DEGREE PROJECT IN INFORMATION TECHNOLOGY, SECOND CYCLE
*STOCKHOLM, SWEDEN 2017*

# Performance Optimization of Virtualized Packet Processing Function for 5G RAN

FILIP ÖSTERMARK

KTH ROYAL INSTITUTE OF TECHNOLOGY
*INFORMATION AND COMMUNICATION TECHNOLOGY*

# Performance Optimization of Virtualized Packet Processing Function for 5G RAN

Filip Östermark

2017-11-19

Master's Thesis

Examiner
Gerald Q. Maguire Jr.

Academic adviser
Anders Västberg

Industrial adviser
Gábor Fényes (Ericsson)

KTH Royal Institute of Technology
School of Information and Communication Technology (ICT)
Department of Communication Systems
SE-100 44 Stockholm, Sweden

# Abstract

The advent of the fifth generation mobile networks (5G) presents many new challenges to satisfy the requirements of the upcoming standards. The 5G Radio Access Network (RAN) has several functions which must be highly optimized to keep up with increasing performance requirements. One such function is the Packet Processing Function (PPF) which must process network packets with high throughput and low latency. A major factor in the pursuit of higher throughput and lower latency is adaptability of 5G technology. For this reason, Ericsson has developed a prototype 5G RAN PPF as a Virtualized Network Function (VNF) using an extended version of the Data Plane Development Kit's Eventdev framework, which can be run on a general purpose computer. This thesis project optimizes the throughput and latency of a 5G RAN PPF prototype using a set of benchmarking and code profiling tools to find bottlenecks within the packet processing path, and then mitigates the effects of these bottlenecks by changing the configuration of the PPF.

Experiments were performed using IxNetwork to generate 2 flows with GTP-u/UDP/IPv4 packets for the PPF to process. IxNetwork was also used to measure throughput and latency of the PPF.

The results show that the maximum throughput of the PPF prototype could be increased by 40.52% with an average cut-through latency of 97.59% compared to the default configuration in the evaluated test case, by reassigning the CPU cores, performing the packet processing work in fewer pipeline stages, and patching the RSS function of the packet reception (Rx) driver.

**Keywords:** 5G, RAN, virtualization, packet processing, NFV, optimization.

## Sammanfattning

Med den annalkande femte generationen av mobila nätverk (5G) följer en rad utmaningar för att uppnå de krav som ställs av kommande standarder. Den femte generationens Radioaccessnätverk (RAN) har flera funktioner som måste vara väloptimerade för att prestera enligt ökade krav. En sådan funktion är Packet Processing-funktionen (PPF), vilken måste kunna bearbeta paket med hög genomströmning och låg latens. En avgörande faktor i jakten på högre genomströmning och lägre latens är anpassningsbarhet hos 5G-teknologin. Ericsson har därför utvecklat en prototyp av en PPF för 5G RAN som en virtuell nätverksfunktion (VNF) med hjälp av DPDK:s Eventdev-ramverk, som kan köras på en dator avsedd för allmän användning. I detta projekt optimeras genomströmningen och latensen hos Ericssons 5G RAN PPF-prototyp med hjälp av ett antal verktyg för prestandamätning och kodprofilering för att hitta flaskhalsar i pakethanteringsvägen, och därefter minska flaskhalsarnas negativa effekt på PPFens prestanda genom att ändra dess konfiguration.

I experimenten användes IxNetwork för att generera 2 flöden med GTP-u/UDP/IPv4-paket som bearbetades av PPFen. IxNetwork användes även för att mäta genomströmning och latens.

Resultaten visade att den maximala genomströmningen kunde ökas med 40.52% med en genomsnittlig latens på 97.59% jämfört med den ursprungliga PPF-prototypkonfigurationen i testfallet, genom att omfördela processorkärnor, sammanslå paketbearbetningssteg, och att patcha RSS-funktionen hos mottagardrivaren.

**Nyckelord:** 5G, RAN, virtualisering, packet processing, NFV, optimering.

# Contents

# List of Acronyms and Abbreviations

| | |
|---|---|
| 3GPP | 3rd Generation Partnership Project |
| 4G | Fourth generation mobile networks |
| 5G | Fifth generation mobile networks |
| C-RAN | Cloud Radio Access Network |
| CN | Core Network |
| CPU | Central Processing Unit |
| CS | Context Switch(es) |
| D-RAN | Distributed Radio Access Network |
| DL | Downlink |
| DPDK | Data Plane Development Kit |
| E-UTRAN | Evolved Universal Terrestrial Radio Access Network |
| EAL | Environment Abstraction Layer |
| eNB | Evolved NodeB |
| HAL | Hardware Abstraction Layer |
| IP | Internet Protocol |
| IPC | Instructions Per Cycle |
| lcore | Logical Core |
| LTE | Long Term Evolution |
| NFV | Network Function Virtualization |
| NIC | Network Interface Card |
| NR | New Radio |
| OS | Operating System |
| OvS | Open vSwitch |
| PMD | Poll Mode Driver |
| PPF | Packet Processing Function |
| pps | Packets/Second |
| RAN | Radio Access Network |
| RSS | Receive Side Scaling |
| Rx | Receive/Reception |
| Tx | Transmit/Transmission |
| UDP | User Datagram Protocol |
| UE | User Equipment |
| UL | Uplink |
| V-RAN | Virtualized Radio Access Network |
| VM | Virtual Machine |
| VNF | Virtualized Network Function |

# Chapter 1

# Introduction

This chapter gives an introduction to the subject of this thesis and includes a brief overview of the latest developments in mobile networks, specifically 5G, along with a description of the problem of implementing efficient and future proof packet processing. A concrete research question is presented along with an explanation of the purpose and goals of the thesis.

## 1.1 Motivation

Mobile networks have rapidly become an omnipresent part of people's daily lives in the past few decades, giving access to quick and easy communication in nearly all parts of the world. As the evolution and integration of technology into our everyday lives continues, the demand for better coverage, quality of service, lower latency, and higher data rates in mobile networks continues to increase. According to a 2016 Cisco* white paper, the global total amount of mobile data traffic reached an estimated 3.7 exabytes (EB) per month during the year 2015 [1]. The same paper predicts an increase from 3.7 EB/month globally in 2015 to 30.6 EB/month by the year 2020, which would mean an increase of 827% in the five years following 2015.

To keep up with the increasing volume of mobile data, mobile networks have to constantly adapt and evolve. Currently, the fifth generation of mobile networks also known as 5G is under development and has been predicted to arrive around the year 2020 [2, 3]. With the transition from the fourth generation mobile networks (4G) to 5G networks comes requirements for higher throughput and lower latency in the Radio Access Network (RAN). A vital part of any 5G RAN architecture is the Packet Processing Function (PPF). The PPF is responsible for the handling data packets flowing through the network. A highly optimized PPF is an important component in order to provide the high throughput and low latency proposed for the 5G standard. To reduce the cost of maintaining and updating network functions such as the PPF, network operators have moved to Network Functions Virtualization (NFV) [4, 5, 6]. Ericsson has followed this trend by developing a 5G RAN PPF as a Virtualized Network Function (VNF). For example, this introduces a need for additional performance considerations regarding packet I/O, as the cost of I/O will generally increase

* Cisco Systems, Inc.

under virtualization with traditional Linux network drivers [7]. To combat this problem, Ericsson's PPF relies on Data Plane Development Kit (DPDK) [8], which reduces I/O cost by giving user space applications direct access to the buffers of the Network Interface Cards (NICs). Additionally, DPDK has functionality to improve the caching performance of network applications, as well as data structures which allows lockless operations in applications running on multicore systems. Ericsson's 5G RAN PPF prototype uses DPDK's Eventdev framework [9] for scheduling and load balancing packet processing events over multiple CPU cores, and is designed to run on multicore systems with at least 4 CPU cores. The DPDK master thread requires one CPU core, while the rest of the CPU cores can be configured to run Eventdev threads. The PPF prototype configures Eventdev to run a centralized software scheduler on its first logical core (lcore), packet reception (Rx) and transmission (Tx) on its second lcore, and worker threads on the remaining lcores. The Eventdev configuration plays an important role in the optimization of Ericsson's PPF prototype and will be described in greater detail in Section 2.2.4. Virtualization and NFV will be described in Section 2.2.1.

This thesis studies the packet processing performance of a 5G RAN PPF prototype developed by Ericsson, by profiling its software and measuring and evaluating the throughput and latency while processing UDP/IP packets. A set of benchmarking and profiling tools were used to used to gain insight about the bottlenecks of the PPF prototype in an attempt to remove or mitigate the effects that these bottlenecks had on the throughput and latency of the PPF prototype. The PPF prototype was optimized using by reconfiguration of the Eventdev framework, including core assignments and consolidated packet processing stages. Lastly, an optimization of the RSS computation in the Rx driver was performed to further improve the throughput and latency performance. The purpose of this study was to gain a deeper understanding of an existing prototype of a 5G RAN PPF, and to optimize a 5G RAN PPF implementation in terms of throughput and latency.

## 1.2 Problem Statement

This thesis answers the question: *"Given an NFV PPF prototype for the 5G RAN based on the DPDK Eventdev framework, what are the bottlenecks of its packet processing path, and how can their effects be mitigated?"*

The investigation looks at implementation details of Ericsson's 5G RAN PPF prototype, and more specifically at its use of the Eventdev framework which is used for the scheduling of packet processing events and packet reception (Rx) and transmission (Tx). To gain insight into the bottlenecks and costly operations performed by the PPF prototype, Linux Perf and IxNetwork was used. The Eventdev framework provides a number of configurable parameters which affect for example the number of reception and transmission buffers, the size of the bursts of events which are enqueued or dequeued for processing, a limit on the number of new events which are allowed to enter the PPF prototype before old events are processed, and more. These parameters, as well as the scheduling quanta for different events, were considered when attempting to optimize the PPF's throughput and latency. The goal of this investigation was to provide an understanding of the effects of different configurations and implementation details of the PPF prototype on its throughput and latency when processing UDP/IP packets, and finally to optimize the performance of

the PPF prototype.

The optimization process was focused on (1) configuration optimization and (2) code optimization. While both methods were used to achieve better throughput and latency performance, there is an intended distinction between the use of the terms as follows:

- *Configuration optimization* achieves increased performance of the PPF prototype by changing the way it performs its tasks, i.e. a reassignment of CPU cores, changes to the scheduling of tasks, or changes to the packet processing stages.
- *Code optimization* achieves increased performance by increasing the speed at which tasks are performed, for example a speedup of the Receive Side Scaling (RSS) computation without changing the way that the PPF prototype operates.

Uses of the term *optimization* alone in this thesis refers to configuration optimization or code optimization interchangeably.

## 1.3 Purpose

The purpose of this thesis project was to aid Ericsson* in their development of a 5G RAN by providing a deeper understanding of the effects that different configurations of the Eventdev framework and other implementation details of a 5G RAN PPF have on the processing of UDP/IP packets. By understanding which parameters have the greatest effect on throughput, latency, and the execution of hot parts of the PPF prototype, the 5G RAN PPF can be optimized. This thesis is a step towards an optimized implementation of the 5G RAN PPF, and by extension a step towards a full deployment of the 5G standard to the market.

## 1.4 Goal

The end goal of this thesis project was to understand the effects of different configurations of an implementation of the Eventdev framework on throughput and latency of a 5G RAN PPF, and ideally to suggest the best 5G RAN PPF configuration from those that were studied. The results are presented in terms of the throughput and latencies achieved for different configurations of the adjustable parameters of the event device.

The thesis will be valuable to the teams at Ericsson working with the 5G RAN PPF by providing a deeper understanding of its functions and configurability, to others who are involved in the development of 5G RANs, and to others who want to implement efficient packet processing functions. A concrete deliverable is a set of example configurations which could aid in the design and evaluation of different packet processing function implementations.

* Telefonaktiebolaget LM Ericsson

## 1.5 Delimitations

This thesis looks at different configurations of a 5G RAN PPF prototype developed by Ericsson, and determines which of these configurations work best in terms of high throughput and low latency of UDP/IPv4 packets. To determine which configuration works best, tests were performed using Ixia's* IxNetwork software to generate UDP network traffic and perform throughput and latency measurements. In addition, the tools described in Section 2.3.2 were used for code profiling and benchmarking. The aim of these tests was to find critical parameters, potential bottlenecks, and costly operations caused by the configuration of the Eventdev framework used by the PPF prototype, and to collect general measurements on throughput and latency. The methodology of the throughput and latency measurements is described in Sections 3.2 and 3.3, and are based on Ericsson's previously configured test cases. The throughput was measured using Ethernet frames of only 90 bytes, which was close to the minimum possible size needed to accommodate a 32 byte payload and all of the required encapsulation. Using 90 bytes as the only Ethernet frame size for testing was deemed appropriate as these test results represent the worst case for throughput performance (more formally *goodput*, due to the small payload size relative to the overhead of the various headers). The number of packets per second (pps) was recorded. For the latency tests, a mix of frames of 86 bytes, 592 bytes, and 1522 bytes were used.

There are several key capabilities of 5G which can all be viewed as performance indicators - such as mobility, spectrum and energy efficiency, etc. This thesis will focus only on the optimization of throughput (peak and user experienced data rate) and data plane latency. In terms of optimizations of these capabilities, this thesis limits itself (1) to parameters within the code of the Eventdev framework and its registered drivers which run inside the Virtual Machine (VM) under the PPF prototype, and (2) potential optimizations of the hypervisor settings.

## 1.6 Methodology

This thesis studies the effects on throughput and latency that different configurations of the Eventdev framework and its configured drivers which are at the core of the PPF prototype developed by Ericsson have. A combination of Linux Perf and IxNetwork were used to profile the PPF to help identify the effects seen following a configuration adjustment. The study was performed in the following steps:

1. Pre-study to find state-of-the-art network analysis and performance profiling tools to help identify potential bottlenecks in the Eventdev framework of the PPF prototype. IxNetwork was chosen for traffic generation and measurements, as it was compatible with Ericsson's testbed equipment and had all the basic measurement functionality needed. Linux Perf was chosen for code profiling due to the relative ease of running it on the PPF prototype, and due to its extensive profiling features.
2. Using the IxNetwork and Linux Perf, a baseline measurement of UDP/IP processing with the 5G RAN PPF using the default event device configuration was produced. The maximum throughput achieved without any packet drops was recorded, as well as the

* https://www.ixiacom.com/products/ixnetwork

minimum, average, and maximum latency. Perf was used to identify expensive and frequent function calls.

3. Iterative reconfigurations of the PPF prototype were made and analyzed using IxNetwork and Linux Perf as described in step 2.
4. The results of these investigation are compiled and discussed in terms of throughput and latency in Chapter 4.
5. Suggestions and considerations for a finished 5G RAN PPF product are presented in Chapter 5.

# Chapter 2

# Background

This chapter explains the theoretical background for this thesis, including descriptions of 4G and upcoming 5G mobile networks, packet processing, code profiling, and NFV.

## 2.1 4G/LTE and 5G Mobile Networks

Mobile networks are currently evolving from their reliance on a variety of 4G standards towards an emerging 5G standard. This section describes the state of current mobile networks and the architecture of their Radio Access Networks (RANs) as well as an overview of the progress being made towards the development of a 5G standard. In this thesis we will focus on the 4G and 5G standards as specified by the 3rd Generation Partnership Project (3GPP).

### 2.1.1 State of Modern Mobile Networks

With the rapid worldwide increase in mobile data traffic and the advent of $5^{th}$ generation mobile networks (5G), the requirements on the performance of mobile networks will drastically increase. The IMT-2020 specification suggested by Working Party 5D of ITU-R lists a set of key parameters to define the performance of future mobile networks [10], of which the first three are the subject of this thesis:

- *Peak data rate* - The maximum data rate under optimal conditions. Data rate means the amount of data which can be sent through a part of a network per unit of time. This thesis uses the terms *throughput* and data rate interchangeably. Throughput is often given in the unit: bits/second (bps).
- *User experienced data rate* - The data rate that is achieved under typical conditions from the perspective of a user.
- *Latency* - The time it takes for a packet to traverse the given part of a network from entry to exit.

Also included in the key capabilities of IMT-2020 5G are requirements on:

- Mobility,
- Connection density,
- Energy efficiency,
- Spectrum efficiency, and
- Area traffic capacity.

Mobile network standards differentiate between downlink (DL) and uplink (UL) capabilities in their specifications. For example, IMT-2020 specifies different data rates and latencies for the DL and UL. The DL in this case refers to data connections from the core network (CN) towards the user equipment (UE), and the UL refers to the traffic from the UE towards the CN. The latency specifications of IMT-2020 also differentiate between what is called the *data plane* (sometimes *user plane*) and the *control plane*. The control plane comprises the functions of the network concerned with for example routing and network topology, while the data plane performs the actual forwarding of packet data according to the rules set by the control plane. Since the data plane performs per packet operations while the control plane generally does not, it is generally more important for data plane functions to be able to achieve the lowest possible latency. The IMT-2020 5G specification suggests improvements over its 4G predecessor IMT-Advanced with respect to all of the key capabilities, with a twentyfold increase in the peak data rate and tenfold improvements to the user experienced data rate and latency as shown in Table 2.1.

Table 2.1: Comparison of the 4G (IMT-Advanced) and 5G (IMT-2020) DL data rate and latency specifications [10, 11].

| | **4G (IMT-Advanced)** | **5G (IMT-2020)** |
|---|---|---|
| **Peak DL data rate (Gbps)** | 1 | 20 |
| **Ubiquitous DL data rate (Mbps)** | 10 | 100 |
| **Data plane latency (ms)** | 10 | 1 |
| **Control plane latency (ms)** | 100 | 10* |

The rest of this chapter introduces current and upcoming mobile network architectures designed to implement 4G and 5G standards.

### 2.1.2 4G - E-UTRAN

The 4G RAN, formally known as Evolved Universal Terrestrial Radio Access Network, abbreviated E-UTRAN), consists of a decentralized system of base stations called Evolved NodeBs (eNBs), which communicate with each other via a standardized interface named X2. The network of eNBs communication with the UEs via the Uu interface on one side and on the other side with the CN via the standardized S1 interface[13].

* Based on use case analysis in [12].

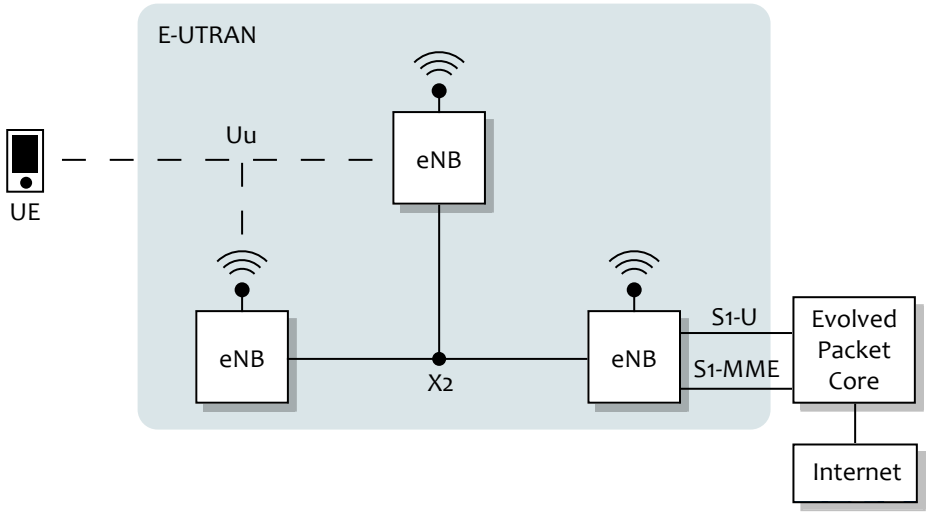



Figure 2.1: Simplified overview of the 4G architecture with E-UTRAN.

Each eNB performs both control plane and data plane tasks. The control plane tasks of the eNB include radio resource management functions, such as radio bearer control, connection mobility control, radio admission control, and dynamic resource allocation [14]. The data plane is responsible for the handling and forwarding of actual user data.

In the data plane, the protocol stack of E-UTRAN is responsible for the communication between UE and eNBs consists of a physical layer protocol generally referred to as PHY, a data link layer consisting of the Medium Access Control (MAC), Radio Link Control (RLC), and Packet Data Convergence Protocol (PDCP) protocols [15]. At the network layer, the UE communicates with E-UTRAN gateways in the Evolved Packet Core (EPC). In the control plane the IP protocol is replaced by the Radio Resource Control (RRC) and Non-Access Stratum (NAS) protocols at the network layer [14]. Figure 2.2 shows the E-UTRAN protocol stack.

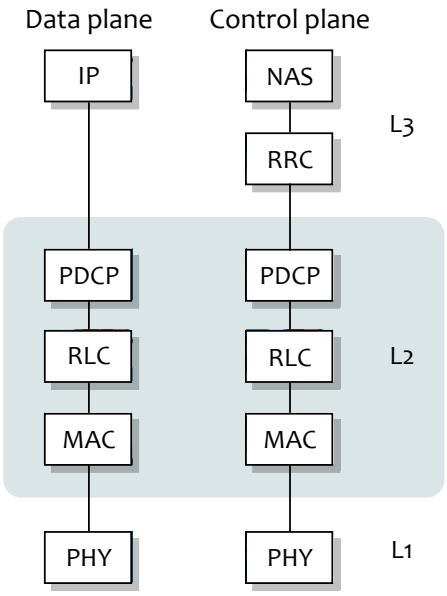



Figure 2.2: The protocol stack of E-UTRAN.

### 2.1.3 5G - New Radio

To achieve the performance and efficiency of the 5G RAN (also called New Radio (NR) [16]) specified in IMT-2020, the different network functions within the RAN architecture need to be highly adaptable and configurable to account for their environment and specific use case. For maximum flexibility and efficiency, 5G NR will likely be implemented as a combination of several different sub-architectures based on different mechanisms. These architectures include Distributed RAN (D-RAN) upon which E-UTRAN is based, Centralized RAN (C-RAN), and Virtualized RAN (V-RAN). [17]

#### 2.1.3.1 D-RAN

As explained in Section 2.1.2, E-UTRAN consists of a flat architecture of eNBs which work together and communicate with both the UE and CN, and lacks centralized controllers. This type of RAN without centralized control is known as D-RAN. D-RAN architectures come with several benefits, such as high mobility by easy implementation of fast handovers of UE connections [13], quick time to market, and easy deployment of individual base stations [17]. However, D-RAN is not optimal in all scenarios. In some cases better throughput and/or lower latency can be achieved by adding some degree of centralization, as will be explained further on. An example of D-RAN was shown in Figure 2.1 on page 8, illustrating the E-UTRAN architecture.

#### 2.1.3.2 C-RAN

In E-UTRAN there is no centralized control to enable fast handovers of user connections from a set of eNBs to another set of eNBs or to provide fast communication between eNBs and UE. However, as the demand for higher data rates increases, the density of nodes must

increase. Without any centralized logic to distribute workloads and allocate appropriate network functions, denser deployment may lead to inefficiency in the utilization of nodes, because of greater variance in workload between Radio Access Points (RAPs). The peak DL data rate of 20 Gbps and latency of 1 ms anticipated for the 5G NR may be achieved by ultra-dense deployment of RAPs together with centralized logic for on-demand allocation of specific network functions. Such functions may for example include high bandwidth packet delivery of streaming media to UEs or ultra-low latency messaging in machine-to-machine communications, for example between driverless vehicles [18]. This type of architecture, where centralized logic is used to allocate network functions, is referred to as Centralized RAN (C-RAN). While decisions on the architecture of the 5G RAN are still being made, it has been suggested that the 5G NR should be deployed with some degree of centralization [17, 19].

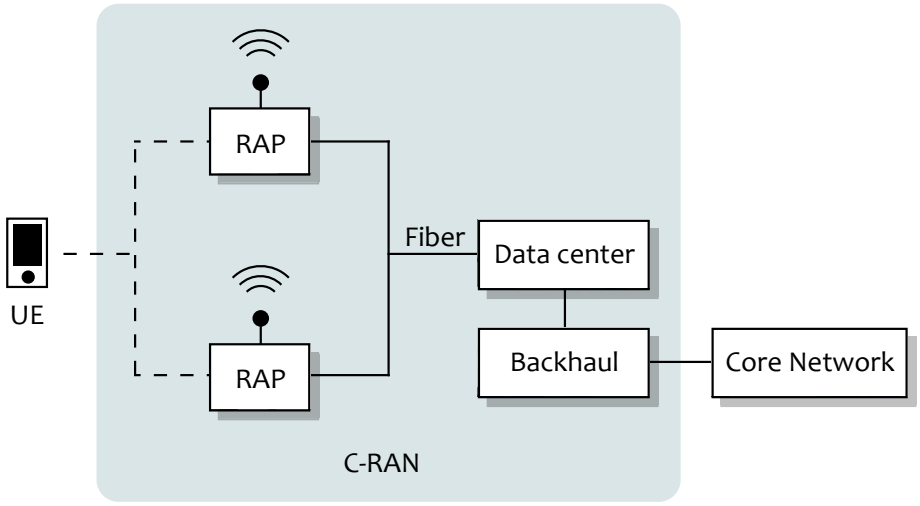



Figure 2.3: An overview of a generic C-RAN architecture.

#### 2.1.3.3 V-RAN

The development of 5G NR includes transitioning parts of the decentralized architecture of E-UTRAN which relies on specific purpose processors, towards centralized logic and general purpose processors with Software-Defined Networking (SDN) and Network Function Virtualization (NFV) [19]. A RAN that runs Virtual Network Functions (VNFs) on general purpose hardware is sometimes called a V-RAN. There are multiple potential benefits of NFV which relate directly to the needs of a future 5G RAN. NFV allows for separation of the logical network resources from physical network resources, hence running network functions which traditionally would run on special purpose hardware on general purpose hardware instead, which leads to [17]:

- Easy and cost effective deployment of new or upgraded VNFs.
- Scalability of the Virtual Machines (VMs) running each network function, leading to increased flexibility in the RAN.
- Increased energy efficiency with several VNFs running on the same physical host

machine and on different cores of the same processor.

#### 2.1.3.4 5G New Radio and Cloud RAN

While the exact specifications of the 5G standard are yet to be fully determined, there have been several suggestions about the details of the 5G NR architecture. Many point to what is often called Cloud RAN [17, 18, 10], which generally means a combination of D-RAN, C-RAN, and V-RAN where network functions can be allocated dynamically based on the needs of the network user. The aim is to use the best suited parts of each type of architecture for each use case in order to achieve the highest performance possible in the vastly different situations that are expected to occur in the deployment of 5G NR.

5G has also been proposed to use Multiple Radio Access Technologies (Multi-RAT) to include unlicensed spectrum for greater spectral efficiency [19]. For example, E-UTRAN can be used for the control plane while 5G NR provides higher throughput and/or lower latency. Multi-RAT solutions will likely also be required to provide coverage during the initial phase of 5G deployment [17].

### 2.1.4 Functional Splitting of the 5G NR

An important factor in the performance of the 5G RAN is the way in which the network is logically split into its different elements. This means the separation of network functions with different characteristics - for example those with less strict requirements on latency (mainly control plane functions) from those that require very low latency (for example data plane functions) or functions that can benefit from NFV from those that cannot. In the context of C-RAN, the degree of centralization, i.e. the decision of which network layers to centralize, is also an important question as it will affect the performance of the RAN. The degree to which the 5G RAN should be centralized is discussed in [17] and [18]. Centralization splits are considered within the PHY layer, between the PHY and MAC layers, and between the RLC and PDCP layers [17]. In the former case, parts of the PHY layer which can benefit from running in a distributed architecture would remain distributed while the rest of the PHY layer processing is centralized along with the higher layers. In the PHY-MAC split, a distributed PHY layer and centralized MAC (and higher) layer is discussed. Similarly the RLC-PDCP split suggests centralization of the PDCP layer and higher layers.

Erik Westberg provides an overview of the current E-UTRAN architecture and the envisioned 5G RAN architecture, with descriptions of their respective functional splits [20], as shown in Figures 2.4 and 2.5. A logical abstraction of the E-UTRAN architecture can be made by splitting it into its Radio Unit (RU) and a Digital Unit (DU) functions. The RU function performs the physical radio communication with the UE and connects to the DU function. The DU function performs both control plane and data plane functions with different levels of urgency.

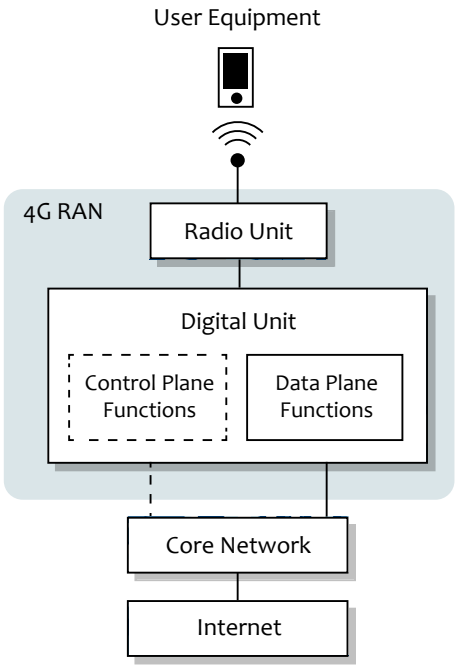



Figure 2.4: An abstract overview of the 4G RAN architecture. Solid lines represent connections between data plane functions and dashed lines represent connections between control plane functions.

As demonstrated in the functional abstraction of E-UTRAN shown in Figure 2.4, the DU function performs both control plane and data plane tasks, where the data plane generally requires lower latency [11]. To pave the way for further optimizations to reduce the latency of data plane functions, it is beneficial to separate architectural functions, increasing the flexibility and configurability of individual functions. For example, the envisioned 5G RAN architecture can be split into the functions depicted in Figure 2.5. In this functional split, the Radio Function remains its own function while the DU function of E-UTRAN is split into a Baseband Processing Function (BPF), Radio Control Function (RCF), and a Packet Processing Function (PPF). While the BPF performs both control plane and data plane functions, the RCF performs only control plane functions and the PPF mostly performs data plane functions. Thus the functions of the RCF and PPF have been completely separated. The suggested functions of the 5G RAN are well suited to NFV. Westberg suggests that the BPF be implemented on specific purpose hardware due to its strict requirements on spectrum efficiency, while the RCF is suitable for NFV. Implementation of the 5G RAN PPF as a VNF is the subject of this thesis. Its configuration will be studied based on code profiling and measurements of throughput and latency.

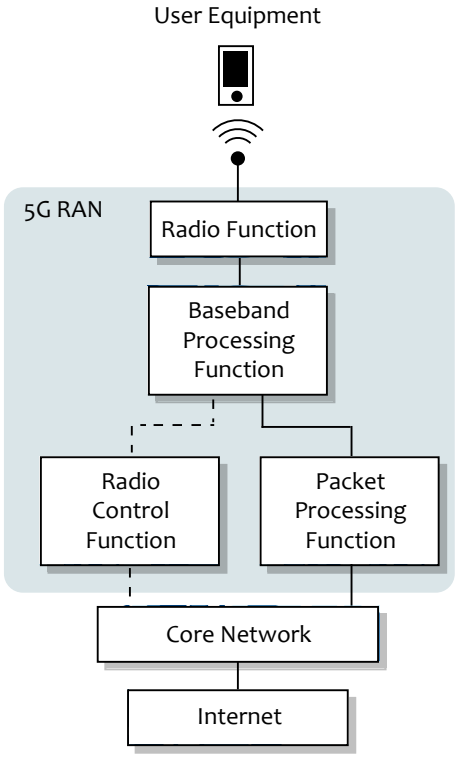



Figure 2.5: An abstract overview of an envisioned 5G RAN architecture. Solid lines represent connections between data plane functions and dashed lines represent connections between control plane functions.

## 2.2 Packet Processing for 5G RAN

In today's mobile networks, user equipment (UE) communicates with other UEs and with the core network via radio base stations called eNBs, which perform different types of processing of the data traffic. Packet processing is performed by an eNB; together with other types of processing such as baseband processing which prepares data for efficient transmission over the radio interface. A network *packet* is a unit of data which contains control information such as the source and destination of the packet, and the actual data transmitted (i.e. the *payload*). A Packet Processing Function (PPF) looks at the content of packets moving through the network and performs operations on the packet's data depending on the content of the packet. The PPF may perform common and simple but time critical functions such as forwarding of user data packets, and less frequent but more complex functions in the control plane such as processing packets which contain information about network topology changes, error handling, etc. The latter types of operations are often less critical in terms of throughput and latency. The path of packets inside the PPF that is made up of the most common and time critical operations is often referred to as the *fast* path and has traditionally exploited some type of hardware acceleration. The corresponding *slow path* is made up of the operations which require more advanced or less time critical types of processing and has traditionally been performed in software running on a general purpose processor.

In E-UTRAN (as described in Section 2.1.2), the packet processing, as well as baseband and

radio control, is performed by the eNB's DU and runs on special purpose hardware. The 5G RAN PPF, which has been factored out of the DU, represents a distinct unit as was shown in Figure 2.5 on page 13. The 5G RAN PPF is a virtualized adaptation of the packet processing elements of the eNB and is designed to run in a Cloud RAN architecture. The reasons for running a virtualized 5G RAN PPF include greater adaptability and configurability, thus the PPF VM guest can be installed on any general purpose machine that fulfills the hardware requirements. This approach makes it easier to deploy a PPF, to configure the emulated hardware of the VM to better accommodate the PPF software, and more.

This section will explain the concepts related to packet processing and how it is being implemented for upcoming 5G standards. These concepts include virtualization, NFV, Event-driven packet processing using DPDK, as well as packet I/O and its related performance implications.

### 2.2.1 Virtualization

Virtualization enables multiple Virtual Machines (VMs) to run on physical machines. There are multiple aspects of virtualization, each with different types of benefits. For example, applications designed for an architecture different than that of the physical machine can be run inside a VM that offers the desired architecture. A VM can also be very flexible because its hardware components can be emulated, hence VMs can be easily changed without having to replace any physical components. This section will explain the concepts of virtualization and Network Function Virtualization (NFV). NFV refers to the virtualization of network functions. NFV is a cornerstone in the development of large parts of the upcoming 5G standard, and is largely the basis of the 5G RAN PPF.

Virtualization can be realized on many general purpose machines. Figure 2.6 shows an overview of a generic virtualization setup. The physical machine, which is generally referred to as the *host machine* or *host*, often runs an operating system referred to as the host OS. This host OS runs a special type of software called a *hypervisor*, which provides support for maintaining and running VMs. Some hypervisors are capable of running on bare-metal, i.e. without a host OS - such hypervisors are called Type 1 hypervisors, while hypervisors running on top of a host OS are called Type 2 hypervisors. A hypervisor may run one or more different VMs at once. A VM consists of emulated hardware components, which can be fully or partly independent of the components of the host. The concept of complete virtualization of all components of the VM is called *full virtualization*. The emulation of a subset of components of the VM is called *paravirtualization*. A VM, sometimes called a *guest machine* or *guest*, can run a guest OS which is often different from the host OS. For example, VMs can be used to run applications and functions which cannot be executed directly by the host OS. Another benefit of virtualization is that functions inside VMs are generally isolated from functions running inside other VMs. This isolation can be emphasized by allocating different host resources to different guests, for example by CPU core isolation where different VMs are allocated different cores of the host's CPU.

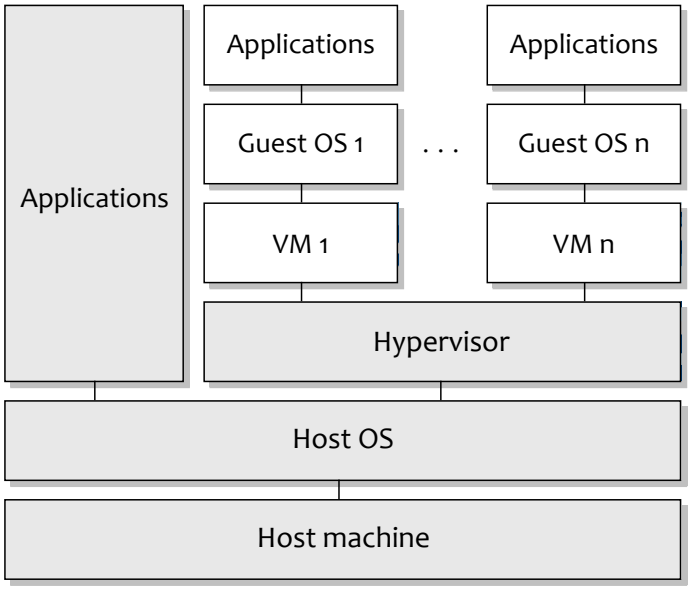



Figure 2.6: An overview of a generic virtualization with a host machine running several guest systems.

Network Function Virtualization (NFV) refers to the virtualization of network functions so that they can run on general purpose hardware instead of having to run on special purpose hardware as was previously the case. Network functions that are virtualized are commonly called Virtualized Network Functions (VNFs). NFV offers several potential benefits, including the following that were listed in a white paper produced by contributors to the ETSI NFV Industry Specification Group (NFV ISG) [4]:

- *Reduced equipment costs* - By running several VNFs on the same physical general purpose machine, the cost of maintenance and power consumption can be reduced. An example of NFV with multiple VNFs running on the same host machine is shown in Figure 2.7.
- *Faster time to market* - New network products and services can be deployed faster as the reliance on special purpose hardware is reduced. For example, new VNFs can be pushed to previously deployed general purpose machines.
- *Scalability of network functions* - Capabilities of VNFs can potentially be extended while still running on the same platform.
- *Innovation encouragement* - The deployment difficulty and cost both decrease, facilitating research driven deployments of experimental network services.

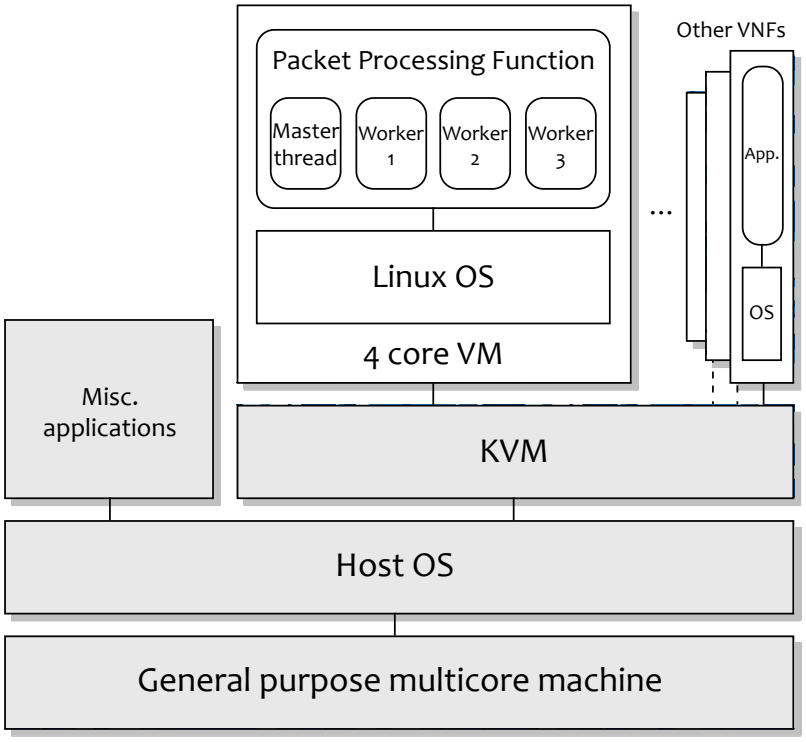



Figure 2.7: Example use of NFV. In this example, a general purpose host machine has been configured to accommodate multiple VNFs. One such VNF is demonstrated: the PPF which runs inside a 4 core VM and is isolated from the other VNFs on the host system.

However, reaping all of the potential rewards of NFV is non-trivial and presents challenges. For example, virtualization usually comes with a performance penalty which VNFs must overcome. This challenge is also emphasized in many applications by the requirements on portability of the VNFs between different hypervisors and host systems. Different hypervisors are available for different host and guest systems, and each hypervisor implements different functionalities and has different characteristics. In the case of the 5G RAN PPF prototype tested in this thesis, the performance penalty is minimized by choosing QEMU/KVM as hypervisor, as this hypervisor has previously shown high performance when running VNFs in similar tests [21]. To realize the performance of the underlying host machine, CPU core isolation can be used. This helps increase guest performance by pinning threads to specific CPUs. This pinning of threads to specific cores has several beneficial effects, including avoidance of CPU migrations and maintaining cache coherence, both of which help maximize performance.

Using LMbench 3 alpha 9 (see Section 2.3.2) on the host and guest machines of Ericsson's PPF prototype, a comparison between the host and guest performance was made. To illustrate the performance penalty of the virtualization, the memory and cache access latency (Table 2.2), memory read/write bandwidth (Table 2.3) and context switch (CS) time (Table 2.4) were measured. Table 3.1 in Section 3.1 shows the specifications of the testbed setup. The configuration described in Section 3.1 LMbench was configured using its *config-run* script with the following settings (descriptions of these settings are available in Appendix A.3):

- MULTIPLE COPIES [default 1]: 1

- Job placement selection [default 1]: 1
- MB [default ...]: 1024
- SUBSET (ALL | HARWARE | OS | DEVELOPMENT) [default all]: all
- FASTMEM [default no]: no
- SLOWFS [default no]: no
- DISKS [default none]: none
- REMOTE [default none]: none
- Processor mhz [default ... MHz, ... nanosec clock]: [default]
- FSDIR [default /var/tmp]: /var/tmp
- Status output file [default /dev/tty]: /dev/tty
- Mail results [default yes]: no

The results show decreased performance in terms of memory latency and bandwidth. However, the context switch times were actually shorter in some measurements. This may be due to the fact that the host and guest OS are built around different kernels which may handle context switches differently, and that fewer processes ran on the guest machine.

Table 2.2: Latencies of L1-L3 cache and main memory reads in nanoseconds as reported by LMbench. The bottom row displays the relative performance of the guest machine and is calculated as $frequency_{guest}/frequency_{host}$ for the CPU MHz frequency column and $latency_{host}/latency_{guest}$ for the cache and main memory latency columns.

| | **CPU clock frequency (MHz)** | **L1 cache latency (ns)** | **L2 cache latency (ns)** | **L3 cache latency (ns)** | **Main memory latency (ns)** |
|---|---|---|---|---|---|
| **Host** | 2899 | 1.3790 | 5.3080 | 24.3 | 130.2 |
| **Guest** | 2460 | 1.6340 | 11.5 | 70.4 | 136.3 |
| **Relative guest performance (%)** | 84.9 | 84.394 | 46.2 | 34.5 | 95.52 |

Table 2.3: Memory read/write bandwidth for the host and guest machines as reported by LMbench. The bottom row displays the relative performance of the guest machine and is calculated as $frequency_{guest}/frequency_{host}$ for the CPU frequency column and $bandwidth_{host}/bandwidth_{guest}$ for the memory read/write bandwidth columns.

| | **CPU clock frequency (MHz)** | **Memory Read (MB/s)** | **Memory Write (MB/s)** |
|---|---|---|---|
| **Host** | 2899 | 10000 | 7450 |
| **Guest** | 2460 | 6640 | 5599 |
| **Relative guest performance (%)** | 84.9 | 66.4 | 75.2 |

Table 2.4: CS time in microseconds for different numbers of processes and process sizes on the host and guest machines as reported by LMbench. The relative performance of the guest machine is calculated as $frequency_{guest}/frequency_{host}$ for the CPU frequency column and $cs_{host}/cs_{guest}$ for the CS columns.

| | **CPU clock frequency (MHz)** | **2p/0K (us)** | **2p/16K (us)** | **2p/64K (us)** | **8p/16K (us)** | **8p/64K (us)** | **16p/16K (us)** | **16p/64K (us)** |
|---|---|---|---|---|---|---|---|---|
| **Host** | 2899 | 4.470 | 3.510 | 3.040 | 2.610 | 1.780 | 2.430 | 1.670 |
| **Guest** | 2460 | 0.630 | 1.050 | 1.240 | 1.840 | 2.500 | 2.160 | 2.600 |
| **Relative guest performance (%)** | 84.9 | 710 | 334 | 245 | 142 | 71.2 | 113 | 64.2 |

### 2.2.2 Packet I/O

Realizing packet processing as a VNF with high throughput and low latency requires efficiency within the I/O and network stack on the packet processing platform. Recent efforts to improve packet processing speeds - such as the Data Plane Development Kit (DPDK) [8] - have been made by reworking the way in which network applications communicate with Network Interface Cards (NICs) on Linux platforms. The PPF implementations tested in this thesis will use DPDK drivers and libraries for packet processing. To better understand DPDK and its effects on packet processing, it is beneficial to also have an understanding of the way in which packets are traditionally handled by Linux; how packets traverse the Linux network stack from arrival at the NIC to an application upon reception, and vice versa during transmission. This section will now explain the general process of packet reception (Rx) and transmission (Tx) of Ethernet frames in the traditional Linux network paradigm using the Linux New API (NAPI) [22], based on the previous work by W. Wu, M. Crawford, and M. Bowden in "The performance analysis of linux networking – Packet receiving" [23].

The packet Rx procedure is illustrated in figure 2.8. When an Ethernet frame arrives at the NIC, it is buffered in the NIC's internal hardware and transferred to kernel space main

memory by the device driver using Direct Memory Access (DMA). The NIC and the Linux kernel each also have buffers of data structures generally referred to as packet descriptors which contain packet metadata, realized in the Linux kernel code as a ring buffer structure called *sk_buff* [24]. A packet descriptor of an arriving Ethernet frame is initially stored in the NIC before being copied into an available sk_buff residing in kernel space main memory. The NIC then informs the CPU of the arriving frame with a hardware interrupt request. The NIC's interrupt handler, which is part of the NIC driver, then schedules a software interrupt. For each CPU the kernel maintains a poll queue of devices that have made software interrupt requests. The kernel polls the NICs referenced in each CPU's poll queue via the NICs' device driver poll function to retrieve new packets contained in the sk_buffs of the ring buffer for further processing by the kernel's network stack. When an sk_buff is taken from the ring buffer, a new sk_buff is allocated and appended to the ring.

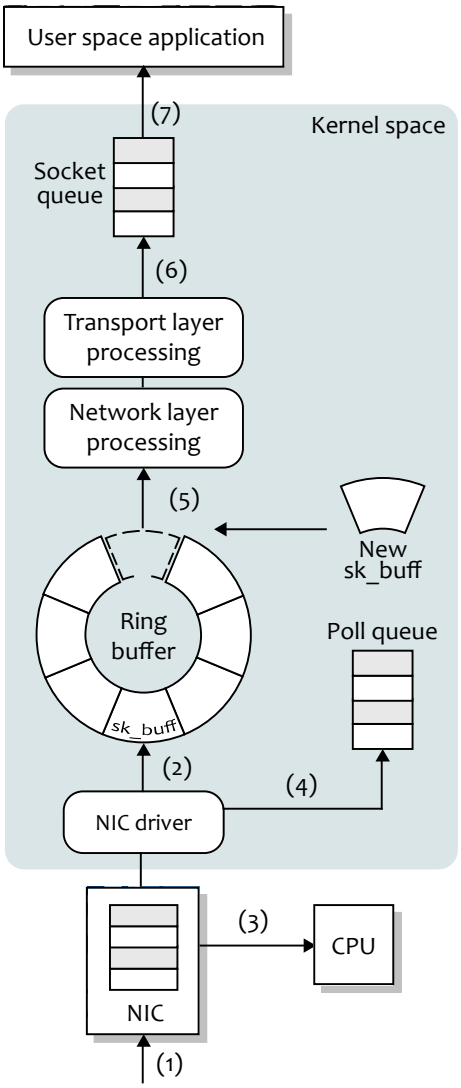



Figure 2.8: shows the following steps in processing of the arriving Ethernet frames:

1. Ethernet frame arrives as physical signals at the NIC and is stored in its internal hardware buffer.
2. Ethernet frame is transferred by the NIC device driver to a ring of packet descriptors (sk_buffs) in kernel space main memory, using DMA.
3. Hardware interrupt from NIC to indicate that a packet has arrived and is available to the kernel. This hardware interrupt causes the NIC device driver to run its interrupt handler.
4. The NIC device driver schedules a software interrupt request for the NIC device in the CPU's poll queue. The poll queue is polled to check if any of the NICs have reported new incoming packets.
5. If a device is scheduled in the CPU's poll queue, sk_buffs are taken from the ring buffer for further packet processing in the network stack. New sk_buffs are allocated and appended to the ring to replace the used descriptors.
6. The processed packet is placed in the application's socket queue.
7. Application retrieves packet via `read/recv` or similar system call.

The Linux network stack will process incoming packets differently depending on the protocols involved. IP packets are removed from the ring buffer and processed by calls to the `ip_rcv()` (`ip6_rcv()` for IPv6) function of the IP stack [25, 26]. This function extracts the IP header, performs checks on the packet, etc. and determines what to do with it, i.e. transport layer processing, forwarding, or dropping the packet. If the packet has arrived at the destination host, then the kernel will proceed with for example TCP or UDP processing before submitting the packet to the receiving application's socket receive buffer. From there, a user space application can acquire the packet, copying the data into user space memory, and removing the packet from the socket receive buffer using for example the `recv()` [27] system call.

Packet Tx can be performed by a call to the `send()` system call (or equivalent) which copies the message from user space to kernel space memory where protocol processing is performed, placing packet information in an sk_buff in a Tx packet descriptor ring, and eventually transmits the packet by calling `hard_start_xmit()`, which is a pointer to a Tx callback function defined by the NIC's device driver [28].

While Linux's traditional approach to packet I/O maintains a clear separation between kernel and user mode and protects the NIC device from direct interaction with user space applications, it involves many operations which slow down the performance on the packet's I/O path as demonstrated by Georgios P. Katsikas in [7]. Key factors in the slow performance of the traditional Linux network I/O, compared to recent alternatives such as the Data Plane Development Kit (DPDK), include the number of expensive system calls and context switches involved in the I/O process, which in large part are caused by the copying of data between user space and kernel space memory.

To mitigate the problem of slow I/O due to excessive data copying and context switching, DPDK (which is described in greater detail in Section 2.2.4) introduces a set of libraries and drivers which allow direct access to a NIC's hardware storage from Linux user space. Traditionally Linux Rx requires a receive system call which involves context switching to the kernel to process the incoming packet before copying the packet data to user space and context switching back to user space. By mapping a NIC's hardware buffers to user space memory and bypassing the kernel, these context switches and data copies can be avoided. DPDK's packet I/O process is shown in Figure 2.9.

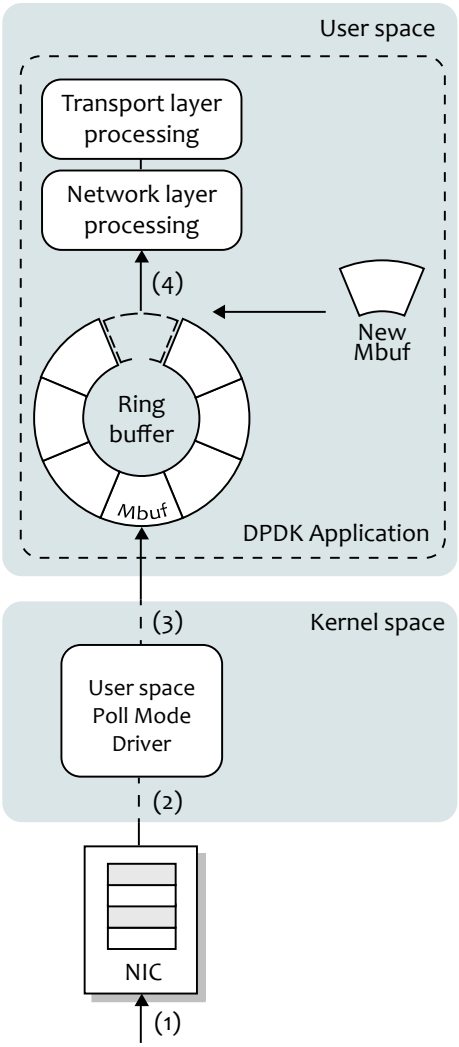



Figure 2.9: The following steps are performed during packet Rx using DPDK:

1. Ethernet frame arrives as physical signals at the NIC and is stored in its internal hardware buffer.
2. DPDK's registered Poll Mode Driver (PMD) polls the NIC directly, looking for arriving Ethernet frames.
3. Frames are received by the application in user space, bypassing the kernel to avoid data copying and context switching between kernel and user space. Frame information is placed in Mbufs in a ring buffer by the application.
4. The application performs the desired processing of packets at the network and transport layers.

DPDK maintains a ring buffer of a structure called *Mbuf*, which corresponds to Linux's sk_buff structure. These are mapped to the packet descriptors of a NIC and are passed to a user space application by a Poll Mode Driver (PMD) registered by DPDK. Polling requires a CPU core to actively query the NIC for incoming packets at some rate. When a NIC receives packets at a high rate, this technique grants good performance by avoiding the requirement for handling interrupt requests [7]. Since the purpose of the 5G RAN PPF is to process incoming packets at a high rate, polling is the preferred method for acquiring packets from

the NIC. However, when the rate of incoming traffic is low, the CPU core used to poll the NIC is underutilized.

While DPDK's network I/O is designed to outperform traditional Linux's network stack, it blurs the boundaries between Linux kernel and user space by allowing user space applications access to the hardware address space of the NICs. DPDK also leaves all of the responsibility for packet processing to the application, whereas the Linux kernel's network stack performs some of the network and transport layer processing. In the case of the 5G RAN PPF, the increased I/O speed and flexibility in the implementation details of the packet processing motivates the use of the DPDK rather than relying on Linux's traditional network stack.

### 2.2.3 Receive Side Scaling

Receive Side Scaling (RSS) is a technology which distributes Rx packets to different CPU cores to achieve good performance scaling in multicore systems. RSS processes the received packet data in the NIC through a hash function, which produces a hash value. The least significant bits of the hash value are used to index an indirection table, which contains the IDs of the available CPU cores. Figure 2.10 shows the general RSS process. [29]

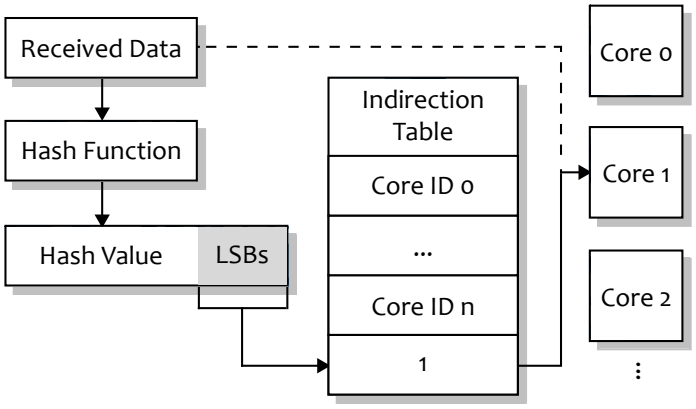



Figure 2.10: An overview of the RSS core selection process. Derivative of the illustration at [29]

DPDK implements RSS as a software function, based on the Toeplitz hash function. The code for DPDK's RSS function which is used by the PPF prototype and was subject to code optimization in this thesis project, is shown below and is available at [30]:

```
static inline uint32_t
    rte_softrss_be(uint32_t *input_tuple, uint32_t input_len,
    const uint8_t *rss_key)
{
    uint32_t i, j, ret = 0;

    for (j = 0; j < input_len; j++) {
        for (i = 0; i < 32; i++) {
            if (input_tuple[j] & (1 << (31 - i))) {
                ret ^= ((const uint32_t *)rss_key)[j] << i |
                    (uint32_t)((uint64_t)(((const uint32_t *)rss_key)[j + 1]) >>
```

```
                    (32 - i));
            }
        }
    }
    return ret;
}
```

### 2.2.4 The Data Plane Development Kit

The Data Plane Development Kit (DPDK) is an Open Source BSD licensed* set of libraries and NIC drivers for fast packet processing in data plane applications [8]. DPDK can be used in Linux systems running on a wide range of processor architectures, including Intel x86, ARM and more†. This section summarizes some of the most important design principles and implementation details of DPDK necessary for understanding this thesis. The information is based on the DPDK Programmer's Guide [31] which contains a more complete description of DPDK and its functions. The components of DPDK which are used by the PPF prototype include:

- *Environment Abstraction Layer* - The Environment Abstraction Layer (EAL) allows the libraries of DPDK to communicate directly with low level resources from user space via an abstract and highly configurable interface. It is responsible for the initialization of DPDK, memory mappings for direct access to the NICs' memory, CPU core affinitization, keeping track of CPU specific properties and functions, and more.
- *Mbuf Library* - Similar to the sk_buff in the traditional Linux packet I/O paradigm, as was described in Section 2.2.2, DPDK uses packet descriptor buffers for incoming and outgoing packets, although they differ from the sk_buff in the details of their implementation.
- *Mempool Library* - The Mempool library handles allocation of Mbufs by DPDK applications. It includes features such as per logical core (lcore) caching of Mbufs to avoid excessive inter-core communication. It also optimizes the spread of objects over the DRAM and DDR3 memory channels by memory alignment.
- *Ring Library* - While traditional Linux ring buffers are implemented as a circular doubly linked list of the packet descriptor struct sk_buff, DPDK implements a ring buffer as a fixed size table of pointers to packet descriptors. This design improves the performance of queue operations at the cost of a larger memory footprint and less flexibility. Some of the features of the DPDK ring manager library include its lockless implementation, multi- and single-producer/consumer operations to append or retrieve packets and bulk operations for appending or retrieving a specified number of packets at once.
- *Poll Mode Drivers* - DPDK allows an application to register a Poll Mode Driver (PMD). This PMD is used to poll a system's NICs in order to receive packets. The purpose of polling the NICs is to avoid the use of interrupts, which can degrade the performance of high throughput VNFs. Using DPDK's hardware abstraction layer (HAL), the PMD can poll the NIC and retrieve packets directly in user space.

* https://opensource.org/licenses/BSD-3-Clause † There is also a port of DPDK for FreeBSD.

- *Eventdev Framework* - A framework which supports an event driven packet processing model. This framework introduces scheduling functionality and dynamic load balancing between lcores.

The 5G RAN PPF proposed by Ericsson is a virtualized adaption of the PPF of the DU inside E-UTRAN's eNBs. This PPF is built using DPDK's Eventdev framework [9] with a software event device driver. The PPF maps a virtual *event device* inside the user space of a Linux guest to the physical NICs of the host machine. An event device is essentially an event scheduler, which configures a set of *event queues* used to hold different types of *events* waiting to be scheduled. An event is a unit of schedulable work, for example a packet that has just been received and awaits further processing, a timer expiry notification, or an intermediate unit of work between packet processing pipeline stages. Linked to each event queue is an *event port*, which is used to configure the enqueue and dequeue operations associated with one or more event queues. An event port can be linked to multiple event queues. The event port configures *enqueue depth* and *dequeue depth*, which specify the number of event objects to enqueue or dequeue in a burst operation where multiple events are enqueued or dequeued at once. This parameter may affect the throughput and latency of packet processing since burst enqueue/dequeue operations reduce the total amount of enqueue/dequeue operations that have to be performed. Depending on the balance between the cost of a single enqueue/dequeue operation and the processing time after enqueue/dequeue, this may increase throughput and in some cases reduce the average packet delivery latency of the application. However, burst operations may also increase the per-packet latency when the operation must wait for a certain number of events to arrive before the enqueue/dequeue. The maximum time to wait for enough events in the event queue to perform a successful burst operation on an event port is specified by the event port structure. Also configured by the event port is the *new event threshold*, which is used to control the number of new events which are allowed to enter the event device. This parameter can be used to make the Eventdev complete processing of older events before accepting new ones. As such, an optimized setting of this parameter may reduce the latency of packets in cases when many new events are produced.

In a multicore architecture, different CPU cores can be assigned to poll the event queues and schedule events of any type or specified types. This approach makes it easy to implement natural and dynamic balancing of workloads between CPU cores. One lcore (the DPDK master lcore) is reserved for program control while the rest of the lcores can be divided as desired with a subset of lcores used for scheduling of events, a subset of lcores for Rx/Tx, and the rest of the lcores as workers for different packet processing stages. The event queues are capable of simultaneously queueing events from multiple flows. This functionality can be configured using different event scheduling types, which specify how events belonging to a flow can be scheduled for processing. The scheduling types are:

**Ordered** Events from a flow handled by the event queue can be scheduled for parallel processing on different lcores, but the original order of the events is maintained. Correct ordering is ensured by the enqueue operation on the destination event queue of the events. The event port handling the events of an ordered event flow will only process events of this particular flow until the burst dequeue operation is performed on the port, or earlier if the burst enqueue operation is performed with an option to release the flow context.

**Atomic** An atomic flow can be scheduled only to a single event port at a time. This differs from the ordered scheduling type by not scheduling events concurrently to different ports. However, the original order of events is maintained also for the atomic scheduling type since only a single port will schedule processing of a flow.

**Parallel** Events from a flow can be scheduled in parallel to different ports. Event order is not necessarily maintained.

The ASCII image in Figure 2.11 displays the functionality of the Eventdev framework.

The PPF software is driven by DPDK and receives and transmits packets directly from Linux user space. Incoming packets are acquired from the NIC's Rx queues using a PMD which is scheduled on a CPU core reserved for handling of packet I/O. When a new packet has arrived, a packet Rx event is generated by the event device and is placed in an event queue for packet processing by the worker cores. The worker cores dequeue new and intermediate events from the event queues based on decisions from the scheduler cores. The worker cores then perform the different stages of packet processing, which eventually finishes and may generate a Tx event for the response to/forwarding of the packet that was initially received.

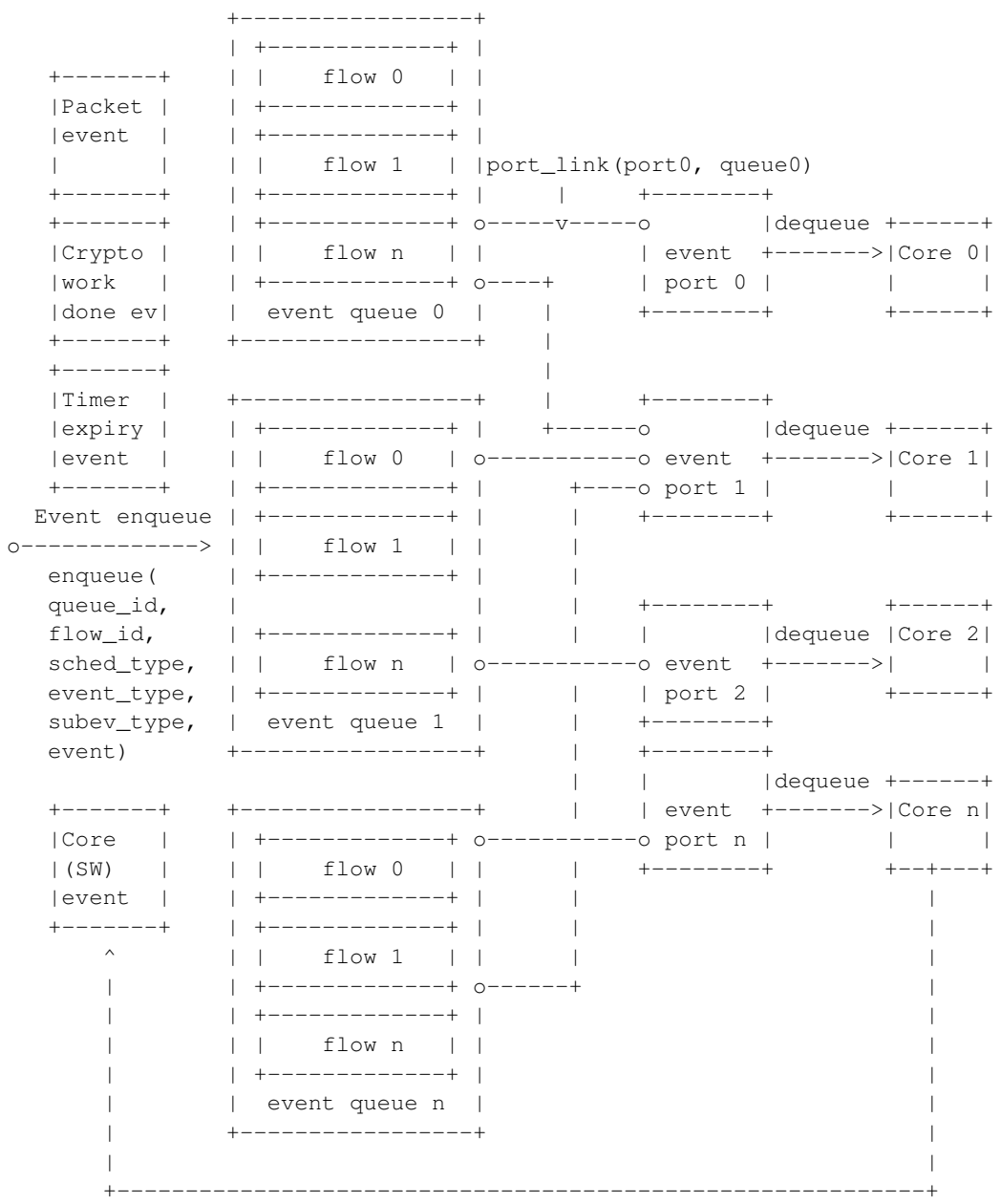



Figure 2.11: Functionality of the Eventdev framework - taken from the DPDK Eventdev documentation [9].

### 2.2.5 A 5G RAN PPF Prototype

The 5G RAN PPF prototype developed by Ericsson is based on the DPDK Eventdev framework as described in Section 2.2.4. This section will further describe the way in which the PPF prototype was configured.

The PPF prototype is designed to run on a general purpose machine using at least 4 CPU cores. The cores are distributed as follows: one master core to process user I/O for PPF control, one core to perform scheduling, one core to perform both Rx and Tx of packets, and one worker core to perform packet processing. If the PPF prototype is offered more than 4 cores, the additional cores are configured as worker cores. The PPF prototype was configured with a centralized software scheduler running on a single core and performing the scheduling for all worker cores. The PPF prototype has 7 event queues: 1 Rx queue, 1 Tx queue, 1 timer queue and 4 worker queues. The Rx and Tx queues are used to enqueue events resulting from packet reception and events of packet transmissions to be executed respectively. The timer event queue is used to enqueue timer expiry events. The worker queues are used to enqueue events for different stages of packet processing. For the purpose of this thesis the PPF prototype was configured with 4 processing stages, each with their own event queue. The first stage performs classification of events to decide how to process it further. This stage also increments the event flow ID of the event, which loops back to 0 at 512. Thus, the PPF prototype creates 512 flows. The second and third stages are dummy stages which consume a number of CPU cycles to simulate the time consumption of packet processing. The fourth and final stage also consumes a number of cycles before applying some actual packet processing and sending the packet out on the Tx event queue for transmission. The priority of each stage decreases with the stage number, i.e. the first stage has the highest priority and the fourth stage has the lowest priority.

The PPF prototype has 4 event ports linked to its event queues. The Rx, Tx and timer event queues each have their own event port and are the only queues linked to their respective ports. The worker event queues all share the last event port.

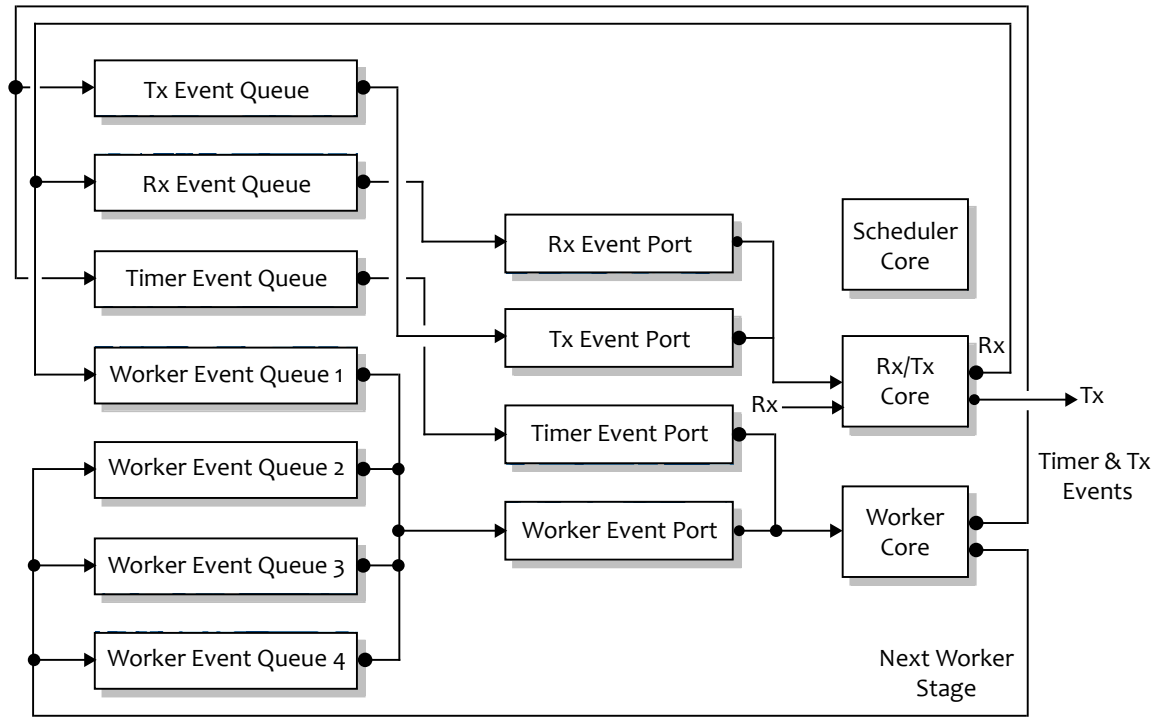



Figure 2.12: The PPF prototype is programmed using DPDK's Eventdev framework and was configured with 7 event queues and 4 event ports. It runs on a minimum of 4 cores. In this figure, the CPU core running the DPDK master thread has been left out since it is not configured by the Eventdev framework.

## 2.3 Performance Optimization

This section describes the ideas that helped optimize the performance of the PPF prototype. Section 2.3.1 gives a brief introduction to some system metrics and their effects on the performance of systems and applications. Section 2.3.2 presents a set of tools which can be used to gather information about the presented system metrics.

### 2.3.1 System Performance Factors

There are a variety of system metrics and factors that are vital to a system's performance. The performance impact of these metrics may be affected by reconfiguration and code optimization. A few of these are described below.

#### 2.3.1.1 Context switch cost

A context switch (CS) means that the state of a process is saved and later restored to the saved state. A CS can occur during multitasking when a process is swapped out for another, at an interrupt, or when switching between Linux user/kernel space. The number of CSs and cost of context switching can have a large impact on the system's performance since

each CS introduces additional overhead in execution time. The cost of a CS can depend on the number of processes in the system, the size of a process, the choice of scheduler, etc. For example, if a CS occurs due to an excessive use of system calls in an application, i.e. the CS is a switch between Linux user and kernel space, then the performance of the application may be optimized by reducing the number of system calls. Figure 2.13 shows general process of a CS.

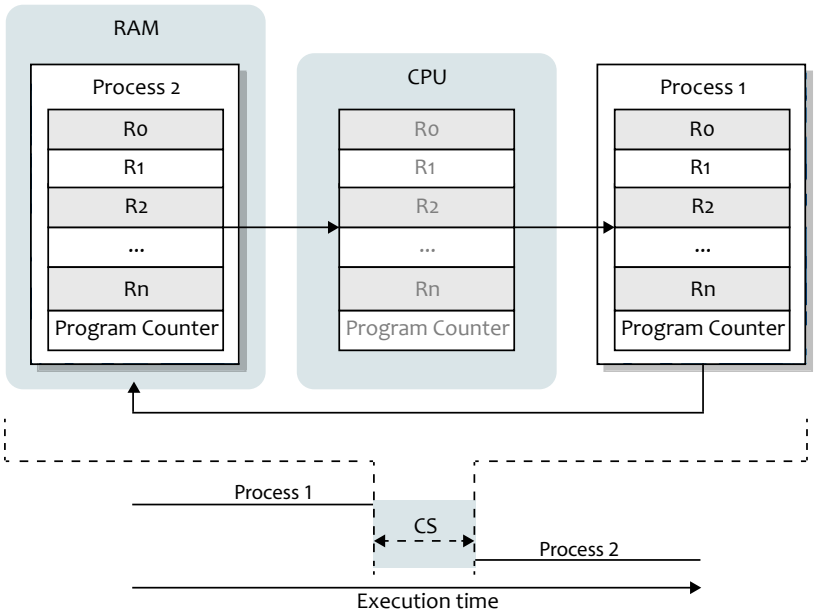



Figure 2.13: The general process of a context switch (CS). Before the CS, the CPU executes process 1. During the CS, the CPU stores its registers in main memory, and loads into its registers the previously stored registers from process 2 from main memory. Process 2 then proceeds to execute. The diagram at the bottom illustrates the execution time overhead of the CS.

#### 2.3.1.2 Memory access cost

The number of memory access and the cost of memory access by an application can have a large impact on system performance. The cost of memory access is determined by the bandwidth and latency of memory access operations, which result from the system's hardware specifications, the amount of contention for memory resources, etc. The cache hit/miss rate also affects memory access, as we shall see next.

#### 2.3.1.3 Cache hit/miss ratio

To reduce the cost of memory access, most modern general purpose processors are equipped with a small amount of fast but expensive memory called cache memory. This cache memory is often arranged in a hierarchy of increasingly larger but slower memories as the distance to the processor increases. A common cache configuration is to have level 1 (L1 cache), 2 (L2 cache), and 3 (L3 or LLC for last level cache) caches. The L1 cache is often divided into separate a instruction cache and data cache. When an application makes a memory access,

a lookup of the accessed memory address will simultaneously be performed in the cache hierarchy. If the address is present in a cache, then the instruction or data will be fetched from the cache instead of the main memory. This is referred to as a *cache hit* and is usually many times faster than accessing the main memory. If the accessed address is not present in a cache, the instruction or data will be fetched from main memory and usually also copied to the cache for future references. This is referred to as a *cache miss*. If the cache is full when bringing a new entry into the cache, old entries must be evicted. Maintaining a high cache hit rate can be an important factor in performance optimization.

Caches generally store instructions and data that have recently been used, or that have been fetched from memory due to their close proximity in the memory's address space. The proximity of accesses in time is often referred to as *temporal locality*, and proximity in address space as *spatial locality*. To utilize the cache as much as possible, it is important to consider the temporal and spatial locality of memory access when programming an application. Effective use of a system's caches can often drastically increase the performance of an application.

#### 2.3.1.4 CPU Load Balance

A balanced workload among the CPU cores is an important factor when optimizing parallel code in a multicore system. DPDK's Eventdev framework performs dynamic load balancing among its worker cores. However, there are still potential bottlenecks when the number of scheduling, Rx/Tx, and worker cores are static. For example, with a large number of worker cores, having only a single scheduling or Rx/Tx core might form a bottleneck. With such a bottleneck, speedup of the packet processing by the worker cores will not help. For this reason, it is important to assign the right number of cores to each functionality.

### 2.3.2 Performance Profiling

This section presents the set of tools that were used to gather information about the metrics described in Section 2.3.1 in order to present a basis for optimization of the 5G RAN PPF.

#### 2.3.2.1 LMbench

LMbench [32] is a benchmark suite for Linux systems capable of measuring a variety of system performance metrics. LMbench provides benchmarks for the bandwidth of cached file reads, memory copying, memory reads, etc. as well as latency benchmarks for context switching, process creation, signal handling, memory reads, etc. This thesis project used LMbench 3 alpha 9 to measure and compare host and guest machine performance (see Section 2.2.1. LMbench can be configured using its `config-run` script, which configures different test parameters and gathers basic performance information such as the CPU's clock frequency. Using the `results` script, the tests are then performed according to the configuration file produced by config-run. Test results can be viewed using the `make see` command from the LMbench root directory. The following process was used to gather performance information on the PPF prototype's host and guest machines:

1. Shut down the PPF process

2. Run LMbench3 script `./config-run` from the lmbench/scripts/ directory
3. Run LMbench3 script `./results` from the lmbench/scripts/ directory
4. Get result report using the `make see` command from the lmbench/ directory

#### 2.3.2.2 Linux Perf

Perf [33] is a code profiling tool shipped with the Linux kernel under `tools/perf`. Perf uses performance counters to present statistics about the execution of an environment or application in a variety of ways. These performance counters include cache references and cache misses, branches and branch misses, instructions, cycles, page faults, CPU migrations, context switches, and more. Perf is capable of monitoring specific processes by passing their process IDs (PIDs) to a Perf command. Some valuable commands of Perf include:

- **perf stat** - The `perf stat` command can be used to gather information from selected performance counters. Which performance counters to report can be specified using the `-e` option. Additionally, a specific PID can be profiled using the `-p` or `-pid` option. The following example output was produced when running the command `perf stat -C 2 sleep 10` on the PPF prototype, which gathers event counts during 10 seconds:

```
Performance counter stats for 'CPU(s) 2':

      10001.032648      task-clock (msec)         #    1.000 CPUs utilized
                                                        (100.00%)
              3866      context-switches          #    0.387 K/sec
                                                        (100.00%)
                 0      cpu-migrations            #    0.000 K/sec
                                                        (100.00%)
                 0      page-faults               #    0.000 K/sec
       28910198243      cycles                    #    2.891 GHz
                                                        (100.00%)
   <not supported>      stalled-cycles-frontend
   <not supported>      stalled-cycles-backend
       50033467534      instructions              #    1.73  insns per cycle
                                                        (100.00%)
        7413056830      branches                  #  741.229 M/sec
                                                        (100.00%)
          14910328      branch-misses             #    0.20% of all branches

      10.000835628 seconds time elapsed
```

- **perf record** - This command gathers counter statistics like `perf stat`, but for later reports. Event counts are saved in a `perf.data` file which can be used to display information about system or program execution.
- **perf report** - The `perf report` command displays the data from a `perf.data` file. Depending on the options given to perf, the data can be displayed as for example a call graph of the functions called in the system or by an application which also shows the percentage of time spent executing each function. This information can be useful when searching for an application's bottlenecks.

#### 2.3.2.3 IxNetwork

Ixia's IxNetwork [34] is a traffic generator application capable of generating a variety of types of network traffic. This thesis uses IxNetwork to measure the throughput and latency of Ericsson's 5G RAN PPF prototype.

# Chapter 3

# Methodology

This chapter describes the methodology used to find an optimized implementation of Ericsson's 5G RAN PPF prototype, including the testbed setup, the parameters which were tweaked and the software that was used for the performance measurements.

## 3.1 Testbed Setup

This section provides an overview of the machines involved in the experiments and their connections, as well as the configuration used for the 5G RAN PPF prototype. For the purpose of the experiments in this thesis, the PPF prototype was configured to use a UDP relay which receives and forwards UDP packets with a four stage packet processing pipeline, instead of the more complicated software used in a real world deployment of the 5G RAN PPF. This was done to focus the optimization effort on the efficiency of the Eventdev and VM configurations rather than on the packet processing code. The testbed setup and test environment were provided by Ericsson. This both makes the results of these measurements more generically usable and makes them *unrepresetative* for the performance of the actual Ericsson 5G RAN PPF.

Figure 3.1 displays an overview of the testbed setup. The PPF prototype runs as a VM with a Wind River Linux 8 guest OS. In the testbed used in the experiments for this thesis, the QEMU-KVM hypervisor was used. Table 3.1 lists the specifications of the testbed.

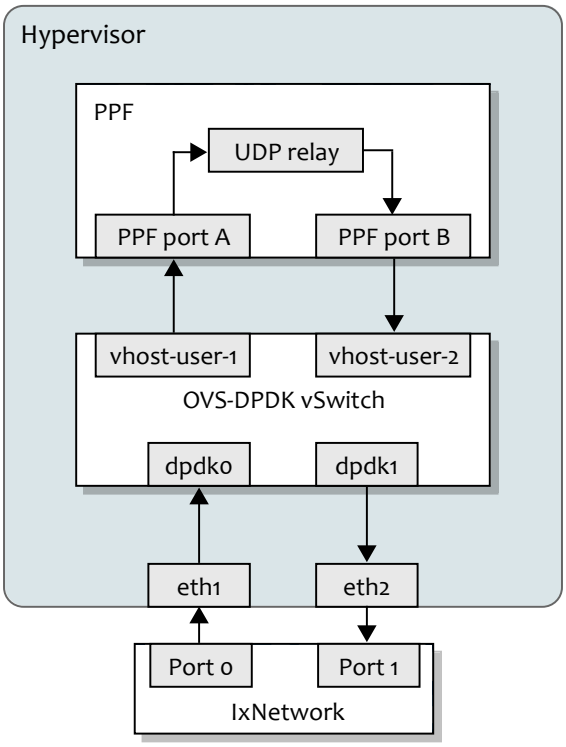



Figure 3.1: An overview of the testbed.

Table 3.1: The specifications of the testbed setup.

| **Testbed Specifications** | |
|---|---|
| **Host OS** | Kernel release: 4.4.0-89-generic,<br>Kernel version: #112-Ubuntu SMP Mon Jul 31 19:38:41 UTC 2017,<br>Hardware platform: x86_64,<br>OS: GNU/Linux |
| **Host CPU** | Intel(R)Xeon(R) CPU E5-2680 v3 @ 2899 MHz |
| **Host NICs** | Intel Corporation Ethernet 10G 2P X520 Adapters ($\times$ 2) |
| **Hypervisor** | QEMU emulator version 2.5.0 (Debian 1:2.5+dfsg-5ubuntu10.14) |
| **Guest OS** | Kernel release: 4.1.27-rt30-WR8.0.0.17_standard,<br>Kernel version: #1 SMP Sun Sep 10 07:08:22 CEST 2017,<br>Hardware platform: x86_64,<br>OS: GNU/Linux |
| **Physical Layer Switch** | Chassi: MRV Media Cross Connect, NC316-288PMCHS,<br>Blade: MRV Media Cross Connect, 10G SFP+ Blade |
| **OvS Version** | ovs-vswitchd (Open vSwitch) 2.7.0 |
| **DPDK Version** | 17.05 |

To prepare the PPF prototype for throughput and latency tests, the hypervisor and VM were optimized to process traffic with minimum packet drops. To reduce the number of packet drops caused by the host and guest OS, the following configuration was used:

- CPU core pinning using the *isolcpus* boot parameter. This reduces unwanted CPU migrations and context switches by isolating the selected CPU cores from the Linux kernel scheduler. By isolating the cores on which PPF threads are running, the kernel will not schedule other tasks unless CPU affinity system calls are explicitly made for this purpose. On the hypervisor, QEMU-KVM processes running the PPF/VM threads were pinned to the physical CPU cores 6, 8, 10, 12, 14, 16, 18, 20, 22 on socket 0. Core 0 was assigned the non-PMD threads of DPDK and QEMU-KVM/VM. The OvS bridge PMD threads (one per NIC) were pinned to cores 2 and 4 respectively.
- The *nohz_full* [35] boot parameter was set for the cores running QEMU-KVM/PPF and OvS threads. This parameter tells the kernel to omit scheduling-clock ticks for selected cores, which can reduce OS jitter by for example reducing the number of context switches. This parameter was set for cores 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, and 22 on the host machine. The VM guest kernel was compiled without this capability.
- The *rcu_nocbs* boot parameter was set for the cores running PPF threads so that they do not receive RCU callbacks, which would take time away from the processing of packets.
- Disabling CPU frequency throttling by the command: `cpupower frequency-set -g performance -d 2.9G -u 2.9G`, to avoid a period of decreased performance when the CPU has been in an idle state before it increases the CPU frequency.
- Disabling automatic NUMA balancing by the command: `echo 0 > /proc/sys/kernel/numa_balancing`. Automatic NUMA balancing optimizes the performance of an application running on a NUMA system by moving the application's tasks closer to the memory that it references. The PPF prototype is configured to run on a single NUMA socket, making NUMA balancing unnecessary. Relocation of a PPF thread would temporarily halt its processing of packets which could decrease performance and cause packet drops.
- Disabling HyperThreading by the command: `./set_ht 0`. Enabling HyperThreading may increase performance. However, for the purpose of reproducibility of the benchmarks in this thesis, all PPF prototype threads were run on physical cores.
- Setting `echo 0 > /sys/module/kvm/parameters/halt_poll_ns` to avoid halting idle virtual CPUs of the KVM guest, which would require to wake them up when new processing work is received. The process of waking threads up may negatively impact the latency performance of the PPF.

## 3.2 Throughput Measurements

This thesis evaluates the maximum throughput of the modified PPF prototype under different configurations. To measure the maximum throughput, IxNetwork was used to

perform a binary search for the greatest throughput at which no packets were dropped. IxNetwork was configured to send Ethernet II/VLAN/IPv4/UDP/GTP-u packets of size 90 as shown in Figure 3.2.

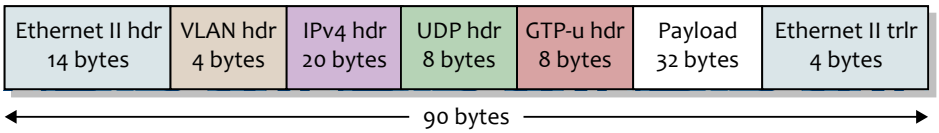



Figure 3.2: The structure of the Ethernet frames used to measure maximum throughput. Hdr and trlr are abbreviations of header and trailer respectively.

The binary search was performed by initially setting a lower bound frame rate of $750\,000$ frames/second (fps) for which the PPF prototype was known to cause zero packet drops, and a higher bound frame rate of $2\,000\,000$ fps - as this later rate was known to exceed the capacity of the PPF prototype. IxNetwork was then used to run at each frame rate for 10 seconds, sending a total of $10 \times t$ frames, $t$ being the frame rate, and moving the middle value up when no packets were dropped and down when packets *were* dropped. The precision of the maximum throughput measurements was set to 100 fps. The procedure is described by the pseudo code below. The high and low bounds were adjusted and verified between tests.

```
t_high = 2000000  // (or adjusted value)
t_low = 750000    // (or adjusted value)
// Verify 0 packet drops for t_low and >0 packet drops for t_high
while t_high - t_low > 100
t_current = (t_high + t_low) / 2
    // Send t_current fps for 10 seconds
    if num_packets_dropped == 0 then
        t_low = t_current
    else
        t_high = t_current
    end
end
return t_low
```

IxNetwork was configured with two traffic flows, one for each physical NIC in the testbed. Each NIC was configured to transmit packets to the PPF prototype which proceeded to process and forward the packets to the other NIC's Rx port. The flows were configured with the settings: Tx mode interleaved, src/dest mesh OneToOne, route mesh OneToOne, uni-directional.

## 3.3 Latency Measurements

The latency of the PPF prototype was measured using IxNetwork. IxNetwork was configured with one flow from the Tx port of one physical NIC of the testbed to the Rx port of the other NIC. The transmitting NIC was configured to transmit packets to the PPF prototype which proceeded to process and forward the packets to the receiving NIC's Rx port. The flow was configured with the settings: Tx mode interleaved, src/dest mesh OneToOne, route mesh OneToOne, uni-directional. The latency mode was set to cut-through, meaning the time from reception to transmission of the first bit of a frame. Ethernet frames were sent

using a custom IMIX configuration with frames of size 86 bytes with weight 7, 592 bytes with weight 4, and 1522 bytes with weight 1. The choice of 86 bytes as the size for small packets was made because this was the smallest frame size to accommodate internal headers. For the large frames, 1522 bytes was chosen as the size as this is the standard Maximum Transmission Unit (MTU) of an Ethernet II frame with VLAN tagging when not using jumbo frames. 592 bytes was chosen by Ericsson as a suitable size of medium frames. Initial tests revealed that the average cut-through latency reported generally stabilized after about 40 seconds, thus 40 second latency measurements were performed in subsequent testing. The throughput was set to $500\,000$ fps because early tests revealed this was a safe number that most configurations could handle.

## 3.4 Code Profiling

Code profiling with Linux Perf was used as a complement to the throughput and latency measurements. Linux Perf was used to look for slow functions, bad cache performance or other performance inhibiting behavior. The main commands used to perform profiling were the "perf record", "perf report", and "perf stat" commands. To gain knowledge about the execution time of different functions in the code, the following command was used: "perf record -C <core IDs> sleep 10". This command records the portions of the total execution time that different functions spend executing on the specified cores during 10 seconds. These statistics can then be viewed using "perf report".

To gain knowledge about cache statistics, the following command was used: "perf stat -C <core IDs> -e L1-dcache-loads,L1-dcache-load-misses,L1-dcache-stores sleep 10". This command reports the cache miss ratio on the specified cores during 10 seconds. This command can also be used with "stat" replaced by "record", which produces a record of where cache hits/misses occur.

## 3.5 Multicore Performance Scaling

To gain some insight into the parallelism of the PPF, multicore performance scaling tests were performed with different numbers of Rx/Tx cores and worker cores. Figure 3.3 shows the event path with the default core assignment with the maximum number of workers.

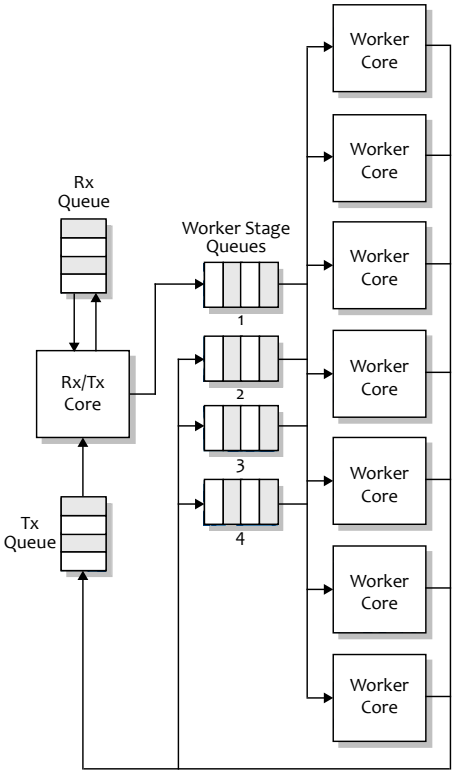



Figure 3.3: The default core assignment and path of events with 1 Rx/Tx core and 7 worker cores.

Each test included measurements of throughput and latency according to the methodology described in Sections 3.2 and 3.3. Tests seemed to reveal that no significant performance gain occurred when increasing the number of workers beyond 4 or 5, which could suggest a bottleneck in the delivery of packets from the Rx/Tx core to the worker cores. To investigate this, the PPF prototype was reprogrammed to support multiple Rx/Tx cores, with an option to reconfigure the core affinity at boot time by reading a parameter file. The throughput and latency scaling tests were then rerun with 2 Rx/Tx cores and up to 6 worker cores. Figure 3.4 shows the event path with the new assignment of cores.

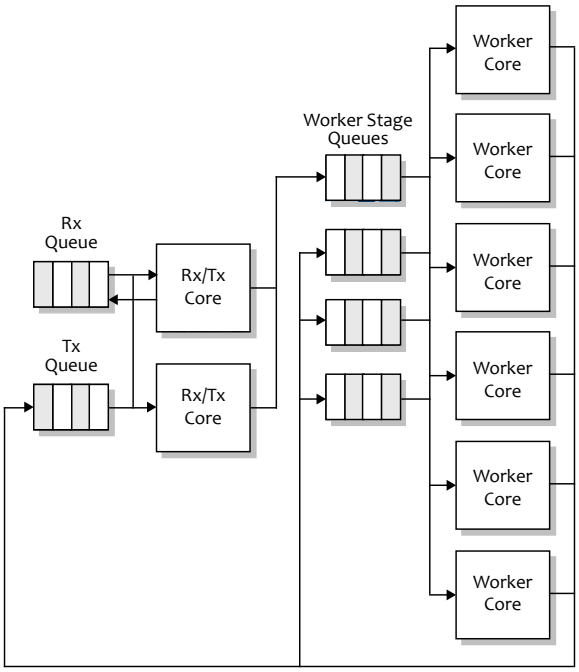



Figure 3.4: An alternative core assignment and path of events using 2 Rx/Tx cores and 6 worker cores.

## 3.6 Packet Processing Stage Consolidation

The Eventdev framework enables the programmer to perform packet processing in different stages. The default configuration of the PPF prototype has workers performing packet processing in 4 stages (illustrated in figure 3.5):

1. A classification stage where events are classified depending on their type and sent on to the next event queue for further processing.
2. A dummy stage which simulates time consumption from packet processing by actively consuming the specified number of instruction cycles. This stage consumes a configurable number of cycles per packet event and forwards the events to the next stage's event queue. For the baseline tests, this stage was configured to consume 0 cycles.
3. Another dummy stage like stage 2. This stage was set to consume 0 cycles for baseline tests.
4. The last worker stage consumes instruction cycles (set to 0 for baseline tests) in the same way as the previous two stages, and continues to process the packets and send them to the Tx event queue.

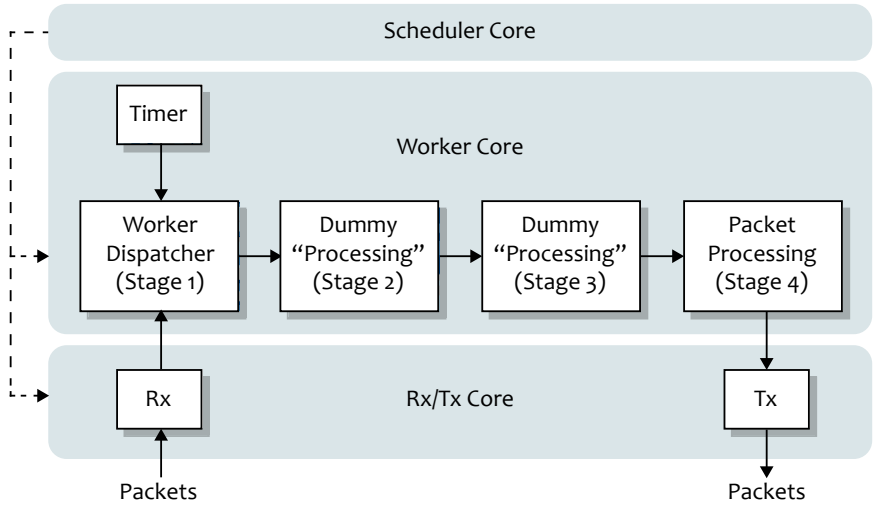



Figure 3.5: The path taken by packet events. The Rx/Tx core receives packets and forwards events into the eventdev. The worker processes packet (and timer) events through its four stages and sends them back to the Rx/Tx core through the event queues. The Rx/Tx core takes Tx events and sends out packets. The scheduler core schedules events for the different cores and decides their processing order.

One of the reasons for performing the packet processing in multiple stages is to present the Eventdev framework with smaller units of work which can be balanced among the worker cores. However, when the processing requires events to maintain their original order between stages, this introduces overhead from a larger number of enqueue and dequeue operations. More stages also introduces a degree of parallelism overhead due to locking mechanisms resulting from the atomic scheduling of events. For this reason, it makes sense to test a configuration with consolidated processing stages which classify packets early and processes them to completion. This approach should reduce the amount of overhead and potentially improve the cache hit ratio by keeping the events in the CPUs' L1 caches for longer. These potential improvements are at the cost of potentially having the workloads unevenly balanced. To test this approach, the PPF prototype was reprogrammed with a reduced number of worker stages, as shown in figure 3.6:

1. The same classification stage as before.
2. A consolidation of stages 2, 3, and 4, where packets need only be dequeued once and enqueued once, and no synchronization between events of the same flow is needed. The same total amount of cycles were consumed as in the default configuration.

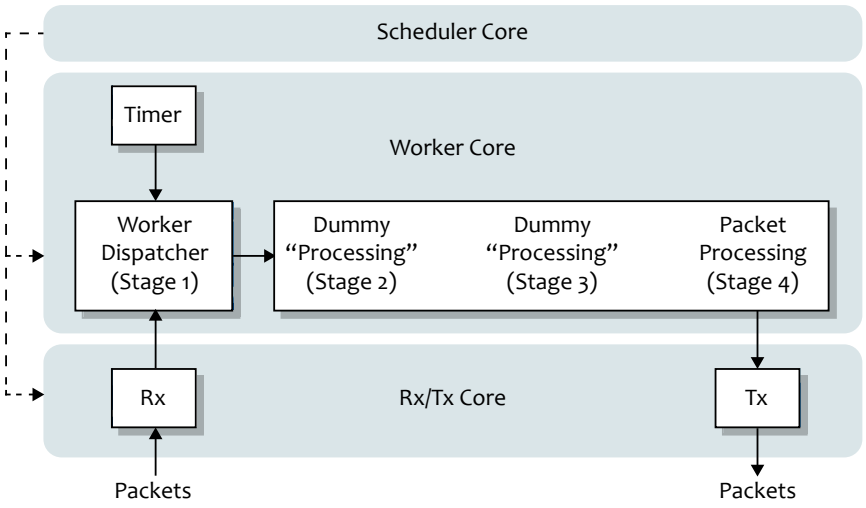



Figure 3.6: The event path with consolidated worker stages. The classification of packets remains in its own stage while the 3 subsequent stages have been merged into a single stage.

To test its throughput and latency, the consolidated configuration was then included in the multicore performance scaling tests described in Section 3.5.

## 3.7 Rx Optimization - RSS Patch

The Rx driver of the PPF prototype makes use of DPDK's RSS functionality, as described in Section 2.2.3. The RSS hash value is computed using the Toeplitz hash function, provided by DPDK's hash library [30]. When profiling the PPF prototype using Linux Perf on the Rx/Tx core after it was discovered that Rx seemed to be a performance bottleneck, it was revealed that a substantial portion of the execution time was spent performing RSS hash computations. When sending packets with a flow setup as described in Section 3.2 but with both flows reconfigured to send continuous traffic at $1\,200\,000$ fps which was close to the maximum throughput achieved for the default PPF prototype configuration using 2 Rx/Tx cores, Linux Perf reported that the portion of time spent computing the RSS hash function was 29% on the Rx/Tx core. The computation loops through each bit of a 32-bit integer, checks if each bit is set, and performs further operations if the bit is set. This is unnecessarily inefficient, since there are alternatives which avoid looping through all the unset bits. An example of this was suggested in a patch by Zhou Yangchao [36]. This patch uses the `rte_bsf32` function, which is an inline function calling GCC's built-in function `__builtin_ctz()` [37]. The `__builtin_ctz()` takes an unsigned integer argument and returns the number of trailing zeros starting with the least significant bit. To mitigate the effects of the RSS computation, Yangchao's methodology was adopted for the PPF prototype's Rx driver. The results using this patch are displayed in Section 4.3.

## 3.8 Optimization Workflow

The optimization workflow used in the experiments can be summarized as follows:

1. Configure the PPF prototype to have only 1 worker core.
2. Perform throughput and latency measurements. Linux Perf was used to gather information about the time consumed by different function calls made by the PPF prototype, as well as overall system statistics including the cache hit ratio and more.
3. Increase number of worker cores.
4. Repeat from step 2 until no further CPU cores are available.
5. After running all measurements with 1 Rx/Tx core - repeat from step 1 using 2 Rx/Tx cores.
6. Optimize code and repeat all measurements.
7. Compilation, comparison, and analysis of the results, as seen in Sections 4, 5, and 6.

The basic idea of the experiments described in this thesis was to isolate the parameters of the PPF which affected the throughput and/or latency the most, and adjusting the values using IxNetwork and Linux Perf as measurement and profiling tools. IxNetwork was used to generate network traffic and to measure throughput and latency while Linux Perf was used for deeper performance analysis of for example cache statistics, Instructions Per Cycle (IPC) statistics, and context switching.

# Chapter 4

# Results

This chapter summarizes the results of the throughput and latency measurements for the tested PPF prototype configurations.

## 4.1 Core Reassignment Results

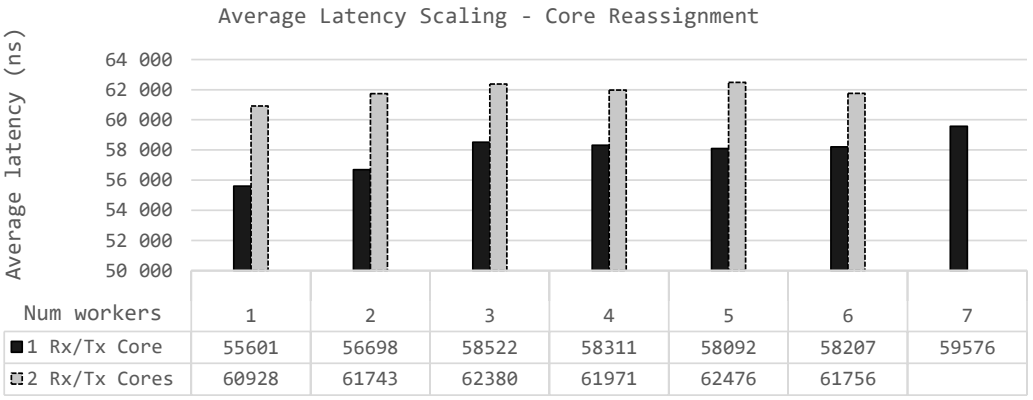
The PPF prototype has access to 10 CPU cores, of which a minimum of 3 are used for the DPDK master core, scheduling, and Rx/Tx. The remaining cores can be used to run worker threads. This section summarizes the scaling of throughput for different assignments of the CPU cores using an otherwise unchanged configuration of the PPF prototype. Throughput was measured according to the methodology described in Section 3.2. Figure 4.1 shows the maximum throughput of the PPF prototype with one worker core reassigned to Rx/Tx. No other changes were made to the PPF configuration. The throughput was lower with 2 Rx/Tx cores for all numbers of worker cores.


| Num workers | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 Rx/Tx Core | 616724 | 1033289 | 1289624 | 1586975 | 1591461 | 1590767 | 1583984 |
| 2 Rx/Tx Cores | 573364 | 991058 | 1237061 | 1273224 | 1291351 | 1278412 | |

Figure 4.1: Maximum throughput results with reassigned CPU cores: 1 Rx/Tx cores and up to 7 worker cores; and 2 Rx/Tx cores and up to 6 worker cores.

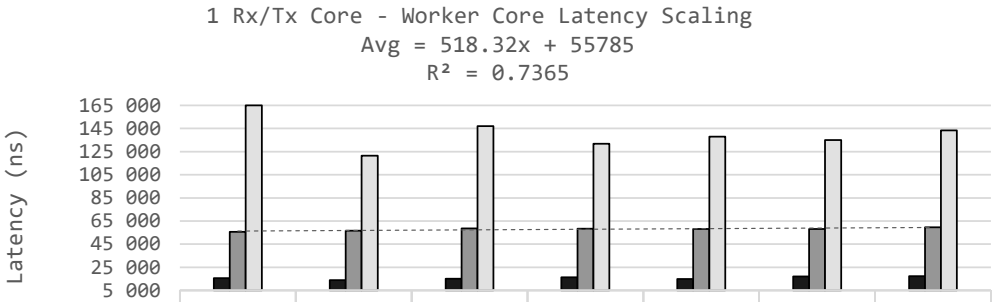
Figures 4.2, 4.3, and 4.4 display the latency results of the core reassignments. The results show that the average latency increased for each respective number of worker cores when

assigning an extra Rx/Tx core. The lowest average latency was seen with only a single worker core with both 1 and 2 Rx/Tx cores.



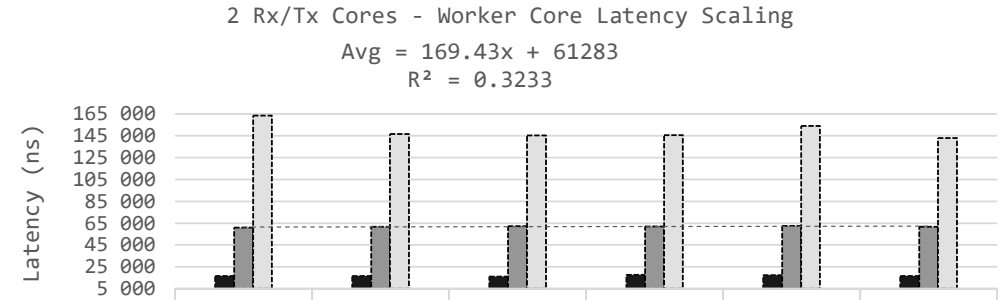

| Num workers | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 Rx/Tx Core | 55601 | 56698 | 58522 | 58311 | 58092 | 58207 | 59576 |
| 2 Rx/Tx Cores | 60928 | 61743 | 62380 | 61971 | 62476 | 61756 | |

Figure 4.2: Comparison of the average latency with reassigned CPU cores: 1 Rx/Tx cores and up to 7 worker cores, to 2 Rx/Tx cores and up to 6 worker cores.




| Num workers | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| Min | 15660 | 13920 | 15100 | 16220 | 14820 | 16960 | 17220 |
| Avg | 55601 | 56698 | 58522 | 58311 | 58092 | 58207 | 59576 |
| Max | 165080 | 121480 | 146960 | 131920 | 137920 | 135000 | 143440 |

Figure 4.3: The minimum, average and maximum average cut-through latency at $500\,000$ pps using 1 Rx/Tx cores and up to 7 worker cores. The dashed line is a curve fit of the average latency data, with its function and $R^2$ value displayed above chart area.

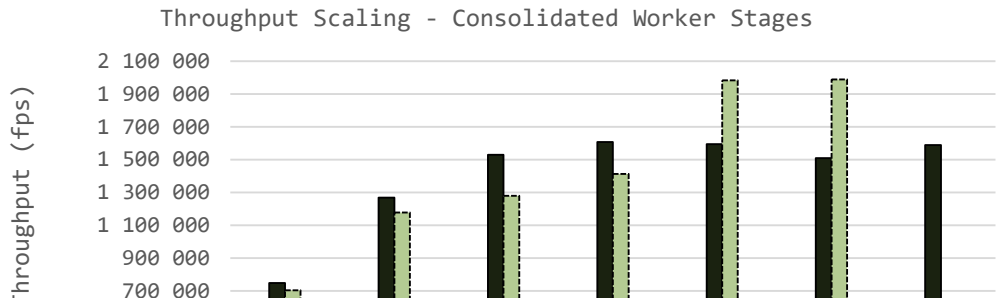



| Num workers | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Min | 16740 | 16700 | 16240 | 17600 | 17520 | 16540 |
| Avg | 60928 | 61743 | 62380 | 61971 | 62476 | 61756 |
| Max | 163560 | 146600 | 145260 | 145680 | 154080 | 143040 |

Figure 4.4: The minimum, average and maximum average cut-through latency at $500\,000$ pps using 2 Rx/Tx cores and up to 6 worker cores. The dashed line is a curve fit of the average latency data, with its function and $R^2$ value displayed above chart area.

## 4.2 Worker Stage Consolidation Results

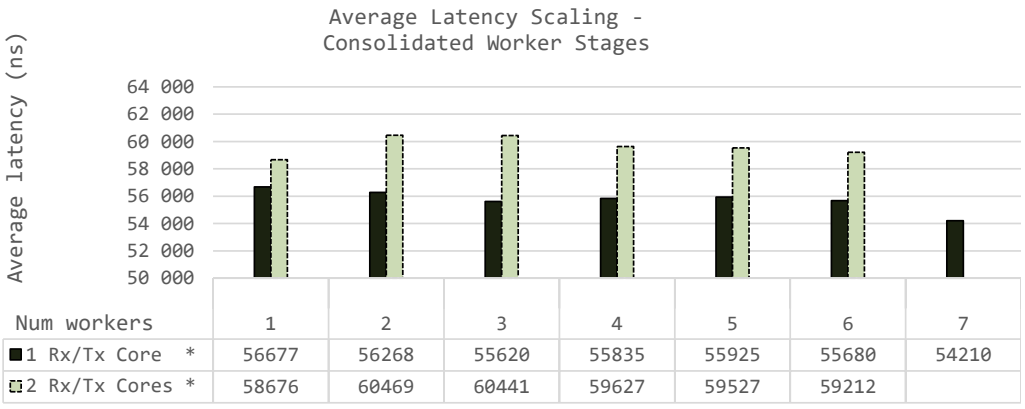
This section displays the results of the tests with consolidated worker stages. Throughput and latency was measured using 1 Rx/Tx core and up to 7 worker cores, and 2 Rx/Tx cores and up to 6 worker cores. The highest throughput was achieved using 2 Rx/Tx cores and 6 worker cores. The maximum throughput using consolidated worker stages was higher than the maximum throughput using the default packet processing stages, at 1.99 Mpps versus the the default maximum throughput of 1.59 Mpps.




| Num workers | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 Rx/Tx Core * | 748413 | 1268652 | 1530566 | 1608203 | 1595117 | 1509180 | 1589453 |
| 2 Rx/Tx Cores * | 704736 | 1179016 | 1279358 | 1412366 | 1982635 | 1989777 | |

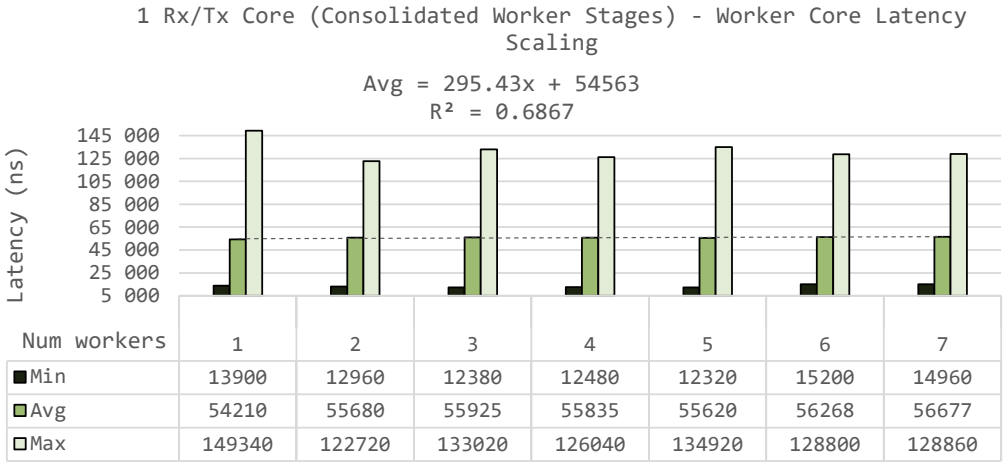
Figure 4.5: Maximum throughput results with consolidated worker stages. Throughput was measured with 1 and 2 Rx/Tx cores and 7 and 6 worker cores respectively.

Figures 4.6, 4.7, and 4.8 show the results of the latency measurements. The average cut-through latency increased for all respective numbers of worker cores when using 2 Rx/Tx cores instead of 1.



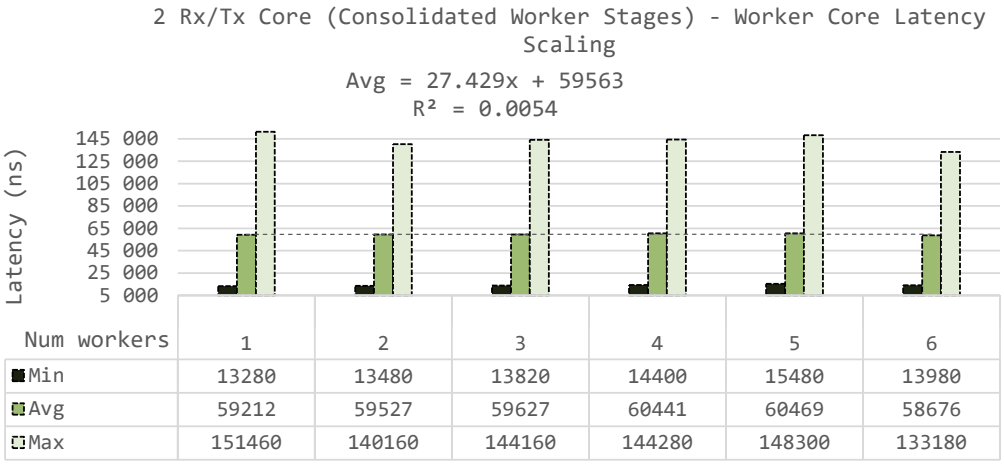

| Num workers | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 Rx/Tx Core * | 56677 | 56268 | 55620 | 55835 | 55925 | 55680 | 54210 |
| 2 Rx/Tx Cores * | 58676 | 60469 | 60441 | 59627 | 59527 | 59212 | |

Figure 4.6: Comparison of the average latency with consolidated worker stages and reassigned CPU cores: 1 Rx/Tx cores and up to 7 worker cores, to 2 Rx/Tx cores and up to 6 worker cores.




| Num workers | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| Min | 13900 | 12960 | 12380 | 12480 | 12320 | 15200 | 14960 |
| Avg | 54210 | 55680 | 55925 | 55835 | 55620 | 56268 | 56677 |
| Max | 149340 | 122720 | 133020 | 126040 | 134920 | 128800 | 128860 |

Figure 4.7: The minimum, average and maximum average cut-through latency at $500\,000$ pps when using 1 Rx/Tx core, up to 7 worker cores and consolidated worker stages. The dashed line is a curve fit of the average latency data, with its function and $R^2$ value displayed above chart area.

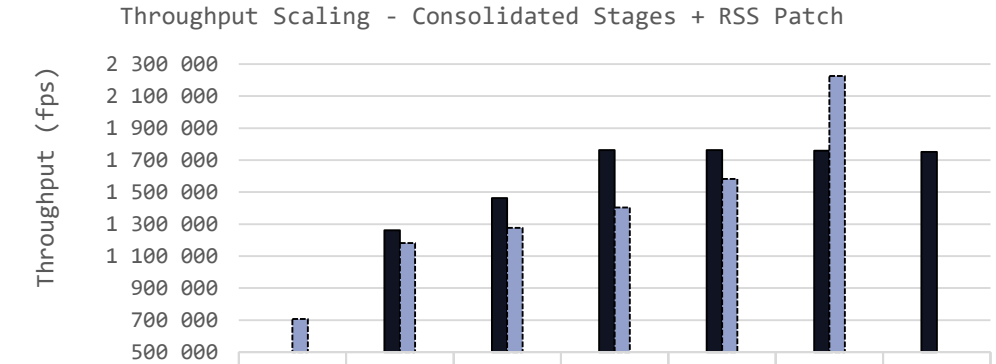



| Num workers | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Min | 13280 | 13480 | 13820 | 14400 | 15480 | 13980 |
| Avg | 59212 | 59527 | 59627 | 60441 | 60469 | 58676 |
| Max | 151460 | 140160 | 144160 | 144280 | 148300 | 133180 |

Figure 4.8: The minimum, average and maximum average cut-through latency at $500\,000$ pps when using 2 Rx/Tx core, up to 6 worker cores and consolidated worker stages. The dashed line is a curve fit of the average latency data, with its function and $R^2$ value displayed above chart area.

## 4.3 RSS Patch Results

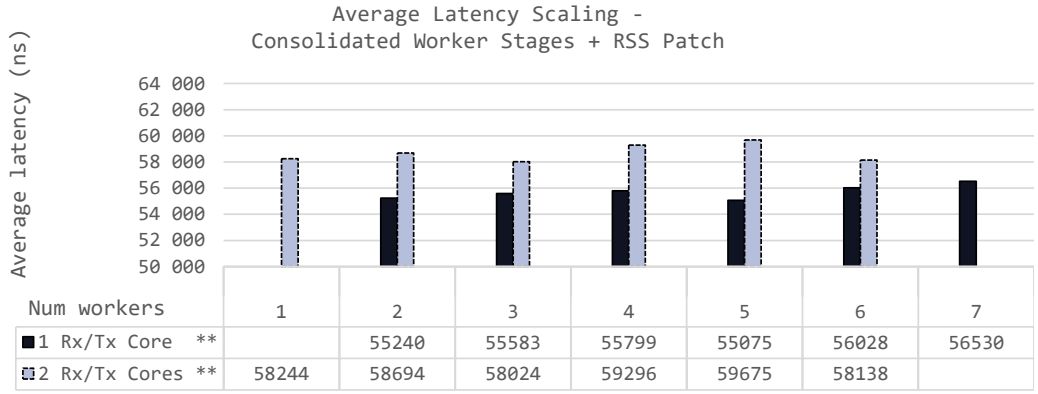
This section displays the results of the tests with consolidated worker stages and the RSS patch applied. Throughput and latency was measured using 2 Rx/Tx core and up to 7 worker cores, and 2 Rx/Tx cores and up to 6 worker cores. The highest maximum throughput with this configuration was achieved using 2 Rx/Tx cores and 6 worker cores, as shown in Figure 4.9. The case with 1 Rx/Tx core and 1 worker core was excluded due to tests failing with strange results.




| Num workers | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 Rx/Tx Core ** | | 1261597 | 1462793 | 1762598 | 1763281 | 1757935 | 1750366 |
| 2 Rx/Tx Cores ** | 707275 | 1182263 | 1276114 | 1403381 | 1582275 | 2225891 | |

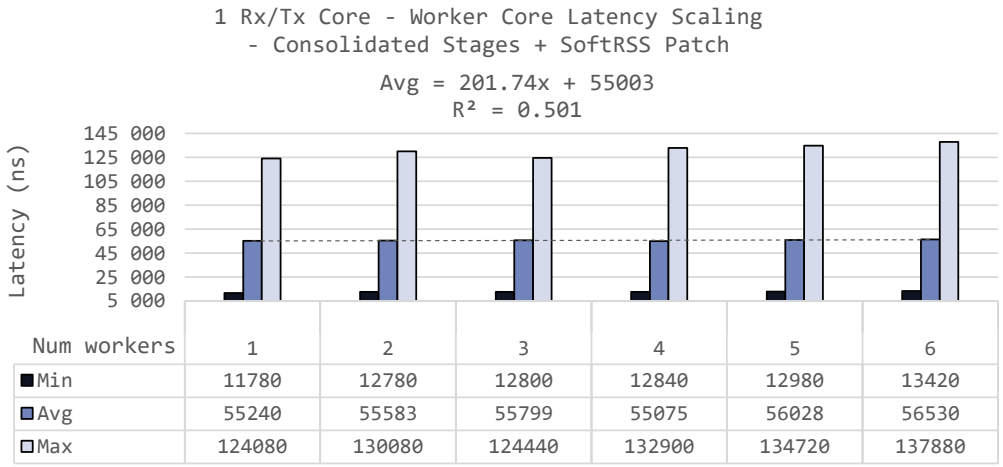
Figure 4.9: Maximum throughput results with consolidated worker stages and the RSS patch applied. Throughput was measured with 1 and 2 Rx/Tx cores and 7 and 6 worker cores respectively.

Figures 4.10, 4.11, and 4.12 display the latency results for the PPF configuration with consolidated worker stages and the RSS patch applied. Reassigning a worker core to Rx/Tx increased the latency for all configurations.



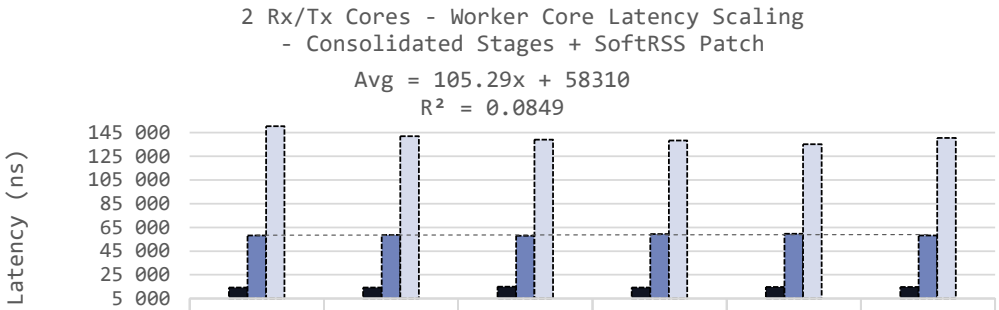

| Num workers | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 Rx/Tx Core ** | | 55240 | 55583 | 55799 | 55075 | 56028 | 56530 |
| 2 Rx/Tx Cores ** | 58244 | 58694 | 58024 | 59296 | 59675 | 58138 | |

Figure 4.10: Comparison of the average latency with consolidated worker stages, the RSS patch applied and reassigned CPU cores: 1 Rx/Tx cores and up to 7 worker cores, to 2 Rx/Tx cores and up to 6 worker cores.




| Num workers | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Min | 11780 | 12780 | 12800 | 12840 | 12980 | 13420 |
| Avg | 55240 | 55583 | 55799 | 55075 | 56028 | 56530 |
| Max | 124080 | 130080 | 124440 | 132900 | 134720 | 137880 |

Figure 4.11: The minimum, average and maximum average cut-through latency at $500\,000$ pps when using 1 Rx/Tx core, up to 7 worker cores, consolidated worker stages, and the RSS patch. The dashed line is a curve fit of the average latency data, with its function and $R^2$ value displayed above chart area.

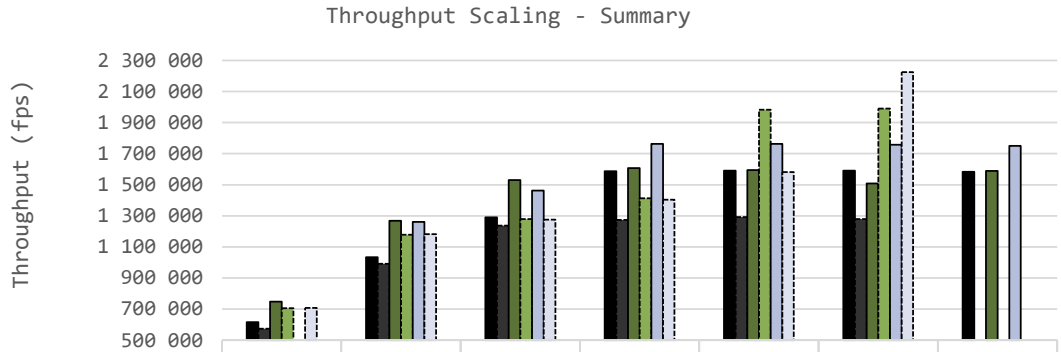



| Num workers | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Min | 14280 | 14240 | 15020 | 14360 | 14700 | 14700 |
| Avg | 58244 | 58694 | 58024 | 59296 | 59675 | 58138 |
| Max | 150460 | 142000 | 139000 | 138280 | 135200 | 140380 |

Figure 4.12: The minimum, average and maximum average cut-through latency at $500\,000$ pps when using 2 Rx/Tx core, up to 6 worker cores, consolidated worker stages, and the RSS patch. The dashed line is a curve fit of the average latency data, with its function and $R^2$ value displayed above chart area.

## 4.4 Summarized Results

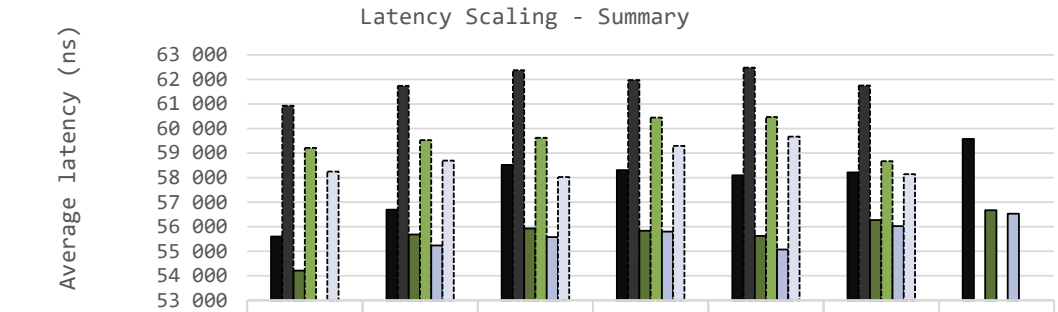
Figure 4.13 shows the throughput achieved when using one scheduling core, 1 Rx/Tx core and 2 Rx/Tx cores, and an increasing number of worker cores up to the maximum of 7 and 6 cores respectively. The highest maximum throughput (2.23 Mpps) was achieved using 2 Rx/Tx cores, 6 worker cores, consolidated worker stages, and the RSS patch applied. These reconfigurations improved the throughput performance by 40.52% compared to the default PPF configuration with 1 Rx/Tx core and 7 worker cores, which achieved a throughput of 1.58 Mpps.




| Num workers | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 Rx/Tx Core | 616724 | 1033289 | 1289624 | 1586975 | 1591461 | 1590767 | 1583984 |
| 2 Rx/Tx Cores | 573364 | 991058 | 1237061 | 1273224 | 1291351 | 1278412 | |
| 1 Rx/Tx Core * | 748413 | 1268652 | 1530566 | 1608203 | 1595117 | 1509180 | 1589453 |
| 2 Rx/Tx Cores * | 704736 | 1179016 | 1279358 | 1412366 | 1982635 | 1989777 | |
| 1 Rx/Tx Core ** | | 1261597 | 1462793 | 1762598 | 1763281 | 1757935 | 1750366 |
| 2 Rx/Tx Cores ** | 707275 | 1182263 | 1276114 | 1403381 | 1582275 | 2225891 | |

Figure 4.13: The throughput performance scaling with increasing number of workers. * denotes the PPF prototype with consolidated worker stages, and ** denotes the PPF with consolidated worker stages and RSS patch applied. The test case failed for 1 Rx/Tx Core **, and the result was excluded.

Figure 4.14 shows a summary of the average cut-through latency results. The lowest observed average latency of 54 210 ns was achieved using the PPF configuration with consolidated worker stages, 1 Rx/Tx cores and 1 worker core. The average cut-through latency reported for the configuration with the highest observed throughput of 2.2 Mpps was 58 138 ns, which is 1 438 ns lower than that of the completely default PPF configuration.




| Num workers | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 Rx/Tx core | 55601 | 56698 | 58522 | 58311 | 58092 | 58207 | 59576 |
| 2 Rx/Tx cores | 60928 | 61743 | 62380 | 61971 | 62476 | 61756 | |
| 1 Rx/Tx core * | 54210 | 55680 | 55925 | 55835 | 55620 | 56268 | 56677 |
| 2 Rx/Tx Cores * | 59212 | 59527 | 59627 | 60441 | 60469 | 58676 | |
| 1 Rx/Tx Core ** | | 55240 | 55583 | 55799 | 55075 | 56028 | 56530 |
| 2 Rx/Tx Cores ** | 58244 | 58694 | 58024 | 59296 | 59675 | 58138 | |

Figure 4.14: The average cut-through latency performance scaling with increasing number of workers. * denotes the PPF prototype with consolidated worker stages, and ** denotes the PPF prototype with consolidated worker stages and RSS patch applied. The test case failed for 1 Rx/Tx Core **, and the result was excluded.

# Chapter 5

# Discussion

## 5.1 Results Analysis

During the beginning of this thesis project, cache optimization and context switching was assumed to play an important role in the optimization of the PPF. When profiling the default PPF using Linux Perf, a miss ratio of 7.08% among the L1 data cache loads was reported on the Rx/Tx core, and 3.55% on the worker cores. However, when looking at the cause of the cache misses, it was discovered that a majority were an intended result of prefetching, and some due to locking mechanisms. Linux Perf reported 387 CS/second on the Rx/Tx core and 336 CS/second on the worker cores. The reconfigurations presented in this thesis did not have a significant effect on the cache miss ratio or number of context switches. DPDK takes several measures to optimize cache utilization and minimize context switching. For this reason, focus was shifted from these aspects towards the multicore performance scaling of the PPF.

By adding support for multiple Rx/Tx cores in the PPF prototype, and measuring the maximum, minimum, and average throughput, and maximum cut-through latency using different core assignments, it was discovered that a plateau in throughput was reached when using only 4 or 5 of the available 7 worker cores. This plateau was assumed to be caused by a bottleneck in the Rx or Tx drivers or the scheduler. However, when adding an additional Rx/Tx core to the default PPF configuration, at the cost of one worker core, both the throughput and the latency performance got worse. The maximum throughput dropped by 23.9% when comparing the configuration with 1 Rx/Tx core and 7 worker cores to the configuration with 2 Rx/Tx cores and 6 worker cores, and the latency increased by 3.66%. One of the reasons for this decrease in throughput seems to be that the Eventdev cannot schedule the Rx/Tx driver completely in parallel as it is not thread-safe. This leads to decreased efficiency in the Rx driver due to synchronization overhead. Linux Perf reported that 24.49% of the cycles on the Rx/Tx cores were spent on atomic scheduling, when using 2 Rx/Tx cores instead of 1. When using the `stat` command of Linux Perf on the Rx/Tx cores while letting the PPF prototype process packets at close to the maximum throughput for two different configurations of the PPF prototype, the output numbers shown in Table 5.1 were produced. This output shows that the IPC went from 1.73 on the single Rx/Tx core to 0.98 over the 2 Rx/Tx cores: roughly a 77% drop. The number of CS per second also dropped by

78%, from 387 to 217 CS per second. Additionally, the number of branches taken dropped by 64% while the number of branch misses almost doubled, from 0.20% to 0.38%. While the reason for the increase in branch mispredictions could have been investigated also using Perf, this investigation had to be left out due to limited time with the testbed equipment.

Table 5.1: Statistics reported by Linux Perf when processing packets at close to the maximum throughput of two different configurations of the PPF prototype: 1.5 Mpps using 1 Rx/Tx core and 7 workers, and 1.2 Mpps using 2 Rx/Tx cores and 6 workers. The perf stat command was executed on all Rx/Tx cores in both cases.

| | **1 Rx/Tx core, 7 workers** | **2 Rx/Tx Cores, 6 workers** |
|---|---|---|
| **CPU core utilization** | 1.000 | 2.000 |
| **Number of CS** | 387 /sec | 217 /sec |
| **CPU migrations** | 0 | 0 |
| **Page faults** | 0 | 0 |
| **CPU clock frequency** | 2.891 | 2.891 |
| **Instructions/cycle** | 1.73 | 0.98 |
| **Branches** | 741.229 M/sec | 453.175 M/sec |
| **Branch misprediction %** | 0.20% | 0.38% |

However, when consolidating the last three worker stages, making them a single unit of schedulable work, the throughput increased for all numbers of workers when using 1 Rx/Tx core, indicating a speedup in the worker cores. When reassigning one worker core to Rx/Tx, the throughput increased with 5 and 6 worker cores. This indicates a maximum throughput of around 1.6 Mpps for a single Rx/Tx core, without the RSS patch. To verify that Rx/Tx was indeed a throughput bottleneck, a 1000 cycle delay per packet was added to the fourth worker stage, while processing at 1.99 Mpps. Linux Perf reported that this increased the cycles consumed by the fourth worker stage from 6.08% to 26.92% of the total cycles on the worker cores, while the additional delay had no apparent impact on the throughput.

With support for the belief that Rx/Tx was the bottleneck, the Rx/Tx core was inspected. When profiling the Rx driver at a throughput of 1.2 Mpps for the default configuration, it was reported that at least 29% of the instruction cycles were consumed by the RSS function of the Rx core. Looking for to speedup the RSS code, the patch described in Section 3.7 was found. When measuring the maximum throughput using consolidated worker stages and the RSS patch, a throughput of 2.2 Mpps was achieved with 2 Rx/Tx cores and 6 worker cores. This is a 40.52% increase in throughput from the default configuration. When profiling the Rx code again at 1.2 Mpps, the portion of instruction cycles consumed by RSS computations had dropped to 18.6%.

The average latency results varied for the different PPF configurations. Assigning an extra Rx/Tx core seemed to increase the average latency for all of the tested configurations. This increase in latency seemed to be a result of the synchronization overhead introduced when running the Rx driver in parallel on two different cores. This overhead was reported to be 24.49% on the dual Rx/Tx cores. However, the average latency increased by at most 5 327 ns, corresponding to 9.58%, of the average latency for the case with 1 worker core and none of the optimizations applied, when going from 1 to 2 Rx/Tx cores. The minimum

recorded average latency recorded was 54 210 ns for the configuration with consolidated worker stages, 1 Rx/Tx core and 1 worker core. However, this configuration achieved the relatively low maximum throughput of 0.75 Mpps, compared to the greatest observed throughput of 2.2 Mpps. For the configuration with the highest throughput, the average latency was 97.59% of the average latency for the completely unmodified PPF prototype with 1 Rx/Tx core and 7 worker cores. Both optimizations, i.e. consolidating the last three worker stages and applying the RSS patch, seemed to reduce the average latency with all core configurations. Most of the line fits in each average latency scaling chart (Figures 4.3, 4.4, 4.7 4.8, 4.11, 4.12) seem to suggest that the average latency increases with more worker cores, although with a fairly large degree of uncertainty. Most of the latency seems to come from the constant term of each respective function.

## 5.2 Methodology Discussion

IxNetwork was used to find the maximum throughput and minimum, average, and maximum cut-through latency for different configurations of the PPF prototype. This tool worked well when measuring the maximum throughput. However, it would have been desirable to also measure the median cut-through latency of the PPF prototype, as well as latency jitter and distribution. The latency measurements could have been performed in another way, perhaps by gathering raw latency measurement data and doing further analysis outside of the measurement tool. The main reason for using IxNetwork was that it was easy to use for traffic generation and measurements by the same tool, and that Ericsson's test equipment was set up this way. The latter was difficult to change, and although it may have been possible to use another tool, that would have required much more effort and time than using IxNetwork.

Another improvement that could have been made to the methodology would have been to perform more consistent profiling with Linux Perf to find the reason for the behavior of the PPF prototype under different configurations. Due to unavailability of the testbed for much of the duration of the project, the time window for testing became very limited. For this reason, the focus was set on completing the throughput and latency scaling measurements with different core assignments, with Perf as a complement where analysis was needed. In the end, this may not have been enough to gain all of the valuable insight that was potentially offered. Systematic apples-to-apples tests using Linux Perf should have been made for each of the configurations of the PPF prototype. This could have provided more information about the exact details of the atomic scheduling overhead and reason for the behavior of the minimum/average/maximum latency as the number of Rx/Tx and worker cores differed.

# Chapter 6

# Conclusions and Future Work

This section states the conclusions made from the results and analysis presented in Sections 4 and 5. Also included in this section are required reflections regarding the ethics and sustainability aspects of this thesis project, and suggestions for future work related to the 5G RAN PPF and its prototype.

## 6.1 Conclusions

The problem statement of this thesis project was: *"Given an NFV PPF prototype for the 5G RAN based on the DPDK Eventdev framework, what are the bottlenecks of its packet processing path, and how can their effects be mitigated?"* The throughput of the PPF prototype in a 2 flow traffic setup was raised by 40.52%, with a latency of 97.59% compared to the default PPF prototype configuration. This was done by reassigning one worker core to perform Rx/Tx, consolidating three of the packet processing stages into a single unit of schedulable work, and optimizing DPDK's software RSS function. An intermediate result suggests that the maximum throughput of an unmodified Rx/Tx core of the PPF prototype is close to 1.6 Mpps. One of the points of having multiple packet processing stages when using DPDK's Eventdev library is to have smaller units of schedulable work, which can be evenly balanced among CPU cores. However, the PPF prototype seemed to benefit from fewer but longer processing stages to outweigh the negative impact that this change could have on load balancing.

## 6.2 Limitations

This thesis project was somewhat limited by the fact that all measurements were performed on a prototype of the 5G RAN PPF, which meant that the full processing path was not included in the tests. For example, the consolidation of packet processing stages may have a different meaning in a real-world application than they had in the prototype tests. Additionally, the tests had to be shaped around Ericsson's testbed setups and what tools were available to use for this setup. Gaining access to the testbed was non-trivial as it had to be shared with Ericsson employees during many periods of the project, which made

continuous testing difficult. This dragged out the duration of the tests and measurements which somewhat lowered the quality of the experiments, since only the most important measurements could be done by the end of the project. For example, a more in-depth analysis of the PPF prototype using Linux Perf could have been made, as well as measurements of the median latency and latency jitter and distribution, if there was more time and easier access to the testbed.

## 6.3 Required Reflections

Optimizing the PPF for throughput may lead to a higher energy efficiency, which is important to the economic and environmental aspects of 5G RAN. With a higher throughput, the amount of CPU cores required by each PPF in the 5G RAN may be reduced. Also, less PPF nodes may be required if a single node can handle the traffic that would otherwise require multiple nodes. Another consideration of the environmental aspect of the PPF prototype is that it utilized 100% of all of its assigned CPU cores, even when it is not processing any traffic. This may keep the PPF ready for incoming packets at all times, at the cost of the energy required to run the CPU at full speed all the time. Perhaps it could be worth some extra initial latency to be able to put the PPF in an idle state when it is not processing traffic, or to scale its performance according to the amount of traffic that requires processing.

## 6.4 Future Work

This thesis answers what some of the bottlenecks were and how to mitigate their effects mainly on the throughput of the PPF prototype, as described by the problem statement, but fails in many cases to report the reasons *why*. A more systematic code profiling analysis could have helped explain the effects seen in the results. Linux Perf was a useful tool, and in future studies it should be utilized more than it was in this thesis project.

A few topics to investigate further emerged during this thesis project, which are summed up below:

- An interesting and desirable result of NFV is the potential portability of network functions between physical machines. For this reason, it could be interesting to test how the PPF performs with different host machines, host OS, and hypervisors.
- This thesis project used a UDP relay to simulate a packet processing pipeline. A complete optimization of the PPF should be performed on an instance with real world software instead of the placeholder code used for testing in this thesis project.
- The PPF runs its software with 100% CPU utilization at all times. If this is to keep the cache hot and to keep the PPF ready for incoming packets, this may lead to increased performance. However, if there is no traffic being processed, this is inefficient as the resources are left unused. To leave room for other VNFs as well as for environmental reasons related to energy consumption, perhaps some allocated resources could be released when traffic is low at the cost of higher latency in the early phase of transmission.

- An interesting optimization to try would be to compile the VM guest OS kernel with NO_HZ=full capability, as this was a pre-test optimization used on the host machine to reduce OS jitter. Having a guest OS kernel with this capability may further reduce OS jitter.
- The measurements in this thesis are based on Ericsson's pre-existing test cases. The throughput tests only set up 2 traffic flows, and the latency test only set up 1 IMIX traffic flow. It could be interesting to evaluate the PPF with a greater variety of network flows, and with different payloads and packet sizes.
- Software RSS proved to be an expensive functionality of the Rx driver. Experiments could be made with RSS offloading. The NICs for the experiments in this thesis project have RSS hardware offloading capability, yet RSS is performed in software. One of the reasons for running the PPF as a VNF is to be able to run it on different types of hardware. However, if RSS offloading capability is detected, this could likely boost the PPF's performance.

Additionally, the DPDK libraries used by the PPF have a number of configurable parameters which were originally intended as a part of the PPF optimization in this project. These parameters could affect both throughput and latency. Some of these parameters are:

- **Enqueue and dequeue depths of Eventdev event ports** - The event device uses event queues to contain events of different types and to allow the scheduler to dynamically distribute the load when processing events. The event queues are linked to event ports which can be set to enqueue and dequeue events in bursts of configurable sizes. These burst sizes are called the enqueue and dequeue depths. Tuning these parameters to find the right balance can affect the throughput and latency when processing events: increasing the burst size may increase both throughput and latency and may also affect cache performance.
- **New event threshold on Eventdev event ports** - This parameter is used to protect the event device from being overwhelmed when the system is under heavy load. It limits the amount of new events that are enqueued in the event queues. If the new event threshold is set too low, the event device will be underutilized due to starvation from backpressure at the ingress, and if set too high the event device may be flooded with new events which can lead to increased latency.
- **Eventdev scheduling quanta** - The number of events scheduled at each scheduling function call. This parameter serves as a hint to the scheduler, which may not adhere to the configured quanta.

# Bibliography


[1] Cisco, "Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2015–2020," Cisco Systems, Inc., Tech. Rep. C11-738429-00, Feb. 2016. [Online]. Available: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.pdf

[2] P. Pirinen, "A brief overview of 5G research activities," in *1st International Conference on 5G for Ubiquitous Connectivity*. Akaslompolo, Finland: IEEE, 2014. doi: 10.4108/icst.5gu.2014.258061. ISBN 978-1-63190-055-6 pp. 17–22. [Online]. Available: http://ieeexplore.ieee.org/document/7041023/

[3] ITU, "ITU defines vision and roadmap for 5G mobile development," ITU, Geneva, Tech. Rep., Jun. 2015. [Online]. Available: http://www.itu.int/net/pressoffice/press_releases/2015/27.aspx#.VYhFl-1VhBe

[4] L. contributors of NFV ISG, "Network Functions Virtualization - An Introduction, Benefits, Enablers, Challenges & Call for Action," in *SDN and OpenFlow World Congress*, Darmstadt, Germany, Oct. 2012. [Online]. Available: https://portal.etsi.org/NFV/NFV_White_Paper.pdf

[5] ——, "Network Functions Virtualisation (NFV) - Network Operator Perspectives on NFV priorities for 5G," in *NFV#17 Plenary meeting*, Bilbao, Spain, Feb. 2017. [Online]. Available: https://portal.etsi.org/NFV/NFV_White_Paper_5G.pdf

[6] B. Pinczel, D. Géhberger, Z. Turányi, and B. Formanek, "Towards High Performance Packet Processing for 5G," in *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*. IEEE, 2015. doi: 10.1109/NFV-SDN.2015.7387408. ISBN 978-1-4673-6884-1 pp. 67–73. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7387408

[7] G. P. Katsikas, "Realizing High Performance NFV Service Chains," Licentiate Thesis, KTH Royal Institute of Technology, Stockholm, Sweden, Dec. 2016. [Online]. Available: http://kth.diva-portal.org/smash/record.jsf?pid=diva2%3A1044355&dswid=3351

[8] "DPDK.org," Jan. 2017. [Online]. Available: http://dpdk.org/

[9] DPDK, "DPDK doc (v 17.05.0) - rte_eventdev.h File Reference." [Online]. Available: http://dpdk.org/doc/api/rte__eventdev_8h.html

[10] ITU-R, "IMT Vision – Framework and overall objectives of the future development of IMT for 2020 and beyond," ITU, Geneva, Schweiz, Tech. Rep. M.2083-0, Sep.

2015. [Online]. Available: https://www.itu.int/dms_pubrec/itu-r/rec/m/R-REC-M.2083-0-201509-I!!PDF-E.pdf

[11] ——, "Requirements related to technical performance for IMT-Advanced radio interface(s)," ITU, Tech. Rep. M.2135, Nov. 2008. [Online]. Available: http://www.itu.int/dms_pub/itu-r/opb/rep/R-REP-M.2134-2008-PDF-E.pdf

[12] W. Xiang, K. Zheng, and X. S. Shen, *5G Mobile Communications*, 1st ed. Springer International Publishing, 2017. ISBN 978-3-319-34208-5. [Online]. Available: http://www.springer.com/gp/book/9783319342061

[13] M. Nohrborg, "LTE Overview," 2017. [Online]. Available: http://www.3gpp.org/technologies/keywords-acronyms/98-lte

[14] G. A. Abed, M. Ismail, and K. Jumari, "The Evolution to 4g Cellular Systems: Architecture and Key Features of LTE-Advanced Networks," *IRACST - International Journal of Computer Networks and Wireless Communications*, vol. 2, no. 1, 2012. [Online]. Available: https://pdfs.semanticscholar.org/8e1c/4295c2533f49ef6e0636ac047afd54a16420.pdf

[15] M. Qian, Y. Zhou, W. Wei, Y. Huang, Y. Wang, and J. Shi, "Efficient design and implementation of LTE UE link-layer protocol stack," in *Wireless Communications and Networking Conference (WCNC), 2013 IEEE*. Shanghai, China: IEEE, Jul. 2013. doi: 10.1109/WCNC.2013.6554682. ISBN 978-1-4673-5939-9 pp. 895–900. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6554682

[16] T. Nakamura, "3gpp RAN work towards '5G'," May 2016. [Online]. Available: http://www.3gpp.org/ftp/Information/presentations/presentations_2016/160527_ICC_5GWS-REV2.pdf

[17] Telefónica and Ericsson, "Cloud RAN Architecture for 5G," Telefónica, White Paper, 2016. [Online]. Available: http://www.tid.es/sites/526e527928a32d6a7400007f/content_entry5321ef0928a32d08900000ac/578f4eda1146dde411001d0e/files/WhitePaper_C-RAN_for_5G_-_In_collab_with_Ericsson_SC_-_quotes_-_FINAL.PDF

[18] P. Rost, C. J. Bernardos, A. De Domenico, M. Di Girolamo, M. Lalam, A. Maeder, D. Sabella, and D. Wübben, "Cloud Technologies for Flexible 5G Radio Access Networks," *IEEE Communications Magazine*, vol. 52, no. 5, pp. 68–76, Sep. 2014. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6898939

[19] 5G PPP Architecture Working Group, "View on 5G Architecture," Jul. 2016. [Online]. Available: https://5g-ppp.eu/wp-content/uploads/2014/02/5G-PPP-5G-Architecture-WP-July-2016.pdf

[20] E. Westberg, "Ericsson Technology Review: 4G/5G RAN Architecture: How a Split Can Make the Difference," *Charting the Future of Innovation*, vol. 93, no. 6, Jul. 2016. [Online]. Available: https://www.ericsson.com/assets/local/publications/ericsson-technology-review/docs/2016/etr-ran-architecture.pdf

[21] Intel, "Intel® Open Network Platform Release 2.1 Performance Test Report - SDN/NFV Solutions with Intel® Open Network Platform," Intel, Tech. Rep., Mar. 2016. [Online]. Available: https://download.01.org/packet-processing/ONPS2.1/Intel_ONP_Release_2.1_Performance_Test_Report_Rev1.0.pdf

[22] "napi," Nov. 2016. [Online]. Available: https://wiki.linuxfoundation.org/networking/napi

[23] W. Wu, M. Crawford, and M. Bowden, "The performance analysis of linux networking – Packet receiving," *Computer Communications*, vol. 30, no. 5, pp. 1044–1057, Mar. 2007. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0140366406004221

[24] A. Cox and F. La Roche, "Ubuntu-kernel, skbuff.h," Aug. 2014. [Online]. Available: https://github.com/Canonical-kernel/Ubuntu-kernel/blob/master/include/linux/skbuff.h

[25] R. Biro, F. N. van Kempen, D. Becker, A. Cox, R. Underwood, S. Becker, J. Cwik, and A. Gulbrandsen, "Ubuntu-kernel, ip_input.c," Jan. 2014. [Online]. Available: https://github.com/Canonical-kernel/Ubuntu-kernel/blob/master/net/ipv4/ip_input.c

[26] P. Roque and I. P. Morris, "Ubuntu-kernel, ip6_input.c," Jan. 2014. [Online]. Available: https://github.com/Canonical-kernel/Ubuntu-kernel/blob/master/net/ipv6/ip6_input.c

[27] O. Zborowski, R. Biro, and F. N. van Kempen, "Ubuntu-kernel, socket.c," Aug. 2014. [Online]. Available: https://github.com/Canonical-kernel/Ubuntu-kernel/blob/master/net/socket.c

[28] J. Corbet and A. Rubini, *Linux Device Drivers*, 2nd ed. O'Reilly Media, Jun. 2001. ISBN 0-596-00008-1. [Online]. Available: http://www.xml.com/ldd/chapter/book/ch14.html#t5

[29] T. Hudek, "Introduction to Receive Side Scaling," Apr. 2017. [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling

[30] V. Medvedkin, "rte_thash.h," 2015. [Online]. Available: http://dpdk.org/doc/api-17.05/rte__thash_8h_source.html

[31] DPDK, "DPDK Programmer's Guide Release 16.11.0," Nov. 2016. [Online]. Available: http://fast.dpdk.org/doc/pdf-guides/prog_guide-16.11.pdf

[32] L. McVoy, "LMbench," Jun. 1998. [Online]. Available: http://www.bitmover.com/lmbench/

[33] "Perf Wiki," Sep. 2015. [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page

[34] "www.ixiacom.com." [Online]. Available: https://www.ixiacom.com/products/ixnetwork

[35] "NO_hz: Reducing Scheduling-Clock Ticks." [Online]. Available: https://www.kernel.org/doc/Documentation/timers/NO_HZ.txt

[36] Z. Yangchao, "[dpdk-dev] hash: optimize the softrss computation," Aug. 2017. [Online]. Available: http://dpdk.org/dev/patchwork/patch/27714/

[37] "Using the GNU Compiler Collection (GCC) - 6.59 Other Built-in Functions

Provided by GCC." [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc-7.2.0/gcc/Other-Builtins.html

# Appendix A

# LMbench Output

This section contains the output when running LMbench on the host and guest machines on which the 5G RAN PPF was evaluated.

## A.1 Host (Physical) Machine Results

```
                L M B E N C H  3 . 0   S U M M A R Y
                ------------------------------------
 (Alpha software, do not distribute)


Processor, Processes - times in microseconds - smaller is better
----------------------------------------------------------------------------
Host                 OS  Mhz null null      open slct sig  sig  fork exec sh
                             call  I/O stat clos TCP  inst hndl proc proc proc
--------- ------------- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ----
selistnh0 Linux 4.4.0-8 2899 0.10 0.13 0.39 0.92 2.40 0.10 0.68 193. 559. 947.

Basic integer operations - times in nanoseconds - smaller is better
-------------------------------------------------------------------
Host                 OS  intgr intgr  intgr  intgr  intgr
                          bit   add    mul    div    mod
--------- ------------- ------ ------ ------ ------ ------
selistnh0 Linux 4.4.0-8 0.5000 0.0400 1.0600 8.1700 9.0600

Basic float operations - times in nanoseconds - smaller is better
-----------------------------------------------------------------
Host                 OS  float  float  float  float
                         add    mul    div    bogo
--------- ------------- ------ ------ ------ ------
selistnh0 Linux 4.4.0-8 1.0300 1.7200 4.6700 2.8800

Basic double operations - times in nanoseconds - smaller is better
------------------------------------------------------------------
Host                 OS  double double double double
                         add    mul    div    bogo
--------- ------------- ------  ------ ------ ------
selistnh0 Linux 4.4.0-8 1.0300 1.7200 7.0700 4.8300
```

```
Context switching - times in microseconds - smaller is better
-------------------------------------------------------------------------
Host                 OS  2p/0K 2p/16K 2p/64K 8p/16K 8p/64K 16p/16K 16p/64K
                         ctxsw  ctxsw  ctxsw ctxsw  ctxsw   ctxsw   ctxsw
--------- ------------- ------ ------ ------ ------ ------ ------- -------
selistnh0 Linux 4.4.0-8 4.4700 3.5100 3.0400 2.6100 1.7800 2.43000 1.67000

*Local* Communication latencies in microseconds - smaller is better
---------------------------------------------------------------------
Host                 OS 2p/0K  Pipe AF     UDP  RPC/   TCP  RPC/ TCP
                        ctxsw       UNIX         UDP         TCP conn
--------- ------------- ----- ----- ---- ----- ----- ----- ----- ----
selistnh0 Linux 4.4.0-8 4.470 6.463 8.46 9.991        12.7        38.

File & VM system latencies in microseconds - smaller is better
-------------------------------------------------------------------------------
Host                 OS   0K File      10K File     Mmap    Prot   Page   100fd
                        Create Delete Create Delete Latency Fault  Fault  selct
--------- ------------- ------ ------ ------ ------ ------- ----- ------- -----
selistnh0 Linux 4.4.0-8 4.9812 4.2954 9.7819 5.5132  5670.0 0.195 0.18030 1.067

*Local* Communication bandwidths in MB/s - bigger is better
-----------------------------------------------------------------------------
Host                 OS  Pipe AF    TCP  File   Mmap  Bcopy  Bcopy  Mem   Mem
                             UNIX      reread reread (libc) (hand) read write
--------- ------------- ---- ---- ---- ------ ------ ------ ------ ---- -----
selistnh0 Linux 4.4.0-8 3936 8360 5103 6370.0 9870.9 8993.3 5875.6 10.K 7450.

Memory latencies in nanoseconds - smaller is better
    (WARNING - may not be correct, check graphs)
------------------------------------------------------------------------------
Host                 OS   Mhz   L1 $   L2 $    Main mem    Rand mem    Guesses
--------- -------------   ---   ----   ----    --------    --------    -------
selistnh0 Linux 4.4.0-8  2899 1.3790 5.3080   23.4       130.2
```

## A.2 Guest (Virtual) Machine Results

```
                L M B E N C H  3 . 0   S U M M A R Y
                ------------------------------------
 (Alpha software, do not distribute)

Basic system parameters
----------------------------------------------------------------------------
Host                 OS Description              Mhz  tlb  cache  mem   scal
                                                     pages line   par   load
                                                           bytes
--------- ------------- ----------------------- ---- ----- ------ ------ ----
vrcs      Linux 4.1.27-                  x86_64 2460    32    128 8.4100    1

Processor, Processes - times in microseconds - smaller is better
----------------------------------------------------------------------------
Host                 OS  Mhz null null      open slct sig  sig  fork exec sh
                             call  I/O stat clos TCP  inst hndl proc proc proc
--------- ------------- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ----
vrcs      Linux 4.1.27- 2460 0.09 0.13 0.61 1.16 2.95 0.12 0.77 129. 419. 1031
```

```
Basic integer operations - times in nanoseconds - smaller is better
-------------------------------------------------------------------
Host                 OS  intgr intgr  intgr  intgr  intgr
                          bit   add    mul    div    mod
--------- ------------- ------ ------ ------ ------ ------
vrcs      Linux 4.1.27- 0.4100 0.0400 1.2600 9.6500   10.7

Basic float operations - times in nanoseconds - smaller is better
-----------------------------------------------------------------
Host                 OS  float  float  float  float
                         add    mul    div    bogo
--------- ------------- ------ ------ ------ ------
vrcs      Linux 4.1.27- 1.2200 2.0400 5.5200 2.9000

Basic double operations - times in nanoseconds - smaller is better
------------------------------------------------------------------
Host                 OS  double double double double
                         add    mul    div    bogo
--------- ------------- ------  ------ ------ ------
vrcs      Linux 4.1.27- 1.2200 2.0400 8.3500 5.7400

Context switching - times in microseconds - smaller is better
-------------------------------------------------------------------------
Host                 OS  2p/0K 2p/16K 2p/64K 8p/16K 8p/64K 16p/16K 16p/64K
                         ctxsw  ctxsw  ctxsw ctxsw  ctxsw   ctxsw   ctxsw
--------- ------------- ------ ------ ------ ------ ------ ------- -------
vrcs      Linux 4.1.27- 0.6300 1.0500 1.2400 1.8400 2.5000 2.16000 2.60000

*Local* Communication latencies in microseconds - smaller is better
---------------------------------------------------------------------
Host                 OS 2p/0K  Pipe AF     UDP  RPC/   TCP  RPC/ TCP
                        ctxsw       UNIX         UDP         TCP conn
--------- ------------- ----- ----- ---- ----- ----- ----- ----- ----
vrcs      Linux 4.1.27- 0.630 3.380 5.16 5.939       7.743        25.

File & VM system latencies in microseconds - smaller is better
-------------------------------------------------------------------------------
Host                 OS   0K File      10K File     Mmap    Prot   Page   100fd
                        Create Delete Create Delete Latency Fault  Fault  selct
--------- ------------- ------ ------ ------ ------ ------- ----- ------- -----
vrcs      Linux 4.1.27- 2.5258 1.5459 5.4627 2.9655  4941.0 0.473 0.22480 1.058

*Local* Communication bandwidths in MB/s - bigger is better
-----------------------------------------------------------------------------
Host                 OS  Pipe AF    TCP  File   Mmap  Bcopy  Bcopy  Mem   Mem
                             UNIX      reread reread (libc) (hand) read write
--------- ------------- ---- ---- ---- ------ ------ ------ ------ ---- -----
vrcs      Linux 4.1.27- 4340 4792 4384 4896.9 7060.4 6465.8 4137.3 6640 5599.

Memory latencies in nanoseconds - smaller is better
    (WARNING - may not be correct, check graphs)
------------------------------------------------------------------------------
Host                 OS   Mhz   L1 $   L2 $    Main mem    Rand mem    Guesses
--------- -------------   ---   ----   ----    --------    --------    -------
vrcs      Linux 4.1.27-  2460 1.6340   11.5   70.4       136.3
```

## A.3 LMbench Config-Run Script Sample Output

```
================================================================

L M B E N C H   C ON F I G U R A T I O N
----------------------------------------

You need to configure some parameters to lmbench.  Once you have configured
these parameters, you may do multiple runs by saying

"make rerun"

in the src subdirectory.

NOTICE: please do not have any other activity on the system if you can
help it.  Things like the second hand on your xclock or X perfmeters
are not so good when benchmarking.  In fact, X is not so good when
benchmarking.

================================================================

If you are running on an MP machine and you want to try running
multiple copies of lmbench in parallel, you can specify how many here.

Using this option will make the benchmark run 100x slower (sorry).

NOTE:  WARNING! This feature is experimental and many results are
known to be incorrect or random!

MULTIPLE COPIES [default 1]:
================================================================

Options to control job placement
1) Allow scheduler to place jobs
2) Assign each benchmark process with any attendent child processes
   to its own processor
3) Assign each benchmark process with any attendent child processes
   to its own processor, except that it will be as far as possible
   from other processes
4) Assign each benchmark and attendent processes to their own
   processors
5) Assign each benchmark and attendent processes to their own
   processors, except that they will be as far as possible from
   each other and other processes
6) Custom placement: you assign each benchmark process with attendent
   child processes to processors
7) Custom placement: you assign each benchmark and attendent
   processes to processors

Note: some benchmarks, such as bw_pipe, create attendent child
processes for each benchmark process.  For example, bw_pipe
needs a second process to send data down the pipe to be read
by the benchmark process.  If you have three copies of the
benchmark process running, then you actually have six processes;
three attendent child processes sending data down the pipes and
three benchmark processes reading data and doing the measurements.
```

```
Job placement selection [default 1]:
=============================================================================

Hang on, we are calculating your timing granularity.
OK, it looks like you can time stuff down to 100000 usec resolution.

Hang on, we are calculating your timing overhead.
OK, it looks like your gettimeofday() costs X usecs.

Hang on, we are calculating your loop overhead.
OK, it looks like your benchmark loop costs X.XXXXXXXX usecs.

=============================================================================

Several benchmarks operate on a range of memory.  This memory should be
sized such that it is at least 4 times as big as the external cache[s]
on your system.   It should be no more than 80% of your physical memory.

The bigger the range, the more accurate the results, but larger sizes
take somewhat longer to run the benchmark.

MB [default XXXX]:
Checking to see if you have XXXX MB; please wait for a moment...
XXXXMB OK
XXXXMB OK
XXXXMB OK
Hang on, we are calculating your cache line size.
OK, it looks like your cache line is  bytes.

=============================================================================

lmbench measures a wide variety of system performance, and the full suite
of benchmarks can take a long time on some platforms.  Consequently, we
offer the capability to run only predefined subsets of benchmarks, one
for operating system specific benchmarks and one for hardware specific
benchmarks.  We also offer the option of running only selected benchmarks
which is useful during operating system development.

Please remember that if you intend to publish the results you either need
to do a full run or one of the predefined OS or hardware subsets.

SUBSET (ALL|HARWARE|OS|DEVELOPMENT) [default all]:
=============================================================================

This benchmark measures, by default, memory latency for a number of
different strides.  That can take a long time and is most useful if you
are trying to figure out your cache line size or if your cache line size
is greater than 128 bytes.

If you are planning on sending in these results, please don't do a fast
run.

Answering yes means that we measure memory latency with a 128 byte stride.

FASTMEM [default no]:
=============================================================================

This benchmark measures, by default, file system latency.  That can
```

```
take a long time on systems with old style file systems (i.e., UFS,
FFS, etc.).  Linux' ext2fs and Sun's tmpfs are fast enough that this
test is not painful.

If you are planning on sending in these results, please don't do a fast
run.

If you want to skip the file system latency tests, answer "yes" below.

SLOWFS [default no]:
================================================================

This benchmark can measure disk zone bandwidths and seek times.  These can
be turned into whizzy graphs that pretty much tell you everything you might
need to know about the performance of your disk.

This takes a while and requires read access to a disk drive.
Write is not measured, see disk.c to see how if you want to do so.

If you want to skip the disk tests, hit return below.

If you want to include disk tests, then specify the path to the disk
device, such as /dev/sda.  For each disk that is readable, you'll be
prompted for a one line description of the drive, i.e.,

Iomega IDE ZIP
or
HP C3725S 2GB on 10MB/sec NCR SCSI bus

DISKS [default none]:
================================================================

If you are running on an idle network and there are other, identically
configured systems, on the same wire (no gateway between you and them),
and you have rsh access to them, then you should run the network part
of the benchmarks to them.  Please specify any such systems as a space
separated list such as: ether-host fddi-host hippi-host.

REMOTE [default none]:
================================================================

Calculating mhz, please wait for a moment...
mhz: should take approximately 30 seconds
I think your CPU mhz is

4267 MHz, 0.2344 nanosec clock

but I am frequently wrong.  If that is the wrong Mhz, type in your
best guess as to your processor speed.  It doesn't have to be exact,
but if you know it is around 800, say 800.

Please note that some processors, such as the P4, have a core which
is double-clocked, so on those processors the reported clock speed
will be roughly double the advertised clock rate.  For example, a
1.8GHz P4 may be reported as a 3592MHz processor.

Processor mhz [default XXXX MHz, X.XXXX nanosec clock]:
================================================================
```

```
We need a place to store a 128 Mbyte file as well as create and delete a
large number of small files.  We default to /usr/tmp.  If /usr/tmp is a
memory resident file system (i.e., tmpfs), pick a different place.
Please specify a directory that has enough space and is a local file
system.

FSDIR [default /var/tmp]:
================================================================

lmbench outputs status information as it runs various benchmarks.
By default this output is sent to /dev/tty, but you may redirect
it to any file you wish (such as /dev/null...).

Status output file [default /dev/tty]:
================================================================

There is a database of benchmark results that is shipped with new
releases of lmbench.  Your results can be included in the database
if you wish.  The more results the better, especially if they include
remote networking.  If your results are interesting, i.e., for a new
fast box, they may be made available on the lmbench web page, which is

http://www.bitmover.com/lmbench

Mail results [default yes]: no
OK, no results mailed.
================================================================

Confguration done, thanks.

There is a mailing list for discussing lmbench hosted at BitMover.
Send mail to majordomo@bitmover.com to join the list.
```

TRITA-ICT-EX-2017:198