Performance Evaluation of eXpress Data Path for Container-Based Network Functions

Tugce Özturk

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 30.12.2020

Supervisor

Prof. Raimo Kantola

Advisor

M. Sc. Riku Kaura-aho

Aalto University
School of Electrical
Engineering

Aalto University
School of Electrical
Engineering

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Tugce Özturk

Title Performance Evaluation of eXpress Data Path for Container-Based Network Functions

Degree programme Computer, Communication and Information Sciences

Major Communications Engineering **Code of major** ELEC3029

Supervisor Prof. Raimo Kantola

Advisor M. Sc. Riku Kaura-aho

Date 30.12.2020 **Number of pages** 65 **Language** English

Abstract

The architectural shift in the implementations of Network Function Virtualization (NFV) has necessitated improvements in the networking performance of commodity servers in order to meet the requirements of new generation networks. To achieve this, many fast packet processing frameworks have been proposed by academia and industry, one of them being eXpress Data Path (XDP) incorporated in the Linux kernel. XDP differentiates from other packet processing frameworks by leveraging the in-kernel facilities and running in device driver context. In this study, the XDP technology is evaluated to address the question of supporting performance needs of container-based virtual network functions in cloud architectures. Thus, a prototype consisting of a data plane powered by XDP programs and a container layer emulating the virtual network function is implemented, and used as a test-bed for throughput and CPU utilization measurements. The prototype has been tested to investigate the scalability with varying traffic loads as well as to reveal potential limitations of XDP in such a deployment. The results have shown that an XDP-based solution for accelerating the data path for virtual network functions provides sufficient throughput and scalability in a system with multi-core processor. Furthermore, the prototype has proven that the XDP-based solution has higher throughput performance than the implementation with Linux networking stack. On the other hand, the prototype has been observed to have performance limitations which are analysed in detail. Finally, future directions for further improvements are proposed.

Keywords Virtual network functions, Packet processing, Linux kernel, eXpress Data Path

Preface

This thesis study was conducted in Nokia Digital Automation Cloud team in Espoo, Finland.

Firstly, I would like to thank my advisor Riku Kaura-aho for introducing me to such an intriguing topic and always supporting me through the thesis challenge. And, I am grateful to my manager Jari Hyytiainen for giving me this opportunity to conduct my thesis in this innovative environment.

Later, I would like to express my gratitude to my supervisor Prof. Raimo Kantola for his guidance and illuminating recommendations.

This work, completed in the last half of such a challenging year, is an end-product of a more than two years long journey. A full journey with tough times, dark days, and frosty paths. But also, rewarding and gratifying with joyous memories, endless summer days, and colorful nature. And, I want to thank all the beautiful people who have made this experience unique. Especially to Eda Genç for always encouraging me since the very beginning, to Raffaella Carluccio for inspiring me in times that I felt drained, and to Gabriel Ly for reminding me to enjoy every moment.

Lastly, I want to thank my precious family, my parents Kıymet & Mehmet and my sisters Tuğba & Aleyna, for making me who I am today and for always being by my side, unquestionably.

Espoo, 30.12.2020

Tuğçe Öztürk

Contents

Abbreviations

3GPP	3rd Generation Partnership Project
5G	5th Generation
API	Application Programming Interface
BGP	Border Gateway Protocol
BPF	Berkley Packet Filter
CNI	Container Network Interface
COTS	Common Off-the-Shelf Server
CPU	Central Processing Unit
DDIO	Data Direct I/O
DDoS	Distributed Denial-of-Dervice
DMA	Direct Memory Access
DPDK	Data Plane Development Kit
DUT	Device Under Test
eBPF	extended Berkley Packet Filter
ELF	Executable and Linkable Format
ETSI	European Telecommunications Standards Institute
FIB	Forwarding Information Base
GPL	General Public License
I/O	Input/Output
IP	Internet Protocol
IRQ	Interrupt Request
ISG	Industry Specification Group
JIT	Just-in-time
MAC	Media Access Control
MTU	Maximum Transmission Unit
NAPI	New API
NF	Network Function
NFV	Network Function Virtualization
NIC	Network Interface Controller
NUMA	Non-Uniform Memory Access
OS	Operating system
P4	Programming Protocol-Independent Packet Processors
PCIe	Peripheral Component Interconnect express
PF_RING ZC	PF_RING Zero Copy
RAM	Random Access Memory
RSS	Receive Side Scaling
RX/TX	Receive/Transmit
SDN	Software Defined Network
SFC	Service Function Chaining
SPDX	Software Package Data Exchange
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VM	Virtual Machine
VNF	Virtual Network Function
XDP	eXpress Data Path

1 Introduction

In recent years, the advancements in telecommunications by the emergence of 5G technologies have provided several nascent use cases such as factory automation, connected drones, autonomous vehicles and smart grids [1]. These use cases promise new opportunities for many industry verticals; however, these new use cases demand ambitious capabilities from the underlying network infrastructure such as, fast and reliable connection, very high throughput, programmability and scalability of the networks cost-efficiently. In order to meet those requirements, the novel telecommunication networks have been transforming towards a service-oriented and modularized architecture [2] by adopting network softwarization [3].

Network softwarization enables the separation of monolithic networking hardware and software bundles by leveraging Software Defined Networking (SDN) and Network Function Virtualization (NFV) concepts. SDN aims to decouple network controlling functions from network forwarding functions which results in the desired flexibility and programmability for service-based networks [4]. Likewise, NFV proposes complete virtualization of network functions (NFs) and gives capability to run on commodity servers (Commercial Off-The-Shelf (COTS)) [5]. NFV virtualizes network functions such as routing, switching, load balancing, firewalls into software programs which can be deployed on a virtual machine (VM) or a container residing on top of a hypervisor or a bare-metal server respectively. This gives flexibility to deploy customized NFs based on the needs of the network, and ability to scale in/out depending on the traffic demand.

In order to realize the true potential of the network softwarization, the telecom industry has been in an architectural shift towards cloud computing together with edge computing. Cloud computing refers to the way of providing computing services through the Internet where the service resources, such as servers, storage, databases and networking, reside in the data centers [6]. In addition to that, edge computing decentralizes the cloud architecture in order to provide data processing facilities closer to the source of the data. By following the trend in cloud computing, one step further in the evolvement of NFV has been the adoption of the micro-services architecture in recent releases [7]. This architecture is practised by decomposing the NFs into smaller functional elements and orchestrating these decomposed functions based on service needs [3]. Such an approach provides a modular architecture which eliminates redundant network processing, enables customized service chains and results in cost-efficiency [3].

One of the main components of the micro-services architecture is the emerging container technology. Containers are lightweight and isolated instances of programs running on top of the mechanisms abstracted from underlying hardware components and operating system (OS). Unlike virtual machines that reproduce a complete operating system on top of the host server [8], containers leverage the capabilities of the core of host operating system, namely ‘kernel’, and skip hardware emulation

overhead [9]. Thus, containers promise higher-resource efficiency, faster boot time and more flexible life-cycle than virtual machines for implementation of virtual network functions (VNFs) [8]. However, the design of kernel networking stack of commodity servers in general is not performance-oriented as in a function-specific hardware. Consequently, in high-speed traffic conditions, the networking stack in commodity server OSes does not provide sufficient performance for the network functions deployed in containers [10].

In order to overcome the incompetence of the networking stack of commodity servers in the context of providing high-performance, several packet processing frameworks have been under development and some of them are widely used to implement in VNFs. Examples are DPDK (Data Plane Development Kit), PF_RING ZC (Zero Copy) and netmap [11]. Whereas all of these frameworks aim to increase the performance of packet processing, they vary in terms of overall performance, convenience of implementation, hardware compatibility and software requirements [12] due to the divergence in their methods. For instance, the highest performing and widest used method DPDK [13] utilizes the kernel-bypass technique which eliminates the kernel networking stack and transfers packet data directly to user space applications for processing. Although this technique boosts the performance significantly, it requires the network stack to be re-implemented from scratch in the OS user space which brings additional cost to developers [14]. Moreover, its implementation demands at least one CPU to be fully dedicated to packet processing as well as huge page memory to be pre-allocated [15, 16, 9]. Such problems necessitate a more efficient and less costly method for development of high-performance network functions.

One promising solution for tackling this issue would be to incorporate eXpress Data Path (XDP) in the architecture of network functions. XDP is a recently developed packet processing framework running in the Linux kernel. Contrary to kernel-bypassing frameworks, XDP programs start packet processing instantly in the network device driver context. Thus, XDP introduce no overhead of moving packet data from network device driver queues to user-space applications.

XDP executes in kernel-space by utilizing eBPF (extended Berkley Packet Filter) technology. eBPF is a built-in Linux kernel tool which has been widely used for network observability, packet filtering and packet capturing [17]. This technology enables XDP programs to run as process virtual machines attached to network device drivers. Thus, XDP programs immediately catch the arriving packets from the receiving queues in the network device driver, right before the packets are transferred to the kernel network stack. As a consequence, the packet-processing performance is improved by the elimination of some costly kernel network stack operations, such as buffer allocation per packet.

This promising feature has led XDP to be used in various contexts both in industry and academia. For instance, one Open Source project called Cilium utilizes XDP for providing security and load balancing to networking between container applications [18], while Cloudflare uses XDP for mitigating DDoS (Distributed Denial of Service)

attacks [19]. The studies [15, 20, 21, 22, 23] evaluate and implement XDP in different cases and report its potential for enhancing packet processing performance. However, this technology is in early stages of development and needs more attention to be further improved. Moreover, there is little research addressing the usability of XDP for implementing virtual network functions, especially using containers and virtual network interfaces.

Therefore, this thesis aims to evaluate the feasibility of using XDP for accelerating the packet-processing performance for container-based virtual network functions, exclusively in the scale of the edge cloud network. To accomplish this task, the thesis implements a prototype consisting of an XDP-based routing and forwarding plane, and on top of it, a container layer which emulates the container-based network functions. The goal of the prototype is to provide a stable environment to observe the overall system performance as well as the packet-processing capabilities when XDP programs are attached to both physical and virtual interfaces. Thus, the prototype implements solely the routing and packet-forwarding functionalities. In order to provide simplicity, network functions, such as load balancing or deep packet inspection, are not implemented in this thesis. Later, the performance of the prototype has been tested with the traffic produced by a packet generator in a test-bed consisting of cloud-grade servers. The performance measurements focus on throughput performance, packet-loss occurrence, and CPU usage in order to evaluate the performance impact of XDP-based packet-processing as well as the practicality of such implementation. The latency measurements are not within the scope of this study.

The rest of the thesis is divided into five chapters. Chapter 2 gives background information on the utilized technologies in order to provide a solid understanding on the overall context of the study. Following chapter reviews the literature which studied XDP and available technologies which utilized XDP. Such review of available works aims to highlight different use-cases which supported the development of the prototype as well as to provide a foundation for performance comparison of the prototype. Chapter 4 explains the architecture and implementation of the prototype. In addition, the method and tools of measurement are described in detail. Chapter 5 presents a thorough analysis and discussion on the performance of XDP based on the results obtained from the measurements performed on the prototype. Final chapter concludes the thesis by summarizing the study and its output, and offers directions for future research.

2 Background

This chapter aims to provide the foundational information to perceive the origin of this study, as well as to comprehend the fundamental constituents of the packet processing in commodity servers. Firstly, the timeline of network function virtualization is summarized including the technological trends affecting its evolution. The contributing technologies are defined briefly in order to depict the current status and future directions of NFV.

Later, the fundamentals of Linux networking from the packet reception perspective are explained extensively. This knowledge base of Linux networking is distinctly required to grasp the difference of the studied technology from other available solutions. Thereupon, a brief introduction to software-based high-performance packet processing frameworks is presented including base-level information on kernel-bypass methods as well. Finally, the studied technology, XDP, and its underlying mechanism, eBPF, are explained thoroughly. Starting from its architecture, the fundamental components and functions of the technology are reported to pave the way for the implementation and further evaluation.

2.1 Evolution towards cloud-native network functions

In telecommunications, the networking infrastructure has been deployed using function-specific hardware until recent years. Although these dedicated equipments provide great performance, such an infrastructure composed of monolith hardware lacks the agility to embrace the ever-changing demands and increasing competitiveness of the market. Consequent to the inert nature of conventional networking hardware, advancements in virtualization technologies, which abstracts the functions from the hardware, has resulted in a paradigm shift in telecom industry towards Network Function Virtualization (NFV) and Software Defined Networking (SDN). Principally, ETSI ISG NFV is the organization that guides the NFV development and defines the specifications for implementation since 2012 [7]. Until today NFV has been adopted by many vendors and service providers in a way to sustain the legacy infrastructure and deploy the softwarized versions of the network functions in virtual machines running on commodity servers. However, this approach does not provide the performance boost from the customized networking chips used in the monolith networking hardware.

Therefore, as of ETSI ISG NFV specification Release 4 [7], the specifications suggest the adaptation of cloud-native architecture in telecom infrastructures. In addition to that, through the tight collaboration between ETSI and 3GPP, the novel 5G core architecture is designed to utilize micro-services approach for implementation of new service-based architecture since 3GPP Release 15.

In the micro-services architecture, services are broken into smaller services running

inside lightweight processes and orchestrated through a controlling mechanism which populates the services with the required amount of resources. This architecture obtains its power mainly from the container technology.

2.2 Containers

Containers run as isolated sets of processes on top of the kernel of the host operating system. In Linux operating system, containers are enabled fundamentally by namespaces and cgroups technologies [24]. Namespaces is a feature of Linux kernel that isolates processes; thus, the processes running in different namespaces cannot access each other. Cgroups is another feature that allows to define resource limits on a group of processes. For instance, such resource limits can be enforced on CPU and memory usage of the regarding processes in that cgroup. Unlike the virtual machines which require to setup the complete operating system and abstracted hardware components, containers are able to run as isolated processes securely on an abstraction layer by utilizing the namespaces and cgroups features.

In general, containers are built on individual network namespaces and communicate with host and other containers through the virtual network interfaces attached [25]. 'veth' interfaces are the most commonly used virtual interfaces in containers. veth interfaces come in pairs that one is attached to the container and the other peer is attached to the host. These peers work like a virtual Ethernet cable where a packet arriving at an interface ends up in the other interface.

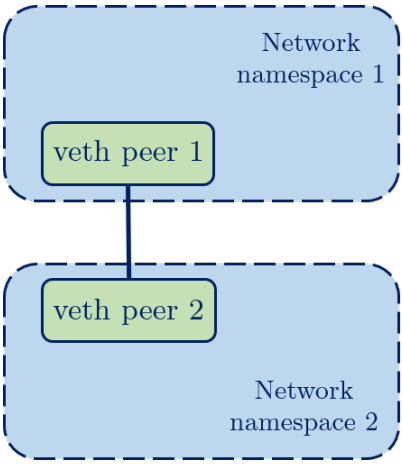


Figure 1: Representation of veth pairs attached to different network namespaces

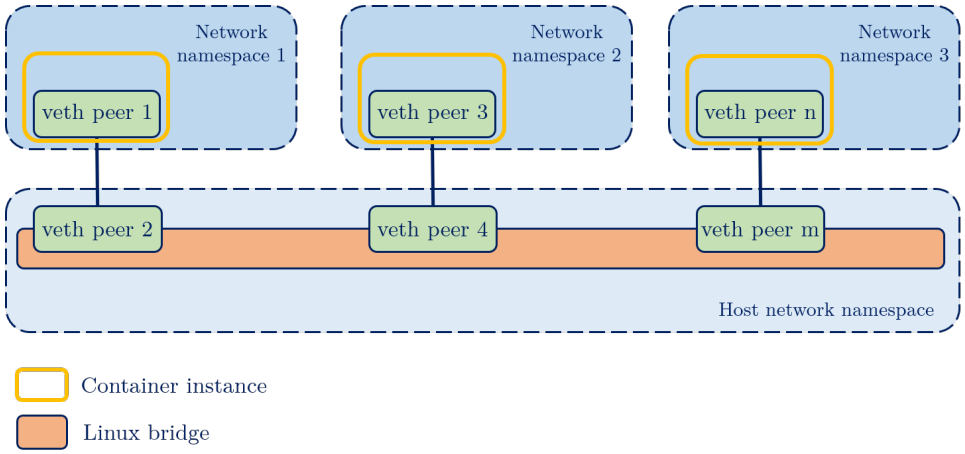


Figure 2: An example of container networking with veth pairs

2.3 Packet processing in Linux kernel networking stack

In this section, Linux kernel networking fundamentals are studied comprehensively in order to depict a clear image of packet processing mechanism in Linux networking stack.

The whole packet processing mechanism is a harmonious cooperation of Linux kernel network stack internals, such as data structures and system calls, as well as the underlying hardware constituents such as the processor, network card and memory. Packet processing is composed of a chain of events traversing through layers of the networking stack, from physical transmission layer to user-space or vice versa.

From a high-level perspective, the fundamental elements of the path of a packet are the Central Processing Unit (CPU) handling significant amount of the processing, the Network interface card (NIC) providing the connection point with outside of the host, memory buffers and data structures that keep the packet and packet information, internal kernel mechanisms than control the flow of process handling.

2.3.1 Impact of hardware components in kernel networking

CPU. CPU is the main processing unit that executes directives to produce output from given input. These directives are machine-level instructions provided by the programs in execution. A CPU has an internal clock which is made of quartz crystals oscillating in a very high frequency. The oscillating frequency determines the ticking of the CPU clock, and CPU executes one cycle in each tick of the clock. Within this cycle, a CPU executes an operation such as fetch or decode instructions, read from memory. Thus, the cycles per second affects the number of instructions executed per unit time, and consequently the speed of the CPU. [26].

Through the evolution of processors, the main goal had been increasing the clock frequency to accelerate the processors until the increasing clock speed has become costly due to limitations, such as very high temperature of the hardware at higher frequencies [5]. Therefore, the processor producers have shifted the design focus towards bundling multiple computational engines [27], referred as cores, on a processor in order to improve overall computing performance, instead of enhancing the speed of a single processor computational unit [28]. Such processors are called multi-core processors, and they enable the execution of multiple and distinct threads of instructions in parallel. For instance, in the Intel Core processor family, which is designed for small scale computing units, such as personal laptops, number of physical CPU cores vary from 2 to 8 in i7 series, and reach up to 10 physical CPU cores in i9 series [29]. Additionally, in multi-core processors targeting data center computing units, such as Intel Xeon processor family, the number of physical cores can reach up to 56 cores [30].

Additionally, as CPU catches the instructions and input data from the associated memory and delivers the output to memory as well, the efficiency of the communication with memory is highly significant. Thus, modern processors have their own local memory areas called cache in addition to the main memory.

Cache memory areas are located close to the CPU logical units in order to eliminate the cost of transferring data to and from a farther location. In modern architectures, CPU cores generally have three level of caches with different proximity and sizes. CPU searches for the instructions first in layer 1 cache which is the closest but smallest sized cache, then layer 2 and layer 3 caches that are relatively farther and bigger. Layer 3 cache is shared among multiple CPU cores in a multi-core system, while L1 and L2 caches are exclusive to the corresponding CPU core. Finally, CPU searches for the required input in the main memory if it can not acquire the information in any level of the caches.

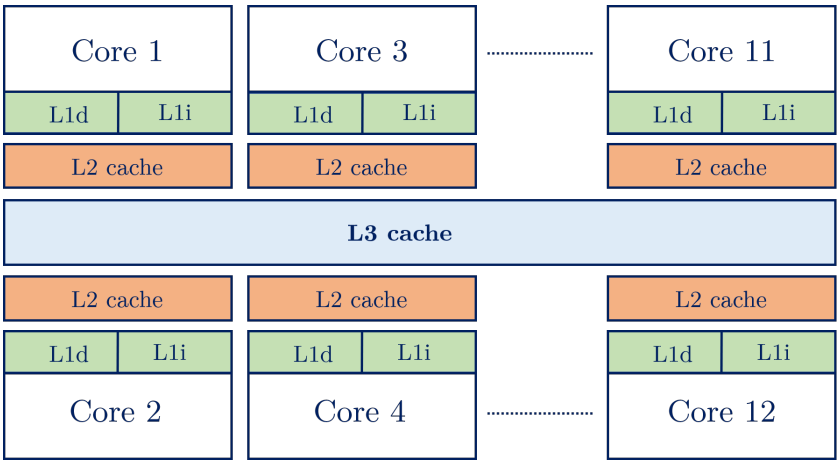


Figure 3: CPU cache levels reference representation [31]

NIC. NIC, called as network adapter as well, is a hardware unit connecting the host to a network by receiving and transmitting data packet signals. The physical

communication between the NIC and the host system is handled over the system bus where NIC card is attached. NIC devices consist of physical ports for data transmission and reception, buffers where the data is stored before transmission to host system, a controller unit and a direct memory access controller which handles transmission of packets to designated memory accesible to the CPU core.

NIC devices are handled by the host kernel system through network device drivers. The network device drivers are responsible from initialization of the network device, handling the communication between host and NIC device through interrupts, allocation of memory area to locate the network packets [32].

2.3.2 Fundamental data structures in Linux networking

struct sk_buff. This is the data structure that stores all required information to manage the packet under process. As packet traverses the network layers, its data is kept in the same buffer allocated in a dedicated memory cache [33]. The sk_buff data structure contains the pointers to specific information fields regarding to this packet in the buffer. Some of those information fields are protocol headers, location of payload, destination, length of data structure, head and tail of data structure, and timestamp.

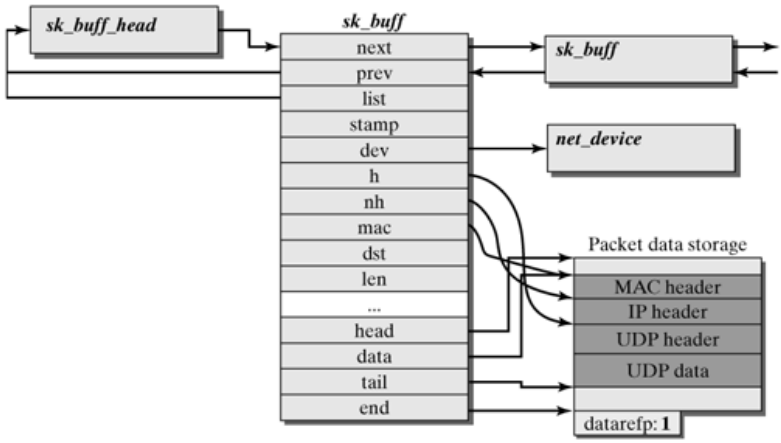


Figure 4: sk_buff structure representation [34]

The 'dev' field points to a net_device data structure and its utilization is subject to change depending on the state of packet. Meaning that, the dev field points to ingress interface if the packet is received, and it points to egress interface if the packet is in transmission [33].

struct net_device. This data structure is utilized to represent each of the network devices registered in Linux kernel. All the hardware and software information regarding to the network device is defined in this data structure since network drivers do not have device files under /dev directory [32]. The fields of the structure hold

information in many categories, such as configuration, statistics, device status, and traffic management [33].

2.3.3 Packet reception in Linux kernel

Packet processing is handled through the interaction between the CPU and NIC device. NIC adapter receives the packets through wire or radio interface, and starts transmitting them into the designated memory area in the kernel via direct memory access (DMA). DMA is a feature that enables NIC devices to transfer the packets in NIC receive buffers directly into the device driver buffers located in RAM or CPU cache. These packet buffers are allocated by network device driver for both reception and transmission queues in a ring buffer data structure. Then, their location in the memory is informed to the NIC device by the driver. Device drivers have to initialize these settings before the packet handling.

After the packets are transferred to the RX ring buffers through DMA, the NIC raises a hardware interrupt request (IRQ) to notify the CPU that there are new frames ready to be processed. This hardware interrupt triggers the network driver to start its processing on the packets. A hardware interrupt causes immediate interruption on any running process on CPU, and CPU cannot process any other interrupts until the hardware interrupt handling finishes [34]. Thus, hardware interrupts can easily consume all the CPU resources if they occur more frequently than they are processed. For this reason, modern systems handle the network packet arrival interrupts through an interrupt management technique called New API (NAPI). When NAPI is invoked, the processor switches to polling mode from interrupt handling, and switches back to default mode when the tasks in NAPI queues are finished.

In case of packet reception, the network device driver triggered by the hardware interrupt invokes the NAPI mode. To enable NAPI, network driver disables the hardware interrupts until NAPI dequeues the awaiting packets. NAPI polling and packet dequeuing are performed through software interrupt mechanism (softirqs). Softirqs differentiate from hardware interrupts since they can be scheduled for later execution and do not require immediate action like hardware interrupts [35].

The softirqs transfer the packets to kernel networking stack by allocating socket buffers and sk_buff data structures. After socket buffer allocation, the Linux networking stack passes the sk_buff data structure through corresponding protocol layers for further processing of the packet.

2.4 High performance packet processing

Linux kernel network stack consists of many tools providing granular networking capabilities, such as the data structures to represent network packets and network devices, various systemcall functions to perform packet modifications, a wide selec-

tion of network protocol implementations, firewall functionality, and routing tables. However, Linux network stack performance decreases dramatically as the network line rate increases due to high overhead of the processing of each packet through the levels of the stack. Consequently, the gap between the packet rate on the line and the processing time allocated for each packet have necessitated the implementation of new high-performance packet processing frameworks, such as netmap and DPDK.

Netmap is a device and operating system independent packet I/O acceleration solution that provides user-space applications with fast access to network packets [36]. It is loaded into kernel as a module and requires modification on network drivers to be able to function in netmap mode. Many widely used drivers have patches to support netmap mode [37]. Netmap acquires its efficiency by several techniques; such as pre-allocating packet buffers and removing runtime memory allocation cost, providing exchange of packets between NIC and user-space with ring buffers in shared memory, and reducing the need for system calls per packet.

By architecture, netmap has three fundamental data structures, which are `netmap_if`, `netmap_ring` and `packet_buf`,that enable direct communication between NIC and user-space program during execution in netmap mode. These data structures are located in a non-swappable shared memory area that is accessible by all processes. The `netmap_if` data structure keeps information related to network interface, while the `netmap_ring` keeps references to `packet_buf` structures in a circular queue model. The `packet_buf` data structures are pre-allocated in the shared memory and each keeps a single network frame. Consequently, netmap eliminates the overhead of system calls for memory allocation and packet copy to user-space. Furthermore, the usage of system calls in netmap is kept limited to device opening operation and updating the NIC about the availability of ring buffers [13].

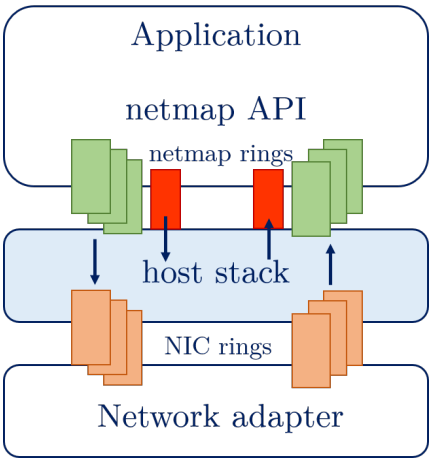


Figure 5: netmap representation [37]

Another widely used framework is the Data Plane Development Kit (DPDK) which consists of libraries for data plane applications [38]. DPDK is a kernel-bypass technique that carries the packet processing tasks to user-space programs by giving user-space the full control over the network devices. Therefore, it completely isolates

the kernel network stack from the packet processing operations, which requires reimplementation of a seperate networking stack in the user-space. Additionally, DPDK eliminates the interrupt processing overhead through constantly polling the network devices instead. However, in such architecture, all the required resources have to be allocated and dedicated before the data plane applications starts execution [38]. For instance, a CPU core dedicated for DPDK applications is always 100% utilized due to constant polling to the network devices. As this accelerates the packet processing performance dramatically, it may lead to wasted CPU cycles and high power consumption since the processor will continue polling even there is no incoming packets.

2.5 In-kernel fast packet processing: eXpress Data Path (XDP)

XDP is an in-kernel packet processing technique which has been merged to Linux kernel in version 4.8 [39] and has been under active development since 2016.

XDP initially operates at Layer 2 (Ethernet) and Layer 3 (IP), and targeted to increase the performance of processing the stateless protocol packets [16]. Thus, as the time of writing XDP does not have a Layer 4 (TCP) implementation; but, it is capable of pushing the packets to the upper layers of the Linux kernel network stack for further processing after the L2/L3 processing is completed.

Fundamentally, XDP programs are capable of processing the received packets at the earliest stage when they arrive to kernel, which is immediately after the reception from NIC device, and before the `sk_buff` data structure allocation in the kernel network stack.

By design, XDP works in cooperation with the existing kernel networking infrastructure unlike the kernel-bypass techniques which isolate the kernel from their operations. Thus, it provides high-performance processing within the kernel-space without the need of additional user-space implementations or kernel modules. Apart from that, XDP programs do not require dedicated CPU units and pre-allocated memory pages solely for packet processing. This gives an advantage in efficient resource utilization and power consumption.

Another important point is that, the XDP programs are prevented to crash the kernel and are programmable on the fly without service discontinuation [16]. Furthermore, XDP programs can create and access to their custom data structures, which provides metadata-sharing with other XDP programs, kernel-space and user-space. Additionally, XDP programs are able to utilize some kernel functionalities, such as routing table look-ups, through helper functions provided in kernel libraries.

In the following subsections, architecture of XDP and the technologies that make up this architecture are explained in detail. Later, XDP operation modes, return codes

and bulk processing feature are briefly described.

2.5.1 XDP architecture

XDP programs are essentially extended Berkley Packet Filter (eBPF) programs attached to a special type of hook on the network device driver. In order to understand the overall architecture of XDP, it is important to understand eBPF technology as well. Thus, the following part presents an introduction to the eBPF technology; then, we explain the XDP architecture and the way it utilizes the eBPF facilities.

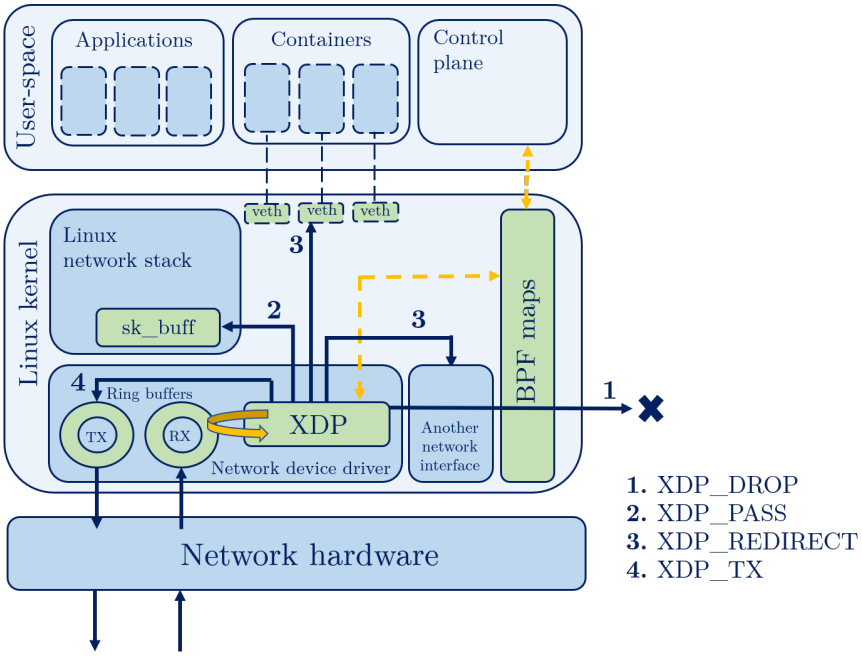


Figure 6: Overview of XDP and XDP actions [16]

eBPF (Extended Berkley Packet Filter) BPF (Berkley Packet Filter), and the ancestor of the eBPF, is a tool inside the Linux kernel, which emerged in 1992 and has been initially used for network packet filtering applications such as well-known tcpdump. BPF has differentiated from other packet filter frameworks due to its ability to run directly in kernel as a process virtual machine and without the need of copying the packets to user-space [40]. Specifically, BPF virtual machines decode and execute given instructions in a separate execution environment, and use the operands located in CPU registers. These instructions are compiled from high-level languages to a mid-level source code, which is called bytecode, in order to be processed through an interpreter to execute as machine code in runtime. Furthermore, BPF programs are ensured to run safely and exit immediately when the execution is completed.

As keeping the fundamental structure same, eBPF is introduced as an extended, accelerated and optimized version of BPF in 2014. This new version, called eBPF,

provides ten 64-bits registers in BPF virtual machines while ancestor BPF provided only two 32-bits registers [40]. As a result, the extended version gives opportunity for more complex in-kernel programs and achieves better compatibility with modern hardware. Additionally, eBPF supports stateful programs by utilizing its own map structures.

Further details regarding to the components and functionality of eBPF is explained within the XDP architecture subject in the following part.

XDP architecture The overall architecture of XDP is composed of the eBPF virtual machine (bytecode), the BPF verifier, XDP driver hook, and BPF maps [16].

Basically, an XDP program is written in restricted C code and compiled into bytecode through the LLVM compiler in the kernel. After compilation, the compiled bytecode has to be verified as safely-executable within the kernel. This verification process is performed by the BPF verifier utility which has to make sure that the program has no loops, does not exceed the program size limits and does not call unreachable functions [16]. Additionally, the verifier has to check if the data exchange with the maps are configured properly and the program does not try to reach out of the valid memory boundaries. It should be noted that this examination ensures only the program safety, and it does not consider if the program performs the expected functionality.

Once the program is verified, the bytecode is compiled into machine code by JIT (Just-in-time) compiler [40]. JIT compiler compiles the bytecode into machine code on-the-fly, which results in performance increase in the execution speed [41].

Further to that, XDP programs are executed based on kernel events like all the BPF programs. In order to receive the designated kernel event, the programs need to be attached to an appropriate execution point, also known as hook point. An eBPF program starts its execution when the hook point is triggered by the occurrence of the registered event. In case of the XDP programs, this execution point is located in the network device driver, and it is triggered by the reception of a packet [42].

At the reception of a packet from the network hardware, the XDP driver hook in fact receives a context object which encloses the pointers addressing to the raw packet and the metadata regarding the packet. That XDP context object is defined in `bpf.h` library as `xdp_md` struct with additional data fields pointing to receive and transmit interfaces.

```
1  struct xdp_md {
2          __u32 data;
3          __u32 data_end;
4          __u32 data_meta;
```

```
5           /* Below access go through struct xdp_rxq_info */
6           __u32 ingress_ifindex; /* rxq->dev->ifindex */
7           __u32 rx_queue_index;  /* rxq->queue_index  */
8           __u32 egress_ifindex;  /* txq->dev->ifindex */
9   };
```

When XDP receives this context object, the received packet still resides in the receiving queue buffer of the driver. Thus, XDP program is able to start processing packets before the Linux networking stack performs socket buffer allocation.

Another BPF component utilized by XDP programs is the BPF maps which are persistent key/value data storages allowing programs to register and read data inside kernel. Furthermore, these maps can be read and written from user-space as well. By this way, BPF maps enable the bidirectional communications between kernel and user-space as well as the data transfer between different eBPF programs running in the kernel [16]. Specifically, they can be used to manage forwarding actions, to collect program metrics, and exchange information with the user-space.

The mechanism that provides access to BPF maps are the BPF helper functions defined in kernel. Not limited to that, helper functions enable usage of some of the existing Linux kernel facilities from within the XDP programs instead of reimplementing these functionalities or evoking system calls. XDP programs can leverage many useful functionalities through the helpers, such as accessing the routing tables, calculating checksum values, modifying map values, and triggering concatenated execution of eBPF programs [43].

2.5.2 XDP driver modes

As mentioned in the XDP Architecture subsection, an XDP program runs in the network device driver context and its functionality depends on the device driver. In accordance with the support of the network device driver, XDP can operate in three different modes: native, generic and offload.

Native XDP. This is the essential XDP operation mode in which the XDP program is directly attached to the device driver. Consequently, it provides the aforementioned benefit of executing the program before the kernel networking stack handles the packet. However, the number of network device drivers that support this mode is limited. Some of the supporting network drivers are Intel i40e and ixgbe, Netronome nfp, Mellanox mlx4, mlx5, and virtual interface veth [39].

Generic XDP. In case of the network driver does not support the native mode, the XDP program can be attached inside the kernel. This mode is called generic XDP. Unfortunately, generic XDP mode has performance penalties compared to native mode due to the fact that the packet is handled within kernel; thus, it brings

the additional overhead of kernel stack processes, such as sk_buff allocation. For this reason, this mode is advised to be used only for testing purposes, and not for production environments.

XDP Offload. The third operation mode offloads the XDP program onto the NIC device. Currently, the support for this feature is limited to smartNICs.

2.5.3 XDP actions

An XDP program starts with reception of a packet and continues with performing its programmed tasks on the packet, such as parsing, encapsulating, and rewriting. Based on this processing, the program has to return a final verdict to the device driver so that it can take an action on the packet. There are five predefined actions in Linux kernel that XDP programs can return to drivers: XDP_ABORTED, XDP_DROP, XDP_PASS, XDP_TX, and XDP_REDIRECT [44].

XDP_ABORTED. This action drops the packet and raises an exception through a tracepoint. Thus, it can be utilized to monitor misbehavior of the program.

XDP_DROP. This return code drops the packet immediately at the driver level. It is useful especially for DDoS mitigation scenarios.

XDP_PASS. This action passes the packet to the kernel networking stack so that the kernel allocates socket buffer and continues with the processing.

XDP_TX. A program returning this action code directs the packet back to the ingress interface the packet was received. For instance, this action is useful to implement a load balancer which sends the packets back to the switch after rewriting them.

XDP_REDIRECT. This action provides the ability to redirect the packets to another physical or virtual network device driver. Additionally, it can redirect the packets to a remote CPU for further processing: in consequence, the CPUs serving to RX queues can continue processing the reception tasks while remote CPUs can continue performing the costly packet processing tasks. This mechanism allows to reduce the load on CPU cores that are serving to receiving queues [44].

2.5.4 XDP bulking in XDP_REDIRECT action

XDP Redirect action handles bulk processing of XDP frames in cooperation with the NAPI mechanism and the BPF device maps. As described in [45], the device drivers with XDP support accommodate an XDP RX handler which has access to BPF device map given to the program. This map provides the connection with the `net_device` data structure which keeps the XDP bulk queue information.

In the packet reception, NAPI mechanism calls the poll function and triggers the XDP RX handler of the device driver. The RX handler executes the `xdp_do_redirect` function upon the REDIRECT verdict from the program, which processes the frames within the bulk queue. At the end of the NAPI poll, the device driver calls `xdp_do_flush_map` action to flush the device map and get the state ready for next NAPI cycle.

This chain of events is defined within the device driver header files such as [46] for i40e device driver. Apart from that, the default bulk queue size is defined as 16 frames in the `xdp.h` kernel header file [47].

3 Related work

Usability of XDP is not only limited to single domain due to being directly integrated to the kernel. It proposes potential improvements in various fields serving to different purposes, from high performance packet processing to network security. In this chapter, both academic studies and industrial cases that implement and investigate XDP are studied in order to provide a better understanding of the abilities of this relatively new technology.

As explained in previous chapter, XDP is actually an eBPF program running on the network device driver context of the kernel. Consequently, primary studies on XDP have focused on the analysis of its impact on packet processing performance. The paper [16], written by principal developers of XDP, extensively covers the technology from different aspects including packet processing performance measurements. It first explains the overall architecture by articulating the details of eBPF, analyzes the performance from different aspects and discusses the current limitations. Additionally, the paper supports the claim of functionality with real life use-cases, such as software routing, inline DOS mitigation and load balancing.

In the study [16], the performance evaluation conducted as a comparison of different packet processing frameworks: Linux kernel networking stack, XDP, and the kernel-bypass method DPDK. The tests pay attention to raw packet processing performance with minimum-sized (64 bytes) packets since the number of packets processed per second is a more relevant metric than the impact of the size of the packets. Furthermore, authors selected the metrics of comparison as packet drop and packet forwarding performance as well as CPU usage. Packet drop tests on one CPU core showed that XDP reached 24 Mpps which is 45% less than DPDK's performance. Nevertheless, it performed almost 6 times better than Linux networking stack which used iptables module to drop packets and reached 4.8 Mpps at maximum. Packet forwarding tests show more interesting results due to two different modes of forwarding with XDP: forwarding to same NIC and different NIC. On forwarding to different NIC case, DPDK performed slightly better than XDP, with 50 Mpps and 40 Mpps throughput respectively with five cores. In CPU usage tests, DPDK used 100% of CPU as expected due the nature of its design that dedicates a full core for processing. Linux networking stack hit the full CPU usage as early as 5 Mpps load while XDP was able to process up to 25 Mpps until reaching 100% CPU capacity.

Another highlight of the study is the decrease of throughput performance in accordance with the XDP functionality under test. Table 1 represents this occurence as the comparison of reported results from different XDP applications tested on a single core. It is clearly seen that the throughput decreases with the added functionality, such as packet forwarding. For instance, packet drop case is the highest performing case since it performs the minimum functionality by simply dropping the received packets. On the other hand, in the routing application case, the performance drops to 3.5 Mpps due to additional costs introduced by the kernel table look-up and the

packet redirection from one physical interface to the other one.

Functionality under test		**Max. packet rate**
Packet Drop		24 Mpps
Packet forwarding	To diff. NIC	9 Mpps
	To same NIC	18 Mpps
Routing	Single table	5,2 Mpps
	Full table	3,5 Mpps

Table 1: XDP benchmark results with single core for packet drop, packet forwarding and router (with kernel routing table look-up) functionalities [16]

Eventually, these measurements have shown that XDP definitely performs more efficiently than Linux network stack in reported cases; however, it still needs more development in order to minimize the gap to DPDK.

Another study [20] presents an evaluation on feasibility of XDP for offloading some of the packet processing functions. For this purpose, the study examines three different offloading options supported by XDP, which are XDP device driver offloading, kernel-bypass with AF_XDP socket, and offloading to programmable NICs. Packet drop tests in XDP driver offload case have supported the claim of achieving 14.88 Mpps on 10 Gbit/s line rate of previous studies. However, adding complexity to offloaded tasks by including checksum calculations decreased the packet drop rate. Nevertheless, driver offload outperformed kernel-bypass with almost double packet rate until their performance converged at 20 checksums. Additionally, programmable NIC got overloaded by checksum calculations very quickly compared to driver offload and kernel-bypass due to its lower processing capacity than host CPU. Another finding in the study is the decrease of throughput when offloading is actualized on VM device driver instead of host driver. This effect results from the overhead of forwarding packets from host to VM driver. On the other hand, if VM driver tasks are offloaded to host driver, it may cause breaking the isolation between VMs sharing the same host. In short, the study showed that offloading by using XDP promises acceleration in packet processing performance in some cases. However, the offloaded tasks need to be kept small and meticulously implemented for each use-case.

As studies have been proving high performance capabilities of XDP, there have been research activities that aim to investigate its usability in design of virtual network functions in order to defeat performance penalties resulting from commodity servers. In this direction, the study [22] extensively evaluates the limitations of eBPF and XDP as well as their advantages in context of creating network functions. Additionally, possible workarounds in order to overcome those challenges are proposed and validated on test environment. As stated by the paper, the main drawback of XDP is the limited number of helper functions compared to other network hook points in Linux kernel. For instance, in the time of writing, XDP lacks the clone

helper function that allows a packet to be forwarded to multiple ports at the same time, which is a feature needed by several network functions, such as bridge and router [22]. Nevertheless, this functionality is currently under active discussion of kernel developers and planned to be supported in future releases [48]. Aside from the challenges, the study validated the improvement of throughput both in native XDP and generic XDP cases, with 65% and 25% performance increase respectively. It is observed that the consumption of CPU has decreased as well with native XDP.

Another study published quite recently [15] has proposed a hybrid architecture for high-performance VNFs by decoupling packet-processing operations into fast path and slow path through leveraging XDP and kernel-bypass respectively. This approach resembles to OpenFlow since a packet takes the fast path and is processed in an eBPF program if it matches a defined rule, similar to OpenFlow switch operation. Otherwise, it is forwarded to user-space for further processing via slow path, like forwarding to OpenFlow controller. However, these two approaches differentiate in terms of scale. The advocated VNF architecture is to be implemented on one host machine, while OpenFlow is implemented through the whole network. Additionally, utilization of XDP proposes shorter communication delay between the forwarder and controller, and provides more flexibility in programming compared to OpenFlow mechanism. The study validates the impact of the hybrid architecture by implementing and testing on network functions, such as load balancer, firewall, and deep packet inspection. The evaluation is performed both on single VNF setup and multiple service-chained VNF setup. The results showed that the proposed architecture with XDP scales efficiently with the number of CPUs; in addition, it is observed that the throughput and latency performance increased. Lead by the promising results, the study suggests more research for implementing such architecture to container-based VNFs as well.

Complementary to reported initiatives of adopting XDP in VNF design, native XDP support for veth devices has been recently added to kernel (since version 4.19) as described in conference paper [48]. Inclusion of native XDP support for virtual interfaces paves the way for accelerated cloud-native network functions as well as service chaining through XDP. Prior to this inclusion, virtual interfaces only supported generic XDP programs which results in the `sk\_buff` allocation which eliminates the fundamental advantage of native XDP. In order to overcome this effect, the native XDP implementation for veth devices incorporates XDP packet buffers and NAPI handler in the veth device driver. Such approach enables redirection of packets between the XDP-loaded physical and veth interfaces without `sk_buff` allocation; thus, boosts the overall packet processing performance.

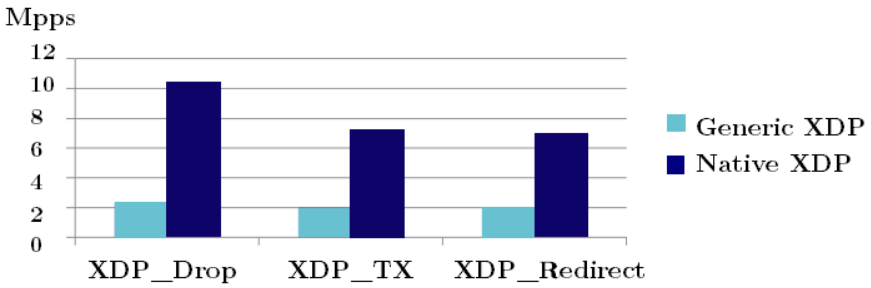


Figure 7: XDP performance on veth device with single core [49]

Finally, the developers of the solution validated the performance enhancement on veth devices with native XDP compared to generic XDP. However, the XDP_TX and XDP_REDIRECT performance on the veth device seems to decrease, shown in Figure 7, when it is compared with the performance of physical interfaces reported on previously discussed studies. For instance, redirect action hits approximately 6 Mpps on veth interface, while physical interfaces are noted as performing up to 9 Mpps in redirect action.

Apart from its usability for network functions, XDP has been adopted in a diverse set of applications. One of the remarkable Open Source initiatives leveraging eBPF and XDP is the Cilium project [50] which is a container networking solution providing enhanced security and monitoring capabilities. Cilium benefits from XDP in order to act very quickly on malicious and unforeseen traffic flows.

As a result of being at the earliest entry point of the host network, native XDP has been under the spotlight for many other security solutions to mitigate the effects of malicious traffic. Cloudflare used XDP for DDoS mitigation [19], while Suricata utilized it to build a network threat detection engine [51].

On top of these, XDP has been utilized in various applications, such as a DNS traffic manager [52] and packet steering engine [21]. Furthermore, Facebook has an open source project, called Katran, which implements a high-performance Layer 4 load balancing forwarding plane by leveraging eBPF and XDP [53].

A very distinctive XDP implementation [23] enables in-network control of industrial machineries by synthesizing P4 programming with XDP offloading on SmartNICs. Authors reported that using XDP provides flexibility in deployment and gives possibility to developing custom protocols for a specific control problem.

4 Implementation and Measurements

In this chapter, the implementation of the XDP forwarding plane prototype and its integration to test environment is explained in detail. In tandem, the outlook of the test environment and the practice of running the test cases are described.

4.1 Implementation of XDP Forwarding Plane

To begin with, the implementation aims to provide a prototype in order to evaluate the usability of XDP technology in the novel container-based architecture of the virtual network functions in edge-cloud scale. Thus, the fundamental attribute of the implementation is the integration of XDP programs that perform forwarding actions on virtual interfaces. The implementation is designed to enable an environment for straightforwardly testing the fundamental considerations of network function performance. Such considered matters are the ability to process high-throughput traffic while utilizing processor power efficiently, usability of existing kernel functionalities within system, programmability from user-space, and complexity.

4.1.1 Architecture of the prototype

Following the fundamental design considerations, the prototype is designed as a separate data forwarding plane laying under the user-plane which is assumed to be inhabiting containerized network functions, such as encapsulation, header modification and deep packet inspection, and a container orchestrator.

The fundamental motivation to keep forwarding functions in a separate plane instead of implementing this functionality within the containers, like for instance a containerized router, is to leverage the routing tables already available at host kernel. Using the host kernel tables gives a unified reachability throughout the system's interfaces and the networks connected to those interfaces. Otherwise, a routing or forwarding application within a container would require additional workarounds to access the routing information of overall system due to the fact that the containers have access to interfaces and routing information only within their own network namespace.

Considering the scope of this thesis, development of such user-plane level network functions and incorporating a container orchestrator is not included in the implementation. Thus, the prototype implements solely the XDP forwarding plane and tests its functionality.

As a requirement of the main purpose of the research, a container instance is placed in the host in order to emulate a network. For the container implementation, a Docker container is used due to its practicality and the convenience of the Docker API command tool options. Furthermore, the container is attached to the host network

through veth pairs. Veth pairs are virtual network interfaces that function like a virtual cable between the attached network namespaces; thus, they are commonly used in order to enable networking for containers. On top of that, veth pairs support native XDP programs since kernel version 4.19.

As seen in the Figure 8, the network interfaces facing the host kernel namespace, including both physical and virtual interface (veth0) constitute together the integrated components of an XDP forwarding fabric. All the network interfaces are attached with an XDP program relevant to their role in the system.

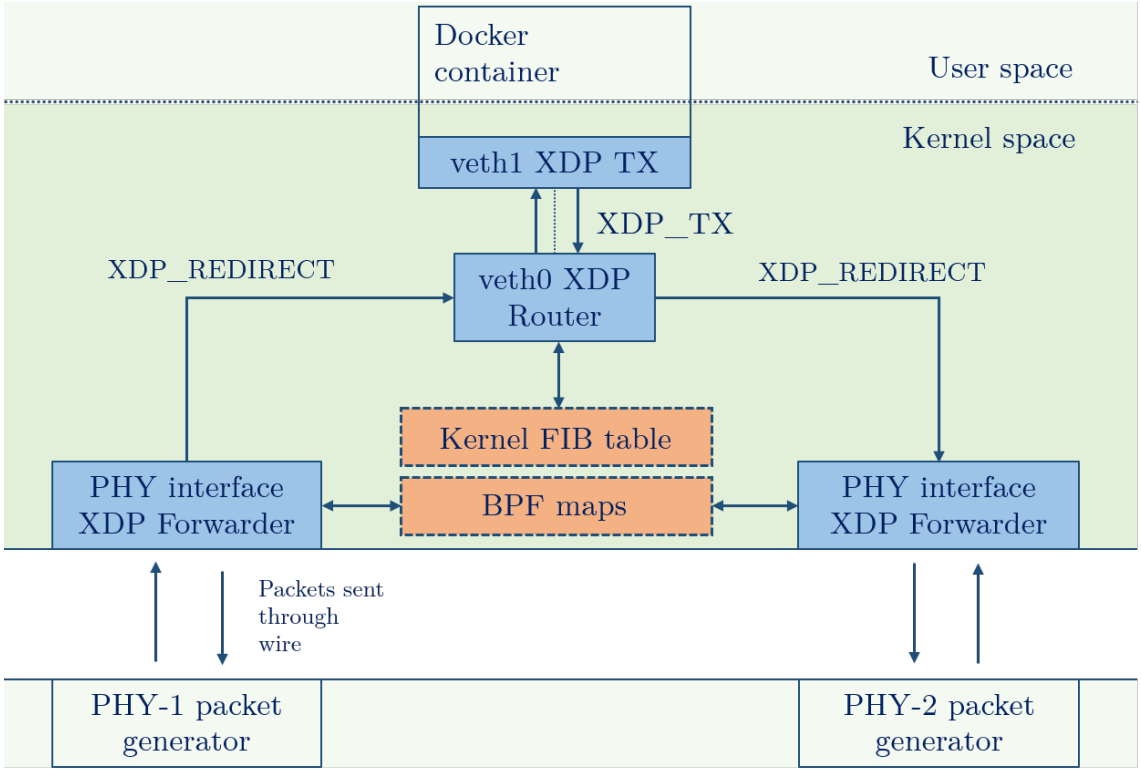


Figure 8: Prototype overview

Physical interfaces of the host machine are attached with XDP Forwarder program that forwards incoming packets to the virtual interface based on the source ethernet address of the packets. The purpose of the source-based forwarding of these forwarder programs is to redirect the incoming traffic to relevant network processing pipeline; thus, they are not intended to make verdict on the packets. In order to forward packets to relevant container, the XDP program performs a look-up to the assigned BPF map; then, sends the packet to designated interface by executing XDP_REDIRECT action. The BPF map, which can be accessed by both of the physical interfaces, has been configured from the user-space by using `bpftool` to store the information regarding to source MAC address and destination MAC address of the next hop. `bpftool` is a command line facility to view and update BPF elements, such as BPF programs and maps [54].

```
# bpftool map update id <map_id> key <source_MAC_address> \
   value <destination_MAC_address>
```

After the successful execution of forwarding from physical interface, the packets are received in the veth peer inside the container (veth1). Here, further network functions could be implemented on the packets. However, since network function development is not within the scope of this work, the packets are directly sent back to the ingress interface without any modification.

In order to send the packets back to the ingress interface, the XDP_TX action has been utilized. The packets sent to TX queue of the inner veth (veth1) arrives to the RX queue of the outer veth (veth0) interface due to the nature of veth pairs. The packets arriving to RX queue of the veth0 triggers the XDP hook attached on the veth0. The XDP Router function attached on this veth operates the actual routing functionality by performing a route look-up on the routing tables of the host kernel. The kernel routing tables are entities that can be configured from user-space and by routing daemons. In this prototype, the routes are statically configured, and the dynamic routing protocols are not included.

After performing routing table look-up, the XDP Router program determines the destination interface and transmits the packets to the TX queues of the designated interface by executing the XDP_REDIRECT action.

4.1.2 The XDP programs in the forwarding plane

In this section, an overview of the XDP programs implemented in this study is given including structural details, such as the context objects, object structures and helper functions. The XDP programs implemented in this study are derived from XDP tutorial codes [55] provided by XDP developers.

An XDP program code can enclose more than one XDP program defined in separate sections. Thus, the common parts of the code are presented first; later, the sections of the code that separate forwarding, routing, and TX functions are explained. Additionally, it should be noted that each network interface can be loaded with one XDP program at a time, which is defined at program loading phase by giving the section name as a parameter to BPF loader.

First of all, the XDP program code starts and ends with SPDX License Identifier that remarks the source code under the GNU GPL 2.0 license. Then, the BPF libraries that contain the required headers and functions are included.

```
1  #include <linux/bpf.h>
2  #include <linux/in.h>
```

```
3  #include <bpf/bpf_helpers.h>
4  #include <bpf/bpf_endian.h>
```

The BPF maps are defined under a separate maps section in the SEC("maps") format. The SEC helper is a macro identified in `bpf_helpers.h` header file, and is used for defining program sections, map objects and open-source license in distinct parts. Thus, the ELF BPF loader can separately interpret each section [56].

```
1  struct bpf_map_def SEC("maps") redirect_params = {
2      .type = BPF_MAP_TYPE_HASH,
3      .key_size = ETH_ALEN,
4      .value_size = ETH_ALEN,
5      .max_entries = 1,
6  };
```

Each executable XDP programs starts with SEC macro where the program name is placed. Then, the program is defined as a function that takes the XDP context object as the function argument. The XDP context object in fact represents the packet received by the XDP hook.

```
1  SEC("xdp_redirect_map")
2  int xdp_redirect_map_func(struct xdp_md *ctx)
```

XDP context object, xdp_md identifier in the source code, is defined as a struct in linux/bpf.h header file as mentioned in Chapter 2

The program continues with a casting operation that assigns the data fields in context object to pointers. Casting means conversion from one data type to another type in programming. This data conversion is needed in order to pass the verifier which checks pointer access to the packet data [57].

```
1      void *data_end = (void *)(long)ctx->data_end;
2      void *data = (void *)(long)ctx->data;
```

Later, the structs and variables, such as Ethernet header type and header pointer, which are required for parsing and storing packet data are created accordingly.

```
        struct hdr_cursor nh;
        struct ethhdr *eth;
        int eth_type;
        int action = XDP_PASS;
        unsigned char *dst;

        /* Parse Ethernet and IP/IPv6 headers */
        eth_type = parse_ethhdr(&nh, data_end, &eth);
        if (eth_type == -1)
                goto out;
```

XDP Forwarder Program

The XDP Forwarder program implemented in this study determines the egress interface by performing a look-up on the attached BPF map. The look-up is realized by the `bpf_map_lookup_elem` function that takes two arguments, the address of BPF map and source Ethernet address of the packet; then, it returns the corresponding value in the map. The information acquired from the map look-up is copied to the destination header of the packet. Eventually, the final verdict on the packet is returned at the end of the program. In successful look-up, the final verdict is XDP_REDIRECT action; otherwise, the default action is set to XDP_PASS that passes the packet to Linux kernel network stack. The packets captured by this program are processed in bulks due to utilization of BPF maps as explained in Sub-section 2.5.4.

```
        dst = bpf_map_lookup_elem(&redirect_params, \
        eth->h_source);
        if (!dst)
                goto out;

        /* Set a proper destination address */
        memcpy(eth->h_dest, dst, ETH_ALEN);
        action = bpf_redirect_map(&tx_port, 0, 0);

    out:
        return xdp_stats_record_action(ctx, action);
```

XDP Router Program

The XDP Router performs additional IP level packet parsing in order to obtain the source and destination IP address as well as the protocol information. Then,

it runs a FIB look-up function with the acquired parameters to determine the destination interface. Additionally, XDP Router program utilizes bulk processing through the BPF maps. The program executes the `bpf_fib_lookup` helper which calls the `fib_table_lookup()` function with the parsed packet information stored in `fib_params` structure and returns zero in successful look-up. `fib_table_lookup()` function is a system function defined in net/ipv4/fib_trie.c source file in the Linux kernel. Additionally, `fib_params` is a `bpf_fib_lookup` type structure defined in `bpf.h` header file.

```
1   rc = bpf_fib_lookup(ctx, &fib_params, sizeof(fib_params), 0);
```

When the look-up is successful, the function writes the destination Ethernet address field of `fib_params` with the corresponding next-hop ethernet address. Then, the obtained next-hop address is copied to the destination field in the actual packet context object.

```
1               if (h_proto == bpf_htons(ETH_P_IP))
2                       ip_decrease_ttl(iph);
3                else if (h_proto == bpf_htons(ETH_P_IPV6))
4                       ip6h->hop_limit--;
5
6                memcpy(eth->h_dest, fib_params.dmac, ETH_ALEN);
7                memcpy(eth->h_source, fib_params.smac, ETH_ALEN);
8                action = bpf_redirect_map(&tx_port, \
9                fib_params.ifindex, 0);
```

After successfully writing the next-hop address, XDP_REDIRECT action is executed.

```
1   return xdp_stats_record_action(ctx, action);
```

XDP TX Program

This program that run in the container instance in the prototype simply executes XDP_TX action on all of the received packets. XDP_TX action has no bulk processing support since it does not utilize BPF maps.

```
1  SEC("xdp_tx_simple")
2  int xdp_tx_simple_func(struct xdp_md *ctx)
3  {
4          return xdp_stats_record_action(ctx, XDP_TX);
5  }
```

4.1.3 Setting up the XDP programs

After the XDP programs considerately are written in restricted C language, they need to be compiled in to BPF bytecode using LLVM+clang compiler. LLVM is a back-end infrastructure providing libraries, header files, tools, code analyzers and optimizers to convert codes into object files [58]. Clang is a compiler of LLVM that compiles C-like languages. The compiled BPF bytecode is stored an ELF (Executable and Linkable Format) object, which is a standard cross-platform file format for executables.

Prior to attaching compiled programs into kernel, a BPF file system needs to be mounted in order to keep BPF maps under a shared file system. This procedure is called pinning and performed with the following command.

```
# mount -t bpf bpf /sys/fs/bpf/
```

Additionally, the JIT compiler should be enabled via below command in order to increase the performance of program execution.

```
# sysctl -w net.core.bpf_jit_enable=1
```

Following the aforementioned actions, the BPF bytecode is ready to be loaded into the kernel. There are two possible ways to load an XDP program into the kernel: using iproute2 BPF ELF loader tools or writing a custom loader using `libbpf` library functions. iproute2 provides a straightforward loading option; however, it does not support BPF map structure. Thus, it becomes a necessity to implement a custom loader when using BPF maps inside the program. In this study, the BPF loader provided by the XDP project maintainers has been used.

```
# ip link set dev lo xdp obj xdp_prog.o sec xdp
```

The BPF loaders load the programs through bpf() system call. If the program is verified by the kernel, it returns a file descriptor for the program which can be attached to a designated hook point [44]. A successfully loaded and attached XDP program can be observed on the interface through `ip` command as below.

```
# ip link show ens1f0
2: ens1f0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 xdp \
       qdisc mq state UP mode DEFAULT group default qlen 1000
       link/ether 3c:fd:fe:9e:93:78 brd ff:ff:ff:ff:ff:ff
       prog/xdp id 898
```

In a similar fashion, the user-space programs that update BPF map elements are custom-made by using the system-call wrappers that enable look-ups from kernel-space to user-space. In this study, the user-space program configuring BPF map values has been obtained directly from the XDP project source code [59].

4.1.4 Setting up the Docker container for XDP

The Docker container image used for the prototype is the latest version of Ubuntu with additional tools and header files required to run XDP programs, such as `clang` and `llvm`. The running container instance is created with –privileged flag in order to give the container root privileges. Although it is not the best practice to give such extended privileges to containers, such permissions are required to load XDP program on the `veth` device inside the Docker container.

After creation of the container, its PID (process ID) has been linked to the network namespace exclusively created for this container instance [60]. This allows to attach custom veth interfaces instead of using the default veth interfaces that docker creates.

```
# mkdir -p /var/run/netns/
# ln -sfT /proc/$pid/ns/net /var/run/netns/[container_id]
```

Later, the veth pairs and additional network driver settings are configured in accordance with the recommendations [49]. The number of RX and TX queues of veth pairs are set to 24 since it is the maximum available number of cores in the system. Additionally, it is observed that the veth devices use only the number of queues they receive the traffic from. For instance, if the physical network drivers are configured to utilize 4 RX queues, the veth pairs that receive traffic from this physical network driver uses 4 RX queues as well although it has 24 available queues.

```
# ip netns add <container_id>
# ip link add veth0 numrxqueues 24 numtxqueues 24 type veth \
   peer name veth1 netns <container_id> numrxqueues 24 numtxqueues 24
# ethtool -K veth0 tx off txvlan off
# ip netns exec ns0 ethtool -K veth1 tx off txvlan off
# ethtool -K PHY_NIC rxvlan off
# bridge fdb add MAC_OF_VETH1 dev PHY_NIC self # Unicast filter
```

Finally, the XDP_TX program has attached to the configured veth peer inside container with aforementioned BPF program loading techniques.

4.2 Performance measurements

In this section, an overview of the test environment is presented and the measurement process is described. In the following, the test setup is depicted including the features of used devices and their placement in the setup along with the system-tuning configurations which are performed before the measurement process. Later, the measurement process is outlined in detail to justify the selected performance metrics as well as the measurement tools.

4.2.1 Overview of experimental test setup

The test environment consists of two Ubuntu servers and a switch device that connects the servers. The model of the switch is Quanta LY6 equipped with Broadcom Trident 2 chip that provides 1.28 Tbps switching capacity [61].

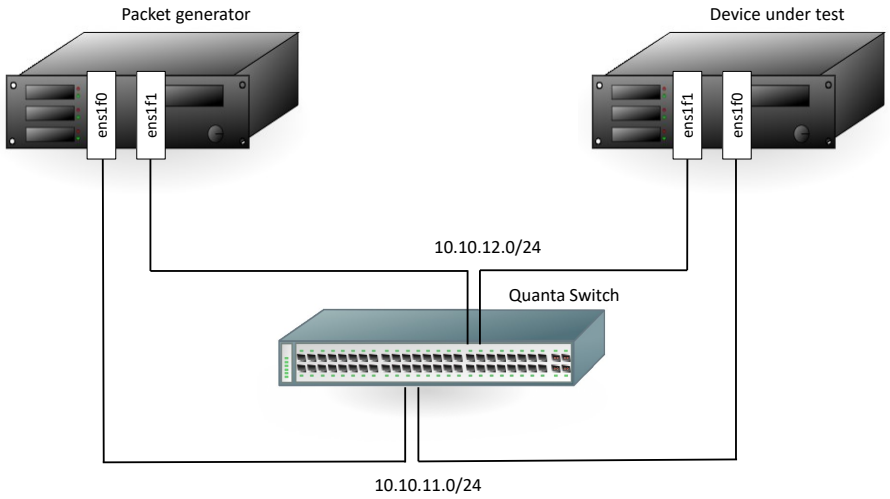


Figure 9: Test setup

Ubuntu servers in the setup have Intel(R) Xeon(R) CPU with 48 cores with hyper-threading. However, hyperthreading functionality is disabled on the server hosting the prototype in order to avoid instability in processor and cache performance. Thus, the system can utilize up to 24 cores. Furthermore, the CPU frequency scaling factor in DUT (device under test) server is set to performance mode which means that

the CPU will be inclined to run at the possible highest frequency to achieve highest performance [62].

```
# echo performance > /sys/devices/system/cpu/cpu0/
                      cpufreq/scaling_governor
```

Both of the servers have NUMA (Non-Uniform Memory Access) system enabled by default. In NUMA systems, CPUs are attached to their own local RAM, which consequently enables fast access from processor to memory [63]. In addition to that, both servers have Intel Data Direct I/O Technology (Intel DDIO) feature enabled which provides direct communication between the NIC device and the host processor cache. Therefore, the NIC device transfers the packets directly into L3 cache. This mechanism boosts the performance significantly due to elimination of the costly visits to main memory [64].

	Packet generator	**Device under test (DUT)**
CPU	Intel(R) Xeon(R) @2.50GHz	Intel(R) Xeon(R) @2.50GHz
NIC	Intel XL710 for 40GbE PCIe v.3 x8, total 64 Gbit/s	Intel XL710 for 40GbE PCIe v.3 x8, total 64 Gbit/s
Memory	DDR4 Synchronous 2133 MHz	DDR4 Synchronous 2133 MHz
Kernel	5.4.0-42-generic	5.4.0-47-generic

Table 2: General features of the test environment

Cache Level	**Cache size**
L1d	768 KiB
L1i	768 KiB
L2	6 MiB
L3	60 MiB

Table 3: Cache levels and sizes in DUT

Additionally, the NIC adapter is connected to a PCIe v3 x8 bus offering full-duplex 64 Gbit/s bandwidth. As this bandwidth is already higher than the maximum traffic rate of 40 Gbit/s supported by the NIC, it is able to transfer the data on wire speed to the main memory.

On the packet generator server side, the packet generation is performed through Netmap packet generator which generates raw network packets [65]. In order to

use netmap packet generator, the kernel is compiled with netmap module and the module is loaded on the network device driver of the corresponding interface.

4.2.2 Variables in the measurements

In this study, the main target is to measure the overall system performance in terms of network throughput and CPU utilization. By the definitions acquired from RFC 1242 [66], throughput corresponds to the maximum data rate processed without any frame loss. Furthermore, frame loss rate refers to the ratio of frames which could not be forwarded to total frames sent in the constant load. Constant load is defined as the traffic of fixed size frames at a fixed rate. In addition to that, CPU utilization refers to the percentage of time that CPU is not idle [67].

To measure these performance metrics in different cases, some variables are decided and configured in accordance with the test case. These variables can be grouped as traffic-related and DUT-related variables.

Traffic-related variables are the packet size, the packet rate as a constant load, and the number of flows. The packet size refers to the total bytes of sent packets including the packet headers and packet data. The netmap pkt-gen utilized in this study sends UDP packets with 14 bytes Ethernet header, 20 bytes IP header, 8 bytes UDP header, and the payload. Packet rate is the number of packets sent from generator to DUT per second. In addition to that, RFC 3917 defines a flow in the variable 'number of flows' as a packet stream consisting of packets with common properties, such as destination IP address, source IP address, destination port, and source port [68].

DUT-related variables are the number of NIC queues, the number of cores, and the queue size. Number of NIC queues are the receive and transmit queues of the interface where the packets are queued for reception or transmission. Number of cores variable is tied to the previous metric since each RX/TX queue pair is served by a core. Queue size is the number of descriptors set for the queue. Descriptor means data segments allocated in memory for the corresponding queue, and there must be an available descriptor for a packet to be received or transmitted.

4.2.3 Data collection instruments

In order to obtain data related to throughput performance, the number of packets received, transmitted and dropped on interfaces are observed through `ethtool` utility. In addition to that, netmap pkt-gen provides the information regarding the sent traffic, such as number of packets, the packet rate, and throughput.

The CPU utilization per core has been measured by `mpstat` tool. For additional system metrics, `perf` utility has been used.

4.2.4 Measurement process

The system has been tested with varying number of CPU cores in DUT and varying volumes of incoming traffic.

At the DUT side, the number of cores to be utilized are configured through `ethtool` command. In essence, this command sets the number of RX and TX queues on the NIC since each queue has a dedicated interrupt which is served by a processor core. An interrupt can be served by a different core each time it is raised, or it can be pinned to a particular core by setting the interrupt affinity. Interrupt affinity refers to the pinning of an interrupt request to a particular CPU core [69]; as a result the IRQ is always raised in the pinned CPU core.

```
# ethtool -L <interface> combined <number of queues>
```

During the measurements, interrupt affinity is set manually after changing the numbers of cores by following the suggestion of the official manual of the NIC device, which is an Intel network adapter from XL710 family with i40e driver. As recommended by the manual [70], the irqbalance service has been disabled and interrupt affinity has been manually set. The irqbalance is a service that distributes the hardware interrupts across the multiple cores to balance the interrupt load [71]; however, manually pinning an interrupt source to a CPU core may provide more deterministic results during tests. Finally, the interrupt affinity has been configured by running the `'set_irq_affinity'` script provided in Intel i40e driver source code [72].

```
#[path-to-i40epackage]/scripts/set_irq_affinity -x local ens1f0
```

In addition to that, Receive Side Scaling (RSS) allows the NIC driver to distribute incoming packets among the available receive queues; thus, among the CPU cores. RSS runs a hash algorithm using the source and destination IP address and ports in order to assign the flows to RX queues with a logic and maintain the packet order [73].

At the packet generator side, netmap packet generator parameters are configured in accordance with the test case. As defined in packet generator manual [65], **-i** sets the execution interface, **-l** sets packet size, **-d** destination IP address, **-s** source IP address, **-D** destination MAC address, **-S** source MAC address, **-f** transmit or receive mode, **-c** number of cores to be utilized for execution, **-p** number of threads to be utilized, **-b** batch size, **-R** number of packets per second to be sent from one thread. At the end of the execution of packet generator, it returns some useful information, such as the number of sent packets, average value of sent packet rate, and duration of the execution.

```
# pkt-gen -i ens1f0 -l 60 -d 10.10.41.1:2000-10.10.41.1:2000 -s \
   10.10.11.2:41000-10.10.11.2:51000 -D <destination_MAC> \
   -S <source_MAC> -f tx -c 24 -p 1 -b 64 -R 1000000
```

For all the measurements conducted for each test case, the test traffic destination address has been set to complete the full data path through the prototype: starting from the physical ingress interface to the container, and then to physical egress interface. Thus, the packet is received and forwarded to container by XDP Forwarder program; then, forwarded to egress interface by XDP Router on container veth peer interface.

Additionally, for every repetition of each test case, the CPU utilization and throughput metrics are collected during 30 seconds as the traffic was flowing. Test cases are performed with 3 repetitions; and in occurrence of outlier values, the repetitions are performed again.

5 Results and Discussion

In this chapter, results of the performed measurements are presented and evaluated. Firstly, the CPU utilization and throughput performance data obtained from initial measurements are discussed; followed by an analysis on the packet drops observed in the system. Later, similar tests have been repeated with different number of flows to reveal the impact of the number of received flows. Throughout these measurements, the system under test has been configured with the default routing table having only the routes for physically attached networks.

Later, the results regarding to the performance tests in the presence of thousands of generated routes, which aim to evaluate the impact of the size of the routing table on the throughput and CPU performance, are presented and evaluated. Finally, an overall evaluation of the tested system and the XDP technology are discussed together with the limitations.

5.1 Analysis of performance at line rate traffic

The network equipment determines the maximum value of data rate the system can receive, and in this test setup the line rate is 40 Gbit/s due to the 40GbE Ethernet card. In order to achieve the highest data rate and observe the resulting performance, the system is tested under traffic with maximum size packets which is 1500 bytes with Ethernet MTU (Maximum Transmission Unit) size 1500. It is possible to achieve line rate of 39.3 Gbps (Gigabit per second) with only 3,3 Mpps (Mega packet per second).

According to measurement results in Table 4, the system has been able to process up to 19.2 Gbps with one core, and it is observed to be dropping packets after exceeding the 1.6 million packets per second. Nevertheless, the system can handle the line rate traffic without any packet loss when utilizing 2 cores at average 55% CPU utilization. Furthermore, the CPU utilization decreases with higher numbers of cores due to distribution of the traffic among multiple processing units.

Num. of Cores	**Throughput (Gbps)**	**Packet Rate (Mpps)**	**Core Util. (%)**
1	19,2	1,6	55
2	39,3	3,3	55
3	39,3	3,3	39
4	39,3	3,3	16

Table 4: Performance with 1500 bytes packet-size

These results have shown that processing line rate is achievable with big sized packets. As stated in previous studies, the packet size has no significant impact on packet processing cost since all the packets are handled by NIC and processor with the same procedure [74]. Another study [37] supports this fact by highlighting the dominance of per-packet cost on system performance. This applies to the XDP programs running in the tested prototype as well, since the XDP programs receive the packets through context objects pointing to packet data and make the forwarding decisions based on the header information regardless of the size of the packet as explained in Chapter 4.

For that reason, in the following test-cases, the evaluation steers the focus towards analyzing the impact of the number of received packets per second by generating the test traffic with smallest packets to achieve higher packet rate traffic, which emulates the most challenging scenario [37].

5.2 Scalability and packet drop analysis

In order to reveal the scalability of the solution, the throughput is tested with varying numbers of RX queues which corresponds to the number of CPU cores utilized for packet processing. Due to the aforementioned impact of the packet size, all the test cases are performed under the traffic with 64 bytes packets in compliance with the minimum possible size for Ethernet frames [75].

The number of RX queues is set up with `ethtool` as described in Chapter 4 in each case for different number of cores. Additionally, the sent traffic from packet generator has been set to have same characteristics for each case. These characteristics are 64 bytes minimum-size raw Ethernet frames, one destination IP address, source and destination port range with value of 1000, and the burst size of 64 packets. The traffic load has been distributed evenly on the RX queues of receiving interface by setting the interrupt affinity manually in addition to RSS.

Figure 10 represents the overview from these tests. The horizontal axis indicates the average CPU usage value of all utilized cores of the corresponding case. The vertical axis gives the packet rate which lead to that amount of CPU utilization.

The overall look shows that the system handles higher packet rates as the number of cores increase. Lower packet rates, up to 3 Mpps, are observed to be producing similar amount of overhead for 3 to 8 cores as supported by the fairly small difference between the CPU usage values below 20%. This results from the fact that in lower packet rates the amount of packets received by each RX queue after equal distribution is closer for different number of cores. Additionally, the number of packets processed during each interrupt is higher at higher packet rates [16]. For example, the 3 Mpps traffic is distributed as 3/6=0.5 Mpps per queue in 6 cores case and 3/8=0.375 Mpps per queue in 8 cores case which results in a proximity in interrupt processing overhead. On the other hand, for the same number of cores, 6 and 8, the 12 Mpps traffic is distributed as 2 Mpps and 1.5 Mpps per queue respectively, resulting in a

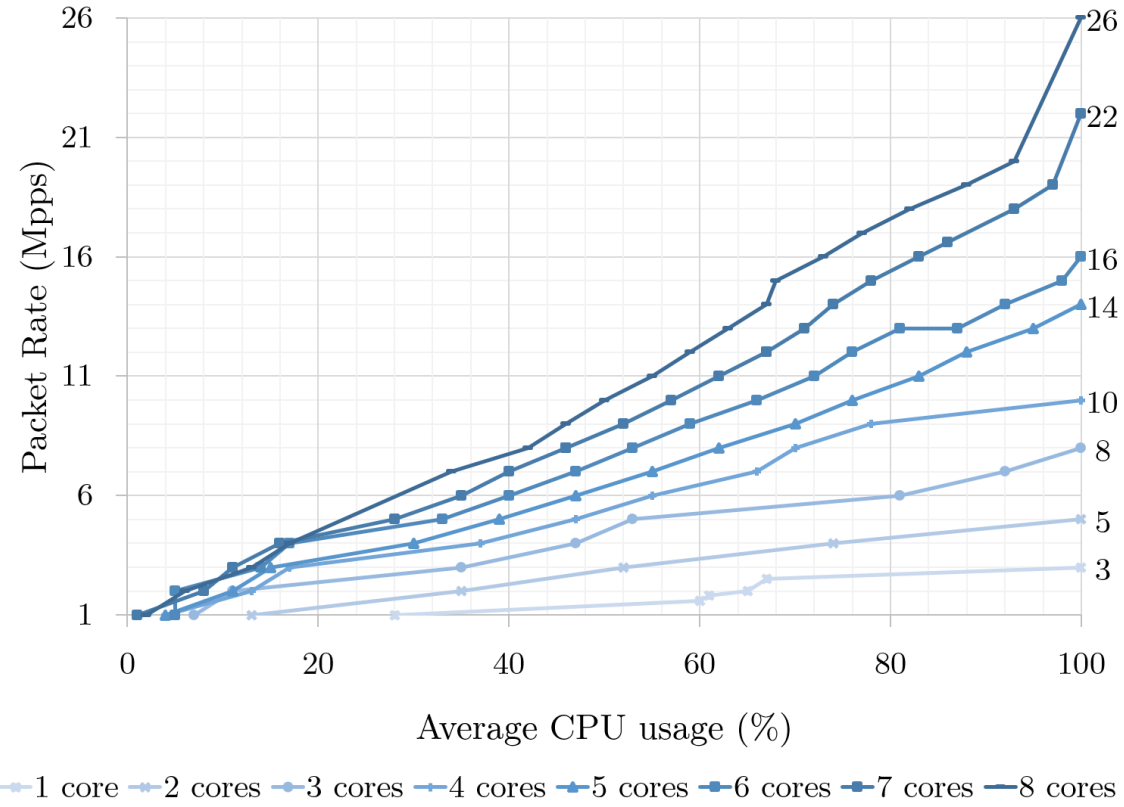


Figure 10: Maximum packet rate with corresponding CPU utilization per number of cores (RX queues)

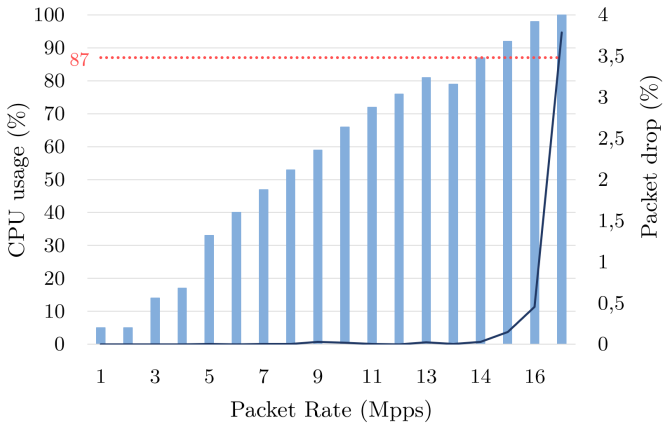
CPU utilization difference around 17%.

These results indicate that the system is scalable for higher packet rates with utilization of more numbers of cores. However, a notable observation during the tests has been the increasing packet drop rates on the receiving physical interface as the incoming traffic rate has been increasing.

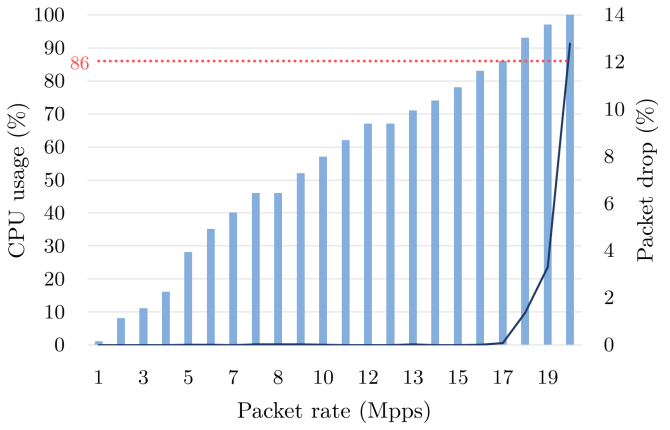
Before moving further with packet drop analysis, it is essential to highlight that the packet drops have been seen solely on the receiving physical interface, which means that all the packets successfully received by the system have completed the data path from XDP Forwarder on physical interface to XDP Router on veth interfaces and to finally intended destination point. Thus, the packet drop metrics analyzed in the following are obtained from the receiving physical interface.

Sent	**Packet loss (%)**							
(Mpps)	1 Core	2 Cores	3 Cores	4 Cores	5 Cores	6 Cores	7 Cores	8 Cores
1	0	0	0	0	0	0	0	0
2	0,006	0	0	0	0	0	0	0
3	03533	0,0006	0,0028	0,0004	0	0	0	0
4		0,0027	0,0003	0	0,0002	0	0,0007	0
5		0,0040	0,0024	0	0,0022	0,0015	0,0040	0,0007
6		0,6900	0,0058	0,0008	0,0010	0,0007	0,0045	0
7		14,7782	0,0035	0,0031	0,0260	0,0039	0,0017	0
8		25,3502	0,0884	0,0057	0,0230	0,0011	0,0255	0
9			4,9270	0,0080	0,0093	0,0280	0,0297	0
10			14,3765	0,0234	0,0345	0,0164	0,0110	0,0004
11			22,3144	0,4476	0,0057	0,0017	0,0010	0,0002
12					0,0299	0,0009	0,0017	0,0016
13					0,6580	0,0042	0,0311	0,0025
14					2,6065	0,0288	0,0030	0,0038
15						0,1478	0,0020	0,0063
16						0,4534	0,0200	0,0033
17						3,7882	0,0897	0,0319
18							1,3561	0,0145
19							3,2971	0,0031
20							12,7759	1,1355

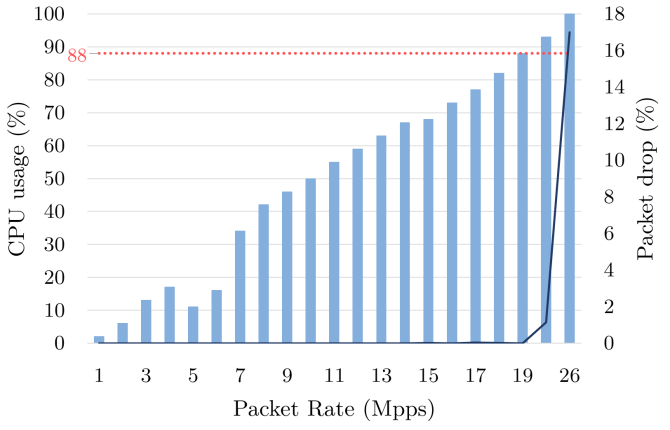
Table 5: Packet drop rates regarding to traffic rate for 1 to 8 cores



(a) 6 cores



(b) 7 cores



(c) 8 cores

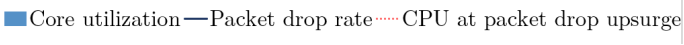


Figure 11: CPU utilization and packet loss rate by packet rate. (a) 6 cores, (b) 7 cores, (c) 8 cores.

Table 5 presents the packet loss percentage with corresponding traffic rate for number of cores from 1 to 8. From the table, it is possible to observe the packet rate where the first packet drop occurs. The outlook of the table which is quite similar to a left triangular matrix indicates that the packet loss starts occuring when each RX queue receives approximately 1 Mpps traffic. This pattern may suggest that the queues are not served fast enough after 1 Mpps which results in insufficiency of RX descriptors in the receive buffer and consequently drop of the packets. In such a case, NIC drops the packets without raising an interrupt for the incoming packets; thus, the CPU is not affected by those packets [74].

This occurrence corresponds with the observations of the study [16], which is depicted in Table 1 in Chapter 3, showing the decrease in throughput performance with added functionality. In the reported study, XDP router running on physical interfaces has reached to approximately 5.2 Mpps with single table, while in the prototype tested in this research has shown packet drops as early as approximately 1 Mpps per core. One explanation to such decrease is the concatenated execution of XDP Forwarder and XDP Router programs, resulting in a summation of BPF map look-up, packet redirecting, and FIB table look-up costs.

In addition to that, Figure 11, shows the packet drop rate with corresponding packet rate and CPU utilization under that traffic for 6 to 8 cores. In these plots, the CPU usage at the point where packet drops upsurge is plotted as a horizontal line. The common CPU usage value at the upsurge point, around 87% CPU utilization, indicates that the system undergoes a state where it is incapable of processing more packets even though the cores have approximately 13% idle cycles.

A possible solution to decrease the packet loss could be to increase the size of the RX queues, due to the fact that the NIC level drops may result from the unavailability of receive queue descriptors [76]. In order to see the impact of this metric, the RX queues are configured with size of 256 and 4096 to be tested with 19 Mpps traffic with 8 cores. The results in Table 6 show that increasing the RX queue size has increased the packet loss rate as well. This probably results from the increasing latency with higher queue size, which affects the number of processed packets per cycle. On the other hand, in smaller RX queue size, the driver faces the packet loss due to unavailability of ring buffers. Thus, changing the RX queue size from the default value, 512, has shown no improvement on the packet loss rate in that case.

RX ring size	**Packet Rate**	**CPU %**	**Packet drop %**
256	19	88,5	0,2007
512	19	88,41	0,0254
4096	19	100	8,7496

Table 6: Impact of RX ring size on performance in 8 cores case

Another remarkable observation has been the increase in utilization of `ksoftirqd/n` threads. `ksoftirqd/n` threads are scheduled to complete the unserved soft interrupts when these interrupts are not served as fast as they arrive [77]. Below is an instance of `perf` statistics, collected at 10 Mpps when 4 cores are utilized, showing the system utilization of individual functions and the responsible parent process running the function. While the ksoftirq/n utilization is almost zero in a system that serves sufficiently, in this instance utilization of ksoftirqd/1 of CPU(1) is already up to 1.67%. This verifies that the system is not able to serve the incoming traffic as fast as it arrives.

```
# perf record -C 0-3 sleep 1
# perf report
...
Overhead  Command     Shared Object               Symbol
7.39%  swapper       [kernel.kallsyms]           [k] fib_table_lookup
4.58%  swapper       bpf_prog_xdp_router_func    [k] bpf_prog_xdp_router
3.22%  swapper       [kernel.kallsyms]           [k] __htab_map_lookup
2.79%  swapper       [kernel.kallsyms]           [k] veth_xdp_rcv_one
2.76%  swapper       [kernel.kallsyms]           [k] i40e_clean_rx_irq
2.45%  swapper       [kernel.kallsyms]           [k] bpf_ipv4_fib_lookup
2.11%  swapper       [kernel.kallsyms]           [k] veth_poll
1.95%  swapper       [kernel.kallsyms]           [k] dev_map_enqueue
1.74%  swapper       [kernel.kallsyms]           [k] i40e_xmit_xdp_ring
1.67%  ksoftirqd/1   [kernel.kallsyms]           [k] fib_table_lookup
```

Listing 1: perf profiling tool report for the overhead per function

On top of that, `perf report` reveals that the impact of XDP Router program on veth device becomes prominent as complementary functions of XDP router, such as `bpf_prog_xdp_router`, `__htab_map_lookup`, `bpf_ipv4_fib_lookup`, and especially `fib_table_lookup`, take the top places in CPU consumption rating. Moreover, `veth_poll` and `veth_xdp_rcv_one` functions, which belong to veth driver and are responsible from collecting packets from veth queues, are observed to consume high CPU power as well. These CPU consumption rates indicate that the XDP Router function together with veth interface becomes the bottleneck of the system.

This occurence may enlighten the packet drop issue on physical ingress interface as well. Due to the fact that the XDP Router and veth device bundle serves the queues slower than the XDP Forwarder program on the ingress interface. Consequently, XDP Router program picks up the packets from the queues more slowly. As a result, the ingress RX queues start dropping incoming packets since there is no available space in RX queues.

In addition to that, examination of the system efficiency in terms of CPU utilization by metrics such as cache-misses [78] and instructions per cycles [74] can reveal further

implications regarding to the data plane performance. Cache-misses emerge when the data needed for an instruction cannot be found in the cache levels; therefore, resulting in latency in completion of execution. Instruction per cycle is another program performance defining metric which corresponds to the instructions executed within one CPU cycle.

```
# perf stat -C 0-7 -e cycles -e  instructions -e cache-references \\
                -e cache-misses -r 15 sleep 1
Performance counter stats for 'CPU(s) 0-7' (15 runs):

    22,716,021,975      cycles
    46,323,579,339      instructions    # 2.04  insn per cycle
       128,580,967      cache-references
           304,673      cache-misses    # 0.237 % of all cache refs

         1.0007954 +- 0.0000207 seconds time elapsed  ( +-  0.00% )
```

Listing 2: perf profiling tool statistics for CPU cores 0-7

In case of 19 Mpps traffic with 8 cores, instruction per cycle rate has been measured to be 2.04 which is an acceptable rate considering the benchmark results of different packet processing systems [79]; thus, it represents no fundamental bottleneck in this case. Additionally, the amount of cache-misses is not dramatically high; thus, it does not explain the observed packet loss.

At this point, further performance and CPU utilization improvements regarding to XDP Router program require detailed analysis and optimization in terms of the consumption of cycles, cache utilization, interaction with device driver and so forth. However, the required code optimization is not performed since it is not within the scope of this study.

5.3 Impact of number of flows

The system has been tested with different numbers of flows in order to observe the behavior under changing flow loads. The testing was conducted with flow numbers starting from 1 to 10000 flows increasing logarithmically. For each flow case, the tests are performed with 1 core, 2 cores, 4 cores and 8 cores with increasing traffic rate until the system is fully-loaded. RX queue size is kept at default 512 and sent packet size is kept at 64 bytes. As highlighted in Chapter 4, the system has been configured with RSS in order to distribute incoming traffic evenly among the RX queues.

Single flow tests have shown that the traffic is not distributed among available

RX queues and only one RX queue has been utilized. This results in the same performance as if there is one available core although there were available 2, 4, and 8 numbers of cores. Thus, the system has performed packet processing using only one core and other available cores have stayed idle.

	Core Utilization (%)			
Packet Rate	1 core	2 cores	4 cores	8 cores
1 Mpps	30	33	33	33
2 Mpps	69	68	68	68
3 Mpps	100	100	100	100

Table 7: CPU utilization of different numbers of cores with 1 Flow

Similar behavior has been seen with 10 flows case which has resulted in uneven distribution of the traffic, and the system has processed packets with only one core. Nevertheless, flow numbers above 100 provided a more remarkable impact on distribution of the traffic among available queues; consequently, they have affected the overall performance positively.

	Core Utilization (%)		
Packet Rate	2 cores	4 cores	8 cores
1 Mpps	16	7	7
4 Mpps	68	32	13
6 Mpps	99	52	22
8 Mpps		65	36
10 Mpps		85	45
12 Mpps		100	59

Table 8: Results for 100 Flows

The results from 1000 and 10000 flows tests have indicated that higher numbers of flows lead to fairer distribution of the traffic among the RX queues; thus, introducing a better CPU utilization and slightly less packet loss. For instance, in case with 8 cores in Table 9, the system performance has stabilized after 1000 flows while it was fluctuating heavily in lower flow rates. The very high packet rates observed in 10 flows case is an impact of the inability to distribute packets to available RX queues.

	CPU Utilization (%)		Packet loss (%)	
Flows	8 Mpps	12 Mpps	8 Mpps	12 Mpps
10	19	22,5	42,5348	55,0046
100	36	59	0,0262	0,0114
1000	38	60	0,006	0,0049
10000	37	59	0,0041	0,0047

Table 9: CPU utilization and Packet loss comparison per flow number for 8 cores

5.4 Impact of the routing table size

The measurements until this point have been conducted in the case where there have been only 9 routes consisting of the default gateways of attached network interfaces and the loop-back address of the host. Thus, with the aim of revealing the cost of FIB look-ups, the system has been tested with a bigger routing table.

To increase the number of routes in the routing table, a bash script is run to populate routes indicating next-hop address towards a few of the interfaces in the neighbor table. In the end, the routing table has grown to have 10,678 route entries. Although the numbers of routes in a BGP (Border Gateway Protocol) router could be up to 814,000 by the time of writing [80], the generated amount of route entries is considered to be sufficient for the target of the study and the scale of the implementation.

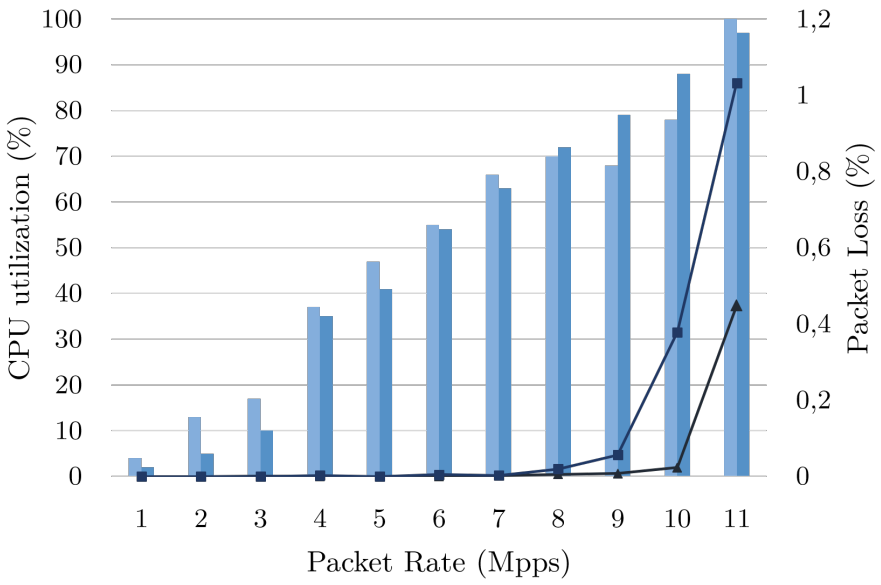
```
1  for i in {20..40};
2  do
3  for j in {1..254};
4  do
5  ip route add 10.10.$i.$j/32 nexthop via 10.10.13.2 dev ens1f1;
6  done;
7  done
```

After increasing the number of routes in the routing table, the system has been tested with two different cases. In the first case, the test traffic has sent with only one destination address; thus, the `fib_table_lookup()` function searches for the same address each time. At this point, it should be noted that the route caching functionality has been removed from Linux kernel since version 3.6 [81]. Thus, the look-up function executes from scratch each time and searching for the same address would result in approximately the same time duration for the look-up of each packet

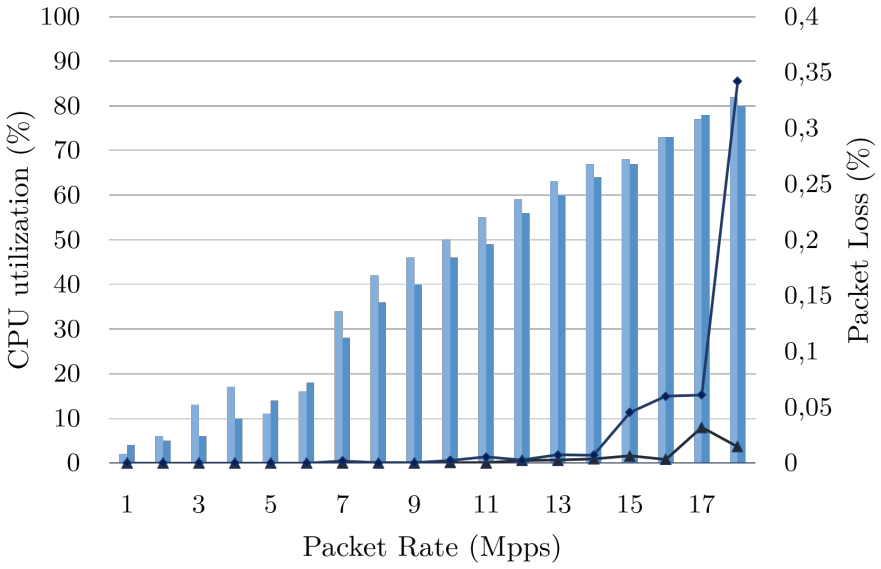
[82]. In the second case, test traffic has been generated to have multiple destination addresses to be searched from the table. Each case has been tested with 2,4 and 8 cores with increasing packet rate, while the flow numbers are kept around 10000 since it provides an even traffic distribution among RX queues.

Figure 12 represents the measurements results from the single destination address case in the bigger routing table. In the figure, the results from the measurements in Subsection 5.2 are used as a reference to see the impact of routing table size since those tests have been performed to look-up for a single destination address in a smaller routing table. The comparison shows that the CPU utilization has not changed remarkably by the increasing number of routes in the routing table. However, the packet drop rate has increased significantly for the same packet rates. This may result from the increase in the time spent for `fib_table_lookup` function since it may traverse more routes to find the right match compared to a table with less routes. Thus, the increasing time spent for each packet causes longer queuing time in receive queues, and consequent packet drops.

In the following, the system has been tested under the traffic directed towards multiple destination addresses on the same bigger routing table. Table 10 shows a comparison of the results with the previous case in order to observe the performance shift when there are multiple different destination addresses to look-up in the table. According to the results, CPU utilization has increased slightly while packet drop rate has surpassed the values from single destination address case in many of the different traffic rates. In the previous case where the `fib_table_lookup()` function has been searching for the same address for each packet, the duration for look-up function should be approximately the same since the address searched for is always at the same location at the routing table. In contrast, the time spent to find a destination address for different addresses would vary regarding to their location on the routing table since the CPU hits a different location in memory for different addresses [82]. Thus, it may result in longer processing time and consequent packet drops on the NIC.



(a) 4 cores



(b) 8 cores

Figure 12: CPU util. and Packet loss comparison between small routing table and bigger routing for (a) 4 cores, (b) 8 cores. Traffic towards single destination address.

	Single Destination		Multiple Destination	
Sent (Mpps)	**Core util. (%)**	**Packet loss (%)**	**Core util. (%)**	**Packet loss (%)**
1	4	0	4	0
2	5	0	9	0
3	6	0	11	0
4	10	0	7	0,0006
5	14	0	10	0,0043
6	18	0	21	0,0020
7	28	0,0018	33	0,0030
8	36	0,0003	41	0,0090
9	40	0,0003	46	0,0594
10	46	0,0025	51	0,0004
11	49	0,0055	54	0,0009
12	56	0,0025	58	0,0084
13	60	0,0073	62	0,0043
14	64	0,0071	65	0,0521
15	67	0,0453	70	0,1441
16	73	0,0599	74	0,2478
17	78	0,0611	78	0,7774
17,8	80	0,3420	82	1,0440

Table 10: Results on bigger routing table when traffic towards single destination address and multiple destination address (with 8 CPU cores)

5.5 Comparison with Linux kernel networking stack

In order to validate the performance gain achieved with XDP, the same data path scenario is tested with bare Linux kernel networking stack without inclusion of any XDP components. To replicate the similar scenario with solely Linux network stack, network namespaces have been utilized. Figure 13 depicts the data path implementation in which there are three separate network namespaces. Two of the namespaces are hosting one physical interface and one peer virtual interface, while

the third namespace is hosting the other peers of veth interfaces. In each network namespace, the default gateway address is defined with the designated next-hop address. For instance, in `netns f0` the default gateway address is defined as `default via <IP address of veth2> dev veth1`. Additionally, each of the veth pairs are created with 24 RX and 24 TX queues.

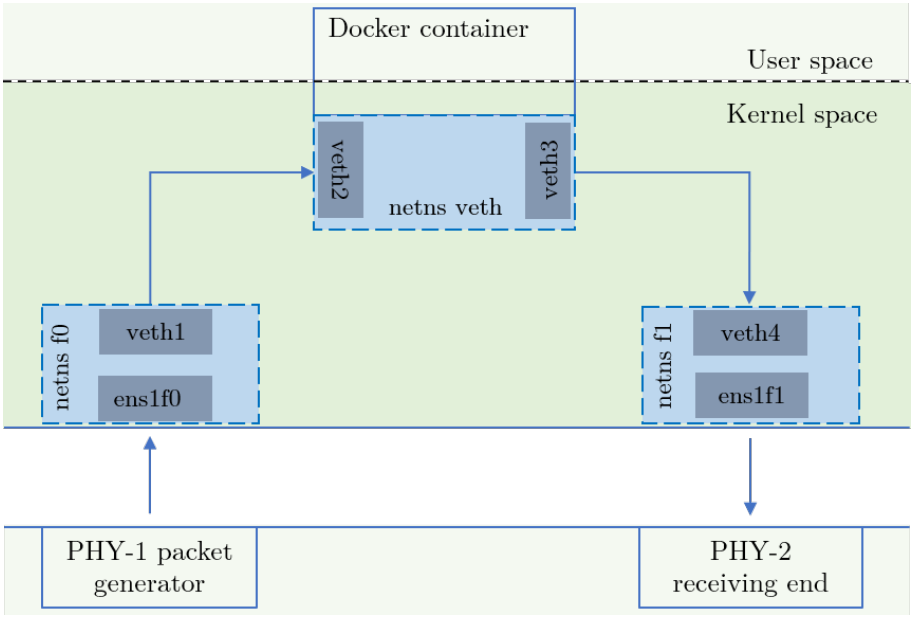


Figure 13: Linux kernel implementation of test scenario using namespaces

The system is tested with numbers of CPU cores from 1 to 8 with gradually increasing incoming packet rate. As reported in Table 11, in single CPU core case, Linux data path processed only 0.4 Mpps without any packet loss, and utilized the 76% of the CPU core. Furthermore, Figure 14 clearly shows that the XDP solution has outperformed Linux network stack data path with more than 50% improvement.

Number of CPU Cores	**Packet rate (Mpps)**	**CPU Util. (%)**
1	0,4	76
2	0,8	72
4	1,4	64
8	3	71

Table 11: Linux networking stack results with 1,2,4,8 CPU cores

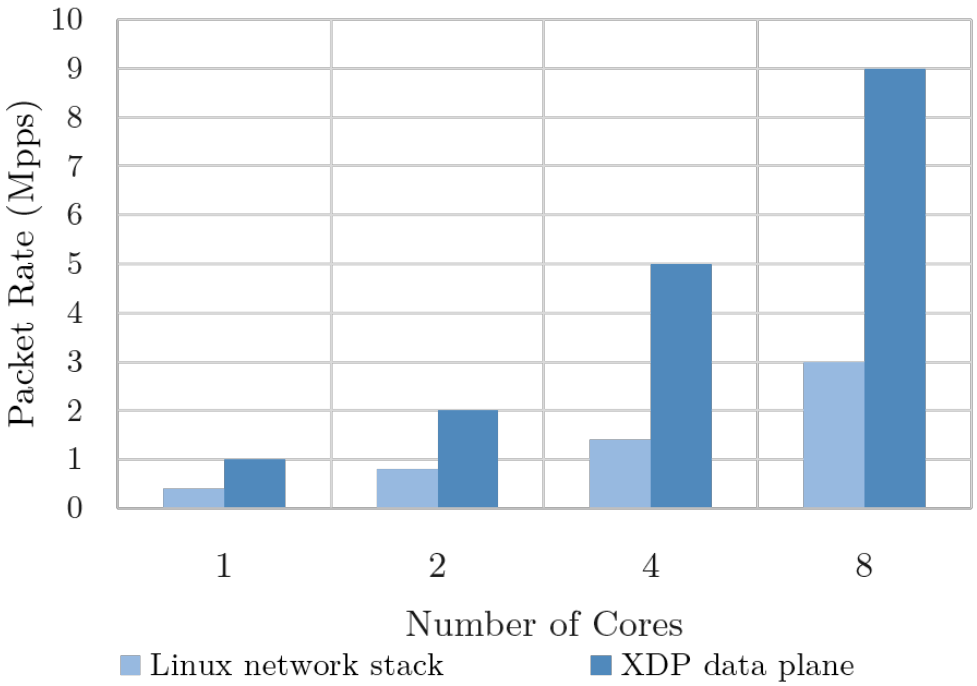


Figure 14: XDP data plane comparison with Linux kernel network stack implementation

5.6 Evaluation of the results

In overall, the presented results have provided an understanding of the behavior of XDP in a data-plane implementation and its interaction with the different elements of the prototype. The main highlights of the observations have been in the areas such as CPU utilization and scalability of the solution, co-dependency with NIC device driver, and program code efficiency.

Firstly, the findings have confirmed that the XDP-based solution is scalable with the number of CPU cores. The prototype at its basic implementation, without any further performance tuning and optimization, has handled more than 1 Mpps traffic per queue without any packet loss. When used with larger sized packets, it can achieve line rate of 40 Gbit/s without requiring no more than 4 cores. In addition to that, flow tests have indicated that the system behavior is more stable under higher numbers of flows due to fair distribution of the traffic among the queues.

Another interesting point was that, the routing table size has introduced no significant effect on the system performance and consequently on the throughput. Therefore, it gives confidence on that such a forwarding plane can be extended for larger scale deployments consisting of more associated networks.

Strikingly, the study has proved that XDP provides more than 50% throughput performance gain in comparison to Linux kernel network stack. Furthermore, the

rate of this performance gain increases with increasing number of CPU cores as well, reaching the triple of Linux network stack in 8 CPU cores case. Therefore, XDP solution was able to achieve 1 Mpps throughput per queue without packet loss, while the Linux network stack implementation remained at 0.4 Mpps per queue. On the other hand, 1 Mpps packet rate achieved by XDP is approximately 80% lower than the performance achieved by the router implementation in the previous study [16]. One of the main reasons of such performance gap is firstly the architecture of the implementation. The prototype implemented in this thesis work redirects the packets multiple times throughout the data path, while the reported study runs one routing program that directly sends the packet to egress interface. Thus, the prototype in this study has been more costly due to multiple hops in data path, which was necessary for the target of the research.

Apart from the promising aspects, the packet loss has remained as an unresolved issue in all of the observed cases as reported. Strikingly, all the packet drops have occurred on the ingress physical interface of the system, and all of the received packets have successfully completed the path through veth interfaces until egress interface. Meaning that, there have been no packet drops by the veth interface itself. On the other hand, the XDP Router application has led to the highest execution cost among other programs integrated in the system due to the cost of look-up practice together with the performance overhead of the veth driver. Considering these facts, the XDP Router has been the defining element in overall system performance.

In this context, the code base utilized for this study, being retrieved from a multi-purpose tutorial code-base, has been a limitation for the performance of the system. Nevertheless, obtained results are promising that a tailored and optimized program can result in a more improved performance. In order to highlight a complementary fact, the experimentations and studying the XDP code-base during the study have stressed that understanding the internals of XDP and developing XDP applications requires a solid knowledge of Linux kernel, kernel networking stack and kernel programming. From a network engineer perspective, this aspect together with the lack of sufficient documentation on the inner-workings of XDP may introduce challenges in igniting the development process.

All in all, the scalability and throughput results have been promising and encouraging for further improvements in the domain of fast packet processing for virtualized network functions exclusively in the scale of edge cloud environments. Additionally, the veth interfaces which are generally utilized for containers have no insolvable restrictions on the performance. Thus, XDP as being a quite new technology deserves further attention and implementations in the research of networking solutions.

5.7 Future research

As the study provided a starting point for the development of XDP-based data plane for virtual network functions, it has revealed the areas that requires further

development as well.

For instance, the system designed as an independent utility in the study should be integrated and tested with a cloud-native container orchestrator such as Kubernetes and CNI (Container Network Interface) plug-ins in order to prove the compatibility with cloud-native environments. Furthermore, the prototype was designed in a way to work as a single tenant system and multi-tenancy has not been considered. Multi-tenancy in cloud context corresponds to a system where the resources are shared among different customers; thus, requiring meticulous work for isolation of the resources such as processing unit and network utilization. As single-tenancy is not always the case, multi-tenancy support should be studied and implemented in the design as well.

Additionally, the latency performance, which is not performed in this study, is an essential metric to determine the performance; therefore, further studies are required to evaluate this aspect as well.

Another point that needs to be clarified is the compatibility with different NIC devices and drivers. Since the XDP infrastructure requires the support of the network device driver, its performance may be in correlation with the features of NIC device. Thus, testing the system with different NIC cards would bring valuable insights in terms of system compatibility and performance dependencies.

Finally, XDP has promising features such as XDP_REDIRECT, which is extensively used in this study, and it possibly enables the implementation of service function chaining (SFC) concept in the data path. Service function chaining refers to chaining of network functions to each other based on the tasks required for a traffic flow. Thus, utilization of XDP data plane for SFC could be an interesting research topic.

6 Conclusions

The research aimed to evaluate the feasibility of implementing eXpress Data Path (XDP) fast packet processing framework in container-based virtual network functions to be utilized in the cloud deployments on the edge. In order to conduct the performance evaluation, a prototype of XDP-based data plane for container-based network functions has been designed and implemented.

The prototype has been composed of forwarder and router programs as well as a container emulating a network function. Then, it was observed with several test cases which aimed to reveal the performance of XDP and its potential limitations.

The results obtained from these tests have shown that an XDP-based solution can provide high throughput in different conditions and scale with the number of processing units. Furthermore, the study has demonstrated that the XDP solution has surpassed the Linux network stack in terms of throughput and CPU utilization. Thus, it has potential to provide high-performance networking for the implementations of network functions on commodity servers.

Crucially, the study has uncovered the limitations and bottlenecks of the XDP-based prototype, and reported the potential solutions for further improvement. In addition to that, promising aspects of the XDP technology have been highlighted and the directions for future research has been discussed.

To conclude, the study has investigated the relatively new technology XDP from different aspects and reported its implications in the context of accelerating the data plane for network functions. And the most importantly, the produced output of the this thesis can serve as a compact knowledge base for the researchers of networking community who would aim to further explore the XDP technology.

References

[1] D. Chandramouli, R. Liebhart, and J. Pirskanen. *5G for the Connected World*. John Wiley & Sons, 2019.

[2] s. Ahmadi. *5G NR: Architecture, Technology, Implementation, and Operation of 3GPP New Radio Standards*. Academic Press, 2019.

[3] S. R. Chowdhury, M. A. Salahuddin, N. Limam, and R. Boutaba. "Re-architecting NFV ecosystem with microservices: State of the art and research challenges". *IEEE Network*, 33(3):168–176, 2019.

[4] E. Tittel. "SDN vs. NFV: Similarities and differences". [Online]. Available: https://www.cisco.com/c/en/us/solutions/software-defined-networking/sdn-vs-nfv.html Accessed: Sept. 17, 2020.

[5] K. Gray and T. D. Nadeau. *Network function virtualization*. Morgan Kaufmann, 2016.

[6] "What is cloud computing?". Official web page. [Online]. Available: https://azure.microsoft.com/en-us/overview/what-is-cloud-computing/ Accessed: Dec. 4, 2020.

[7] European Telecommunications Standards Institute (ETSI). "Network Functions Virtualisation (NFV)". [Online]. Available: https://www.etsi.org/technologies/nfv Accessed: 25 Nov., 2020.

[8] Z. Li, M. Kihl, Q. Lu, and J. A. Andersson. "Performance overhead comparison between hypervisor and container based virtualization". In *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, pages 955–962. IEEE, 2017.

[9] R. Nakamura, Y. Sekiya, and H. Tazaki. "Grafting sockets for fast container networking". In *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems*, pages 15–27, 2018.

[10] J. García-Dorado, F. Mata, J. Ramos, P. Santiago del Río, V. Moreno, and J. Aracil. *High-Performance Network Traffic Processing Systems Using Commodity Hardware*, pages 3–27. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[11] T. Zhang, L. Linguaglossa, P. Giaccone, L. Iannone, and J. Roberts. "Performance Benchmarking of State-of-the-Art Software Switches for NFV". *arXiv preprint arXiv:2003.13489*, 2020.

[12] G. Lettieri, V. Maffione, and L. Rizzo. "A survey of fast packet I/O technologies for network function virtualization". In *International Conference on High Performance Computing*, pages 579–590. Springer, 2017.

[13] S. Gallenmüller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle. "Comparison of frameworks for high-performance packet IO. In *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 29–38. IEEE, 2015.

[14] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker. "NetBricks: Taking the V out of NFV". In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 203–216, 2016.

[15] N. Van Tu, J. Yoo, and J. W. Hong. "Accelerating Virtual Network Functions with Fast-Slow Path Architecture using eXpress Data Path". *IEEE Transactions on Network and Service Management*, 2020.

[16] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller. "The express data path: Fast programmable packet processing in the operating system kernel". In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, pages 54–66, 2018.

[17] B. Gregg. *BPF Performance Tools*. Addison-Wesley Professional, 2019.

[18] Cilium Authors. "Cilium github page". [Online]. Available: https://github.com/cilium/cilium Accessed: Sept. 17, 2020.

[19] G. Bertin. "XDP in practice: integrating XDP into our DDoS mitigation pipeline'. In *Technical Conference on Linux Networking, Netdev*, volume 2, 2017.

[20] O. Hohlfeld, J. Krude, J. H. Reelfs, J. Rüth, and K. Wehrle. "Demystifying the Performance of XDP BPF". In *2019 IEEE Conference on Network Softwarization (NetSoft)*, pages 208–212. IEEE, 2019.

[21] P. Enberg, A. Rao, and S. Tarkoma. "Partition-Aware Packet Steering Using XDP and eBPF for Improving Application-Level Parallelism. In *Proceedings of the 1st ACM CoNEXT Workshop on Emerging in-Network Computing Paradigms*, pages 27–33, 2019.

[22] S. Miano, M. Bertrone, F. Risso, M. Tumolo, and M. V. Bernal. "Creating complex network services with eBPF: Experience and lessons learned". In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–8. IEEE, 2018.

[23] J. Rüth, R. Glebke, K. Wehrle, V. Causevic, and S. Hirche. "Towards in-network industrial feedback control". In *Proceedings of the 2018 Morning Workshop on In-Network Computing*, pages 14–19, 2018.

[24] J. Fernandes. "Containers are Linux", 2017. [Online]. Available: https://www.openshift.com/blog/containers-are-linux Accessed: Nov. 29, 2020.

[25] J. Langemak. *Docker Networking Cookbook*. Packt Publishing Ltd, 2016.

[26] "How Computers Work: The CPU and Memory". [Online]. Available: https://homepage.cs.uri.edu/faculty/wolfe/book/Readings/Reading04.htm Accessed: 25 Nov., 2020.

[27] Intel Corporation. "Multi-core Introduction". Online article, Jun 2012. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/multi-core-introduction.html Accessed: Dec. 11, 2020.

[28] R. Parthasarathi. *Computer Architecture*. INFLIBNET Centre, Jul 2018. [Online]. Available: http://www.cs.umd.edu/~meesh/cmsc411/CourseResources/CA-online/chapter/computer-architectureintroduction/index.html Accessed: Dec. 11, 2020.

[29] Intel Corporation. "Intel Core Processor Family". [Online]. Available: https://www.intel.com/content/www/us/en/products/processors/core.html Accessed: Dec. 30, 2020.

[30] Intel Corporation. "Intel Xeon Platinum Processor". [Online]. Available: https://https://www.intel.com/content/www/us/en/products/processors/xeon/scalable/platinum-processors.html Accessed: Dec. 30, 2020.

[31] R. Prabhakar. "Myths Programmers Believe about CPU Caches", apr 20. [Online]. Available: https://software.rajivprab.com/2018/04/29/myths-programmers-believe-about-cpu-caches/ Accessed: 25 Nov., 2020.

[32] S. Venkateswaran. *Essential Linux device drivers*. Prentice Hall Press, 2008.

[33] C. Benvenuti. *Understanding Linux network internals*. " O'Reilly Media, Inc.", 2006.

[34] K. Wehrle. The linux networking architecture: Design and implementation of network protocols in the linux kernel, 2004.

[35] "Monitoring and Tuning the Linux Networking Stack: Receiving Data", 201. [Online]. Available: https://blog.packagecloud.io/eng/2016/06/22/monitoring-tuning-linux-networking-stack-receiving-data/ Accessed: Nov. 29, 2020.

[36] L. Rizzo. "Netmap Github Page". [Online]. Available: https://github.com/luigirizzo/netmap Accessed: Sept. 19, 2020.

[37] L. Rizzo. "Netmap: a novel framework for fast packet I/O". In *21st USENIX Security Symposium (USENIX Security 12)*, pages 101–112, 2012.

[38] "DPDK Programmer's Guide". [Online]. Available: https://doc.dpdk.org/guides/prog_guide/index.html Accessed: 25 Nov., 2020.

[39] IOVisor Authors. "BPF Features by Linux Kernel Versions". [Online]. Available: https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md Accessed: 25 Nov., 2020.

[40] D. Calavera and L. Fontana. *Linux Observability with BPF: Advanced Programming for Performance Analysis and Networking*. O'Reilly Media, 2019.

[41] S. Sharma. "Just In Time Compiler", 2018. [Online]. Available: https://www.geeksforgeeks.org/just-in-time-compiler/ Accessed: 5 Nov., 2020.

[42] M.A.M. Vieira, M. S. Castanho, R. D. G. Pacífico, E. R. S. Santos, E. P. M. Júnior, and L. F. M. Vieira. "Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges, and Applications". *ACM Computing Surveys (CSUR)*, 53(1):1–36, 2020.

[43] Linux man-pages project. "bpf-helpers(7) — Linux manual page". [Online]. Available: https://man7.org/linux/man-pages/man7/bpf-helpers.7.html Accessed: Jul. 14, 2020.

[44] Cilium Authors. "BPF and XDP Reference Guide". [Online]. Available: http://https://docs.cilium.io/en/latest/bpf/ Accessed: Jul. 14, 2020.

[45] J. D. Brouer and A. Gospodarek. "A practical introduction to XDP". Linux Plumbers Conference (LPC), 2018. [Online]. Available: https://linuxplumbersconf.org/event/2/contributions/71/attachments/17/9/presentation-lpc2018-xdp-tutorial.pdf Accessed: Nov. 27, 2020.

[46] Intel Corporation. "Intel Ethernet Controller XL710 Family Linux Driver source code". [Online]. Available: https://elixir.bootlin.com/linux/v4.2/source/drivers/net/ethernet/intel/i40e/i40e_txrx.c Accessed: Nov. 27, 2020.

[47] J. D. Brouer. "Linux xdp.h file source code". [Online]. Available: https://github.com/torvalds/linux/blob/master/include/net/xdp.h Accessed: Nov. 27, 2020.

[48] T. Makita and W. Tu. "Veth XDP: XDP for Containers". [Online]. Available: https://www.files.netdevconf.info/f/a63b274e50f943a0a474/?dl=1 Accessed: Sept. 17, 2020.

[49] T. Makita. "Veth XDP: XDP for containers". Presentation slides. Netdev 0x13, Prague, Czechia, 2019. [Online]. Available: https://www.files.netdevconf.info/f/224dbe2ec1a447be918b/?dl=1 Accessed: Nov. 27, 2020.

[50] Cilium Authors. "Cilium Architecture". [Online]. Available: http://https://docs.cilium.io/en/v1.6/architecture/ Accessed: Sept. 17, 2020.

[51] E. Leblond and P. Manev. "Introduction to eBPF and XDP support in Suricata". White Paper, 2020. [Online]. Available: https://cdn2.hubspot.net/hubfs/6344338/Resources/Stamus_WP_Intro_to_eBPF_and_XDP_in_Suricata_Online.pdf Accessed: Nov. 24, 2020.

[52] S. Rivera, V. K. Gurbani, S. Lagraa, A. K. Iannillo, and R. State. "Leveraging eBPF to preserve user privacy for DNS, DoT, and DoH queries". In *Proceedings of the 15th International Conference on Availability, Reliability and Security*, pages 1–10, 2020.

[53] Facebook Incubator Authors. "Katran github page". [Online]. Available: https://github.com/facebookincubator/katran Accessed: Sept. 17, 2020.

[54] "bpftool man page", 2017. [Online]. Available: https://lwn.net/Articles/739357/ Accessed: Nov. 27, 2020.

[55] XDP Authors. "XDP Project: XDP Tutorial". [Online]. Available: https://github.com/xdp-project/xdp-tutorial Accessed: Dec. 4, 2020.

[56] BPF Authors. "Linux bpf_helpers.h file source code". [Online]. Available: https://github.com/torvalds/linux/blob/v4.19/tools/testing/selftests/bpf/bpf_helpers.h Accessed: Nov. 27, 2020.

[57] XDP Authors. "Tutorial: Packet01 - packet parsing". [Online]. Available: https://github.com/xdp-project/xdp-tutorial/tree/master/packet01-parsing#the-data-and-data_end-pointers Accessed: Nov. 27, 2020.

[58] LLVM Authors. "Getting Started with the LLVM System". [Online]. Available: https://llvm.org/docs/GettingStarted.html Accessed: Nov. 27, 2020.

[59] "xdp_prog_user.c source code", 2019. [Online]. Available: https://github.com/xdp-project/xdp-tutorial/blob/master/packet-solutions/xdp_prog_user.c Accessed: Nov. 29, 2020.

[60] "How to Access Docker Container's Network Namespace from Host". [Online]. Available: https://www.thegeekdiary.com/how-to-access-docker-containers-network-namespace-from-host/ Accessed: Nov. 29, 2020.

[61] Broadcom. "High-Capacity StrataXGS® Trident II Ethernet Switch Series". [Online]. Available: https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56850-series Accessed: Nov. 28, 2020.

[62] R. J. Wysocki. "CPU Performance Scaling", 2017. [Online]. Available: https://www.kernel.org/doc/html/latest/admin-guide/pm/cpufreq.html Accessed: Nov. 27, 2020.

[63] C. Hollowell, C. Caramarcu, W. Strecker-Kellogg, A. Wong, and A. Zaytsev. "The effect of NUMA tunings on CPU performance". In *Journal of Physics: Conference Series*, volume 664, page 092010. IOP Publishing, 2015.

[64] Intel Corporation. "Intel® Data Direct I/O Technology: A Primer", 2012. [Online]. Available: https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology-brief.html Accessed: May 5, 2020.

[65] G. V. Neville-Neil. "netmap pkt-gen Manual Page", 2016. [Online]. Available: https://github.com/luigirizzo/netmap/blob/master/apps/pkt-gen/pkt-gen.8 Accessed: Nov. 29, 2020.

[66] S. Bradner. "Benchmarking Terminology for Network Interconnection Devices". RFC 1242, July 1991. [Online]. Available: https://tools.ietf.org/html/rfc1242 Accessed: Dec. 2, 2020.

[67] B. Gregg. "CPU Utilization is Wrong". Blog, May 2017. [Online]. Available: http://www.brendangregg.com/blog/2017-05-09/cpu-utilization-is-wrong.html Accessed: Nov. 29, 2020.

[68] J. Quittek, t. Zseby, B. Claise, and S. Zander. "Requirements for IP Flow Information Export (IPFIX)". RFC 3917, October 2004. [Online]. Available: https://tools.ietf.org/html/rfc3917 Accessed: Dec. 2, 2020.

[69] "Interrupt and IRQ tuning". [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/s-cpu-irq Accessed: Nov. 29, 2020.

[70] Intel Corporation. *"Intel Ethernet Controller X710/ XL710 and Intel Ethernet Converged Network Adapter X710/XL710 Family Linux Performance Tuning Guide"*, 2016.

[71] "irqbalance man page". [Online]. Available: https://manpages.debian.org/testing/irqbalance/irqbalance.1.en.html Accessed: Nov. 27, 2020.

[72] "set_irq_affinity source code". [Online]. Available: https://github.com/intersvyaz/i40e/blob/master/scripts/set_irq_affinity Accessed: Nov. 27, 2020.

[73] T. Herbert and W. de Bruijn. "Scaling in the Linux Networking Stack". [Online]. Available: https://www.kernel.org/doc/Documentation/networking/scaling.txt Accessed: Nov. 29, 2020.

[74] D. Raumer, F. Wohlfart, D. Scholz, P. Emmerich, and G. Carle. Performance exploration of software-based packet processing systems. *Leistungs-, Zuverlässigkeits-und Verlässlichkeitsbewertung von Kommunikationsnetzen und verteilten Systemen*, 8, 2015.

[75] IEEE Standards Association et al. IEEE Standard for Ethernet. *IEEE Std*, pages 802–3, 2012.

[76] W. Wu, M. Crawford, and M. Bowden. "The performance analysis of Linux networking–packet receiving". *Computer Communications*, 30(5):1044–1057, 2007.

[77] J. Madieu. *Linux Device Drivers Development: Develop customized drivers for embedded Linux*. Packt Publishing Ltd, 2017.

[78] D. Barach, L. Linguaglossa, D. Marion, P. Pfister, S. Pontarelli, and D. Rossi. High-speed software data plane via vectorized packet processing. *IEEE Communications Magazine*, 56(12):97–103, 2018.

[79] M. Konstantynowicz, P. Lu, and S. M . Shah. "Benchmarking and analysis of software data planes'. White paper, 2017. [Online]. Available: https://fd.io/docs/whitepapers/performance_analysis_sw_data_planes_dec21_2017.pdf Accessed: Dec. 18, 2020.

[80] G. Huston. "BGP in 2019 – The BGP Table", 2020. [Online]. Available: https://blog.apnic.net/2020/01/14/bgp-in-2019-the-bgp-table/Accessed: 25 Nov., 2020.

[81] D. S. Miller. "Merge branch 'kill_rtcache'". Git commit, Jul 2012. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/netdev/net-next.git/commit/?id=5e9965c15ba88319500284e590733f4a4629a288 Accessed: Dec. 18, 2020.

[82] A. Kwatra et al. "Modelling the impact of CPU properties to optimize and predict packet processing performance". Case Study, 2018. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/case-studies/att-cpu-impact-on-packet-processing-perfomance-paper.pdf Accessed: Dec. 3, 2020.