
           (version & 0xFFFF0000) >> 16, (version & 0x0000FF00) >> 8, 
           version & 0x000000FF);
    
    if(clusterId > 0) {
        int rc = pfring_set_cluster(pd, clusterId);
        printf("pfring_set_cluster returned %d\n", rc);
    }
```

after the packets are filtered, they are then balanced across the sockets. It is very effective to use load balancing across applications in multi-core architecture.

Code example for balancing:

```c
typedef struct {
    ..... 
    u_int8_t balance_id, balance_pool;

    /* If balance_pool > 0, then pass the packet to PF_RING caller only if 
       (hash(proto, sip, sport, dip, dport) % balance_pool) = balance_id */
    ..... 
```
Remove Ring from Cluster: int pfring_remove_from_cluster(pfring *ring)
This call allows a ring to be removed from a cluster.
The input parameters include; ring, the PF_RING handle to be cluster.
It will return 0 on success and a negative value on failure.

PF_RING: Packet Sampling : int pfring_set_sampling_rate(pfring *ring, u_int32_t rate)
This call uses the direct implementation of sampling in the kernel.
Sampled packets are those that pass all filters.
Input parameters are; ring, PF_RING handle on which sample is applied, rate, which is the sampling rate.

Packet Filtering:
Two approaches are used to handle packet filtering in a PF_RING. One method is called Hash filtering and the other is known as Wild card filtering.

Hash Filtering: int pfring_handle_hash_filtering_rule(pfring *ring, hash_filtering_rule* rule_to_add, u_char add_rule)
This call is used to add or remove hash filtering rule.
Input parameters are; ring, PF_RING handle from which to read, rule_to_add, rule that is to be added to the filter, add_rule, if value is positive the rule is added and if zero then the rule is removed.

BPF Filtering: int pfring_set_bpf_filter(pfring *ring, char* filter_buffer)
This will set the BPF filters through the PF_RING API
Input parameters are; ring, PF_RING handle on which the filter will apply, filter_buffer, which is the filter to set.
2.3 Netmap

Netmap is a framework for fast packet I/O that enables commodity operating systems to process a high number of packets without the necessity to make changes to the applications. It provides the user-space applications with a very fast access to network packets, both on the transmit and receive side, including packets to or from the host stack. It provides a simple to use API that is very well integrated with the existing operating system mechanisms, and which is not tied to a specific device or hardware features [7].

In order to achieve a high performance Netmap uses lightweight metadata representation. This method of representing data is easy to use, very compact and is able to hide some device features. Lightweight metadata representation is also able to process large number of packets in each system call. The memory buffers are of a fixed size and are only pre allocated if the device is open. The user-space applications have direct protected access to the packet buffers hence it removes the cost of data copying into the buffers. The main advantage of using Netmap is that programs using Netmap require no inner detailed knowledge of the network controllers or libraries [9]. Additionally, the memory buffer and ring format are also independent of the card.

2.3.1 Netmap Architecture

The architecture of Netmap is fairly simple. The framework is built around a shared memory region. The region can be accessed both by the user-space application and the kernel. It contains memory buffers and descriptors for all the packets managed by an interface. They are allocated only once, namely, when the interface is brought up and they remain bound to the interface. The buffers are of fixed size and they are sufficient to store the maximum size packets. On the other hand the descriptors are eight bytes each in size and are very compact [8]. These descriptors are part of netmap_ring data structures and are stored in a circular array that maps one to one to the NIC ring.

The rule to access shared data structure is very simple; the ring is always owned by the user-space application except during the execution of a system call. The same rule applies for the buffers. A general idea of the Netmap architecture is shown below in Figure 2.6. [7];
It can be seen from Fig.2.6 that when the kernel is running in the Netmap mode, the NIC rings are deliberately disconnected from the host stack (as shown in the diagram) and the packets are directly exchanged between the NIC rings (network adapter) and the Netmap API (application). This is achieved by adding two more rings to the application [7].

Fig.2.7 shows the data structures in the Netmap architecture. These data structures are designed to provide; efficient communication between the host stack and the NIC, decrease the per packet overhead, forwarding of packets between interfaces and support for multi-core systems.
In order to deploy these features, each interface is linked with three types of user objects as shown in figure 2.7: netmap_if, netmap rings and pkt_buf. All objects for the interfaces, enabled by Netmap, are located in the same memory region by the kernel, and are shared by all the user processes. Since the shared memory is mapped by the kernel threads and processes in different virtual address spaces, all the memory reference in the region must have relative addresses. One means of dealing with this is to make references as offsets between the primary and secondary data structures. Thus the pointers can be calculated in a position independent manner.

Pkt_buf (packet buffers) are of a fixed size and these buffers are shared between the user processes and NICs. These buffers are pre-allocated when the interface is running in the Netmap mode, so that during the network I/O there is never a requirement to allocate them. In the memory, each buffer is represented by a unique index [9]. The index can be translated into a physical address, used by the NIC’s DMA engines as well as into a virtual address by the kernel or user process. The data describing the buffer are stored in slots that form part of the Netmap Ring.

As can be seen from Fig.2.7, the netmap ring is device independent and includes; ring_size, the number of slots in the ring, cur, the current read/write position in the ring, avail, available buffers, flags, used to request special operations, buf_ofs, offset value between the ring and the beginning of the array of pkt_buf’s and flags, len, index fields. The netmap_if contains some read only information describing the interface.

All the data structures in Netmap are shared between the kernel and the user-space application but, they have proper ownership of data. For example a netmap_ring is always owned by the user-space application except when a system call is executed. It is also not interrupted by handlers and other kernel threads. Pkt_buffs are also owned by the user-space application. The remainder of the buffers are owned by the kernel and these buffers only access the NIC.
2.3.2 Netmap User API

In order to use Netmap, the programs must place the interface in Netmap mode. This is achieved by calling `open (“/dev/netmap”)` and issuing an `ioctl()` on the file descriptor to switch the NIC to “netmap” mode. Executing these lines will disconnect the NIC from the host stack although standard `ioctl()` is still in action, which allows the operating system to configure the interface but, they are no longer able to exchange packets with the host stack. On success, the function returns the size of the shared memory, where all data structures are located and the offset of `netmap_if` within the same location [8]. The function `ioctl(.., NIOCREG, arg)` argument contains the name of the interface, and indications regarding which rings should be controlled via this file descriptor. A call to `mmap()` will make the shared memory region accessible to the process.

2.3.3 Efficient Packet Processing

Using Netmap will reduce the processing cost and improve the throughput making a system more efficient. First of all, all the buffers and Netmap rings are in the shared memory location, which is shared by both the kernel and Netmap clients hence handling more than one NIC within a process is fairly simple. A zero copying method can also be deployed between the interfaces. Secondly, data copying is also removed and the only remaining operation is to update the NIC ring after validating the information in the `netmap_ring` and start sending packets by writing to the NIC registers.
2.3.4 Multiple Cores/NIC’s

The design of the shared memory region reflects that the Netmap based interfaces can have multiple netmap rings. Applications are presented with a choice to either attach all the rings or, just one ring, to the file descriptor while binding a file descriptor to an NIC. In a conventional way, network cards implement multiple NIC rings to distribute the load to multiple CPU cores without lock contention. In the case of Netmap, the NIC rings map to an equal number of netmap rings, that can be associated to file descriptors (independently) and can be used by different threads/processes. Setaffinity(), is a system call which allows the mapping of different threads to cores without any special mechanism. Netmap also uses two more rings per NIC to communicate with the host stack. Packets from the host are visible in an Rx Netmap ring and packets written to Tx (extra Netmap ring) are passed to the host.

2.3.5 Blocking I/O Primitives

In Netmap blocking I/O is supported by two systems calls select() and poll(). Netmap descriptors can be passed to these system calls which are unblocked when the ring has empty slots. The system updates the status of the ring before returning from these system calls. Hence, applications spinning on an event loop require only one system call per iteration. Although using poll() on a Netmap descriptor has the same complexity as an ioctl() system call, plus poll() also requires some additional optimization to reduce the number of system calls in an application.

2.4 USDPA

USDPA stands for User Space Data Path Acceleration Architecture (a merger of “User Space” i.e. user-space Linux and DPAA – Data Path Acceleration Architecture) is a framework used to gain direct access to software portals for a DPAA buffer manager (BMan) and a DPAA queue manager (QMan) from the Linux user space. These software portals are memory mapped hardware components containing dequeue ring, message ring, etc. These items in the portals have specific addresses in the physical address map. The library operates by allowing the physical address ranges to be mapped into the virtual address space of the user space processes [13].
In the case of hardware mapping, the memory range for the portals must be mapped using the correct cache attributes and, the correct instruction sequences must be used in order to access the portals.

![Diagram showing USDPAA Software Portals](image)

**Fig.2.8**- USDPAA Software Portals [13]

This project uses USDPAA with a SoC such as P4080 and the framework assumes the context of a single SMP Linux instance. The instance may have many user space processes, which can be multi threaded using pthread in Linux. The processor P4080 has 10 QMan software portals and 10 BMan software portals which means that the USDPAA threads can be limited by the number of software portals that the particular SoC provides. However, a thread may share many resources so a process written with awareness and the ability to share the hardware can have a finite number of threads to one hardware.

### 2.4.1 USDPAA Components

USDPAA contains several components, which can be used in the context of standard SMP Linux on the Freescale family. Some of the components are listed below;

### 2.4.2 Device Tree

Device Tree used to boot Linux contains some configuration and resource details. The kernel uses this data structure to determine a system’s attributes and devices. Both the portals, QMan and BMan are declared within the device tree, as are the Ethernet interfaces and other attributes from the Frame Manager (FMan).
In the current Open Firmware driver implementation, it is the application’s responsibility to initialize the driver layer prior to initializing or using any other driver layers that may be dependent on it. This is achieved by calling of \texttt{init()} API. The Queue Manager (QMan) and Buffer Manager (BMan) are the vital parts of USDPAA. They are both similar in the broadcast sense [13].

### 2.4.3 Queue Manager (QMan) Driver

QMan operates under these set of rules:

- Management and initializing of all aspects of QMan that is global.
- Management and initializing of per software portal QMan operations.

The driver can be split into two parts; QMan software portal driver and QMan global driver. There is an instance of the software portal driver for every QMan software portal that exists physically on the SoC and there is only one instance of the global driver per SoC.

User space processes access QMan using software portals that are dedicated to the user space. The QMan API abstracts it, but the access is implemented using a standard Linux facility known as UIO. UIO is a framework for user space drivers. A user space process opens such a device and makes a request to map the device directly into the process address space. At this particular point, the process can access the device hardware using normal load and store instruction [13].
2.4.4 Buffer Manager (BMan) Driver

QMan and BMan are similar in many ways. For example, the BMan driver and its software support are very similar to QMan. The software portals of BMan may be used in the kernel or in user space in a similar manner to that for the QMan portals. As is the case for the QMan, the same BMan software API is available on both the kernel and in the user space.

2.4.5 DMA Memory

The DPAA (Data Path Acceleration Architecture) hardware from Freescale provides several peripherals such as FMan, SEC, and PME that read and write memory directly using DMA. Buffers used for peripherals DMA must be allocated from memory with special characteristics:

- The memory must be contiguous physically.
- The memory must be addressable by the peripheral.
- The memory must not be swapped out by Linux while accessing it.
- There must be an efficient mechanism to convert the physical addresses used by the peripheral hardware to and from the effective addresses used in a user process.
- It is easy for software to allocate memory from physically contiguous memory regions for both BMan and DPAA usage.
- In order to increase performance, it is good to use the Power core’s TLB1 mechanism to map large physically contiguous memory regions of this sort.

A memory that meets all the above characteristics is known as a DMA memory. Drivers for all DMA capable devices must use a DMA memory for buffers [13]. This is true for both conventional in kernel drivers and user space drivers.
2.4.6 Assumptions and Use cases

The main objective of USDPAA is to enhance application performance. USDPAA provides low level access to I/O functionality. It does not employ complex and costly abstractions. Applications deal directly with QMan and BMan via their software APIs. This creates a helpful layer between the application and the hardware.

The USDPAA software makes some assumptions which are listed below;

- A thread should be made affine.
- USDPAA threads may make use of arbitrary Linux system calls.
- Threading is via standard Linux pthreads and the standard Linux GNU C lib.
- USDPAA threads are compatible with general Linux services such as debuggers e.g. gdb.
- Although it is assumed, there are no examples of using more than one USDPAA thread per core but it is still possible to use more than one USDPAA thread per core.
- A fixed chunk of 64 Mbytes of memory is allocated for USDPAA use. However, it is still for a user to also configure according to the application requirements.

USDPAA is intended to support multiple use cases which are listed below;

Run-to-Completion

The following are the attributes of a run-to-completion case;

- Cores hosting USDPAA threads may be configured to run nothing in the user space other than the core’s USDPAA thread.
- USDPAA threads poll their portals for packets.
- Hardware based priority scheduler in QMan distributes work to cores.
- Multiple USDPAA threads per core are allowed.

Interrupt Driven Model

USDPAA also supports an interrupt driven model that allows benefits such as co-operating with other threads on the same core.
- Allows co-operation with other threads on the same core.
- Potential power savings by not polling all of the time.
- Multiple threads (USDPAA or otherwise) execute on the same core.

Developers can choose the model based on their individual requirements [12].

### 2.4.7 Linux Support

The main goal of USDPAA is to provide an API, as it provides user space processes access to DPAA functionality via a user space device driver and device oriented C language APIs which are layered upon them. Legacy applications must be modified in order to use these APIs. It is possible that some legacy user space facilities could be accelerated in a manner that automatically applies to legacy user space applications, if means other than or, in addition to, USDPAA are used. USDPAA’s objective is to maintain Light-Weight-Executive style support for run-to-completion processing with operating system overheads held to very low levels through careful configuration of Linux and careful use of Linux features. LWE style run-to-completion processing is assumed to actively use few standard Linux features and this fact can be considered in order to avoid the overheads of those features.

However USDPAA is also designed to be more flexible than the LWE and to support a wider variety of use cases. USDPAA’s general idea is to allow and encourage the use of standard Linux features and reasonably arbitrary configurations of Linux, but, developers also must keep in mind the performance impact of their use. USDPAA also imposes scheduling constraints since applications could not move between cores due to their hardware ties. They generally also execute alone on a core although this is not a requirement.

### 2.5 QorIQ P4080 Processor

The Freescale Semiconductor QorIQ P4080 communications processor contains eight Power Architecture cores, each integrated with L1 and L2 caches. It also contains a variety of memory and I/O controllers. It also supports a shared L3 cache. On-chip components are connected via CoreNet Coherency Fabric, which provides the connectivity between all the hardware components. Figure 2.12 shows the block diagram [11],
Fig.2.10- QorIQ P4080 Processor [12]
3 **Methodology**

This Chapter describes the methodology of the implementation and evaluation of the libraries discussed in Chapter 2. Some of these tools are very hardware specific and are not compatible with every network device, so different approaches were taken to ensure that these libraries are configured correctly. These methods are an alternative to the default packet processing mechanism provided by Linux. In order to send a packet from a device driver to the application layer, a packet must pass through the entire network stack in the kernel which is very time consuming as some applications do not require this feature.

Thus, in order to avoid the kernel stack and take the Fast Path, advantage of these alternative libraries will be taken to totally avoid the kernel stack and adopt different approaches to processing packets to the application layer in the user space at a faster rate.

### 3.1 Software/ Hardware detail

It is important to mention, at this point, both the hardware and software details used during the experiment, as they had a significant impact on the outcome of the results. A study of this nature only represents a snapshot of the entire reality and the details regarding the software/hardware become important. The machine types and their specifications are given below.

**PF_RING | Broadcom BCM5752**
- Processor: Intel Dual core CPU T2400 @ 1.83 GHz, 32-bit
- Memory: 1 GiB (DDR2, 800 MHz)
- Kernel: Linux 3.2.0-31
- Network Card: NetXtreme BCM5752 1-Gigabit Network card.

**Netmap | Intel Pro/1000 GT**
- Processor: Intel Core i7-2600K CPU @ 3.40GHz, 32-bit
- Memory: 8 GiB (DDR3, 1333 MHz)
- Kernel: FreeBSD kernel
- Network Card: Intel Pro/1000 GT 1-Gigabit card.

**USDPAA | Ethernet SGMII controller**
3.2 PF_RING

This approach, as discussed, comes from Luca Deri’s PF_RING protocol family. Recognizing the cost of per packet system calls, the PF_RING creates a ring buffer in the kernel and uses a kernel/user memory map to share that buffer with the user-space program. The PF_RING was implemented in this case on Ubuntu 12.04 LTS by installing and configuring it. PF_RING is compatible with most of the Network Interface Cards. In order to send/receive packets on PF_RING, two user-space applications, PF_SEND and PF_COUNT were used.

Tools used during the experiment

Ubuntu 12.04 LTS, OpenSUS, PF_RING, Linux Kernel, Wireshark, gcc, Cygwin
3.3 Netmap

The second part of this project is to implement and evaluate Netmap. However, as Netmap is not natively built in any version of Linux/Ubuntu Operating systems, the task was highly complex and the decision was taken to use another operating system called FreeBSD in order to configure Netmap. Netmap is a part of FreeBSD Operating system at this point and the code is available to reconfigure it with the OS kernel. The entire kernel of FreeBSD was recompiled with Netmap built inside the kernel. This task was somewhat time consuming as there was a requirement to determine compatible hardware and make it available in addition to updating all the kernel ports to ensure that the kernel configuration and the build worked in an appropriate manner. Netmap is not yet available for all the NIC drivers which means that not every NIC works with Netmap. Netmap only supports FreeBSD 8/9 and HEAD and, additionally, the support for the following NICs is included, em (Intel Pro/1000 GigaE), ixgbe (Intel 10GigaE), e1000 (Intel) and re (Realtek) 1GigaE drivers etc.

Once all the configuration and installations were ready, the next step was to test it on a network and regarding this aspect, there were two terminals and a switch connected. A user-space application called ‘pktgen.c’ was used to send/receive packets on the network.

Tools used during the experiment
FreeBSD 9RC-1, Kernel Configuration, em (Intel Pro/1000 GT), Ethernet Port, CVSUP tool

3.4 USDPAA

USDPAA is a framework used to gain direct access to software portals for a DPAA buffer manager (BMan) and a DPAA queue manager (QMan) from the Linux user space. The third phase of this project involved testing this software on the network. USDPAA was tested on a Freescale QorIQ P4080 processor. The P4080 was connected to the terminal (my PC) via a Serial Port, and Ubuntu 12.04 was installed on the terminal. Tera-Term was used to communicate to the device via the serial port. The network consisted of a switch, two terminals and a P4080 device. The two terminals were connected via the switch and terminal 2 was connected via a serial port to the device. The USDPAA and Linux images were loaded to the P4080 device by using TFTP server
services. Packet transmission was carried out on the Ethernet port. USDPAA has 8 different Ethernet ports.

After the ports were selected and U-boot was set up, the connection was tested via ping and a trial tftp transfer, and the following is a text capture showing a successful test.

```
=> ping $serverip
Using FM1@DTSEC2 device
host 10.172.72.30 is alive
=> tftpboot 01000000 usdpaa/u-boot.bin
Using FM1@DTSEC2 device
TFTP from server 10.172.72.30; our IP address is 10.172.72.33
Filename 'usdpaa/u-boot.bin'.
Load address: 0x1000000
Loading: ####################################
done
Bytes transferred = 524288 (80000 hex)
```

Again, a user-space application ‘hello_reflector.c’ was used to transmit packets onto the network.

**Tools used during the experiment**

USDPAA, QorIQ P4080 Processor, U-Boot Utility, Tera-Term, TFTP Server, Ubuntu, Nor Flash bank4, Minicom, Yocto Project

**3.4.1 hello_reflector**

hello_reflector is a built-in USDPAA packet processing application and it is used as a benchmark to test the performance of this software tool on the different Freescale processors. USDPAA is specially designed for Freescale products and it achieves a very high performance. hello_reflector is basically an I/O which means it sends/receives frames but does very little processing on the frames. It best demonstrates the I/O capabilities of the DPAA hardware and the corresponding user-space drivers. hello_reflector always starts up on CPU 0 and it starts a single thread.

As shown in figure 3.2, the DTSEC 1..4 are the Ethernet controllers on the board and they support working in an internal loopback mode. In loopback mode, packets sent out from Ethernet Tx port are looped back to the Rx port. The flow of the reflector application is shown in figure 3.2;
User-space reflector application (USDPA)

Core 0  Core 1  Core 2  Core 7

Man - Qman - BMan

DTSEC1  DTSEC2  DTSEC3  DTSEC4

Loop back

Fig. 3.2 - Reflector applications and loop back modes
4 Design and Implementation

This chapter describes the design approach, for the development of a system that will take advantage of the open source fast path technology and work with the Freescale P4080 device. Currently USDPAA is used for packet processing on a P4080 device but USDPAA is a Freescale proprietary device which means it can only work with the Freescale products and, only user-space applications that are written for USDPAA can run on this system and they cannot be ported to any other device. The idea, in this case is to design a system that can provide more flexibility to the user space applications, which means that they can be ported to USDPAA as well as to any other open source Library. This will simplify the application developer’s task as, presently, it is necessary to re-write the entire application from the beginning in order to move to a different hardware or application interface.

After testing the two open source libraries, the decision was made to use Netmap with USDPAA for designing the new tool. The reasons for choosing Netmap over PF_RING, which are explained below [8][5];

- Netmap has much higher packet rates as compared to PF_RING on the same hardware.

- PF_RING uses socket buffers (skbufs) in the device driver and this is a major source of overhead. On the other hand Netmap uses memory mapping.

- More importantly Netmap cannot crash the operating system because it runs in the user-space and has no direct access to the critical resources.

- In Netmap, the applications use only standard system calls e.g. non-blocking ioctl() call to synchronize with the hardware.

- Programs using Netmap do not require having the depth of detail regarding the network controller and the special libraries. This applies to not only the ring and buffer format being independent of the NIC card but, also, any operation that requires the programming of the card is performed within the kernel.
In addition, other techniques such as a card’s packet buffers, modelling the send/receive queues as circular buffers and I/O batching are used to enhance the throughput.

The next step is to design a model that works best with USDPAA. Thus the idea is to use Netmap with USDPAA for efficient packet processing as well as to make the user-space application portable to both USDPAA and Netmap. This means that the same application can work on both Netmap and USDPAA and hence it will be portable. The design must also ensure that the same functionality is achieved with different performance results. This is the most difficult task since it requires a great deal of coding analysis and optimization of the existing code in such a way that it still maintains its complete functionality. USDPAA uses different data structures and coding style as compared to Netmap, so an API design is required to create an interface between the two libraries. However dealing with open source software has some restrictions since there are legal issues in open source and free software codes. Both of these softwares are maintained by means of different GNU general public license terms. Some licenses allow users to use the code in proprietary software without any issue, but some licenses requires that the original open source code must be declared as open source and some licenses require that the entire software program must be declared as a open source software. In this case, the two libraries are maintained by different users/ organizations and to develop a new piece of software program and use it with a Freescale product will require agreement between both the parties, which is far beyond the scope of this project.

4.1 API Design
The API design describes how USDPAA and Netmap can coexist and use the same application with the same functionality. According to this design, Netmap is used on top of USDPAA, so the application interfaces with Netmap and then Netmap interfaces with USDPAA and the hardware. The application used for this design is called ‘pkt-gen’ from Netmap which can both send and receive packets over the network.
Figure 4.1 shows the model. The hardware layer consists of Freescale Processors Services and the Processor, the Kernel layer consists of the Operating System (Linux) with memory mapping and the user-space layer consists of a packet processing library (USDPAA and Netmap) and an application. As previously explained, the idea is to use Netmap on top of USDPAA and thus an API is required to communicate between USDPAA and Netmap. USDPAA and Netmap use the same coding language, but the data structures, system calls, memory management etc are all different. A detailed diagram is given below in Figure 4.2 which illustrates the user-space layer of the design.
In the above diagram, on the left side is the native Netmap design and on the right side is the native USDPAA design. The model in the middle is proposed for the design of a new software program that can use both Netmap and USDPAA for packet processing on a P4080 Freescale device. Pkt-gen is the user-space application that will run on Netmap as well as on USDPAA with the same functionality.

In order to utilize this, it is necessary to add an API or Netmap Wrapper layer between the Netmap and USDPAA. Since the application pkt-gen is initialized with Netmap, simply using it on top of USDPAA will not work. In order to accomplish this, a ‘hello_reflector’ is required, which is an application initialized with USDPAA portals. ‘hello_reflector’ is used to initialize the USDPAA portals (BMan and QMan) with the P4080 device and then pkt-gen is used to perform the fast packet processing [13]. The application pkt-gen, which runs on Netmap natively, is now able to run on USDPAA with the same functionality. Hence the user-space application is now portable.

![Diagram](image)

**Fig.4.3- User-space portable application**

It now becomes possible, without changing the original application, to use it with both Netmap and USDPAA. If the requirement is to use it on USDPAA, it merely becomes necessary to run it on top of the wrapper layer and it will work in a satisfactory manner. Hence, with the addition of a Netmap wrapper layer to the USDPAA, it is now possible to run an open source user-space application with a proprietary tool such as USDPAA and Freescale P4080 Processor.
4.2 **Wrapper layer**

A wrapper is a software or a data structure that contains parts of other software in such a way that the contained elements can exist in the newer system. A wrapper is basically a collection of libraries that acts as an interface/bridge between a client application and a library that is written using an incompatible technology [10]. As in this case, the Netmap uses different calling conventions, memory management, initialization etc. as compared to USDPAA, in order to execute that code with USDPAA, a wrapper had to be created to act as an interface or bridge between its caller and the wrapped code. In order to write a wrapper for a data source, it is necessary to know the basic flow of development in order to create a wrapper module. Wrapper writing involves certain phases e.g. Concept, Design, Code, Build and package and finally Test Phase.

In order to initiate the building of a wrapper, the primary task is to understand the Netmap user-space application (pkt-gen.c) and optimize the current code, which is about 2000 plus lines of C code. Different techniques were used to optimize this code. Firstly, the code was very generic, thus it was changed to make it more specific to the current requirements and static in relation to the hardware. The size of the code was enhanced by using Input/Output bound techniques. Other techniques such as minimizing the use of global variables, declaring functions other than global variables as static within the file, use of macros instead of small functions to save CPU time, use of header files etc were also used. Such methods were very useful as the code was reduced by almost 50%. The same approach was used with the USDPAA user-space application ‘hello_reflector.c’, which was also approximately 2000 lines of code written in C.

With the P4080 device, the initialization of the Buffer Manager (BMan), Queue Manager (QMan) and Frame Manager (FMan) is achieved by using the code from the ‘hello_reflector’, which is natively built on USDPAA. The following is part of the code from ioctl(), which performs the initialization.

```
/* Load the Qman/Bman drivers */
ret = qman_global_init();
if (ret) {
    fprintf(stderr, "Fail: %s: %d\n", "qman_global_init()", ret);
    exit(EXIT_FAILURE);
```

/* Load the Qman/Bman drivers */
ret = bman_global_init();
if (ret) {
    fprintf(stderr, "Fail: %s: %d\n", "bman_global_init()", ret);
    exit(EXIT_FAILURE);
}
ret = qman_alloc_pool_range(&pchannels[0], NUM_POOL_CHANNELS, 1, 0);
if (ret != NUM_POOL_CHANNELS) {
    fprintf(stderr, "Fail: no pool channels available\n");
    exit(EXIT_FAILURE);
}
/* Compute SDQCR */
for (loop = 0; loop < NUM_POOL_CHANNELS; loop++)
    sdqcr |= QM_SDQCR_CHANNELS_POOL_CONV(pchannels[loop]);
/* Load dma_mem driver */
dma_mem_generic = dma_mem_create(DMA_MAP_FLAG_ALLOC, NULL, sz);
if (!dma_mem_generic) {
    fprintf(stderr, "Fail: %s:\n", "dma_mem_create()");
    exit(EXIT_FAILURE);
}
printf("DMA region created of size %zu (0x%zx)\n", sz, sz);

The critical part is to deal with the OPEN() and IOCTL() calls. In USDPAA, the ‘hello_reflector’ does not use an IOCTL() call for a device control but, in Netmap ‘pkt-gen’ both OPEN() and IOCTL() are used to open the ring and then to control the devices on the system and to manage the buffers in the memory. Thus an IOCTL() call is introduced in the wrapper which is used to initialize the BMan and QMan portals using the native call method e.g. call with default arguments.
Here IOCTL() contains all the initialization of the software portals for the USDPAA and the THREADFUNC() call contains all the thread execution in the USDPAA. After the Netmap has been initialized with all the parameters, it will execute the IOCTL() call, which contains the code for the USDPAA initialization of the software portals and after the portals (BMan and QMan) are initialized, the compiler will then execute the threads by calling the THREADFUNC() function.

Figure 4.4 is a design model of the wrapper program also known as ‘netmap_wrapper.c’. This is the new piece of software code that can be used as an API or bridge to communicate between Netmap and USDPAA. Using a pkt-gen application on top of netmap_wrapper will ensure that the application executes correctly with the same functionality and that it is portable, which means that it is able to run on both Netmap and USDPAA.
5 Results

The results are based on the test and performance evaluation of these user-space libraries. An evaluation report is presented in the first section of this report. Results obtained from the testing are also included in this chapter.

The performance of these libraries depends on both external and internal parameters. To evaluate the performance a measurement study is conducted. There are many factors that can be used to maximize the throughput of a system and some are listed below [6].

- Network Interface Card driver
- Network card line rate
- Type of Processor
- Packet size and packet rate
- Packet processing software
- Linux Kernel version
- Linux Kernel configuration

Starting from the hardware, a network interface card on the receiver terminal can have a huge impact on the performance of the system e.g. an Ethernet card with a higher data rate such as 100 Gbps can gain more throughput as compared to that for 10 Mbps. It also depends on how well the NIC driver is optimized and, in this regard, different mechanisms are activated to use fast packet input/ output schemes. The P4080 device is equipped with eight 1 Gbps Ethernet cards and two 10 Gbps Ethernet cards. The type of processor in terms of speed, design, architecture and instruction set can also change the performance results. The processor rate at which the processor and NIC can access the memory to read/ write the packets can also have an impact on the overall performance at the hardware level. The P4080 device is designed with eight Power Architecture e500mc Core processors connected through the CoreNet Coherency Fabric.

In relation to the software, for greater packet processing, the version of the Linux kernel can have an impact on the performance and, additionally, the fact that it is constantly developing, can offer more improve-
ments to the concept of fast packet processing. On other hand, how the
kernel is configured is also important since the piece of code added to
the native kernel, can cause an overhead, hence, slowing down the
system. Other factors can also affect the outcome such as system load of
the packet receiver and the size and speed at which the packets arrive at
the network card [6]. A network can receive up to 1.4 million of 64 byte
packets per second. The following are the factors and levels of study and
implementation for this project.

<table>
<thead>
<tr>
<th>Factor</th>
<th>Level1</th>
<th>Level2</th>
<th>Level3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel Configuration</td>
<td>Custom</td>
<td>Custom</td>
<td></td>
</tr>
<tr>
<td>Machine- CPU</td>
<td>Dual Core</td>
<td>Core i7</td>
<td>P4080</td>
</tr>
<tr>
<td>Processing Software</td>
<td>PF_RING</td>
<td>Netmap</td>
<td>USDPAA</td>
</tr>
<tr>
<td>Network Card</td>
<td>Broadcom</td>
<td>Intel Pro/1000 GT</td>
<td>1/10 Gbps Ethernet</td>
</tr>
<tr>
<td>Operating System</td>
<td>Ubuntu</td>
<td>FreeBSD</td>
<td>Linux</td>
</tr>
<tr>
<td>User-space application</td>
<td>pf_send.c/pf_count.c</td>
<td>pkt-gen.c</td>
<td>hello_reflector.c</td>
</tr>
</tbody>
</table>

Table 5.1- Factors and level of testing

5.1 **PF_RING test**

As explained in the methodology section, the PF_RING test was per-
formed on two user-space applications, namely, pf_send and pf_count.
The information about the PF_RING ($cat /proc/net/pf_ring/info) is
given below.
Table 5.2- PF_RING version information

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PF_RING Version</td>
<td>5.4.4 ($Revision: 5533$)</td>
</tr>
<tr>
<td>Ring Slots</td>
<td>4096</td>
</tr>
<tr>
<td>Slot version</td>
<td>13</td>
</tr>
<tr>
<td>Capture TX</td>
<td>Yes [RX+TX]</td>
</tr>
<tr>
<td>IP Defragment</td>
<td>No</td>
</tr>
<tr>
<td>Socket Mode</td>
<td>Standard</td>
</tr>
<tr>
<td>Transparent mode</td>
<td>Yes (mode 1)</td>
</tr>
<tr>
<td>Total rings</td>
<td>0</td>
</tr>
<tr>
<td>Total plugins</td>
<td>0</td>
</tr>
</tbody>
</table>

The result for pf_send is shown below.

Figure 5.3 shows the output of the pf_send application. The PF_RING does not use zero-copy transmission; there is another version of the PF_RING known as the PF_RING-DNA which uses the zero-copy TX methods. As in the image, the PF_RING uses an Ethernet port eth0 for transmission with a CPU frequency of around 187698500 Hz. The packet size is 64 bytes. When there is 10Gbps, the rate is 14880952.38. The ‘current’ shows current the TX rate, ‘average’ shows the average TX rate and ‘total’ shows the total amount of packets sent. During this TX,
at the end of the window, about 3121600 packets are being transmitted. The following are the snapshots of packets captured on the receiver terminal. These packets were captured with the assistance of the Wireshark tool.

![Figure 5.2: Packets captured from pf_send (PF_RING) via Wireshark](image)

The packets arrive from a source with an ip 10.0.0.0 and are received on the terminal with an ip 192.168.0.1 and these ip addresses are set on the switch. The length of the packets captured is 60-bytes. It uses a DIS protocol for the TX of the packets. DIS or Distributed Interactive Simulation is an IEEE standard for real-time communication between multiple computers over a network.
### 5.2 USDPA A test

The physical setup of P4080 device is shown in the Figure 5.3.

![Physical setup of P4080 device](image)

**Fig.5.3- Physical setup of P4080 device [18]**

A host with Tera-term is connected to the serial consol port on this P4080 device and the adapter motherboard 1G connector is connected to a switch network on which there is a tftp (Trivial File Transfer Protocol) server that contains the USDPA A images.

The P4080 System Configuration is listed below.

<table>
<thead>
<tr>
<th>PARAMETER</th>
<th>CONFIGURATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Board</td>
<td>P4080 with SDK 1.2</td>
</tr>
<tr>
<td>OS</td>
<td>SMP Linux 2.6.35 (32-bit)</td>
</tr>
<tr>
<td>Core</td>
<td>8x e500mc ver.2 cores @ up to 1.5GHz</td>
</tr>
<tr>
<td>Cache</td>
<td>L1: 32 Kbytes D-cache and I-cache with 64byte size</td>
</tr>
<tr>
<td></td>
<td>L2: 128 Kbytes backside L2</td>
</tr>
<tr>
<td></td>
<td>L3: 2Mbytes shared CPC</td>
</tr>
<tr>
<td>Memory</td>
<td>649.994 MHz @ 1300MT/s Data Rate</td>
</tr>
<tr>
<td>U-Boot</td>
<td>U-boot 2011.09 with CPU22 enabled</td>
</tr>
</tbody>
</table>
Test configurations were also setup for the packet TX. Frame sizes are 64, 128 etc. Acceptant packet loss rate is 0.001%.

Selecting the proper Ethernet interface for USDPAA is an important part of the experiment as the Ethernet driver is always involved in the configuration of FMan. It is the Ethernet based Linux device tree entries that determine the use case. The following device tree code shows a Linux private interface, which is also used by USDPAA.

```c
ethernet@0 {
    compatible = "fsl,p4080-dpa-ethernet-init", "fsl,dpa-ethernet-init";
    fsl,bman-buffer-pools = <&bp7 &bp8 &bp9>;
    fsl,qman-channel = <&qpool4>;
    fsl,qman-frame-queues-rx = <0x50 1 0x51 1>;
    fsl,qman-frame-queues-tx = <0x70 1 0x71 1>;
    fsl,fman-mac = <&enet0>;
};
ethernet@1 {
    compatible = "fsl,p4080-dpa-ethernet", "fsl,dpa-ethernet";
    fsl,qman-channel = <&qpool1>;
    fsl,fman-mac = <&enet1>;
};
```

The table below sums up the output for the P4080 device/ interface:

<table>
<thead>
<tr>
<th>Device Tree Name</th>
<th>U-boot Name</th>
<th>U-Boot variable</th>
<th>Linux Name</th>
<th>0xe position on the device</th>
</tr>
</thead>
<tbody>
<tr>
<td>ethernet@0</td>
<td>FM1@DTSEC1</td>
<td>ethaddr</td>
<td>fm1-gb0</td>
<td>not used</td>
</tr>
<tr>
<td>ethernet@1</td>
<td>FM1@DTSEC2</td>
<td>eth1addr</td>
<td>fm1-gb1</td>
<td>Motherboard RGMII</td>
</tr>
<tr>
<td>ethernet@2</td>
<td>FM1@DTSEC3</td>
<td>eth2addr</td>
<td>fm1-gb2</td>
<td>not used</td>
</tr>
</tbody>
</table>
The USDPAA application was also configured with cores 1 to 7, dedicated to USDPAA reflector threads. Linux OS runs on Core 0. Static ARP entries were also setup with a number of static route entries. Figure 5.4 shows the flow configuration for 2x1G connections on the device.

<table>
<thead>
<tr>
<th>Port ID</th>
<th>Source IP Address</th>
<th>Destination IP Address</th>
<th>Default Gateway IP</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10.172.72.14</td>
<td>10.172.72.220</td>
<td>10.172.72.254</td>
</tr>
<tr>
<td>1</td>
<td>10.172.72.3</td>
<td>10.172.72.221</td>
<td>10.172.72.254</td>
</tr>
</tbody>
</table>

Table.5.5 Summary of the IP addresses

Table.5.5 shows the IP addresses assigned to the interface. The network mask is assumed to be 255.255.255.0. In order for the application to
reflect the packets successfully, traffic destined for the P4080 device ports must have the appropriate source and destination addresses.

The following is the image of a Tera-term output testing hello_reflector application on a P4080. The tera-term is connected via a serial port (COM7) with a baud rate = 115200.

![Tera-term output](image)

**Fig.5.5- Tera-Term output**

The results are measured with two 1 Gbps and two 10 Gbps ports. These results are zero loss data rates and they are full duplex since the application is reflecting the frames. The results displayed are for 64 byte frames.

<table>
<thead>
<tr>
<th>cores</th>
<th>Port</th>
<th>Total packets</th>
<th>Pass rate</th>
<th>Throughput (Kpps)</th>
<th>Throughput (Mbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>22Gbps</td>
<td>16256535</td>
<td>24.8%</td>
<td>8126</td>
<td>5460</td>
</tr>
<tr>
<td>2</td>
<td>22Gbps</td>
<td>32253068</td>
<td>49.3%</td>
<td>15931</td>
<td>10840</td>
</tr>
<tr>
<td>4</td>
<td>22Gbps</td>
<td>61736491</td>
<td>94.3%</td>
<td>30868</td>
<td>20743</td>
</tr>
<tr>
<td>8</td>
<td>22Gbps</td>
<td>65479277</td>
<td>100%</td>
<td>32739</td>
<td>22000</td>
</tr>
</tbody>
</table>

**Table.5.6- 64 bit Frame throughput**

As shown in table 5.6, the cores are dedicated to reflect packets on the network. There are a total of eight cores in the P4080 processor. Each
core has a single thread. The user-space applications are designed in such a way that they can bind a thread to a particular core for TX or RX packets. The application reflector receives and sends frames on all Ethernet interfaces.

For 64 bytes frames the throughput is measured in Kpps (Kilo packets per second) and the incoming Ethernet port is 1G. Thus, with 100 percent of the incoming port traffic rate it can achieve up to 8126Kpps on one core and in a similarly manner as the cores are increased, the throughput becomes higher.

The throughput of the system can be scaled against the frame size. As the size of the frame is increased and more cores are involved, the throughput will rise. This is a linear behaviour and the P4080 exhibits almost linear scalability for reflector like applications.
6 Conclusions

The purpose of this study was to determine the packet processing mechanism in Multi-core systems, which results in the highest throughput for a given system. User-space fast packet frameworks (PF_RING and Netmap) and Freescale USDPAA framework are analyzed in detail and evaluated in terms of functionality, support, performance, ease of use, software/hardware dependencies and project stability etc. These Linux based user-space libraries are an alternative to the traditional network stack used for packet processing in Linux.

After evaluating the two libraries Netmap was discovered to be superior to PF_RING in a number of ways. Bearing in mind that there is a better version of PF_RING called PF_RING-DNA, which has the same performance as Netmap but, using PF_RING-DNA for the evaluation falls outside the scope of this project. One advantage that Netmap has over PF_RING is that it uses Memory Mapping for handling packets whereas PF_RING uses socket buffers in the device driver and it is a major source of overhead. Secondly, Netmap cannot crash the operating system because it runs in user-space and has no direct access to the critical resources whereas an application which misbehaves in the PF_RING is able to crash the system. Netmap had much better performance results on the used network as compared to the PF_RING.

In this project a sketch showing how to apply the framework approach to using Netmap to design Wrapper software has been provided. The idea is to use Netmap on top of USDPAA and test an application on a Freescale P4080 processor in order to leverage the DPAA hardware acceleration. It was ensured that the new application was portable, which means that it was able to be used with both Netmap and USDPAA.

The goals of this project have been accomplished as it was not meant to provide a complete solution but rather an analysis and a glimpse into how the new framework addressed the issue. Although the wrapper software is not a complete solution, the idea of using an open source solution with USDPAA to leverage the Freescale proprietary DPAA is fully justified by the extensive research conducted within this project.
The idea of using an open source solution with legacy software for fast packet processing in Multi-core systems can be very useful and beneficial for both the customers and the companies. It provides portability to the user-space applications, for example, a customer will not be required to entirely rewrite an application in order to move from open source to a legacy platform. In addition, since most of the user-space applications are written in low level programming languages e.g. C, moving an application from one platform to another will only require the writing of a Wrapper layer application that has the ability to create an interface between the two libraries.

Impact on Society

This is a relatively new area of research within computer technology researchers are attempting to discover new means of accelerating packet processing in user-space. Thus the solution to this problem will be more software driven. Although it is difficult to design, it also means that it will be easy to maintain, have a low cost and possess a reasonably good performance. Since no more dedicated hardware is required in order to achieve fast packet processing, it will save energy consumption and space on the silicon. Additionally, in relation to the user it will provide very fast services both to the applications and the operating systems which will save a significant amount of time. It is also the case that the use of an open source solution on top of a proprietary device will facilitate the porting of different consumer applications to different hardwares.
7 Future Work

The wrapper model presented in this report is a solution that has not been completely tested on a P4080 device. Actually, it can be stated that to some extent, no complete solution has been presented for the various API related problems. If a complete API solution is delivered, it can be incorporated into the USDPAA model and the actual performance can be tested on P4080 by varying different factors. The current wrapper model can prove to be more productive with regards to developing a complete API solution.

The new solution can be used to make a better application that is able to use the complete functionality of DPAA and is thus able to utilize the hardware acceleration and, simultaneously, perform the same functionality on Netmap. Testing such applications on P4080 can offer a better understanding of the physical performance of the system. Therefore, this work can be extended and advanced in relation to this idea, which yield more fruitful results.

- A full wrapper layer software solution can be proposed which can utilize the complete Freescale DPAA functionality on a P4080 processor. E.g the software layer must be capable of automatically tuning the CPU affinity settings, which is very important for high throughput.

- The current fast packet I/O frameworks are only designed to support raw packets. Classifying a packet as raw or TCP/IP or UDP is conducted in the hardware. A software approach, to classify a packet as raw or TCP/IP, would be interesting although, it will bring more overhead into the kernel software but, this can result in a complete user-space application package. However, this must be achieved in a careful manner since, the current hardware tools are performing well, under normal circumstances.

- It will be interesting to test such an application on multi threaded embedded systems, where each core has more than one thread.
References


[8] Luigi Rizzo, Marta Carbone, Gaetano Catalli, “Transparent Acceleration of Software Packet Forwarding using Netmap”, University of Pisa, Italy


Appendix A

Core parts of the code

Only some core parts of the code are shared here just to give the idea of the work and the coding style;

Please bear in mind, that to run this code proper header files needs to be included in the package and some function files are also needed to be in the package.

This code is for ioctl() function only:

```c
static void ioctl()
{
    /* Load the device-tree driver */
    int loop, tmpret, teardown, ret = of_init();
    /* Parse FMC policy and configuration files for the network*/
    netcfg = usdpaa_netcfg_acquire(PCD_PATH, CFG_PATH);
    if (!netcfg) {
        fprintf(stderr, "Fail: usdpaa_netcfg_acquire(%s,%s)\n", PCD_PATH, CFG_PATH);
        exit(EXIT_FAILURE);
    }
    if (!netcfg->num_ethports) {
        fprintf(stderr, "Fail: no network interfaces available\n");
        exit(EXIT_FAILURE);
    }

    // Install ctrl-c handler
    if (signal(SIGINT, handle_sigint) == SIG_ERR) {
        fprintf(stderr, "Fail: %s\n", "signal(SIGINT)" );
        exit(EXIT_FAILURE);
    }

    /* Allocate the worker structs */
    ret = posix_memalign((void **)&workers, MAX_CACHELINE, ncpus * sizeof(*workers));
    if (ret) {
        fprintf(stderr, "Fail: %s: %d\n", "posix_memalign()", ret);
        exit(EXIT_FAILURE);
    }
```

The Code below describes, how the ioctl() function is called from main() function of the program:

```c
int main(int argc, char *argv[]) {
    int loop, tmpret, teardown, ret;

    while (ARGINC() > 0) {
        if (!strcmp(*argv, "-n")) {
            unsigned long val=2;
            if (!ARGINC()) {
                fprintf(stderr, "Missing argument to -n\n");
                exit(EXIT_FAILURE);
            }
            ncpus = val;
        } else if (!strcmp(*argv, "-p")) {
            PCD_PATH = *argv;
        } else if (!strcmp(*argv, "-c"))
```
{  CFG_PATH = *argv;  }
else if(!strcmp(*argv, "-s"))
  {
    unsigned long val;
    val = strtoul(*argv, &endptr, 0);
    sz = (size_t)val;
  }
else {
    fprintf(stderr, "Unknown argument \"%s\"\n", *argv);
    exit(EXIT_FAILURE);
}

printf("Starting hello_reflector, ncpus=%ld\n", ncpus);
if (ncpus < 1) {
  fprintf(stderr, "Fail: # processors: %ld\n", ncpus);
  exit(EXIT_FAILURE);
}
if (ret) {
  fprintf(stderr, "Fail: %s: %d\n", "of_init()", ret);
  exit(EXIT_FAILURE);
}

ioctl(); //the call to the function is initiated here

/* Start up the threads */

for (loop = 0; loop < ncpus; loop++) {
  struct worker *worker = &workers[loop];
  worker->quit = worker->init_done = 0;
  worker->cpu = loop;
  ret = pthread_create(&worker->id, NULL, worker_fn, worker);
  if (ret) {
    fprintf(stderr, "Fail: %s(%d): %d\n", "pthread_create",
            loop, ret);
    while (--loop >= 0) {
      (--worker)->quit = 1;
      tmpret = pthread_join(worker->id, NULL);
      if (tmpret)
        fprintf(stderr, "Fail: %s(%d): %d\n", "pthread_join",
                 loop, tmpret);
    }
    exit(EXIT_FAILURE);
  }
}

/* Wait for thread init to complete (the first thread will
 * complete global init as part of that) */

while (!worker->init_done) {
  pthread_yield();
  if (!pthread_tryjoin_np(worker->id, NULL)) {
    fprintf(stderr, "Fail: primary thread init\n")
    exit(EXIT_FAILURE);
  }
}
/* Threads are created, now manage them (and catch ctrl-c) */
printf("\nS E N D I N G P A C K E T S !\n\n.......\n");
printf("\nfrom ioctl() function outside main().. \n");
printf("\nHit Ctrl-C (or send SIGINT) to terminate.\n");
teardown = 0;
while (ncpus) {
    if (!teardown) {
        if (received_sigint) {
            /* Ctrl-c signal triggers teardown */
            teardown = 1;
            printf("Ctrl-C, ending...\n");
            /* Non-primary threads should quit first */
            for (loop = 1; loop < ncpus; loop++)
                workers[loop].quit = 1;
        } else
        /* No teardown, no signal, this is where we can
         * pause */
        sleep(1);
    } else {
        /* Once the primary thread is the only thread, it can
         * quit too (global cleanup) */
        if (ncpus == 1)
            workers[0].quit = 1;
    }
}
/* Reap loop */
loop = 0;
while (loop < ncpus) {
    struct worker *worker = &workers[loop];
    if (!pthread_tryjoin_np(worker->id, NULL)) {
        fprintf(stderr, "Exit: thread %d\n",
                worker->cpu);
        if (--ncpus > loop)
            memmove(worker, worker + 1,
                    (ncpus - loop) *
                    sizeof(*worker));
    } else
        loop++;
}
qman_release_pool_range(pchannels[0], NUM_POOL_CHANNELS);
printf("Finished hello_reflector\n");
return 0;"
Code below from Netmap explains how to register an interface on the Netmap device.

```c
// Register the interface on the netmap device
nmr.nr_version = NETMAP_API;
if (ioctl(fd, NIOCREGIF, &nmr) == -1) {
    D("Unable to register interface %s", ifname);
    //continue, fail later
}
```

```c
// Print some debug information.
fprintf(stdout,
    "%s %s: %d queues, %d threads and %d cpus.\n",
    (td_body == sender_body) ? "Sending on" : "Receiving from",
    ifname,
    devqueues,
    g.nthreads,
    g.cpus);
if (td_body == sender_body) {
    fprintf(stdout, "%s -> %s (%s -> %s)\n",
        g.src_ip, g.dst_ip,
        g.src_mac, g.dst_mac);
}
```

```c
// Exit if something went wrong.
if (fd < 0) {
    D("aborting");
    usage();
}
```

```c
// Wait for PHY reset.
D("Wait %d secs for phy reset", wait_link);
sleep(wait_link);
D("Ready...");
```

```c
// Install ^C handler.

global_nthreads = g.nthreads;
signal(SIGINT, sigint_h);
targs = calloc(g.nthreads, sizeof(*targs));
```

```c
// create the desired number of threads
for (i = 0; i < g.nthreads; i++) {
    struct netmap_if *tnifp;
    struct nmreq tifreq;
    int tfd;
```

```c
    // register interface.
    //tfd = open("/dev/netmap", O_RDWR);
    //if (tfd == -1) {
```
D("Unable to open /dev/netmap");
continue;
}

bzero(&tifreq, sizeof(tifreq));
strncpy(tifreq.nr_name, ifname, sizeof(tifreq.nr_name));
tifreq.nr_version = NETMAP_API;
tifreq.nr_ringid = (g.nthreads > 1) ? (i | NET
MAP_HW_RING) : 0;

if ((ioctl(tfd, NIOCREGIF, &tifreq)) == -1) {
D("Unable to register %s", ifname);
continue;
}

tnifp = NETMAP_IF(mmap_addr, tifreq.nr_offset);

// start threads.

bzero(&targs[i], sizeof(targs[i]));
targs[i].g = &g;
targs[i].used = 1;
targs[i].completed = 0;
targs[i].fd = tfd;
targs[i].nmr = tifreq;
targs[i].nifp = tnifp;
targs[i].qfirst = (g.nthreads > 1) ? i : 0;
targs[i].qlast = (g.nthreads > 1) ? i+1 : (td_body == sender_body ? tifreq.nr_rx_rings :
tifreq.nr_tx_rings);
targs[i].me = i;
targs[i].affinity = g.cpus ? i % g.cpus : -1;
if (td_body == sender_body) {
    // initialize the packet to send.
    initialize_packet(&targs[i]);
}
}

for (i = 0; i < g.nthreads; i++) {

    //Join active threads, unregister interfaces and close file
descriptors.

    pthread_join(targs[i].thread, NULL);
    ioctl(targs[i].fd, NIOCUNREGIF, &targs[i].nmr);
close(targs[i].fd);
    if (targs[i].completed == 0)
        continue;
}
return (0);