Optimization of Software Framework to Improve Packet Processing Throughput through Customized Configuration for Small Data Packet

By

SAM POH MUN

A Dissertation submitted for partial fulfilment of the requirement for the degree of Master of Science (Electronic Systems Design Engineering)

August 2016

ACKNOWLEDGEMENTS

I would like to express my heartiest appreciation and gratitude to my supervisor, Dr. Teoh Soo Siang who has been with me throughout my postgraduate study period in Universiti Sains Malaysia (USM). He has shown tremendous effort in providing me guidance and recommendation throughout my research proposal, scope of contents and thesis write up. His research mindset and skills have nurtured me to keep on improving in my research thesis to be a more recognizable thesis. I've learnt a lot under his supervision.

I would also like to extend my gratitude to Universiti Sains Malaysia (USM) for scheduling students with numerous talks and writing workshops to improve in writing skill sets for a well-structured thesis. USAINS Holdings Sdn. Bhd, has also been a great help in facilitating this mixed mode course for industry students like myself. Last but not least, a big thank you for those who have helped me along my way from the beginning of this research especially my family, friends and fellow course mates who have stand by me at all times.

ABSTRAK

Penyelidikan ini mengkaji kaedah untuk meningkatkan kecekapan pemprosesan penghantaran data dalam sistem rangkaian. Penyelidikan ini didorong oleh pembangunan rangkaian komunikasi yang kian pesat yang telah menimbulkan keperluan untuk mengendalikan data komunikasi saiz paket yang lebih rumit akibat kerumitan aplikasi di rangkaian. Penyelidikan ini akan mengkaji pelbagai saiz paket yang biasanya digunakan dalam penghantaran data melalui rangkaian pada masa kini. Rangka rangkaian komunikasi tidak lagi sesuai untuk dipraktikkan dalam saiz paket yang rawak kini, terutamanya saiz paket yang kecil. Oleh itu, pengoptimuman rangka rangkaian komunikasi diperlukan untuk mengembalikan kecekapan pemprosesan paket. Kajian ini menfokuskan kepada penyelidikan seni bina unit kendali dan rangkaian jaringan terpadu supaya meningkatkan kadar memori mengakses tempatan. Di samping itu, masa penyalinan alamat memori dari memori ke “cache” juga akan dikurangkan dengan penggunaan “Huge Page Table”. Penyelidikan juga mengkaji kesan pemprosesan akibat timbunan tukaran konteks daripada system dan peranti sampukan. Maka, pengenalan mekanisma pengundian di peranti dalam penerimaan dan penghantaran paket dapat mengurangkan timbunan tukaran konteks. Selain itu, kumpulan paket diperkenalkan supaya paket diterima secara pukal bagi mengurangkan pengaksesan memori oleh satu per satu paket. Penyelidikan ini dilaksanakan dalam rangka kerja “Data Plane Development Kit (DPDK)”. Akhirnya semua pengubahsuaian dan pengoptimuman telah terbukti dapat meningkatkan dan mengoptimumkan pemprosesan paket sebanyak 61% daripada rangka kerja asalnya.

ABSTRACT

This research investigates the method to improve the data transmission throughput in a networking system. The reason of this research proposal is because of computer network has grown tremendously in recent years and it requires to handle more complex data packet with different sizes from applications. In this research, the throughput for various packet sizes used in a data transmission system were analyzed. The common network frameworks may not be efficient in handling the data traffic with random packet size especially when there are many small packet frames. Therefore, the optimization on the network framework is necessary to improve the packet processing throughput for small packet size. Investigation in methods to improve the local memory accessing rate in processor and network device were conducted. In addition, the duplication of memory address from system memory to cache is reduced by implementing Huge Page Table. Then, the impact on low throughput due to the context switching overhead originated from the system and device interrupts were analyzed. The interrupt polling mechanism were implemented on receive and transmit paths of the network driver for reducing the overheads. Another improvement introduced was by enabling the burst mode in the receive port. This will make the incoming packets being received and processed in bulk, and therefore removing the latency of processing each packet individually. The proposed improvements have been implemented on a Data Plane Development Kit (DPDK) framework and tested on the hardware using simulated data traffic. The results showed that the improved framework is able to achieve better network throughput by 61% compared to the conventional framework particularly for small packet size.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ASPM	Active State Power Management
BIOS	Basic Input or Output System / System BIOS
CPU	Central Processing Unit
DPDK	Data Plane Development Kit
EAL	Environment Abstraction Layer
ID	Identification
IO	Input or Output
IP	Internet Protocol
IRQ	Interrupt Request
ISR	Interrupt Service Routine
L3FWD	Layer 3 Forwarding
LLC	Last Level Cache
MAC	Media Access Control
NAPI	New Application Program Interface
NIC	Network Integrated Controller
NUMA	Non Unified Memory Access
OS	Operating System
OSI	Open System Interconnection
PCI	Peripheral Component Interconnect
PCIe	Peripheral Component Interconnect Express

PMD	Poll Mode Driver
QPI	Intel® QuickPath Interconnect
Rx	Receive
SUT	System Under Test
TCP	Transmission Control Protocol
TLB	Translation Lookaside Buffer
TPT	Tapered Fat Tree
Tx	Transmit
VM	Virtual Machine
VMM	Virtual Machine Monitoring
vNIC	virtual Network Integrated Controller

CHAPTER 1

INTRODUCTION

1.1 Project Overview

Cloud data centers have increasing demand for higher computing power, due to the continuous increase in various data workloads. This has driven the increase in the number of Central Processing Unit (CPU) cores in the servers to increase the computing power significantly. On the other hand, there is also a need for server virtualization, due to the physical resources available in a server system can accommodate many applications and functions for many users than just a single user. Therefore, to fully utilize the capability available in a server system which is designed to handle various workloads, virtualize environment is required. In a server virtualization environment, multiple Virtual Machines (VMs) run independently as virtual servers on a single physical server. The number of VMs depends on the capability of the physical server and also the workload requirements, which could range up to thousands of VMs in a single physical server discussed in [1].

Therefore, the conventional data centers will need to subdivide the physical resources available in a server to accommodate a number of VMs in a virtualized environment. These subdivision of resources would require a process called Virtual Machine Monitoring (VMM) to play a role in resource management and to ensure proper isolation. This is done by emulating the VMs through the abstraction of physical resources

into logical resources. Each VM could utilize certain resources and act as a platform itself with its own guest Operating System (OS), memory and network function independently as shown by Paul et al. [2]. VMs are required to initiate system calls to the VMMs prior to accessing the hardware resources. Examples of current available VMMs are such as *KVM* and *Xen* described in [1], [2]. To use these types of VMMs, it require massive modification in the guest OS kernel. To overcome this, processors makers like Intel and AMD have come out with a hardware assisted virtualization technology to trap the guest OS call to the VMMs. This improves over the virtualization where modification in the guest OS is not needed. It allows the host OS to run directly in the hardware layer and thus reducing the software efforts in performing binary translation as needed in the full virtualization environment. This technique is illustrated in Figure 1.1 where a new root and rings are introduced to the virtualization stack within the processors virtualization technology.

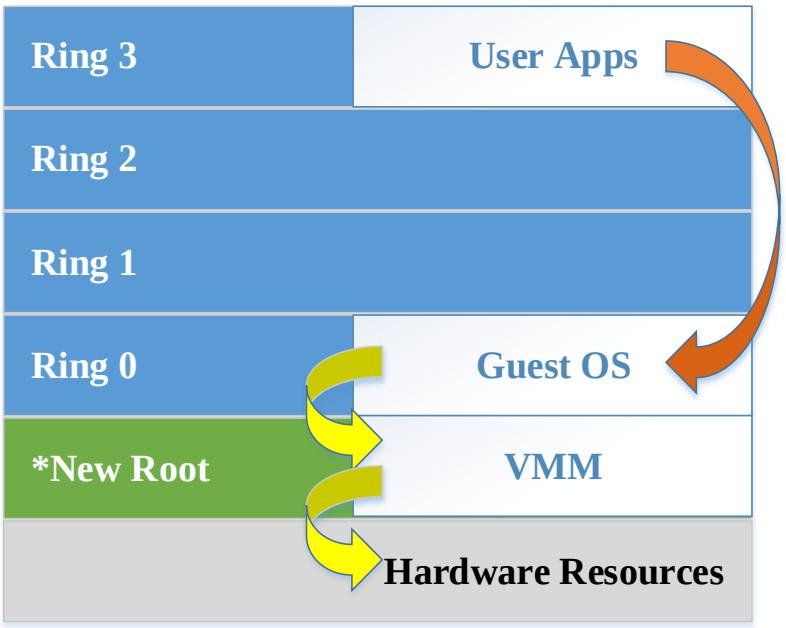

Figure 1.1: Rings of processor in a hardware assisted virtualization stack

Current server applications are mainly focused on the customization of resources available in the hosting platforms to their client's requirement based on the applications domain and workload intensities. The integration of network devices and the Inputs or Outputs (IOs) will provide an improved solutions for the VMs to map and segregate the resources in the hosting platforms. It will allow better utilization of the physical resources by each VM and to achieve better networking throughput.

The block diagram in Figure 1.2 shows the components in a virtualized networking framework of a server. At the bottom layer, the hardware resources are the common physical components normally found in a networking server structure. It comprises of a NIC, for ethernet connectivity, a memory unit for mapping of addresses as well as for data storage and lastly a processor for network packets pre-caching and processing. The middle layer is the VMM. It is needed for server virtualization as discussed earlier. The VMM provides a virtual switch to monitor and manage the switching of the networking path as well as other customized virtual functions in and out of each VMs. Example of virtual switch environment residing in the VMM are like the *Open vSwitch* discussed in [3], [4] and also *KVM* and *Xen*. These modules allow the VMs to share the similar physical resources, and creating networking throughput path. It also customize distribution of configuration and mobility of resources function as well as visibility of these resources among VMs with necessary isolation consideration as demonstrated in [3]. The virtualized Network Integrated Controller (vNIC) which reside on the top layer, is an example of virtualized resources made available to the VM by abstracting the hardware resources from physical NIC.

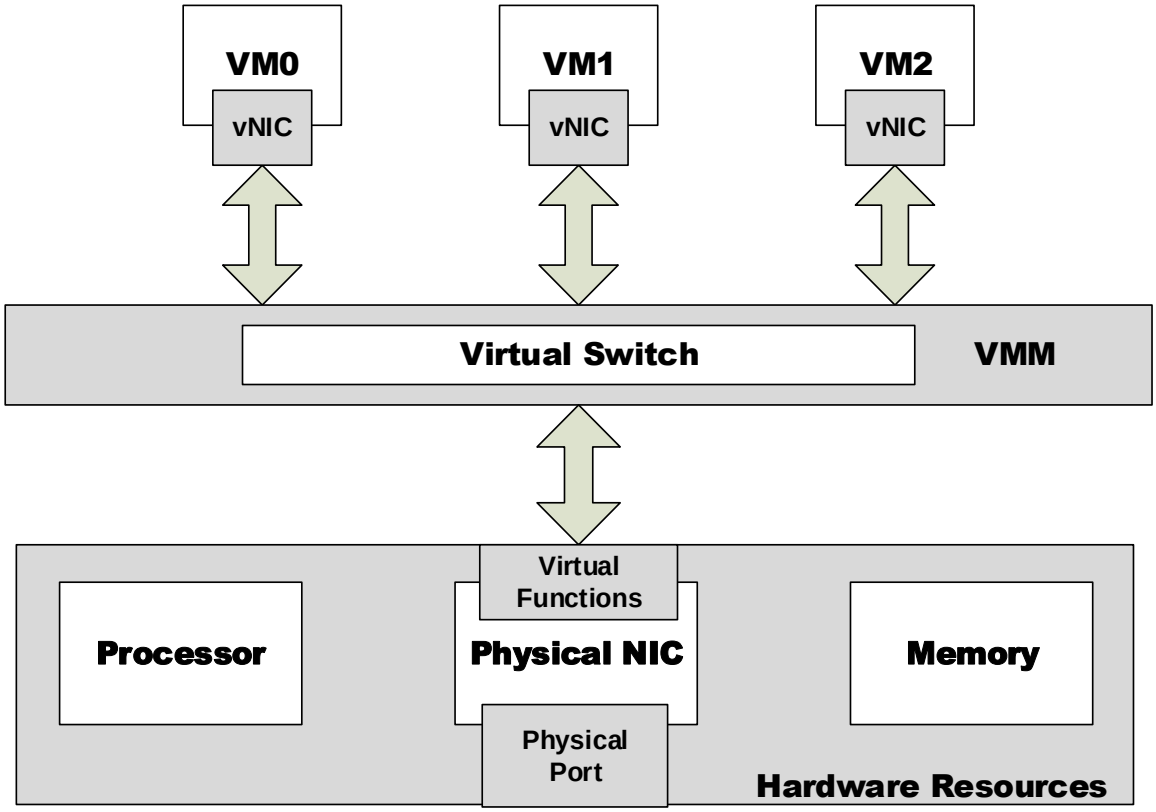

Figure 1.2: Virtualization Framework in Servers

Figure 1.3 illustrates how an application run in a guest OS in the User Space (top layer) transmits data to host OS in the Kernel Space (middle layer), before passing the data to the physical NIC (bottom layer). Transportation of data from application to the physical NIC will go through the networking layers. When an application running in the guest OS, requires to send information to the kernel OS, it will create the user data and follow by a system call which is the *write()* function to copy the user data to the kernel memory. Next, it will add a *send socket buffer* at the beginning of the user data for sending data in sequence, or vice versa *receiver socket buffer* is added for data receiving mode. Transmission Control Protocol (TCP) and Internet Protocol (IP) header is then added to

the socket, to indicate the destination IP address and the transportation method to carry the data. It is then sent to the Ethernet Layer where a Media Access Control (MAC) address of the next destination is added to the Ethernet header. After that, the NIC driver will perform packet transmission by copying the data from Kernel Memory and distribute it to the network through physical NIC network port.

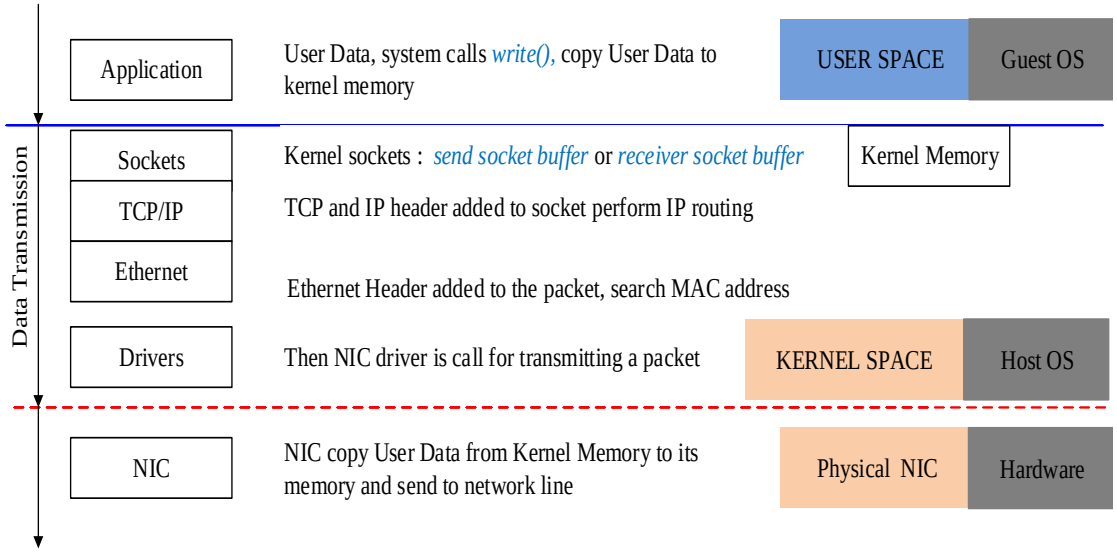

Figure 1.3: Data transmission in between User Space, Kernel Space and Physical NIC

The process of packet transmission in between guest OS and Host OS as shown in Figure 1.3 will create multiple duplicate memory copies of packet data in the OS memory. This will cause additional overhead to the memory management unit in the processors. In addition, a series of interrupts and interrupt handling have to be served during the transmission of data through the multiple network layers in the Kernel Space. In a virtualized environment, due to the numerous VMs and emulated virtual IOs resources, on top of the physical devices, there are even more memory duplicated and interrupts

routines that need to be handled as mentioned in [6]. All these overheads will reduce the packet transmission throughput. Therefore this research proposed a customized configuration for the networking framework to optimize the packet throughput up to the limit that the hardware can support. This work is relevant since many server systems are not optimally configured by default to deliver the optimum packet processing rate.

1.2 Problem Statement

Network frameworks is an application that provides a set of functions to handle networking tasks in the OS. It has grown into a complex structure to meet the networking requirements by data centers in cloud computing which are the performance and isolation. However there were also needs for simplicity and optimization of specific applications, workloads and usage model for VMs in networking. Therefore, analyzing the best framework to be used and proper configuration of the framework is crucial in a networking system.

In addition, a network system designer needs to consider the hardware limitation in conjunction with the processors architectures, ethernet controller and Peripheral Component Interconnect Express (PCIe) bus bandwidth for packet processing. Knowing these limitation, the maximum packet throughput was estimated. Multiple memory access and memory address copies during the data transaction, will cause excessive CPU cycles and increase power consumption. By improving the memory access method, it can help to reduce the number of random memory accesses and thus reduce the CPU cycles required for data transaction. Lastly, the context switching overhead during the packet

transmission through the kernel and user space due to various interrupts can be further optimized by understanding the device specification and the options in the OS kernel of a networking system.

1.3 Objectives

The objectives of this research are;

1) To analyzed the best framework for packet processing to be used in this research.

2) To optimize the framework through customized configuration to improve the packet processing rate for small packet size.

3) To evaluate the improvement in the packet processing rate of the customized framework compared to the original framework.

1.4 Scope

In data centric cloud computing, different VMs could run with a diversified workload. Therefore this research will be conducted for networking in cloud computing, where VMs are running on top of a physical system with various common packet sizes. Customized configuration in the devices operation mode will be studied and analyzed to reduce the overheads of processes during data transmission. Optimization is done in the system calls and functions in the network drivers and packet receive (Rx) and transmit (Tx) mode to accelerate the packet throughput.

This research study will focus in Linux OS kernel modules, due to the Linux operating system availability as an open source system. The networking software framework is integrated with the Linux OS kernel modules. The libraries made available in software framework and Linux OS kernel modules are customized to improve the packet processing rate. This is done by research on NIC, processor and memory in hardware and the software abstraction layers. The hardware limitation of a NIC card with dual network port availability in a system, would limit the performance of throughput sharing this network controller via the network ports. The similar custom configured software framework proposed by this study could be applied by future developers to customize and map to different architectures with similar hardware capabilities. The depth of the network security, encryption and protocol of the packet transferred will not be considered since this research will focus more on native packet transmission in a networking. In addition, scalability will be another factor not considered in this study as well, due to large variance of silicon architectures availability.

1.5 Thesis Organization

This thesis is organized into 5 Chapters. Chapter 1 discusses about server virtualization and network infrastructure required in networking. Server virtualization environment are defined from available methodology in industry and trends for networking applications. Networking infrastructure is described in a manner how a packet is transferred from an application through user and kernel space all the way to the network. Besides current industry problem is also described, which lead to objectives of this thesis.

Chapter 2 brings out the common available network framework defined in industry and common usage model for packet transmission path access from network port through kernel space. In addition, this chapter gives an overview on all available architectures and examples of customization methodology used for optimizing the packet transmission. This chapter will also look into application and tool used to run the packet throughput tests.

Chapter 3 discuss about the hardware tools, software framework and methodology used to obtain the optimized packet throughput. The chapter also explains in details the proposed method to optimize the memory mapping, interrupts call and packet transmission mode in Rx and Tx of the network driver in order to improve the throughput.

Chapter 4 will show packet throughput results from native framework and also after integrating the modified network driver into the modified network framework. The results would then be compared to validate the packet throughput compare to the physical network system capability. Hence all these results will be discussed on the methods used on framework and its impact to the throughput with hardware limitation in consideration.

Finally Chapter 5 will conclude the findings of the research and provide some recommendations for future work.

CHAPTER 2

LITERATURE REVIEW

2.1 Introduction

Network stack is described in the form of layers, while network framework is a software developed to access these layers. The native frameworks may not be an issue for large packet size and low traffic congestion, due to time required to process a packet would be sufficient. However, network communication doesn't come this ideal, which will lead to the need of this research. The best method to study and modify the framework was through the analysis of common frameworks and identify the problems in the network framework. Therefore prior to discussing the methodology of this research, understanding on how packets are processed and forward in network infrastructure from network ports to VMs will be studied to identify the common problems. In this chapter, the OS kernel modules, the physical system hardware architectures and the current techniques will be explored. Then, techniques learnt in memory mapping, interrupts mode and application on how packets are transmitted will further assist in this framework optimization. Lastly, similar research will be discussed on the latency and throughput studies on memory mapping methodologies

2.2 Industry Network Framework

There are many network frameworks in the industry. In this research focus is on the frameworks which are available as open source. These frameworks are reviewed to understand how memory mapping works as packet transfer from kernel space to user space. It is important for developers to understand the framework and provide a solution from current native frameworks to obtain the optimized framework for optimum packet processing.

2.2.1 Linux OS Kernel

As explained by Luigi et al. in [7], [8], packet processing can reach 10Gbit/s Ethernet by utilizing features on NIC, CPU and memory commodity between the application and network. This is an important fact for developers to focus in such commodities to reach the optimum throughput of a network system. With the introduction of multi-core systems, application can improve packet processing effectively especially with the Non-Uniform Memory Access (NUMA) design, where each core can be directly mapped to memory individually. This will provide isolation to applications when several cores try to access the same memory. Linux operating system (OS) has come out with solutions like Linux PF_PACKET, in aid of reducing CPU cycles. This is achieved by passing packets effectively from kernel space to user space. However, memory still cannot release its address for incoming packets as its still in use for packet memory allocator, therefore not much in reducing memory access cycles using PF_PACKET. In addition, due to various request from OS like data fragmentation, buffer sharing and offloading of tasks to network controller, legacy network device driver in OS has functions like *(mbuf,*

skbuf and NdisPacket) to support it, which will cause overhead for large packet size. Although modified drivers could remove these overhead by removing the software dependencies and direct mapping the hardware for control as mentioned in [7]. However, this will expose the system to network intrusion if drivers are not modified properly, due to the dependency of network driver to kernel and system memory, when an applications crash.

2.2.2 Netmap framework

Similar to what Linux OS Kernel offers, Netmap software framework also try to provide shared memory buffer between user application and hardware NIC. The advantage of Netmap is that it can reduce the overhead by pre-allocating memory resources, system call (syscall( )) overheads and memory copies impacting OS kernel and user application as shown in Figure 2.1 also shown in [7]. Yet kernel configuration controlling NIC hardware is still maintained, making it flexible for portability to other NIC hardware in the future.

The data path of Netmap memory layout consists of 3 types data structure. They are packet *buffers, netmap rings,* and *netmap if* as shown in Figure 2.2 as discussed in [8]. All these data structures are located in shared memory region between user space and OS kernel only. The focus on memory locality simplifies the resource management and making the system more robust and controllable.

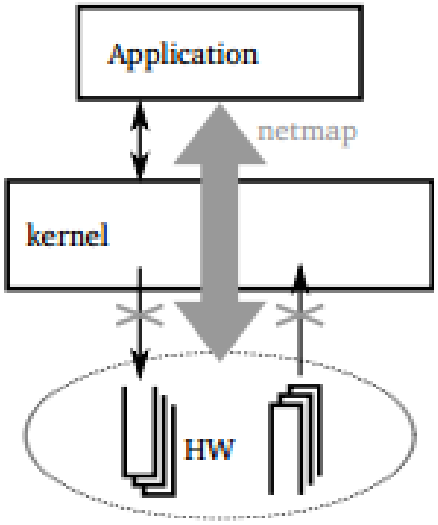

Figure 2.1: Implementation of netmap versus conventional Linux OS kernel [7]

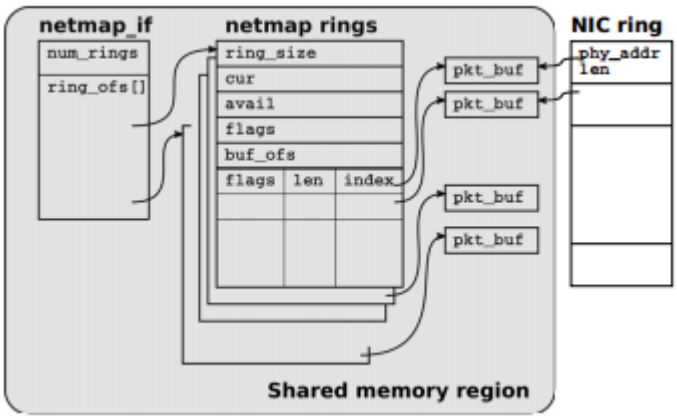

Figure 2.2: Memory Layout of netmap [9]

However the Netmap framework does not accelerate the TCP. Modification is required to accelerate the TCP or IP stack to create a faster path from applications to network port through IO batching and memory mapping techniques. This will be discussed in more details in the next Section 2.2.2 using Data Plane Development Kit (DPDK) framework. In addition, there are limited research or method discussing the Netmap usage model in virtualized network which is required especially in communication and network infrastructure configuration. Netmap has investigated on virtualization as well, known as VALE, which is currently available for e1000 network driver only. However, the adaptation to other drivers has not been commercialized. Therefore, Netmap framework is not an optimized framework for certain packet sizes on its own especially on a virtualized environment as discussed in [10].

2.2.2 Data Plane Development Kit (DPDK) framework

Similar to Netmap, DPDK framework is developed in user space. This gives user the opportunity to modify and scale the model to various system architecture, in order to maximize the packet throughput and the communication workload performance. In addition, DPDK is also a free framework in open source community. There are some major industry players like 6WIND, Calsoft Labs, Intel, Tieto and Wind River together with open platform community that continue to maintain, contribute and adopt it in their system. Therefore the libraries and modules in DPDK framework is better maintained compare to Netmap or native Linux framework for networking.

DPDK is just a developer kit, therefore user will require to build the outlines of the framework with the Environment Abstraction Layer (EAL) and the DPDK service

layer libraries prior to creation of the network framework. These libraries helps communicate the network port to user application based on user customization. However DPDK focus on 4 main pillars of network infrastructure which are the application and services, data plane processing, packet processing and signal processing. The libraries include Buffer/Queue Management, Packet Flow Classification and Poll Mode Drivers are in the user space above the Linux kernel space were shared in [11]. Once these are loaded, EAL will handle the synchronization of processor cores used for data plane processing. Then as in Figure 2.3 shows, function mmap () which will map the abstracted hardware to user application via EAL reserving memory specific for DPDK usage as demonstrated in [12], [13]. Then the service layer will initiate the processors to process the packet in data plane application based on the binding of the memory allocation mapped earlier and hardware devices.

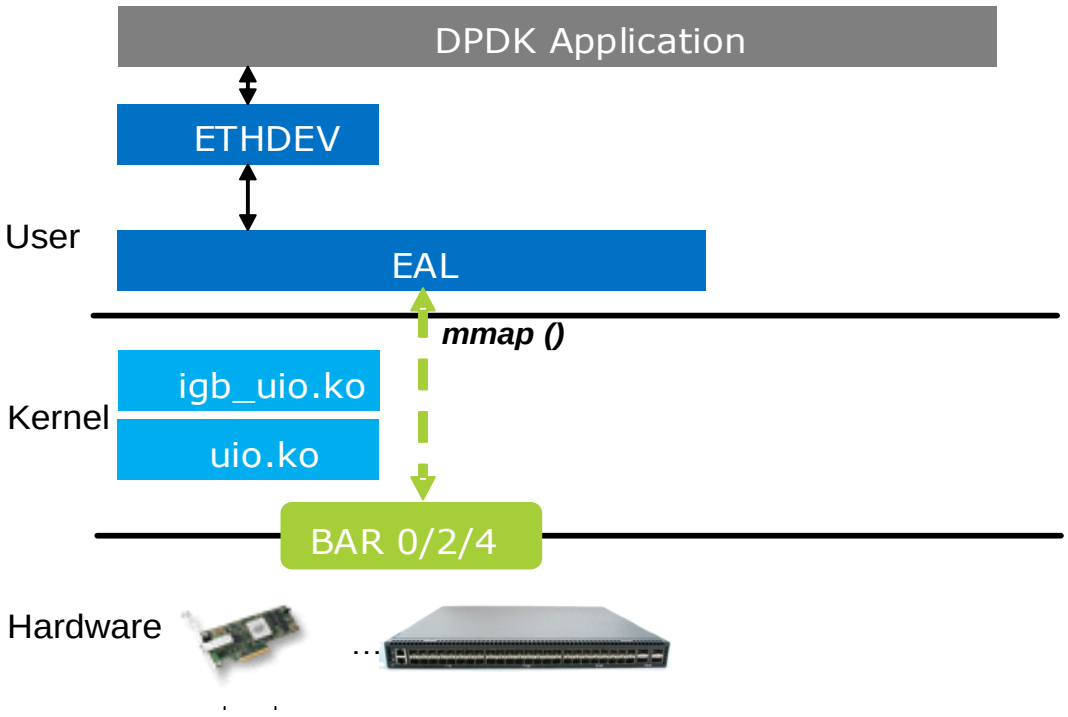

Figure 2.3: Memory Layout in DPDK

However, there is a pre-requisite, which often being considered as a disadvantage in using DPDK framework. The pre-requisites is that the binding of the network ports has to be some specific IGB_UIO modules in kernel. In other words, these network ports would no longer be available to be accessed by user through Peripheral Component Interconnect (PCI) device identification (ID). However, there is a method to retrieve information on the NIC ID or network device IDs that are available in the system. This is done through *dpdk_nic_bind.py* script to check the status of binding network ports for DPDK.

The reason of binding these through software threads is to improve performance, bringing hardware closer to OS kernel. In details, binding core to specific application to NIC hardware, is for lockless queues which reduces unnecessary wait times. While in memory pre-allocation, it improves the packet processing and throughput by using routing tables and hash map. Another advantage of DPDK framework are TCP and IP headers are integrated into DPDK application in user space binding them together, reducing overhead and context switching during transportation of packets as described in [14].

2.3 Non Unified Memory Access (NUMA) architecture

For multiple processor architecture system, Non Unified Memory Access (NUMA) is introduce, to provide faster local memory access to each processor. In multiprocessor system, each processor has its own local memory unit, which could also be a remote memory unit to other processors. Through NUMA methodology in processor nodes architecture, a system can be configured in orders of memory accessing done locally

as described in [15]. This can also bring the system processors, memory and IO devices closer together and therefore improving the latency and bandwidth of packet processing. This is important for system designs with more than 1 socket, as the placement of the nodes to the memory plays an important role to latency and bandwidth. The further apart the processor nodes from the IO devices and its local memory, the larger the latency require to access memory and it will reduce the bandwidth performance as discussed in [16].

In [17], A. Banerjee et al. talked about NUMA in a virtualized environment. As mentioned in Chapter 1, the physical NIC and IOs are shared among the VMs in a virtualized environment. Therefore in NUMA implementation, a scheduler is required to make sure that the NUMA nodes on the memory and the network devices are scheduled in the same queue to achieve NUMA affinity and keeping all these components local and intact. Same goes for virtualized environment if the virtual functions or modules are placed in a different NUMA nodes or other processors nodes, since this environment will create non uniform remote memory access. This is shown in Figure 2.4, where packet will go through Intel® QuickPath Interconnect (QPI) interface which connect between 2 CPUs before accessing the network devices modules at a different node.

Therefore it is critical on the placement of the scheduler in VMs virtual functions that require NUMA nodes to schedule the queue accordingly. Otherwise, the hardware devices might have large number of VMs that share the NUMA nodes and eventually hit higher cache misses, due to scheduler need to access the vNIC of each VMs in a shared NUMA nodes path. Therefore to achieve a high packet processing output, NUMA nodes

have to be mapped as a path without resolving to remote memory access which will cause unnecessary latency in a network system.

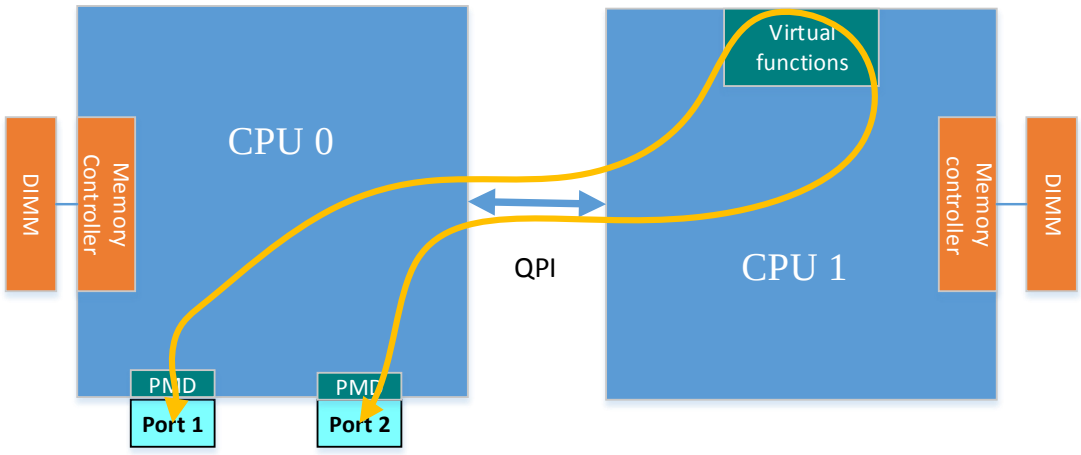

Figure 2.4: Remote Memory Access in non-NUMA architecture

2.4 Memory Mapping

C. D. Killebrew [18] proposed an off processor memory interconnect solutions. When cache size in the processors is no longer scalable to bigger size a memory traffic topology called Tapered Fat Tree (TFT) is used to balance the memory mapping access between cache and external memory.

Page tables are usually used in mapping the physical memory addresses to the virtual memory addresses. Whenever a process or application requires to access a memory in a virtual environment, it will go through the Huge Pages Table first. If it's not mapped in the Huge Pages Table, the extra cycle of procedures will be needed to access the System Memory Address Page Table as shown in Figure 2.5. This process increases CPU cycles

thus reducing the packet processing performance. Therefore a cache like table called Translation Lookaside Buffer (TLB) is introduced in Linux environment. If the pointer to the address could not be found in the Huge Pages Table, it would remap the Huge Pages Table after retrieval of the translated address from the system memory. In other words, a TLB miss will occur if the address is not found in Huge Pages Table. Thus the larger the Huge Pages table is, the more addresses translation it can keep in the cache, providing faster access of translation address and reduce system latency.

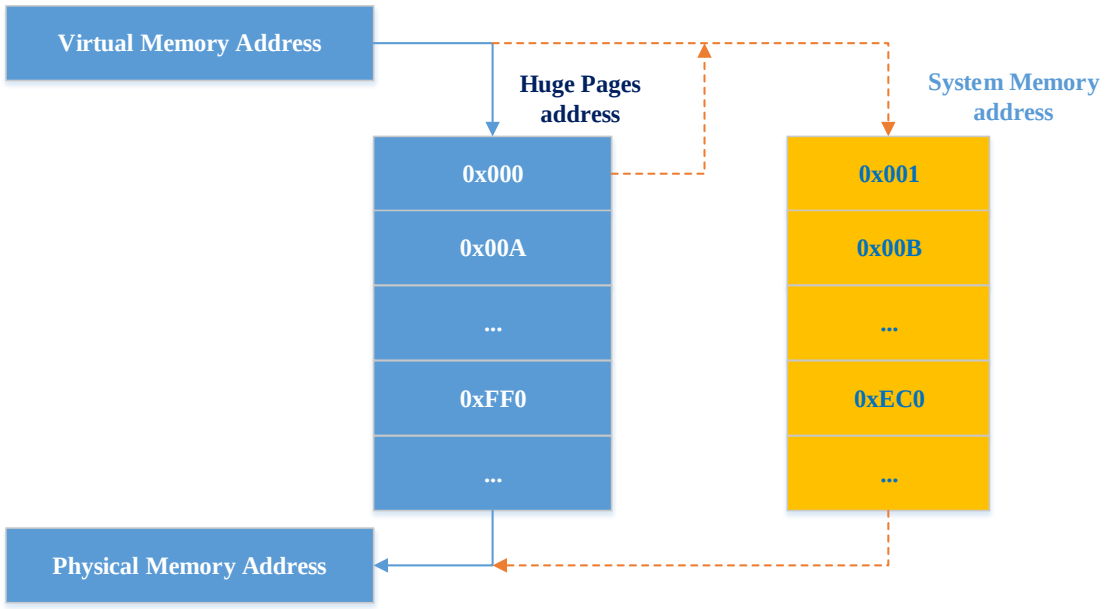

Figure 2.5: Huge Pages Table and System Memory address retrieval

2.5 Interrupt Routine

This section will discuss on the OS kernel interrupt. Each interrupt in a system will intercept the processor to stop its current function and perform the higher priority function.

Therefore if there are too many Interrupt Request (IRQ), as shown in Figure 2.6, the longer time is needed for the packet processing [19]. However the OS kernel provides numerous effort from transferring or disabling the hardware interrupts and transfer it to software interrupts for performance improvement.

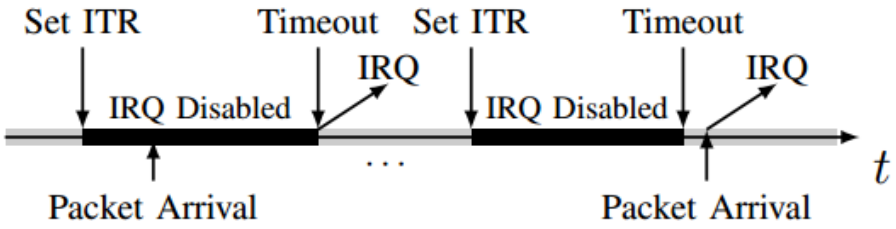

Figure 2.6: Interrupt Throttling Rate (ITR) of a packet [19]

In a network traffic, there are tremendous amounts of interrupt calls to the system. For example, consider the packet transfer through the network. As soon as the packet enters the OS, several Interrupt Service Routines (ISR) are called to the system. The processor interrupt handler performs storing the current operation, switching the process, performs the packet transfer and also releases the current state back to the system upon routine completion. Then device interrupt is called again upon moving into the ethernet device to invoke the device to transfer packet to the network. All these interrupts being mentioned are called context switching of functions and system calls.

The number of interrupts occur with just one packet transfer can be up to ten depending on the network framework, especially when security comes into the picture with more levels of accessibility control. Therefore, when network traffic becomes complex, these interrupts will create context switching overhead to the processing cycles

and affect the efficiency of the whole system. It will also increase the number of packet drops due to not meeting the minimum time for a processor to process a packet prior to the next packet's arrival. In [19], Paul et al. discussed about the New Application Program Interface (NAPI) network driver which can mitigates and reduces the number interrupts that occur in a packet transaction compare to the conventional method.

In addition, polling mechanism also comes into the picture of this research to reduce the devices context switching as interrupt occurred. This is done within the system, where the system will look at the devices from time to time for any event occurred. With this, the processors will handle the devices events in a more controllable manner. It can be configured to handle events when certain system calls or function is triggered, instead of handling each interrupts from the devices. This specific mechanism is utilized in this research to configure each Rx and Tx path of the network port, whereby packet access to the user space is through polling and not using interrupt. This will reduce the context switching overhead caused by excessive number of interrupts. This polling mechanism is built as a driver called Poll Mode Driver (PMD), which is built in the network drivers like IGB or IXGBE as shown in Figure 2.7.

The virtualization environment as shown in Figure 2.7, consists of a VIRTIO block which is a virtualization IO block for guest virtual machine physical resources abstraction, a multi-thread Kernel NIC Interface (KNI) for interaction with kernel with NIC, and a specific driver for connection between NIC in kernel IGB-UIO to physical NIC IXGBE drivers. The numbers of interrupts would drastically increase due to the access blocks in kernel is much more compare to the native networking path. Therefore a similar PMD is

needed for the virtualization environment in the Rx and Tx path which will be further discussed in Chapter 3.

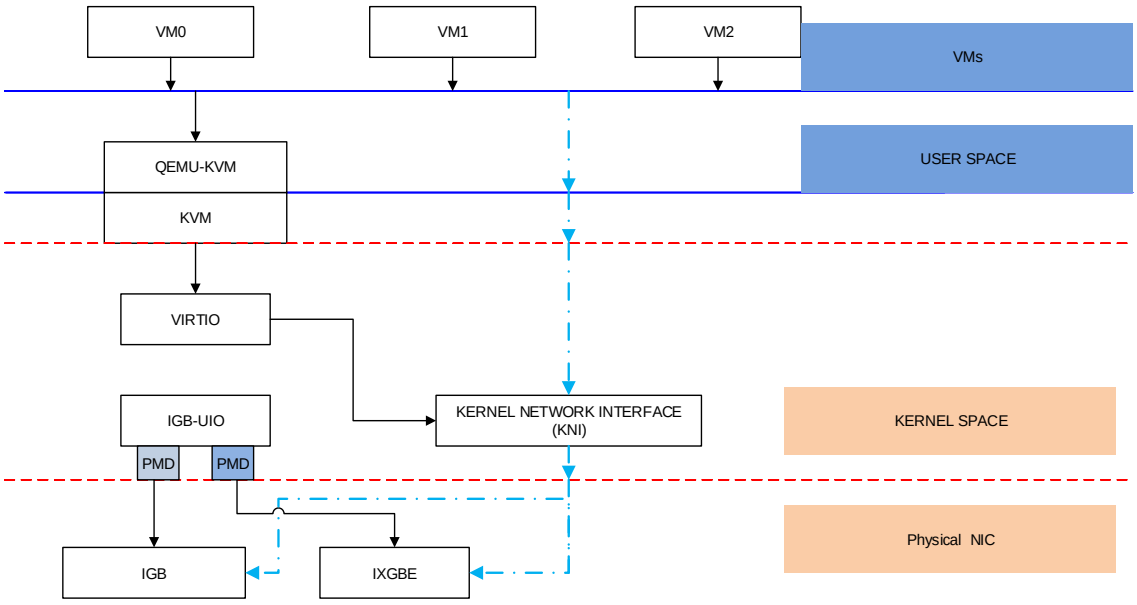

Figure 2.7: Linux Modules and Libraries

2.6 Application and Tools to verify the optimized framework

This research requires to compare the packet processing rate before and after the framework modification. Therefore several applications and tools are used to measure the packet processing rate for various packet size in the experiment. An application called L3FWD is used to pass the traffic from one network port to another network port completing a full loopback mechanism of packet transmission. Last but not least is the Spirent System, which is an industrial certified standard throughput test equipment to verify the optimized framework.

2.6.1 DPDK L3FWD application

Open System Interconnection (OSI) standard model refers to the network communication layer which consists of 7 layers from physical network port (layer 1) all the way up to the application layer (layer 7) as shown in Figure 2.8. In this research focus is on layer 1 to layer 3 due to our subject is on Rx network port send packet processing and routing back to Tx network port native packet transmission not involving other layers of security, encryption, transportation or protocols.

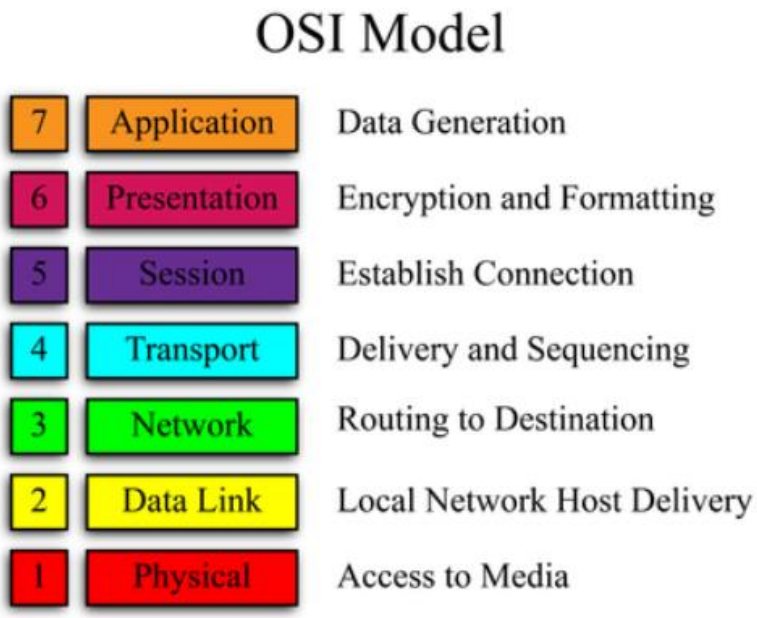

Figure 2.8: Open System Interconnection Model [20]

Layer 3 forwarding (L3FWD) is an application to forward a packet from transmit host to a receiver host or device though a network media part, involving physical layer, data link layer and pass to the network layer to be delivered to destination. Example a packet arrive in a Rx port of a System Under Test (SUT), a forwarding application will be needed to transfer and structure the packet to frame and IP destination to the Tx port of a SUT before sending out to the network completing the networking loop.

In [21], Paul et al. perform studies on the multiple applications of forwarding a packet in a system. Studies shown that the native OS kernel network framework exhibits unnecessary overhead if the network is predefined to run the applications compare to DPDK predefined application running in the processor and network port. This paper [21], Paul et al. also discuss on how this application can improve packets transmission with cores and queues options in the system architecture critical for throughput improvement.

2.6.2 Spirent System Tool (packet generator), Open Source packet generator application, and RFC 2544 standard

The Spirent equipment is used in this research due the limitation of the open source packet generator. The benefit of this equipment is that it is able to analyze the traffic generated especially on the Rx side generating numbers of packet transmitted log. Meanwhile this equipment can also generate some crucial parameters like latency and transmission rate, in which such results are not available in the open source packet generator application. In addition, the tools also has embedded the Request For Comments (RFC) 2544 standard analysis described in [22] for benchmarking results of network