Centro de Informática UFPE

Pós-Graduação em Ciência da Computação

Eduardo Felipe Fonseca de Freitas

Experimental Evaluation on Packet Processing Frameworks under Virtual Environments

Recife
2021

Eduardo Felipe Fonseca de Freitas

Experimental Evaluation on Packet Processing Frameworks under Virtual Environments

A M.Sc. Dissertation presented to the Centro de Informática da UFPE in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Concentration Area: Computer Networks

Advisor: Prof. Dr. Djamel Fawzi Hadj Sadok

Recife
2021

Catalogação na fonte
Bibliotecária: Mônica Uchôa, CRB4-1010

F866e Freitas, Eduardo Felipe Fonseca de.
Experimental Evaluation on Packet Processing Frameworks under Virtual Environments / Eduardo Felipe Fonseca de Freitas. – 2021.
58 f.: il., fig., tab.

Orientador: Djamel Fawzi Hadj Sadok.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn. Programa de Pós-graduação em Ciência da Computação. Recife, 2021.
Inclui referências.

1. DPDK. 2. XDP. 3. Kernel Linux. 4. Processamento de Pacote de Rede. 5. Kernel Bypass. I. Sadok, Djamel Fawzi Hadj (Orientador). II. Título.

681.3 CDD (23. ed.) UFPE- CCEN 2021 - 175

Eduardo Felipe Fonseca de Freitas

“Experimental Evaluation on Packet Processing Frameworks under Virtual Environments”

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação. Área de Concentração: Redes de Computadores e Sistemas Distribuídos.

Aprovado em: 26/08/2021.

I dedicate this work to my family for all support and love they give me throughout my life and especially during the making of this dissertation.

ACKNOWLEDGEMENTS

I want to thank my family for all their support throughout the making of this work, which required so much of their support especially in such an adverse year. I also want to thank my girlfriend for patience, believing and helping me during the making of this dissertation.

I want to thank my advisor, Djamel Sadok, and also Judith Kelner, for guidance into making this research and for the opportunity to continue my academic studies, making this dissertation and being part of GPRT. I also thank their support during this pandemic year, which has not been easy for any of us.

I also thank my team from GPRT, Assis and Pedro, who also helped me during the research and writing process of this dissertation.

I thank my friends from graduation, who have been with me since we discovered together our passion for technology, science and freedom.

RESUMO

O kernel Linux é um componente central das aplicações de rede, estando presente na maioria dos servidores em *data centers*. Com o tempo, à medida que servidores e placas de rede evoluíram para atender tecnologias de rede com demandas de alto *throughput* e baixa latência, o kernel tornou-se um gargalo, impedindo as aplicações de rede de utilizarem a capacidade máxima do hardware. Nesse cenário, diferentes *frameworks* de processamento de pacotes surgiram para solucionar esse gargalo. Os dois principais são o DPDK e XDP, com propostas diferentes para atingir altas taxas de processamento. DPDK adota o *by-pass* do kernel, excluindo-o do processamento e levando os pacotes para o *user space*. Já o XDP, por outro lado, processa os pacotes dentro do kernel, de forma antecipada comparada ao processamento padrão. Em conjunto com isso, o paradigma de computação em nuvem, atualmente disponível na maioria dos *data centers*, traz a virtualização como tecnologia fundamental. Com múltiplas aplicações e sistemas sendo executados no mesmo *host*, surge outro problema, o de competição de recursos. Assim, essa dissertação executa experimentos que buscam avaliar como a presença de um ambiente virtual de computação em nuvem pode interferir no desempenho de ambos DPDK e XDP. Os resultados mostram que embora o processamento "dentro do kernel" traga mais segurança e integração com sistema, essas exatas medidas de segurança causam perda de desempenho ao XDP. Além disso, o XDP também demonstra ser o mais afetado pela presença do ambiente virtual, considerando a taxa de throughput e também a perda de pacotes. Por outro lado, existe um dilema ao utilizar o XDP, que não somente é possível alcançar maior segurança, mas também em relação ao uso de recursos, já que o DPDK aloca um núcleo de CPU completo para utilizar no processamento de pacotes. Também, dependendo do processamento sendo feito pelo framework, como quando depende de uso intenso de CPU, o DPDK oferece uma perda considerável de desempenho do throughput.

Palavras-chaves: DPDK; XDP; Kernel Linux; Processamento de Pacote de Rede; *Kernel Bypass*.

ABSTRACT

The Linux kernel is at the heart of network applications, being present in most servers across data centers. With time, as servers and network cards evolved to enable high-throughput and low-latency network technologies, the kernel became a bottleneck, preventing network applications from operating at maximum hardware capacities. In such scenario, several “packet processing frameworks” emerged to solve this bottleneck. The two main ones are DPDK and XDP, adopting different approaches to achieve high processing rates. DPDK consists of bypassing the kernel and processing packets in user space. XDP, in contrast, processes packets inside the kernel, at an early stage in the processing path. Alongside this, the cloud computing paradigm, currently available in most data centers, brings virtualization as its most important technology and enabler. With multiple applications and systems running in the same host, comes another concern, that of host resource competition. Thus, this dissertation creates experiments that evaluate how the presence of a cloud computing virtualization environment can interfere in both DPDK and XDP's performance. Results show that even though the in-kernel processing from XDP may assure system security and integration, these exact security measures interfere in throughput performance and packet loss. Also, XDP seems to be the most effected by the presence of virtual environment. However, there is a trade-off when using XDP, not only for the system security but for resource usage, since DPDK allocates full CPU core utilization for packet processing. Also, depending on the processing tasks at hand, such as those that require heavy CPU usage, DPDK does not offer an optimal throughput performance.

Keywords: DPDK; XDP; Linux Kernel; Network Packet Processing; Kernel Bypass.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

BPF	Berkeley Packet Filter
cBPF	classic Berkeley Packet Filter
DAG	Directed Acyclic Graph
DDoS	Distributed Denial-of-Service
DMA	Direct Memory Access
DoE	Design of Experiment
DPDK	Data Plane Development Kit
DPI	Deep Packet Inspection
DuT	Device under Test
EAL	Environment Abstraction Layer
eBPF	extended Berkeley Packet Filter
FIFO	First-In, First-Out
IoT	Internet of Things
IPC	Instructions Per Cycle
ISA	instruction set architecture
IT	Information Technology
JIT	Just-In-Time
KVM	Kernel-based Virtual Machine
NFV	Network Function Virtualization
NIC	Network Interface Card
OS	Operating System
PPVE	Packet Processing under Virtual Environments
PS	PPVE Server
SDN	Software Defined Networks
SS	Sender Server
TSN	Time Sensitive Networks
VM	Virtual Machine
VNF	Virtual Network Function
XDP	eXpress Data Path

CONTENTS

1 INTRODUCTION

With the growth and popularity of the Internet, its services became critical to people's social and business activities. Alongside this, emerging technologies like 5G, 6G, Time Sensitive Networks (TSN) for Industry and Entertainment, Network Function Virtualization (NFV) and Software Defined Networks (SDN) enabled networks often demand high-throughput and low-latency traffic processing to cope with their new services. To keep up with these expectations, network hardware gained more processing power, with the deployment of 40, 100 and even 200Gbps network cards in data centers, and also the inclusion of Network Interface Card (NIC) with support for processing power, the programmable NICs, often called SmartNICs (LIU et al., 2019).

However, high network performance does not rely only on hardware. The software side, responsible for operating the hardware, is also critical. The GNU/Linux Operating System (OS) is the most frequently used one in data centers (SURVEYS, 2021), making the Linux Kernel a performance decisive component for servers, since it is responsible for the interaction between hardware and user space applications. Regardless, the Linux Kernel is general-purpose, supporting a wide range of protocols and drivers with diverse use-cases, varying from Internet of Things (IoT) sensors, embedded devices, smartphones, access points, switches, and data center servers. This broad coverage gives Linux a platform-agnostic network processing that relies on costly packet copying among the different packet buffering areas and operating system mode switching between user and privileged kernel modes, as well as a full network and transport layer implementation. These and other properties often translate to network overhead in data center environments (RIZZO, 2012) (BELAY et al., 2014) (HØILAND-JØRGENSEN et al., 2018), and pose a concern on the new bandwidth and low delay requirements from the emerging technologies mentioned before such as NFV.

Founded on these concerns, software frameworks, commonly referred to as packet processors or packet processing frameworks, emerged in an attempt to enable fast network processing in these OSs. They create new mechanisms to exchange packets between the NIC and user space applications. Examples of these frameworks are Data Plane Development Kit (DPDK) and eXpress Data Path (XDP). DPDK was one of the first packet processors to implement a technique called kernel bypass. XDP emerged in response to kernel bypass frameworks, proposing the in-kernel processing technique. Later chapters will detail how each of them work and explain their main counter-points. Overall, these frameworks aim at delivering packets to a user space application developed specifically to interact with them, so instead of using ordinary OS system calls, the application uses the framework's API to exchange network packets. These network applications range from firewalls, software routers, Deep Packet Inspection (DPI), Virtual Network Functions (VNFs), and others.

However, as these frameworks present different implementations on how to assess fast packet processing, understanding their performance is critical to shape their deployment on real data center environments. As a result, this research field attracted researchers to experiment on with different application contexts and scenarios in order to assess their main performance considerations.

The remainder of this chapter further details this research context along with the motivation to this work. Also, it explains the objectives and relevant research questions, as well as the methodology applied to conduct the research.

1.1 MOTIVATION

Cloud Computing is a predominant paradigm in current Data Centers. Approximately 81% of Information Technology (IT) organizations use cloud computing or have applications deployed in the cloud, according to a survey from IDG (IDG, 2020). A key technology that enables Cloud Computing is Virtualization, which puts virtual environments at the center of all these organization's infrastructure.

Therefore, the context exposed at the beginning of this chapter becomes even more critical: the overhead imposed by the kernel inside these Data Center servers degrades network performance for all Virtual Machines (VMs), regardless of what application/service they execute (Dantas et al., 2015) (Liu, 2010). This issue increases if the server executes network applications like a firewall, checksum offloading or packet routing before sending the packets to the VMs, since the processing of these packets involves kernel's packet I/O.

These conditions propose that if a sysadmin implemented a network application with one of the packet processing frameworks mentioned before in a real Data Center environment, it would likely be implemented in hypervisor servers, that host multiple VMs. This means that this application would share the host resources with the VMs and its load, which may in return degrade the framework's performance. Currently there is a lack of ground covered by publications and experiments that addresses the issue of how these frameworks perform in the existence of a virtual environment, and also that compares their performance as single packet processing frameworks. Moreover, as mentioned earlier, understanding the performance of the packet processors is critical to shape their deployment on real data center environments.

1.2 RESEARCH QUESTIONS

Considering the context exposed previously, this work seeks to answer the following research questions:

- Is the performance of **in-kernel** processing approach proposed by frameworks like

XDP surpassing the performance of **kernel bypass** technologies from frameworks like DPDK?

- How is the **virtual environment** from cloud computing servers affecting the performance of fast packet processing frameworks?
- Is a cloud computing data center with **high disk I/O** usage like database or storage VM servers affecting the performance of fast packet processing frameworks?
- Is a cloud computing data center with **high network and CPU** usage like VNFs servers (running demanding application such as Distributed Denial-of-Service (DDoS) protection, routing, DPI, Load Balancing) affecting the performance of fast packet processing frameworks?

This research conducts experiments to address and answer these questions. The general and specific objectives are addressed in the next section.

1.3 OBJECTIVES

Given the motivation described in the previous sections, **the main objective of this work is to evaluate DPDK and XDP's performance when sharing host's resources within a Cloud Computing virtual environment**. Moreover, there are specific objectives of this work, namely:

- Build a packet processing architecture using DPDK and XDP that implements packet processing features with different resource usage.
- Provide a report of issues and challenges found during the process of developing applications that use DPDK and XDP as packet I/O.
- Update literature with experiments using newer versions of XDP, since most experiments conducted by literature uses initial versions of XDP.

1.4 METHODOLOGY

The methodology applied in this work complies with the following phases:

- Theoretical study of the main concepts related to network packet processing and data analysis.
- Literature review.
- Development of the packet processing architecture that implements packet processing based on DPDK and XDP.

- Definition of a testbed that simulates a cloud computing data center with storage and database servers, VNFs and virtual environment.
- Execution of experiments and analysis of collected data.

The theoretical study allows familiarization with the history of network packet processing inside Linux and other frameworks, and also develops basic knowledge of data analysis using basic statistical methods that a researcher may be unfamiliar with. During literature review, it took place revision of the works that compare some packet processing frameworks to understand where the state of the art currently stands when it comes to packet processors evaluation especially in virtual environments, and also to understand important metrics used to evaluate these frameworks.

1.5 STRUCTURE OF THE DISSERTATION

The remainder of the structure of this dissertation is: Chapter 2 presents the background concepts for this work. It describes the network path a packet traverses inside the Linux Kernel as well as gives details of DPDK and XDP's history and architecture. Chapter 3 describes the state of the art through related works, discussing similar works and the way they differ from the one presented here. Chapter 4 presents the testbed scenario, the experimental design and the experiment itself, and discusses the obtained results. Chapter 5 discusses the conclusions of this work as well as the main contribution and future works.

2 BACKGROUND

This chapter presents the main concepts needed to understand the work undertaken in this research. First it describes in detail how the Linux Kernel processes network packets inside a host, with packets flowing from the NIC to the user space application. It also outlines how this processing can slow down network performance and become a bottleneck in some scenarios. Next, it describes the DPDK framework, explaining its design principles and benefits. Finally, it depicts the XDP framework, explaining the motivation behind its development as well as shows how it works and details its components.

2.1 LINUX NETWORK PATH

When a packet reaches a host, the NIC receives it and its course across this host begins. The NIC and its driver processes Layers 1 and 2. After receiving the raw packet signals, transforming and placing them as packets inside the NIC memory, they are then sent to the kernel for processing by upper layers (WU; CRAWFORD; BOWDEN, 2007). This is done firstly by allocating the packet in system memory. To do this, the kernel uses a circular queue of buffer descriptors known as *ring buffers* to manage incoming and outgoing packets via a device driver (separate rings for incoming and outgoing packets) in a First-In, First-Out (FIFO) manner. The buffer descriptors in the ring references a memory slot, holding information like address space and length (RIZZO, 2012). These memory slots are the socket buffers, used to allocate incoming packets, that are stored in a previously allocated `sk_buff struct` data structure. Figure 1 illustrates the network processing path, to ease understanding of this process.

When a packet arrives, the driver searches for slots marked as "ready" in the ring buffer. Upfront, if no slots are ready, the NIC drops every new incoming packet, until there are new "ready" slots available. Then the packet is copied to the `sk_buff` buffer addressed in the ring via Direct Memory Access (DMA), the I/O mechanism that allows hardware devices to transfer data to the memory system without the need of the system processor. This mechanism depends on interrupt handling (CORBET; RUBINI; KROAH-HARTMAN, 2005), therefore, after the packet is copied and made available to the kernel, the NIC issues an interrupt to inform the CPU that the packet is ready. The CPU then receives this packet by adding a reference of the NIC to its *poll queue*, making it accessible via the *poll* method (WU; CRAWFORD; BOWDEN, 2007).

Afterwards, the packet is taken to the kernel protocol stack. Layers 3 and 4 processing take place, running the implementation of protocols like IP, TCP, ICMP or UDP. When such protocols are processed accordingly, the packet can be passed to user space. First, the application process in user space issues a receive system call (e.g. `recv()`) in

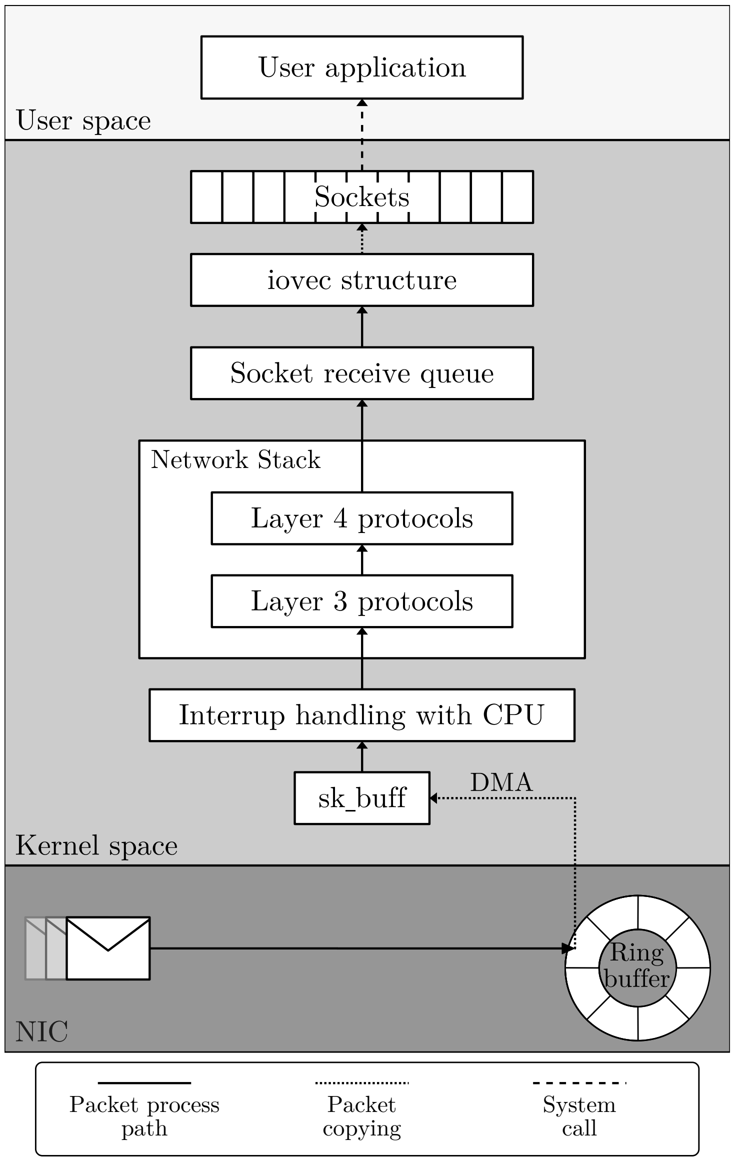

Figure 1 – Linux network packet processing path

advance, containing the appropriate data's destination information like memory address and number of bytes to be transferred. This information is usually supplied by an `iovec struct` via system call to the socket's data receiving process. Then, after the packet leaves the network stack, it is sent to the socket receive queue, where it will be processed as a **network data**: sometimes, different packets compose a single data that was fragmented due to TCP's fragmentation support, for example. This data is reassembled in the socket receive queue. When data is ready, it is copied by the receive system call from the socket receive queue through the `iovec` structure and is delivered to the user space process that issued the system call.

2.1.1 Performance issues

The Linux network path poses some issues related to performance, that can be critical in scenarios like data centers, where frequent packets are received and transmitted. The following main issues are discussed next.

As the kernel uses the `sk_buff` buffer to hold packets, its frequent allocation and reallocation causes CPU stress. Studies conducted in (HAN et al., 2010) show that approximately 5% of CPU cycles are spent to initialize `sk_buff` buffers, 8% are spent dealing with allocation and reallocation and 50% is used to control requests for these memory operations.

System calls can also delay packet processing and lower performance. First, a delay is caused by the frequent mode switching between user and kernel-mode (TSUNA, 2010), that happens every time a packet is sent to a user space process, for example. Additionally, processor structures like L1 data and caches are occupied by kernel-mode state, and the replacement to/from user-mode leaves a footprint in the form of *processor state pollution*, that results in wasted CPU cycles and structures entries (SOARES; STUMM, 2010). Furthermore, system calls can impact user-mode Instructions Per Cycle (IPC) due to the mentioned processor state pollution and also processor pipeline flushing.

In addition, memory usage is also a problem inside Linux. Initially, the `sk_buff` buffer can be considered too large, primarily because it carries information about different protocols and their headers. This can become a bigger problem to certain applications that will not need such protocols headers allocated in memory. Also, as explained before, a packet goes through at least two copying processes inside the kernel. These copying mechanisms also increase CPU processing and resource consumption, that decline performance. Studies carried out in (BRUIJN; DUMAZET, 2017) show that depending on the buffer size, it is possible to save up to 39% of system cycles when processing packets using mechanisms that avoid copying buffers.

As Linux is a general purpose kernel, it must support as much layer 3 and 4 protocols as possible, in order to maintain the most diverse types of scenarios/applications and devices. This opposes having an efficient kernel, as such processing is often unnecessary to certain user space applications (JEONG et al., 2014). Ultimately, the per-packet processing is the main bottleneck issue, especially when dealing with small packets (HAN et al., 2010) (RIZZO, 2012) (BELAY et al., 2014). Linux processes one packet at a time, meaning that all of the additional overhead and resource usage discussed above are repeated for every new packet, which can represent overkill in scenarios with high network load.

Under these conditions, different technologies emerged in an attempt to solve packet processing issues. One of them is the **Kernel Bypass**. This mechanism determines that the NIC is exposed to user space and then packet I/O is conducted directly between the application and network card. Therefore, the kernel itself is *bypassed* and not used to exchange data. In addition, some processing preferences are taken to enable acceleration, which brings some benefits. The main examples are:

- Pre-allocation of packet buffers during initialization, avoiding memory allocation during execution of packet processing.
- Avoidance of packet copying, fixing the packet in a memory location accessible throughout the whole processing path and user space application.
- Batching packets, processing multiple packets at a time.
- Simplified network stack, leaving Layers 3 and 4 processing to the user space application.
- Stepping away from the kernel's complexity and overhead like frequent mode switching, system calls and semantics like POSIX.

Some examples of frameworks that implement the Kernel Bypass mechanism are netmap (RIZZO, 2012), DPDK (FOUNDATION, 2018) and PF_RING (NTOP, 2018). As for the scope of this work, the DPDK framework will be detailed in the next section.

2.2 DPDK

The Data Plane Development Kit is a set of user space libraries and drivers created to enable the Kernel network stack to bypass and achieve high packet processing rates. It creates an Environment Abstraction Layer (EAL) that is responsible for interacting with low-level resources like network cards and memory space, managing its components to provide operations like memory management, time reference or atomic/lock operations. The EAL main purpose is to offer a generic environment so that the user space programs can interact with and do not worry about implementing such low-level features. The EAL can be compared with the Kernel's System Calls. DPDK also uses a run-to-completion model to process packets, allocating all resources used during execution before the user space application execution. It also gains access to network devices by polling instead of using schedulers like Linux, avoiding interrupts and consequently removing the overhead they impose (GROUP, 2017a). Figure 2 illustrates DPDK's network processing path as explained next.

DPDK mainly builds on top of the *core components*, a set of libraries providing the components that enable the kernel bypass and packet processing. The first component is the **Ring Library**, implemented in `librte_ring`. The Ring library provides a ring buffer to manage packet buffer queues, storing the objects in a table, with a fixed maximum size and a FIFO management. It is responsible for interacting with the packet buffers received by the NIC. The ring also supports batch processing, allowing the enqueue and dequeue

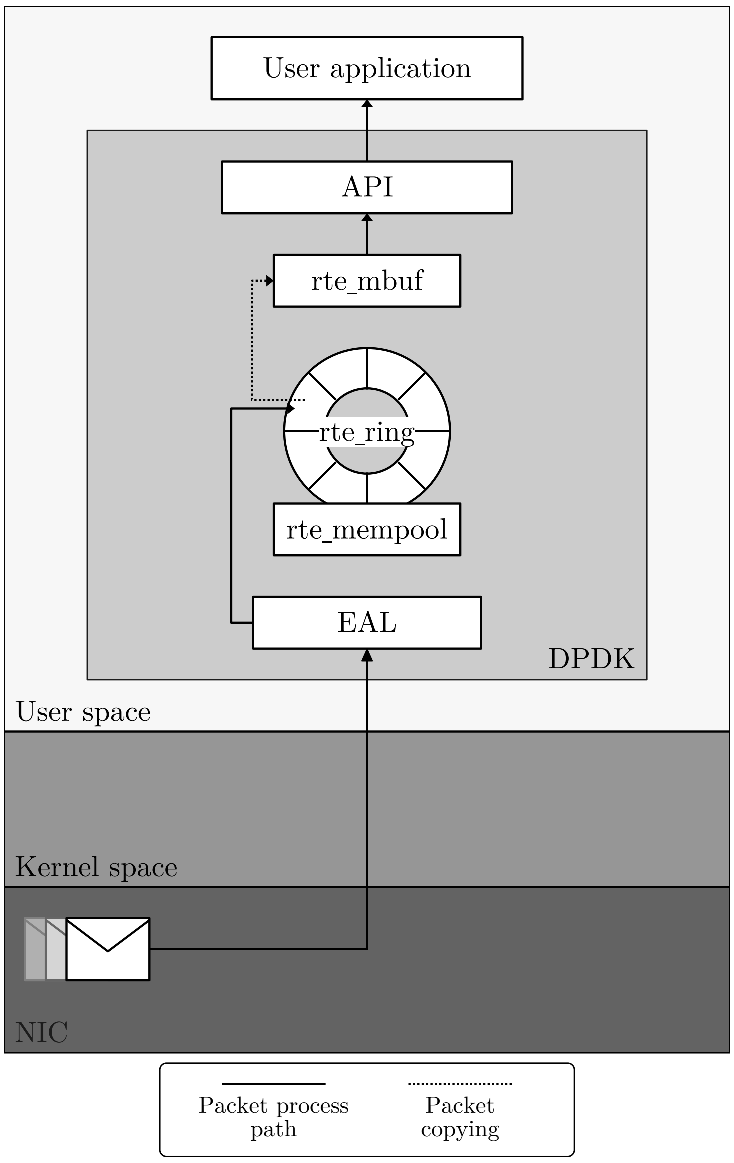

Figure 2 – DPDK network packet processing path

of multiple packets in a single procedure, which improves speed and cache misses (GROUP, 2017b).

The next component is the **Memory Pool Manager**, implemented in `librte_mempool`. As the name implies it manages the memory pool, allocating in memory the pool of objects. It uses a ring as the mempool handler, to store free available objects. Those objects are the third core component, the **Network Packet Buffer Management**, implemented in `librte_mbuf`. This component coordinates the allocation of memory buffers used to store a network packet, and they are stored in the memory pool.

DPDK also uses a special driver that runs in user space and maps the device's memory region into user space (GALLENMüLLER et al., 2015). When a packet arrives at the NIC, the packet is initially mapped in the card's memory region and the `mempool` consults which `mbuf` object is free within the `ring` and stores the packet in the next available one. Next, the user space program can interact with the packet processing via the DPDK API. This effectively implements a kernel bypass solution, where a user space framework directly interacts with the network card and exposes network packets to the application.

However, moving away from the kernel also brings some concerns. The first is that applications using kernel bypass have to re-implement components already provided by Linux, such as device drivers, layer 3 and 4 protocols or routing tables, which creates a security and isolation concerns:

> *This leads to a scenario where packet processing applications operate in a completely separate environment, where familiar tooling and deployment mechanisms supplied by the operating system cannot be used because of the need for direct hardware access. This results in increased system complexity and blurs security boundaries otherwise enforced by the operating system kernel. The latter is in particular problematic as infrastructure moves towards container-based workloads coupled with orchestration systems such as Docker or Kubernetes, where the kernel plays a dominant role in resource abstraction and isolation* (HØILAND-JØRGENSEN et al., *2018).*

Also, the kernel offers security mechanisms to protect the OS, like protected memory region, process isolation and controlled hardware I/O. Using a kernel bypass solution discards all of these mechanisms, leaving the application in charge of re-implementing them, which cannot always be reliable or bug free.

2.3 XDP

As discussed previously, kernel bypass solutions raise security concerns as they discard the kernel and its protection features. The XDP framework brings back packet processing **to** the kernel, instead of removing network control **from** the kernel, and it can be presented as a new kernel hook that provides programmable packet processing inside the kernel.

XDP uses the extended Berkeley Packet Filter (eBPF) infrastructure to leverage its secure and fast processing environment inside the kernel. This infrastructure is the foundation of XDP, and constitutes one of the main components responsible to enable the XDP system. It consists of some important elements like the **eBPF virtual machine**, **eBPF maps**, and the **eBPF verifier**. To better organize the description of how these components work, eBPF will be detailed in the next section, along with its elements, followed by the XDP structure.

2.3.1 eBPF

eBPF is the rework of the Berkeley Packet Filter (BPF), created in 1992, that provided a native way to inject bytecode from user space into the kernel, to perform network filtering tasks. It attached itself in a socket and filtered every incoming packet from there. With time, the rework of BPF began and in around 2014, eBPF surfaced, turning BPF into the classic Berkeley Packet Filter (cBPF) (MONNET, 2016).

In general, eBPF is a safe kernel environment, that allows user space code to be executed inside the kernel. This is done executing a register-based virtual machine inside the kernel, with Just-In-Time (JIT) compilation and an in-kernel verifier that checks user space code's security, to ensure it will not harm the kernel. It also provides structures to allow communication between user space and the program inside the kernel, as will be described next.

The eBPF virtual machine is a register-based instruction set architecture (ISA), that mimics native hardware instruction set. This allows the creation of function calls that receive parameters through registers, and is able to map an eBPF function to a specific hardware instruction, reducing overhead (VIEIRA et al., 2020). This VM is the environment where the dynamic translation (JIT) and loading happens, allowing the user space program to be executed.

Nevertheless, in order to be executed in kernel space, the program must be secure and pose no harm to the kernel. To ensure this, the kernel has only one way to receive eBPF programs, that is through the `bpf()` system call. This system call passes the program to the **eBPF verifier**, that is responsible for inspecting the program's byte code in search for possible properties that could damage the kernel (HØILAND-JØRGENSEN et al., 2018). To do this, the verifier first creates a Directed Acyclic Graph (DAG) of the programs instructions. This graph is used to analyse the program's steps to ensure it has no cycles, meaning it performs no backward jumps or undefined size loops. This is useful to analyse the program bounds, looking for unverified loops or unsupported/unreachable instruction calls. Initially, the verifier didn't allow loops to protect the kernel from programs that could never terminate, or that would take too much time or processing resources. However, recent updates added support for *bounded* loops (RYBCZYńSKA, 2019). The verifier analyses the size of the program by its instruction number, and with this update this instruction limitation is increased, from 4096 to one million instructions. This enabled the verifier to check for bounded loops by simulating all iterations as a collection of states, directly allowing defined-size loops to be inserted in the program. Lastly, the verifier inspects all possible paths of instruction calls, to ensure every memory access is safe and limited to the program's local variables. This also forces the program to perform bound checking when accessing packet bytes, to guarantee that the memory access is performed in checked addresses.

In addition to this, eBPF programs cannot use the system's default memory storage. To achieve such feature, **eBPF Maps** are available. Maps are key-value stores, defined during program loading with fixed sized values. Theses stores allow user-defined data structures to be loaded into the kernel, and are available to be accessed by user space programs or other eBPF programs (VIEIRA et al., 2020). This means it can also behave as a way for multiple eBPF programs to communicate between each other or between eBPF and user space programs. This specific functionality can enable a chain of programs, that each can be triggered according to the change of an specific value in a Map (HØILAND-JØRGENSEN et al., 2018).

2.3.2 XDP driver hook

An XDP program is executed by a hook in the device driver, triggered by the event of a new packet issued by the network card. This means that an XDP program is executed in an early point, right after the packet is received by the NIC, even before the `sk_buff` allocation.

An XDP program starts with an input context object in the form of a `struct xdp_md` structure, that contains different pointers to different parts of the raw packet. These parts are the beginning and the end of a packet, pointed by the `data` and `data_end` pointers. Theses pointers serve especially security purposes, following the same disposition of eBPF for bounding all memory operations to ensure they are made in fixed and known memory area. The third pointer `data_meta` holds a free memory address that can be used to exchange packet metadata. With this context object, the program can parse packet data, read metadata fields and also add metadata to the packet, adding or removing content or headers to the packet data (VIEIRA et al., 2020).

An XDP program can also access kernel functions through *helper functions*, that enable the program to use specific kernel functionalities like check-summing or routing table look-ups. This is useful especially when such functionalities are needed but parsing the packet by the normal network stack is not an option. Figure 3 summarizes the explained XDP's network processing path.

After executing all possible packet processing operations, the XDP program must return an integer value. Four different values are available, each of them representing a different code that will define what will happen to the packet after it leaves XDP:

- **`XDP_DROP`**, this will make XDP drop the packet;
- **`XDP_PASS`**, sends the packet to the kernel network stack, to be processed as it would normally be;
- **`XDP_TX`**, causes the packet to be re-transmitted out via the same network interface it came from;
- **`XDP_REDIRECT`**, allows the packet to be redirected to other locations. This code needs other parameters to indicate the target destination of redirection, which can be:

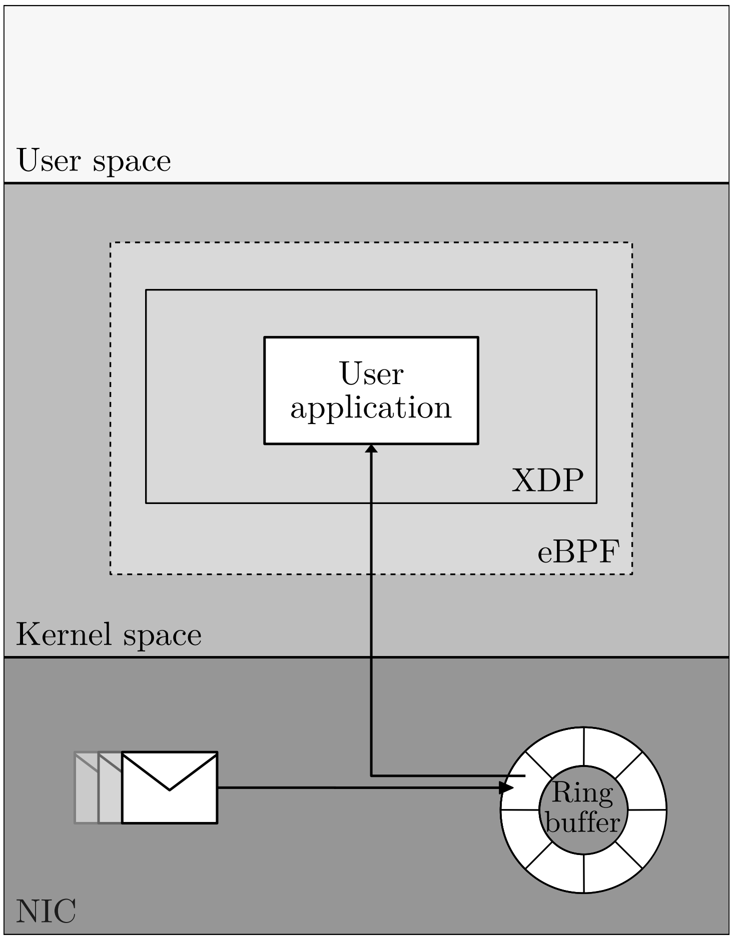

Figure 3 – XDP network packet processing path

- **Network interface**: re-transmitting the packet to a different network interface from what it came from. This includes virtual interfaces connected to virtual machines in user space;
- **CPU**: passes the packet for further processing on another CPU, to achieve load balancing for example;
- **User space**: redirects the packet directly to user space to be used by other application, through the new `AF_XDP` socket family. This approach can be compared to a kernel bypass, as it avoids the standard kernel stack and most generic packet processing, and delivers the packet to user space.

Alongside this standard packet processing, XDP supports the offloading of applications to a *programmable NICs*. In other words, XDP can send the user space application to a specific NICs, and it will run inside the network card. This is only possible when using programmable NICs, which are specific network devices with powerful processing units like CPU and memory (LIU et al., 2019), often referred to as “SmartNICs”. To execute an application inside the NIC via XDP, it cannot require kernel helper functions and it needs to be able to operate outside the host, this way XDP acts as an enabler to send the application to the network interface through its device driver.

This functionality can enhance packet processing by mainly two ways. First, it can achieve high throughput performance since it is using dedicated hardware to process only network packets (Hohlfeld et al., 2019) (LIU et al., 2019). Second, as the NIC is responsible for packet processing, it frees the host's resources and enables it to dedicate processing power to other tasks (Miano et al., 2019). Such functionality may be critical to cloud computing environments, that need to process the highest amount of packets while retaining host resources to the cloud applications.

3 RELATED WORK

This chapter surveys the state-of-the-art in performance evaluation of packet processing frameworks, more specifically the ones including XDP and/or DPDK. In the remainder of this chapter, the works surveyed contribute in some ways to this dissertation, either as an insight for the experiments and methodology adopted or as a theoretical reference.

The work made in (GALLENMüLLER et al., 2015) evaluates the performance of three packet processors that adopt the kernel bypass techniques, DPDK, PF_RING and netmap. The experiments measure the achievable throughput by the frameworks in packets per second (pps) metric. Their work first evaluates each framework's throughput when executing tasks with variable CPU load in each processed packet. Results show that with tasks that consume around 100 CPU cycles, netmap looses throughput performance from 14.88Mpps – their NIC line-rate – to around 11Mpps. This performance loss only happens with DPDK and PF_RING with tasks of 150 CPU cycles. Their work also evaluates the influence of batch sizes in throughput performance, and DPDK shows that with CPU tasks of up to 150 CPU cycles, a batch size of either 8 or 32 packets results in the same line-rate throughput. Only when performing tasks from 150 until 400 CPU cycles the bigger batch size offers a higher throughput, despite the difference between the two batches is around 1Mpps. From 400 CPU cycles and up, the two batch sizes perform equally at a low throughput of around 6.5Mpps or less.

The authors also perform an evaluation of how the batch size influences the latency in each processed packet. With small batch sizes of 8 packets, higher latency occurs of around $130\mu s$ with DPDK since it processes only 8 packets per API call. With batches of 16, the latency is as little as $10\mu s$. Further increasing the batch size causes an interesting trade-off, it consequently increases the latency, since the time a packet spends queued is higher, as it has to wait for new packets to arrive and fill the batch size. So a batch size of 256 packets has a latency of $40\mu s$.

The developers of XDP wrote the research paper in (HØILAND-JØRGENSEN et al., 2018), describing the framework's details followed by a performance evaluation. Their paper explains how XDP works, shows its interaction with eBPF and the rest of the kernel. Afterwords, they conduct different experiments to directly compare XDP and DPDK. The first experiments test performance with packet size of 1500 bytes, and both frameworks could process the packets at line-rate. Then, they perform experiments with minimum-sized packets of 64 bytes. The study starts by comparing the throughput (packets per second) of both frameworks with no processing tasks and an increasing number of CPU cores. DPDK performs better than XDP in all cases, but XDP has an increase of performance as the CPU cores increase. These measurements do not include any host resource usage like CPU or memory. The authors also run experiments measuring the CPU usage with a growing offered load, where DPDK is always consuming 100% while XDP scales CPU usage accordingly to the offered packet load. The experiments also run packet forwarding tasks, which also do not perform host resource usage but impose latency for the packet processing. In these experiment, XDP outperforms DPDK only when it forwards packets to the same NIC and DPDK to a different NIC. When both forward to different NICs, DPDK shows a better throughput performance and a lower latency in forwarding.

The research study described in (KOURTIS et al., 2015) focuses on the NFV context. It provides experiments that compare the performance of a DPI VNF with a normal DPI program in user space, while alternating between the utilization of DPDK and the ordinary Linux kernel network stack. Their work focuses on benchmarking how virtual network functions differ from bare-metal network programs in terms of performance. They do not attempt to investigate if the independent virtual environment influences DPDK, instead they use DPDK to test if it can improve the VNF's performance as much as it improves the bare-metal program.

The results from their work show that DPDK achieves the 10Gbps NIC line-rate in bare-metal environment, while the standard Linux kernel maximum throughput is at 1Gbps. When using virtualization, the standard Linux kernel stays at 1Gbps throughput and gains more instability, creating a variable throughput ranging from 1000Mbits/s to 600Mbits/s. DPDK also looses stability and does not achieve line-rate, with approximately 19% throughput performance degradation when compared to the bare-metal corresponding experiment.

The work in (Hohlfeld et al., 2019) focuses on evaluating performance of different execution points of XDP, including, at user space with `AF_XDP`, the standard XDP at the device driver and special offloading to SmartNICs. Note that all previous works that studied XDP use the standard device driver. The first executed experiments evaluate processing tasks with no resource usage, like (HØILAND-JØRGENSEN et al., 2018). The results show that the standard device driver XDP can only achieve line-rate throughput when using multiple cores. The NIC XDP always achieves line-rate since it has separate processing functionalities, and the user space XDP never achieves line-rate, but its performance scales linearly with the presence of more CPU cores, as expected. When processing packets with tasks that perform CPU usage, the NIC offloading has the worst performance as the CPU from the network card is not as powerful as the ones in the host.

Their work also measures response time from each execution point. When using XDP at the NIC, response time is close to zero with a maximum of $16\mu s$ at any incoming packet rate. When using XDP at the device driver or user space, the response time is around 0.3ms with an incoming packet rate of 100pps, since the interrupt delivery dominated CPU processing. With a higher packet rate of 1Mpps, CPU changes to polling mode and the response time decreases to around 0.05ms. The authors also analyse response time when the processing tasks perform memory access, where the NIC offloading again presents the smaller response time of 0.02ms with an incoming packet rate of 2Mpps, and

0.9ms when the packet rate is 14Mpps. The device driver execution point comes in second place, with 0.04ms and 1.5ms with the same corresponding incoming packet rate.

They also perform experiments on a VM, but at a point of view from inside the virtual machine, with additional execution points: at the VM user space with `AF_XDP`, at the VM device driver, at the host device driver and at the physical NIC. It is important to notice that although this work performs experiments with a virtual machine, it does not consist of a cloud computing environment. This is because it focuses only on one VM execution and testing XDP's performance inside this VM. It does not create a virtual *environment*, with multiple VM execution with different types and intensity of resource usage, as encountered in real cloud computing environments. Results with 4 CPU cores show that XDP inside the VM at user space and device driver has a similar performance with a mean throughput of 874kpps and 921kpps, respectively. Throughput in the host device driver stays at line-rate in this same scenario, along with the NIC. Their work also studies the VM and host CPU usage. Results show that when the processing task has no resource usage, the host CPU usage is not affected with whether XDP is executed at the VM user space or at the VM device driver. But when the processing task performs CPU usage, the host decreases CPU usage when XDP execution point changes from the VM user space to the VM device driver.

The research paper in (SCHOLZ et al., 2018) evaluates the possibility of enhanced packet filtering inside the kernel. It does this by evaluating firewall rules inside XDP followed by the evaluation of socket filtering with eBPF and systemd init daemon. To align to this dissertation's objective, only the first evaluation part of their work is discussed. It is worth noting that the experiments executed in their work used an initial and experimental version of XDP. This version considered the JIT compiler experimental and therefore had it disabled by default. The evaluation first analyses a sample XDP application that forwards packets or drops them, depending on the packet rule. The evaluation tests three types of incoming packets where XDP drops either 10, 50 or 90% of the packets. Results show that the throughput performance is better in the case of 90% of packets to drop, with a 7.2Mpps throughput. This occurs due to the additional processing needed to forward packets, that is not the case when dropping them. They follow with measuring the CPU utilization and conclude that when offering a 10Mpps incoming packet rate, XDP uses approximately 60% of CPU load only for packet processing with eBPF-related functions. Finally, they perform latency measurement, with results showing a median latency of around 50μs with an incoming packet rate from 1 to 6Mpps. When the offered rate is closer to 7Mpps – the maximum throughput XDP could process in their experiments – the latency increases to 1000μs.

Based on the showed literature, Table 1 summarizes the characteristics of each work in perspective to this dissertation's contribution. It is noticeable that no previous work exposes both frameworks to a virtual environment as seen in cloud computing, as well as none takes into account the amount of dropped packets as a response metric. It's worth mentioning that packet loss is a critical metric in a Cloud Computing scenario. Therefore, is should be part of such experimental evaluation to better represent this scenario. Also, most of the reviewed works do not use in the same experiment scenario together the DPDK and XDP frameworks.

Table 1 – Summary of related works and this dissertation's main contributions

Work	**Frameworks**	**Tasks with resource usage**	**Cloud computing environment**	**Measures throughput**	**Measures packet drop**	**Measures latency**
GALLENMüLLER et al., 2015	DPDK, netmap, PF_RING	X		X		X
HØILAND-JØRGENSEN et al., 2018	DPDK, XDP	X		X		X
KOURTIS et al., 2015	DPDK	X		X		
Hohlfeld et al., 2019	XDP	X		X		X
SCHOLZ et al., 2018	XDP			X		X
This dissertation	DPDK, XDP	X	X	X	X	

4 EXPERIMENTAL EVALUATION

This chapter presents how the experiments were designed, implemented and executed. First, it details the packet processing architecture designed to conduct experiments with DPDK and XDP, followed by the explanation of the testbed configuration along with the Design of Experiment (DoE) created and the execution steps. Then, it presents the results obtained, followed by the discussion and evaluation of these results.

4.1 THE PPVE PACKET PROCESSING ARCHITECTURE

The development of a packet processing architecture is a central part of this work since the main goal is the evaluation of DPDK and XDP. An existing architecture that uses these two frameworks in a manner that fits our testbed scenario and research environment is difficult to locate in the literature. There are some related simple tools built as a proof-of-concept with simple packet processing tasks as well as other far too complex production level applications with different layers of configurations not compatible with our scope. This led to the creation of the *Packet Processing under Virtual Environments (PPVE) architecture*, a packet processing architecture that uses DPDK or XDP to process packets under different host resource usage modes.

PPVE processes packets with different tasks that perform different types and intensity of hardware resource usage. The idea is to emulate a typical and a more realistic data center environment. For example, it is feasible that in these real environments, packet processing frameworks perform resource intense tasks like DDoS protection (Miano et al., 2019), load-balancing or deep packet inspection (TU; YOO; HONG, 2019). Hence, when assessing an experiment that simulates such case, it is desirable that the frameworks also perform heavy-loaded tasks. These tasks are configured as parameters parsed when starting PPVE, along with the framework it will use. Figure 4 illustrates the PPVE architecture.

The first parameter defined is the framework used, which determines the selected technology either DPDK or XDP. This loads one of the frameworks in the OS and starts receiving packets from the NIC. Then, the “PPVE Mode” configures the type of resource usage being either CPU or memory access. This means that the PPVE architecture will process each received packet while performing tasks that consume either CPU or memory resources. These represent the resources most commonly used when running packet processing frameworks inside dedicated servers (Hohlfeld et al., 2019) (Miano et al., 2019) (TU; YOO; HONG, 2019) (GALLENMüLLER et al., 2015). Finally, the “PPVE Stress Level” defines the intensity of each resource usage, being either “High” or “Low”. These values are an abstraction of real intensity values explained in the remainder of this section. Each stress

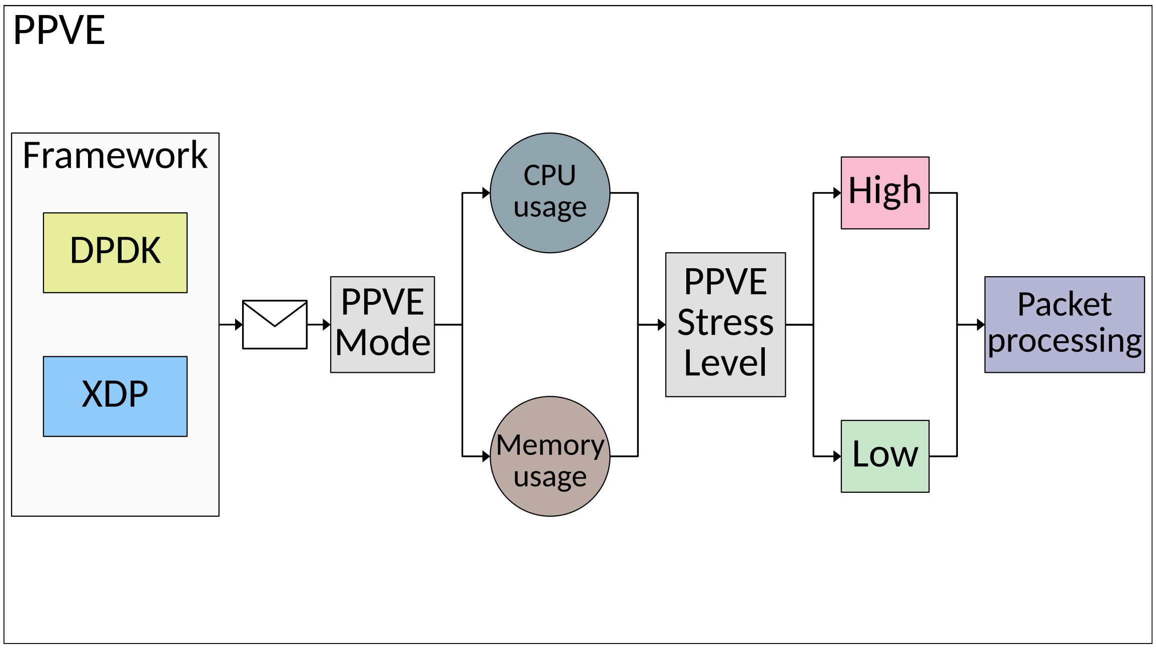

Figure 4 – The PPVE architecture

level regards the intensity of the chosen PPVE Mode. Hence, for example, if the PPVE mode is CPU and the Stress Level is High, it means that for each packet received by the PPVE architecture, the architecture performs a heavy-loaded CPU processing task.

The CPU PPVE Mode (often referred simply as CPU Mode), performs CPU usage by running checksum validation for *each* packet received. Checksum validation refers to the validation of the Checksum field[1] in IPv4 Headers. This field contains a checksum of the IPv4 header and every network node that receives the IP packet can (re)calculate the header checksum and compares it with the one available in the checksum field, to validate its integrity. In other words, the CPU Mode calculates the received packet's IPv4 header checksum for every packet received, therefore performing CPU usage. Checksum calculation is not applicable to every type of CPU processing, but it serves as a representative measure for processing power usage and complexity. The Stress Level abstraction for the CPU mode simply repeats the checksum calculation for each packet to consume more CPU and simulate heavy-loaded CPU tasks. The "Low" stress level for the CPU mode runs **one** checksum calculation per packet, and the "High" stress level runs **ten** checksum calculations per packet.

The memory mode simulates a traditional firewall rule: it holds an IP list of allowed IP addresses and for every new packet, PPVE retrieves its source IP and searches for it in the allowed IP address list. This operation performs memory access because the IP list is an in-memory hashmap list, which means that consulting an IP on the list triggers memory access and reading. The Stress Level abstraction in this mode acts by changing the size of the IP list, so the "Low" level uses a 512KiB list whereas the "High" level uses

[1] https://datatracker.ietf.org/doc/html/rfc1812#section-4.2.2.5

only a 16MiB list. Table 2 summarizes the PPVE modes, stress levels and their tasks.

Table 2 – Meaning of the PPVE Stress level for each PPVE Mode

PPVE Mode	PPVE Stress level	Meaning on the experiment
CPU	Low	1 Checksum calc.
	High	10 Checksum calc.
Memory	Low	Hashmap list size of 512KiB
	High	Hashmap list size of 16MiB

The PPVE source code is available in the remote SourceHut git repository[2] and licensed as GPLv3+.

4.2 TESTBED

As stated in previous chapters, the basic goal of this research is to evaluate and compare the performance of XDP and DPDK packet processors when inserted in a virtual environment. To assess this goal, this research assembled a testbed along with the DoE, basing the execution of the experiments. This is detailed in the remainder of this section.

The testbed adopts a P2P topology, with two servers directly connected to each other. The first server, called PPVE Server (PS), is the Device under Test (DuT) and is responsible for the execution of the PPVE architecture. The other server is the Sender Server (SS), which *sends* a burst of packets to PS, where PPVE processes them. Figure 5 illustrates this testbed scenario.

In order to simulate a more realistic Cloud Computing scenario and to expose DPDK and XDP to this scenario, the PPVE Server runs a virtual environment with Kernel-based Virtual Machine (KVM) VMs. This environment focuses in simulating different data center cases in-line with the scope of this research, like database, storage or VNF servers. Hence, the virtual environment may host from 0 to 128 VMs, and each of them executes a workload to stress the server and compete for hardware resources. This workload simulates the mentioned data center server cases, and as so varies between disk I/O, CPU and Network. Its important to remind that the PPVE architecture runs directly on the server, and not inside the VMs, as the goal of the experiments is to expose both DPDK and XDP to resource competition with the virtual environment.

The Sender Server uses the MoonGen[3] (EMMERICH et al., 2015a) packet generator to create and send the burst of packets to PS. MoonGen is a software developed specifically for academic and industrial experimentation testbeds that need to generate burst of network packets. It can send packets at line-rate on 10Gbps NICs, and it allows the

[2] https://git.sr.ht/~eduardofreitas/ppve
[3] https://github.com/emmericp/MoonGen/commit/525d9917c98a4760db72bb733cf6ad30550d6669

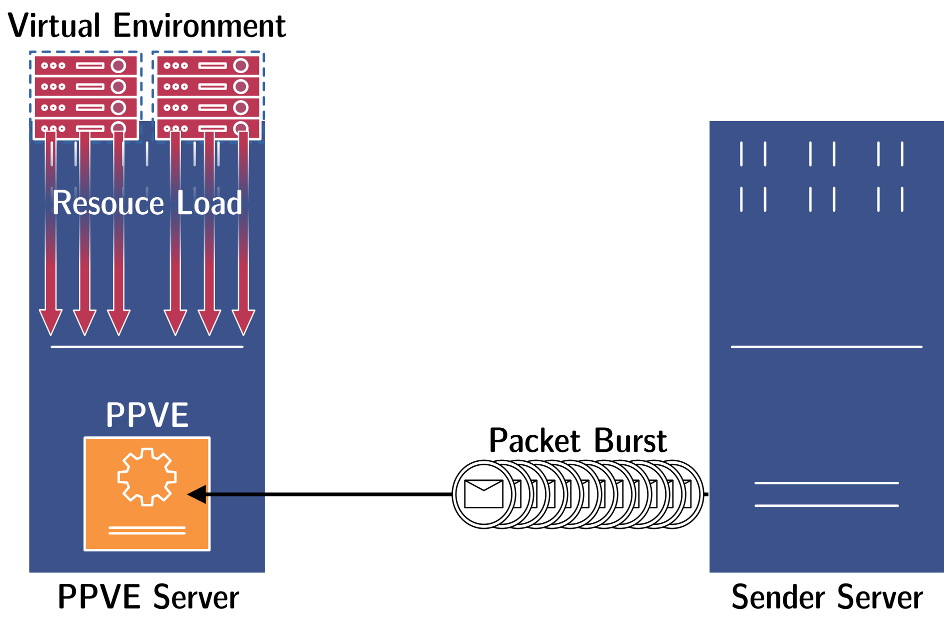

Figure 5 – Experiment's testbed

customization of packets from different protocols such as IPv4, IPv6, ICMP, TCP and UDP.

Table 3 specifies the configuration setup of both hosts and the version of the software used.

Table 3 – Testbed Hardware and Software Specification

Component	**Specification**
Processor	Intel Xeon Bronze 3204
RAM	64GiB
HD	Dell BOSS VD
NIC	Intel Ethernet Controller X540-AT2
OS	GNU/Linux Ubuntu 20.04.2 LTS
Kernel	Linux 5.4.0-70-generic
KVM	Version 2.11.1
DPDK	Version 20.11.1
MoonGen	git commit 525d991

4.3 EXPERIMENTAL DESIGN

The experimental design or DoE, as stated in (NIST/SEMATECH, 2012), is an effective way to plan the experiments that will be executed, so that analysing the responses can provide solid conclusions. The design of experiments conducted in this work follows a **Comparative Experiment** that compares DPDK and XDP's performance against adverse conditions detailed next. This comparison is the main goal of the DoE.

The experiment considers a total of seven variables. Five are controllable input factors and the other two are the responses, in other words, the data produced by the experiment. The input factors and their levels are summarized in Table 4 and described next:

- **PPVE Mode**, which is the mode in which the PPVE architecture will run, as explained in Section 4.1. It alternates between "CPU" and "Memory".
- **PPVE Stress Level** is the intensity level of the PPVE Mode resource execution, as detailed Section 4.1. Its levels are "high" and "low".
- **VM Load** is the workload executed inside the VMs and varies between CPU, I/O and Network.
- **VM Number** is the number of VMs running and performing the resource workload, ranging between 0, 8, 32 and 128. This number of VMs starts as 0 serving as the baseline for the experiments, where the frameworks could perform with free resources and no competition. The successive number of VMs follows the notation of 2^n, where n is an odd number starting from 3. The option of $n = 1$ is discarded since it represents a small number of VMs (only 2) and would not serve the scope of the experiments. Finally, the maximum of 128 VMs is due to hardware limitations.
- **Packet Size** is the size of all packets sent from SS to PPVE, and varies between 64 and 1500 Bytes. It is the norm to usually test packet processing frameworks with small sized packets since the goal is to assess the higher amount of processed packets as possible, as shown in works like (RIZZO, 2012), (HØILAND-JØRGENSEN et al., 2018), (HAN et al., 2010) or (JEONG et al., 2014). However, since the goal is to simulate a real Cloud Computing environment, that is not at all dominated by small sized packets, the case for bigger packets of 1500 Bytes becomes valid.

As for the responses, the collected metrics are:

- **Throughput of Packets Processed per Second**, denoted in unit of Mega packets per second (Mpps), is the throughput of packets that PPVE processes in each experiment.
- **Total number of Packets dropped by the NIC**, represented in packets (pkts) unit, is the total amount of packets that the NIC driver dropped in each experiment.

With the variables explained before, the DoE categorizes as a **mixed-level full factorial design**. This is the result of having a different number of levels for the factors and also running all available factor combinations in the experiment.

Table 4 – DoE's Factors and Levels

Factor	Levels			
PPVE Mode	CPU	Memory		
PPVE Stress	Low	High		
Packet Size	64	1500		
VM Load	CPU	Network	I/O	
VM Number	0	8	32	128

When calculating the combination of all factors, a total of 96 runs is necessary for each packet processor framework. To enhance data reliability, each run replicates 30 times, resulting in 2880 runs for each framework. The execution of the experiment is also separated in a randomized block design especially to avoid external influences that may interfere in the responses in similar factors. Two blocks separate the execution based on the packet size, the first block executes all combinations with packets of 1500 Bytes whereas the second one uses packets of 64 Bytes.

The packet burst lasts 30s, meaning that after PS assembles the virtual environment and starts PPVE, SS starts MoonGen and sends the burst of packets for 30s at the a rate of 10Gbps, the maximum capacity of the network card. This results in a total of 26000000 packets (26Mpkts) sent when the packets are 1500 bytes long, and 420000000 packets (420Mpkts) when they are 64 bytes long.

As mentioned in Section 2.2, DPDK can process packets through batches, meaning it can use one API call to the NIC to receive multiple packets. This feature is not optional and programming any application that will use DPDK must include a batch size. To decide which batch size to use in PPVE for the experiments, a simple experiment evaluates the influence of the batch size in the throughput performance. The experiment runs within the same testbed using the "Low" Stress Level and small packets with 10 replications. The goal is to understand the impact of using the two batch sizes of 4 and 32 packets. The 4 packet batch size is the minimal size available for the testbed's network device driver, and the 32 batch size refers to reported related work, that shows that this size offers the highest performance (GALLENMüLLER et al., 2015), even though the difference may not be excessive. Figure 6 displays the results from this test. The difference in batch size has no direct influence in the throughput performance, as it is visible how the median values remain constant around 14.20Mpps. The scatter plot helps visualizing each experiment response. It is noticeable that in cases with more VMs, there is higher variation, creating some data responses below 14.20Mpps, but not enough to influence the overall median value. These results are inline with mentioned related work that performed batch size comparisons with DPDK and other packet processors and evaluated that there is little difference between batch sizes when analysing throughput performance. Note that this difference may be higher when analysing other network metrics such as latency. Because of this, to better match the features from XDP and create more comparable experiments, the experiments run with 4 packets batch size in DPDK in all experiments. Also, both framework runs with no feature tuning, such as memory reservation or multiple CPU allocation.

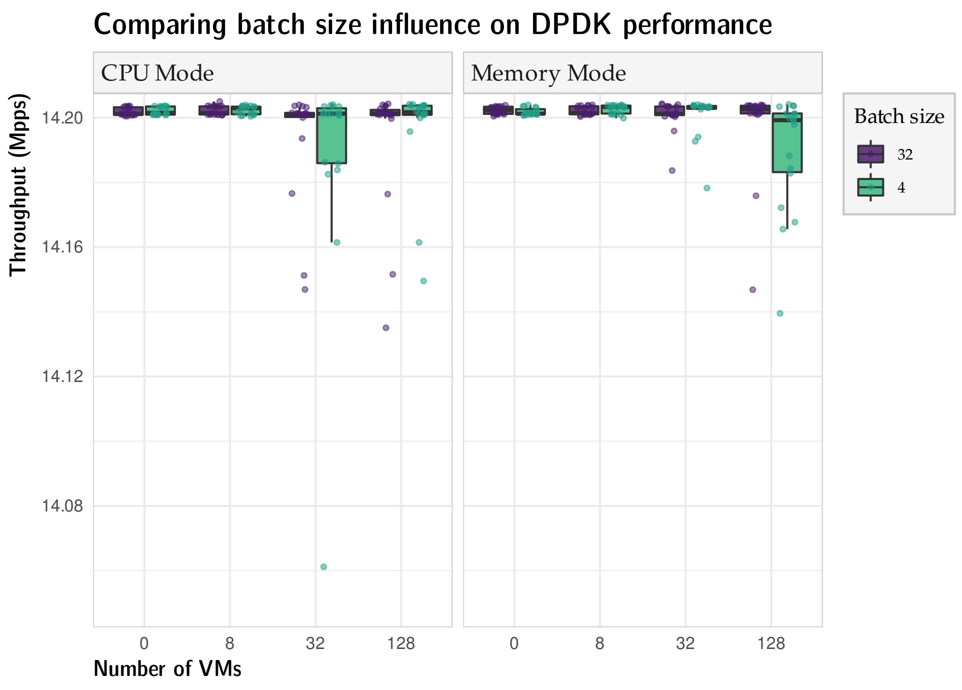

Figure 6 – Box plot of the DPDK throughput by the number of VMs with an overlaid scatter plot. This plot compares two distinct batch sizes to evaluate its influence on throughput performance

4.4 RESULTS

This section presents the results collected from the experiments described in the previous one. Based on the research questions this work seeks to answer, the presentation of the results is divided in two parts. The first one compares the two frameworks with no virtual environment, regarding only the performance for each PPVE mode and stress level. The second part of the results explores the performance of both frameworks with the virtual environment under different VM loads. To further clarify the observed results, Table 5 presents the NIC line-rate throughput in millions of packets per second, which represents the theoretical expected throughput for the packet processing frameworks, i.e., if they process every received packet at NIC line-rate. In other words, the values in Table 5 act as a baseline and provide upper bounds and guidelines for comparison.

Table 5 – Description of expected line-rate throughput

Packet size	Packet size in physical layer	Expected line-rate throughput for each framework
64 Bytes	84 Bytes	14,880Mpps
1500 Bytes	1520 Bytes	0,822Mpps

4.4.1 Frameworks comparison

As this first part deals with a scenario with no VMs, it starts by describing experiments with a large packet size of 1500B, as the expected throughput is different than when using small packets of 64B. Figure 7 shows a box plot of the throughput of each packet processing framework according to the PPVE Modes and Stress Levels. The achieved throughput in any case provides little variation, as ensured by the standard deviation of 0.00007Mpps, calculated from the throughput considering all the samples displayed in the box plot. Also, the throughput around 0.82Mpps is close to the expected line-rate, showing a satisfactory result. These results are expected as they represent the "best" scenario for both frameworks, since there is no background VM executing any resource load or overhead and the packet size of 1500B provides a low rate of incoming packets, which enables an achievable high packet rate. This experiment helps concluding that both frameworks have a similar near line-rate performance when receiving big packets of 1500 bytes and have no virtual environment execution. This conclusion also considers both when the framework performs CPU or memory usage.

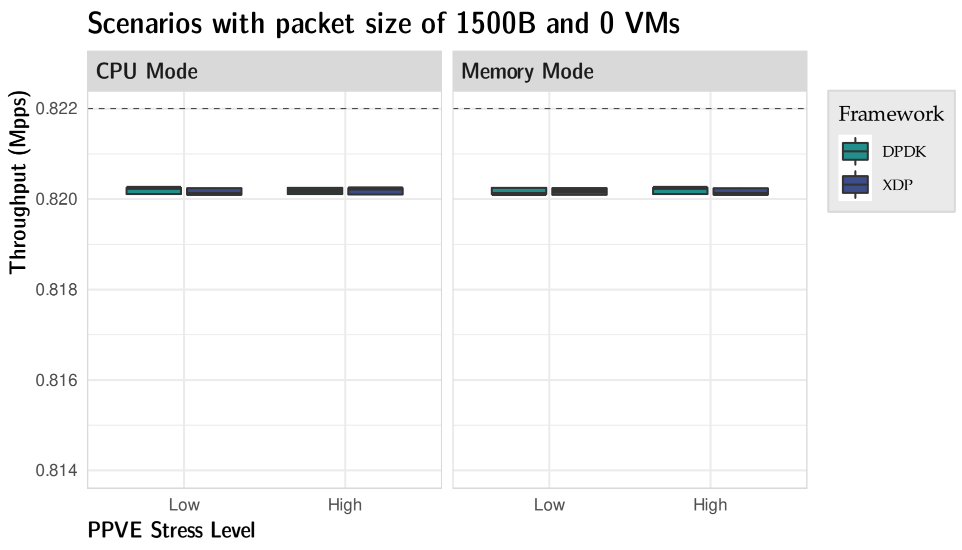

Figure 7 – Experiments comparing the throughput of frameworks according to the modes and stress levels when using big packets. The dashed line indicates the NIC line-rate

However, the case for small packets draws a different picture as demonstrated by the results displayed in Figure 8. The PPVE stress level emerges as an important factor when it comes to the achievable throughput. In both frameworks, the stress level of the memory mode causes little impact in the median throughput. But when analysing the stress level of the CPU mode in both frameworks, it translates to a direct performance loss. Considering the median, when both frameworks perform memory reading tasks, the size of the memory list has no impact on the throughput performance, maintaining the same median and overall spread. On the other hand, when performing tasks with CPU usage, DPDK's performance drops 24.8% when increasing from low to a high stress level and as much as 57.8% when using XDP. This mainly happens because the DPDK's standard behaviour consumes 100% of one CPU core with busy polling for packet processing, while XDP scales its CPU consumption according to the received packet rate. Also, even though DPDK has the mentioned CPU resource usage, it is visible that its performance suffers if the processing requires heavier CPU usage. This would require tuning more CPU cores for both frameworks to enhance its performance, which may not be ideal if it is running in a production environment dedicated for server applications. This helps inferring that the usage of host memory should not interfere in both frameworks performance, if there is no virtual environment execution. On the other hand, the usage of host CPU causes an impact on both frameworks throughput, when performing intense CPU usage.

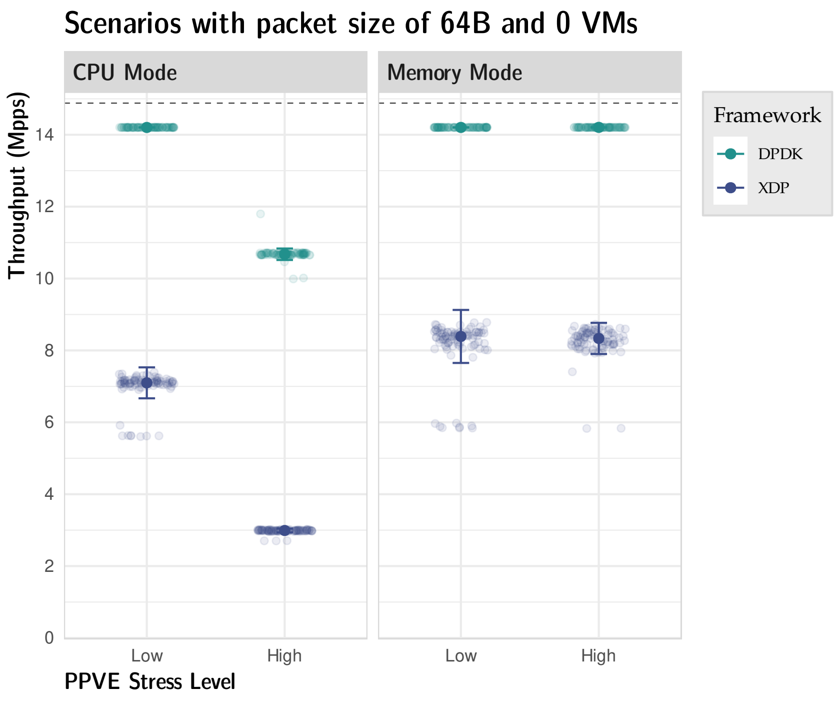

Figure 8 – Point plot representing the median with the standard deviation range. A scatter plot is also displayed in the background to help understand the spreading of the data. The dashed line indicates the NIC line-rate. This scenario compares the throughput of frameworks according to the modes and stress levels when using small packets

Moreover, DPDK's throughput performance is better in all cases. It exhibits less variation and can achieve near line-rate performance in some scenarios, while XDP fails in doing so. This suggests that although the in-kernel packet processing approach may assure system security and integration, the processing performance may still suffer from these same security measures.

Alongside these measures, the in-kernel processing requires using the standard kernel network driver. This driver runs in kernel space with all the security and isolation features discussed in Chapter 2, which can translate to processing degradation. As discussed in that chapter, whenever the driver ring buffer queue is full and has no empty slots available, it causes the NIC to drop new incoming packets until the queue finds new space left. The experiment also measures the total amount of packets sent, packets received by PPVE and also the amount of packets dropped by the network driver and calculated the percentage of dropped packets. Figure 9 shows this percentage according to the PPVE mode and stress level for each framework when using small packets. As observed in the bar plot, DPDK drops less packets than XDP in any case. DPDK uses special network drivers that run in user space, which translates to greater memory space management. Also, as it supports batch processing, the user space driver is able to retrieve more packets from the queue at once, which again improves performance. However, DPDK is not always safe from dropping packets, which is the case when performing a high stress level on CPU mode, when it drops approximately 25% of received packets. In all other cases, the drop percentage is approximately 0.003%, small enough to make it difficult to see in the plot.As for the XDP drop percentage, there is a performance tendency similar to the one in Figure 8. One may observe that when increasing stress level on CPU mode the drop percentage also increases, dropping around 78% of the packets when changing from a low to a high stress level. On the other hand, when increasing the stress level on the memory mode, the drop percentage stays constant around 42%. This brings the conclusion that the increasing CPU usage leads to a higher packet dropping regardless of the framework used. However, the increase of memory access and reading does not increase the packet dropping in any framework. But, it is important to highlight that XDP does drop more packets than DPDK in any of the applied resource usage.

On the other hand, when using 1500B packets, this percentage is lower in all cases. Because of this, this case is displayed in Table 6 rather than in graphics, to ease visualization. DPDK drops no packets while XDP drops some packets. The drop rate remains nonetheless low in this scenario, being comparable to the ones from DPDK when using the memory mode with 64B packets. As the packet size is bigger, the incoming packet rate is lower, allowing the ring buffer queue to fill less often, thus lowering the percentage of dropped packets.

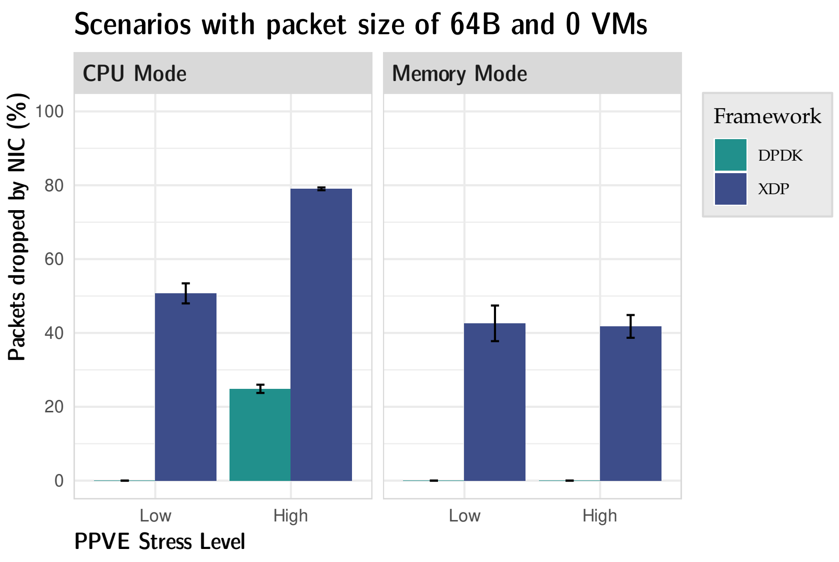

Figure 9 – Bar plot representing the mean with the standard deviation range. This scenario compares the NIC packet drop percentage for each framework according to the modes and stress levels when using small packets

Table 6 – This scenario compares the NIC packet drop percentage according to the modes and stress levels when using big packets

Framework	**PPVE Mode**	**Stress Level**	**Packets dropped by NIC (%)**			
			Min.	Median	Max.	Std. Dev.
DPDK	CPU	Low	0.0	0.0	0.0	0.0
		High	0.0	0.0	0.0	0.0
	Memory	Low	0.0	0.0	0.0	0.0
		High	0.0	0.0	0.0	0.0
XDP	CPU	Low	0.0	0.006	0.528	0.150
		High	0.0	0.007	0.541	0.160
	Memory	Low	0.0	0.004	0.558	0.170
		High	0.0	0.009	0.545	0.166

4.4.2 Virtual Environment Exploration

This second part of the results analyses the framework's performance when subjected to the increase in the number of VMs. The study often separates these results according to the type of resource load performed by the VMs while it maintains the previous separation following the size of packets. To begin with, Table 7 shows the throughput of DPDK for each VM number and PPVE mode. It is straight forward to see that almost no variation takes place in any case. There is a presence of constant central values and standard deviations. This scenario includes both stress levels in all samples, which helps infer how little variation each case has. This scenario helps conclude that when using DPDK to receive large packets of 1500 bytes, whether the framework performs CPU or memory usage tasks, the virtual environment causes no throughput performance degradation.

Table 7 – This scenario compares the throughput of DPDK according to the VM number and PPVE mode used, considering all stress levels and receiving 1500 bytes packets

Framework	**Mode**	**Number of VMs**	**Throughput (Mpps)**				
			Min.	Median	Max.	Mean	Std. Dev.
DPDK	CPU	0	0.8201	0.8203	0.8203	0.8202	0.0001
		8	0.8201	0.8203	0.8203	0.8202	0.0001
		32	0.8201	0.8201	0.8203	0.8202	0.0001
		128	0.8201	0.8201	0.8203	0.8202	0.0001
	Memory	0	0.8201	0.8203	0.8203	0.8202	0.0001
		8	0.8201	0.8203	0.8203	0.8202	0.0001
		32	0.8182	0.8203	0.8203	0.8202	0.0002
		128	0.8201	0.8203	0.8203	0.8202	0.0001

A different case emerges when using XDP. Figure 10 shows the same scenario but with the adoption of the XDP framework when using a high stress level. There is a data concentration in the scenario with no VMs where the values are all concentrated near line-rate. But, with the presence of the virtual environment these results become unstable, spreading the concentration of the data away from the line-rate with also the presence of low value outliers. XDP deals better with the VM network load than with CPU or especially I/O. It spreads the data with lower throughput values whilst the network load concentrates these at a higher rate. This scenario also contributes to the results seen previously where the CPU mode imposes a higher performance drop than the memory mode and where the throughput is less unstable and more values stay near the line-rate. It also differs with the results from (HØILAND-JØRGENSEN et al., 2018), extending the experiments that show a line-rate performance with 1500 bytes packets with a half-idle CPU. Summarizing these results yields the conclusion that XDP's throughput suffers from the presence of a virtual environment when it processes 1500B packets. When performing CPU usage, this degradation is higher than when performing memory usage, even though the memory access also decreases the throughput performance.

Figure 11 shows the median of the throughput along with the standard deviation of both DPDK and XDP when using memory mode with 64B packets. In the case of 0 and 8 VMs, it is noticeable how concentrated the data is when using DPDK, independently of the applied load type. Yet, when the VM number increases to 32 and 128, this concentration decreases lightly, creating some outliers with smaller throughput, but not enough to increase the standard deviation, except for one outlier in the I/O load with 32 VMs. This is an indication that the virtual environment interferes with DPDK's performance in this scenario when it implements host memory usage, but does not cause an intense interference, enabling the concentration of the throughput to still remain near line-rate.

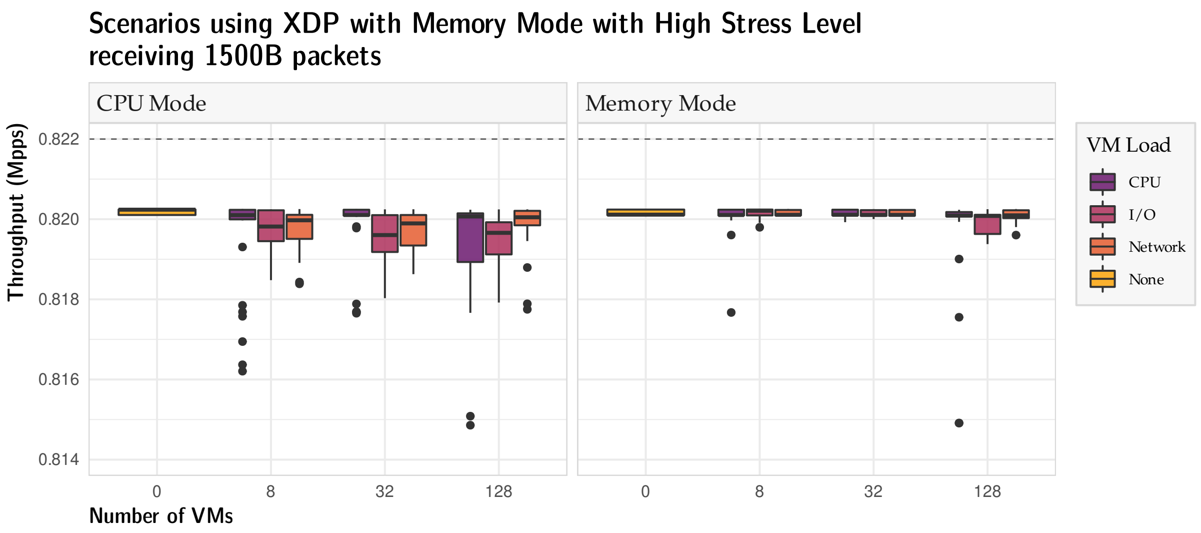

Figure 10 – Experiments comparing the throughput of XDP according to the number of VMs based on the load type. The dashed line indicates the NIC line-rate

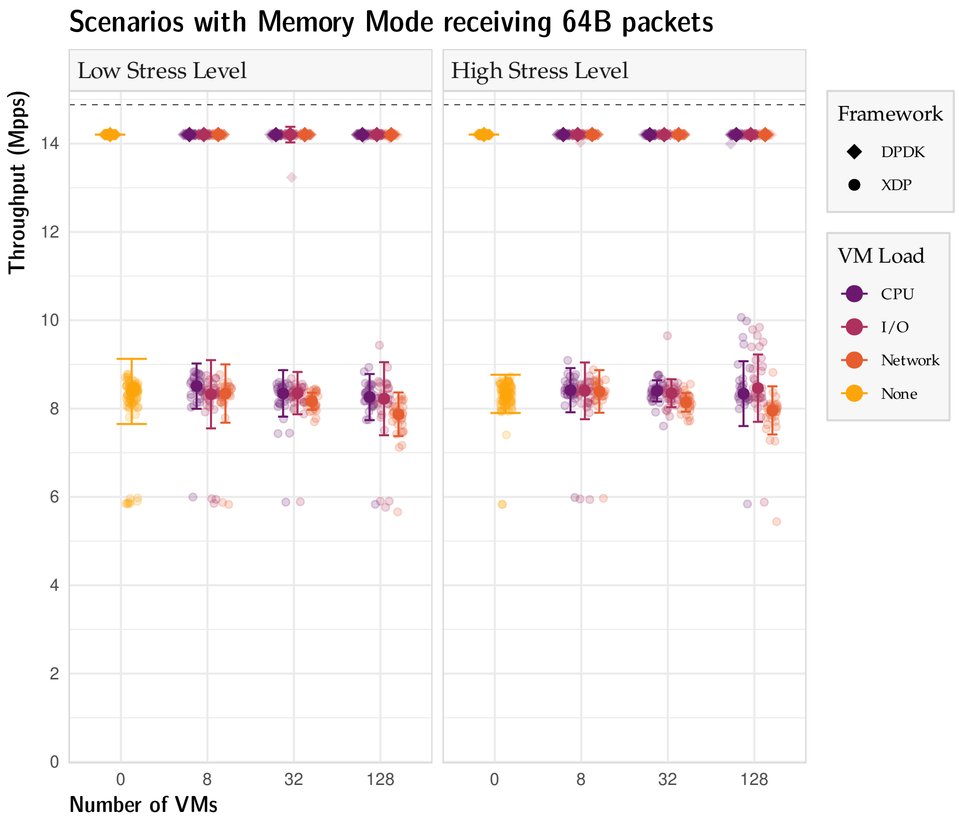

Figure 11 – Point plot representing the median with the standard deviation range. A scatter plot is also displayed in the background to help visualize the spreading of the data. The dashed line indicates the NIC line-rate. This scenario displays the throughput of both frameworks according to the number of VMs based on the load type

But, this is not the case when using XDP. No sample could achieve the NIC line-rate or any close value. The throughput performance is variable, creating a higher standard deviation of 0.59Mpps for both stress level and VM numbers, whereas for DPDK this variation is 0.04Mpps. As for the median throughput, not only it is overall lower when using XDP, but the virtual environment causes an interference in the processing performance especially when conducting network load. This differs from the scenarios using 1500B packets, where the network load would perform the least impact among the other loads. When employing CPU load, there is an interesting small performance boost observed when increasing from 0 to 8 VMs. This happens due to the CPU wake-up time for interrupt delivery and response. With a low CPU usage, as in the case for 0 VM, the CPU stays in interrupt-based mode, which represents a significant overhead in response time and therefore in the per packet processing time. When the CPU load increases to 8 VMs load, the CPU switches to polling mode and delivers a better per packet processing performance. This result aligns with the ones in (EMMERICH et al., 2015b). In summary, these results converge with the ones observed previously, showing that the memory usage in both frameworks does not interfere in the throughput performance. They also complement the previous result, showing that the memory usage causes no performance interference even in the presence of a virtual environment, when using DPDK. Also, the results indicate that when using XDP with memory usage, the virtual environment does interfere in the throughput performance.

On the other hand, when DPDK performs tasks that require CPU usage, which is the case for the PPVE CPU mode, the virtual environment has a direct impact on throughput. Figure 12 displays such scenario, but only when performing a high stress level. The first observation is that the throughput has a higher variation in the presence of VMs, creating a wider standard deviation and outliers. The virtual environment makes the framework's performance unstable, which leads to such behaviour. Also, the overall throughput decreases with the insertion of VMs. The I/O load type is the first one to drop with the presence of the virtual environment, when using 8 VMs, keeping a "constant" drop along with the CPU load, as opposed to the network load that causes a considerable performance drop only when using 32 and especially 128 VMs. The network load drops from 10.7Mpps with 0 VMs to 9.8Mpps with 128 VMs, considering the median of these cases. As the network load competes for resources from the same NIC, more VMs sending packets creates an overhead that causes this performance loss of approximately 8%. This scenario demonstrates that when using DPDK to perform intense CPU tasks, the presence of a virtual environment populated by network load causes the throughput performance to drop consistently when using higher number of VMs. Also, when the VMs perform CPU or I/O load, the throughput drops smoothly as the number of VMs increase.

However, when performing CPU mode with low stress level, the amount of VMs creates almost no throughput interference. This result is visible in Table 8. The results barely

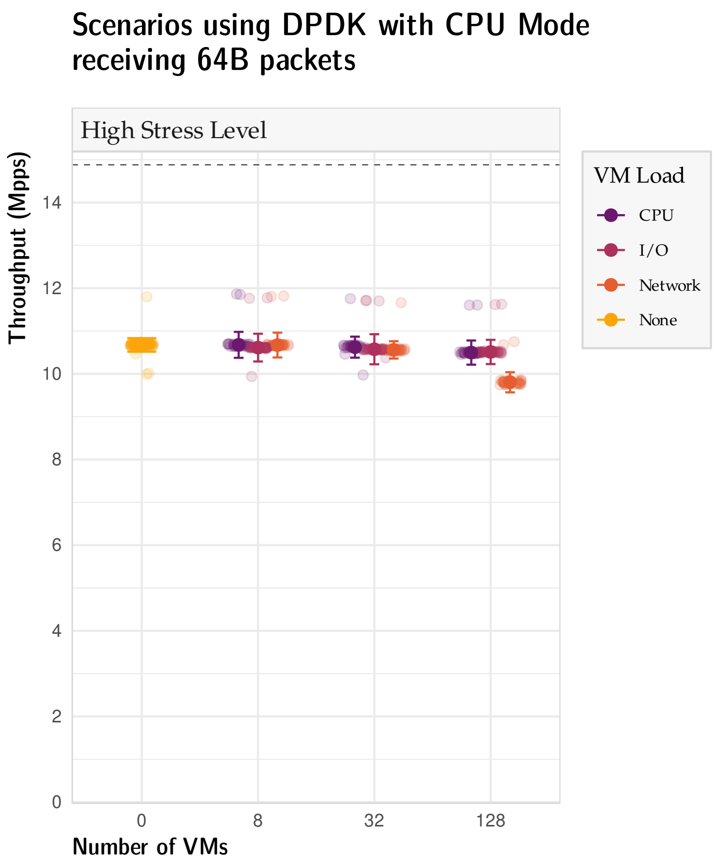

Figure 12 – Point plot representing the median with the standard deviation range. A scatter plot is also displayed in the background to help visualize the spreading of the data. The dashed line indicates the NIC line-rate. This scenario displays the throughput of DPDK according to the number of VMs based on the load type

Table 8 – This scenario compares the throughput of DPDK according to the VM number when using CPU mode with low stress level. All VM load types are included each VM number

Framework	Mode	Number of VMs	Throughput (Mpps)				
			Min.	Median	Max.	Mean	Std. Dev.
DPDK	CPU	0	14.202	14.203	14.205	14.204	0.001
		8	14.203	14.205	14.205	14.204	0.001
		32	14.185	14.205	14.205	14.204	0.003
		128	14.196	14.203	14.205	14.203	0.002

change the median or the mean, affecting only the third decimal place. The standard deviation is also minimal especially when compared to the ones in scenarios with high stress level. Some outliers appear when using 32 and 128 VMs, which indicates a virtual environment interference, but a small one, since the median is constant and only the standard deviation increases from 0.001 to 0.002Mpps. These results from Table 8 and Figure 12 support the ones seen in Section 4.4.1. They confirm the fact that higher CPU usage is critical to the DPDK's performance, and now it is visible that the virtual environment causes a performance degradation as well, intensified by heavier CPU usage.

As for the XDP framework, the CPU usage is more critical, as depicted in Figure 13. It is obvious how the increase of the CPU usage influences XDP's throughput, creating an overall drop of 57.85% considering the median. The heavier CPU usage by XDP is responsible for a higher performance degradation than the virtual environment, although the VMs are not free from causing an impact. The CPU VM load is the least impacting load from the virtual environment in this scenario, with a constant throughput value along the number of VMs. The network load though, has a higher interference, as expected, especially when using 128 VMs in both stress levels. This result demonstrates how much the CPU usage by the XDP framework impacts the throughput performance, and also that when combining this with the virtual environment, the throughput can decrease even further.

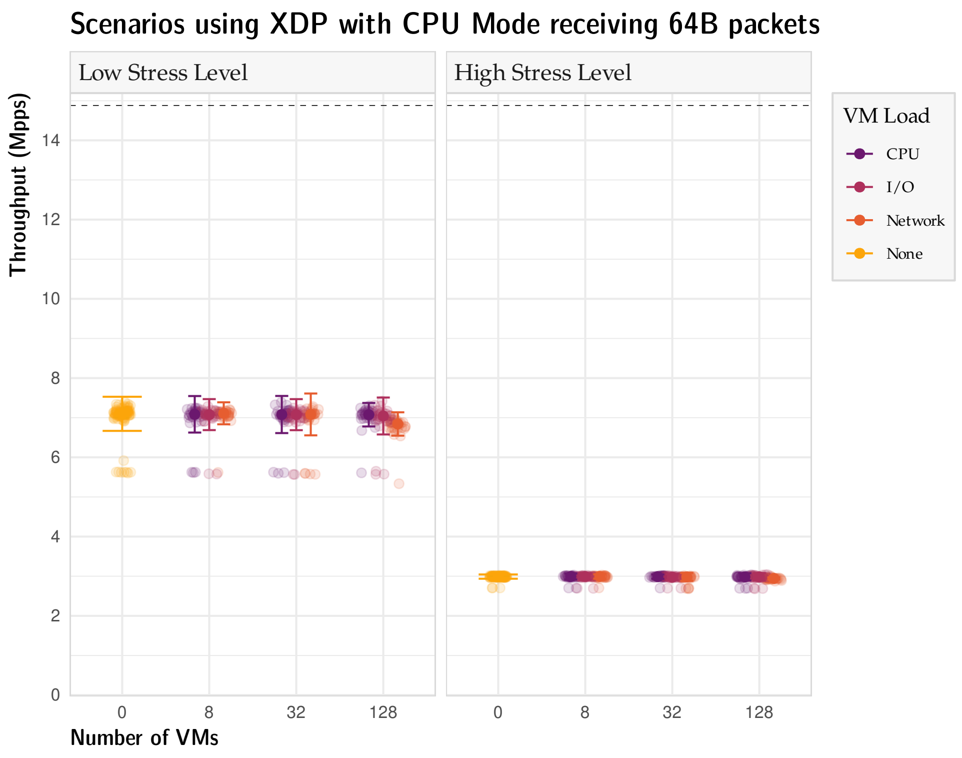

Figure 13 – Point plot representing the median with the standard deviation range. A scatter plot is also displayed in the background to help visualize the spreading of the data. The dashed line indicates the NIC line-rate. This scenario displays the throughput of XDP according to the number of VMs based on the load type

As for the case of packet dropping by the network card's driver, mentioned earlier when analysing the first part of the results, the virtual environment also increases the percentage of packets dropped, as illustrated in Figure 14. Recall the conclusion made in the first part of this chapter, where the packet size is a major factor when considering this response variable. This happens because it is directly related to the incoming packet rate that is responsible for the higher number of packets that fill the network driver buffer queue. But, it is possible to conclude that when processing packets of 1500B, the virtual environment lightly increases the packet drop percentage especially with the XDP framework. Also, when using the CPU mode with DPDK, the virtual environment imposes a higher drop percentage.

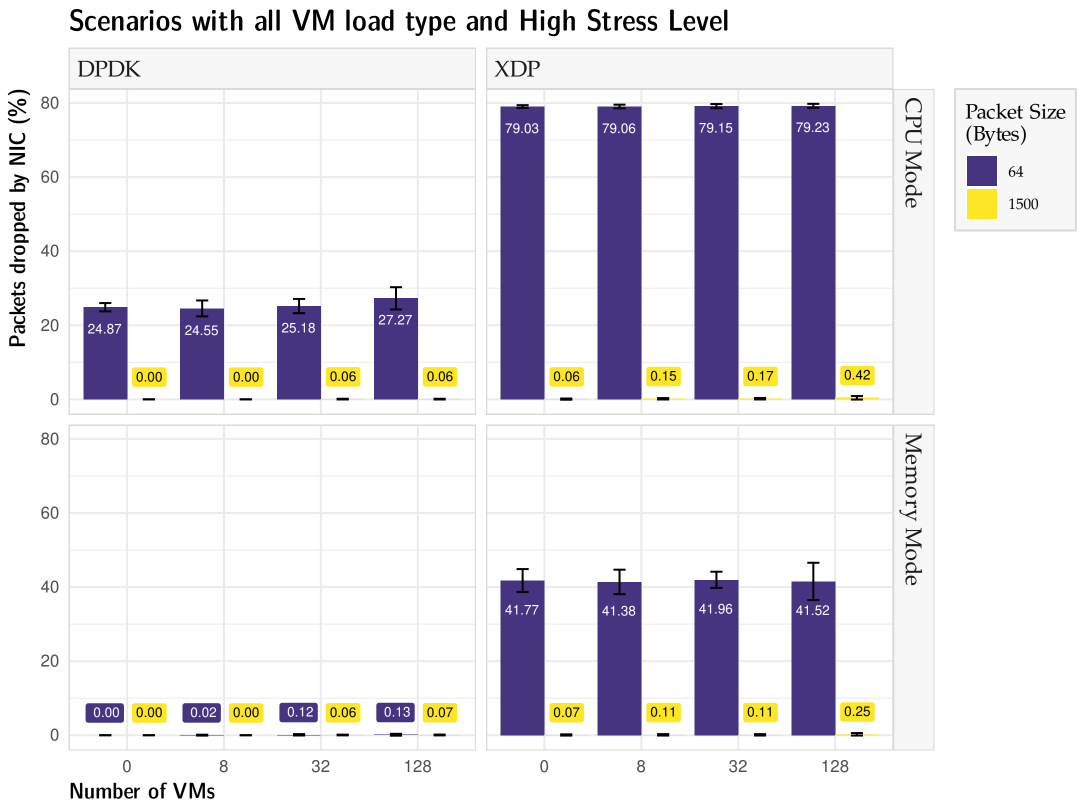

Figure 14 – Bar plot representing the mean with the standard deviation range. Each mean is labeled with its percentage value to ease the visualization of small values. This scenario compares the NIC packet drop percentage based on each VM number when using stress level 2, for each framework according to the packet size and PPVE mode

Alongside the presence of the virtual environment, the CPU mode again plays an important role in decreasing the performance of both frameworks and even more with XDP: when performing memory mode, XDP has a drop percentage mean of approximately 41.77% and when performing CPU mode this percentage reaches 79.03%, both when using no VMs. When deploying 128 VMs, XDP drops 79.23% of the packets sent. An analogous behaviour happens with DPDK, which drops no packets when using the memory mode but drops 24.87% when using CPU mode. With these results, it is possible to conclude that the virtual environment affects the NIC's driver packet drop rate, increasing the packet drop count in both frameworks but even more so in the case of DPDK. Also, the CPU usage by both frameworks is a major factor that contributes to increasing the packet drop rate, even more for the VMs. Additionally, the packet size influences this response, leading to a higher drop percentage when receiving small packets to process.

Calculating confidence intervals enables the usage of the range of values for comparison between the scenarios with the most highlighted setup. This is possible using bootstrap re-sampling with 1000 iterations to create a bootstrap distribution of the mean. Bootstrap is a statistical procedure based on randomly re-sampling with replacement the existing data set and calculating certain statistical measures for each new sample, in this case, the mean. The re-sampling process repeats n times creating a new normal distribution that is obtained from the original data set. The bootstrap procedure is well established in statistics (EFRON; TIBSHIRANI, 1994) (HESTERBERG, 2011), because it does not create new data nor substitutes the original data set during data analysis. Instead it uses the re-sample means to estimate how the original sample mean varies, since it uses random sampling. This new data set is used to calculate the confidence interval with 95% confidence level, illustrated in Tables 9 and 10.

Table 9 – Confidence Intervals for the throughput mean (in Mpps) with confidence level of 95%. Scenarios including CPU mode with low stress level

Framework	**Conf. Interv.**		**Mean**	
	0 VMs	128 VMs	0 VMs	128 VMs
DPDK	(14.2039, 14.2039)	(14.2033, 14.2033)	14.2039	14.2033
XDP	(6.9896, 6.9951)	(6.9016, 6.9063)	6.9924	6.9040

Table 10 – Confidence Intervals for the throughput mean (in Mpps) with confidence level of 95%. Scenarios including CPU mode with high stress level

Framework	**Conf. Interv.**		**Mean**	
	0 VMs	128 VMs	0 VMs	128 VMs
DPDK	(10.6774, 10.6795)	(10.3330, 10.3384)	10.6785	10.3357
XDP	(2.9807, 2.9814)	(2.9517, 2.9526)	2.9810	2.9522

Comparing the scenarios using the CPU mode is more interesting since it appears to draw the biggest impacts in different scenarios. Table 9 shows the results for 0 and 128 VMs for each framework, when using the low stress level. It is possible to conclude according to the obtained confidence interval that the interference created by the virtual environment when using DPDK is minimal, changing only the forth decimal place. However, for the XDP framework this interference is higher, dropping from a range between 6.99 and 7Mpps to 6.90 and 6.91Mpps. When performing high stress level, illustrated in Table 10, the scenario inverts from 0 to 128 VMs, where DPDK's performance degrades more than that of XDP. However, it is clear that the XDP's throughput performance is lower than DPDK's in all scenarios. The virtual environment's impact reflects only at decimal values, while the increasing CPU usage – assured by the stress level – drops the throughput by an overall general value of 4Mpps.

4.4.3 Discussion

In summary, this work performed experiments that compared XDP and DPDK packet processors under a standard configuration and no feature tuning like CPU pinning or memory reservation. The experiments programmed each framework to execute a different type of host resource usage named “mode”, varying from CPU or memory, each of them with a different usage intensity that called “stress level”. Alongside, each experiment contains a different number of running VMs to impose additional host resource usage. Each VM executes a resource load varying between CPU, I/O or network, called “VM load”. Each packet processor receives a burst of packets at NIC line-rate, varying each burst with two packet sizes, 64B or 1500B. The measured results are the achieved throughput and packet loss for each experiment scenario.

In the scenarios with no virtual environment execution, DPDK outperformed XDP regardless of the CPU or memory resource usage, but only when using small sized packets. This result is in accordance with those reported by the work in (HØILAND-JØRGENSEN et al., 2018). This is an expected result, due to the standard behaviour of DPDK to process packets in user space with custom drivers and full CPU single-core usage. However, when each framework receives big sized packets of 1500B, both frameworks achieve an equivalent throughput performance. The higher packet size consequently reduces the packet rate, which explains the fact that both frameworks achieved a similar result.

The results also point to the fact that throughput for both frameworks is highly impacted under heavy CPU usage. Heavier CPU tasks impose a throughput decrease of approximately 24.8% when compared to the throughput under lighter tasks when using DPDK. The same situation happens with XDP but with an approximate and larger 57.8% performance decrease. This is not the same for memory usage, that maintains a certain “constant” performance as the memory usage increases. These results shed light on how the CPU usage is critical to the packet processors. When performing heavy CPU usage during packet processing, the time processing of a single packet not only delays the processing of the following new packets, affecting the throughput, but also reflects on the network card buffer queue, that fills quicker and therefore drops more packets.

The virtual environment also has a considerable performance impact, but only across some specific cases. When XDP receives 1500B packets, the virtual environment has a direct performance impact, destabilizing the concentration and values of the throughput and increasing the NIC packet drop count. DPDK’s throughput in this same scenario is not affected by the VMs as much as the NIC packet drop count, that also increases as the number of VMs increases. This is explained again by the standard configuration of DPDK of fully consuming one CPU core for packet processing. Also, joining this feature with the low packet rate provided by the big packet size, the virtual environment does not impose a throughput constraint, allowing DPDK to stay in near line-rate packet processing.

An analogous situation happens with small sized packets. The virtual environment drops XDP's throughput especially when using the VM network load. However, this impact is not as intense as expected, especially when compared with the impact that heavy CPU usage imposes. This raises the conclusion that the framework's task may be more critical than the execution of a virtual environment. Heavy CPU usage degrades XDP's performance greater than a virtual environment with 128 VMs executing network load, for example.

When using DPDK, this tendency is similar only when it uses heavy CPU tasks. The performance decrease for scenarios with 0 and 128 VMs is larger than when using XDP, even though the throughput values themselves are higher with DPDK. When DPDK performs memory or lightweight CPU tasks, the virtual environment causes almost no impact on the throughput, only creating outliers on the analyzed samples. This yields to the conclusion that only in some scenarios the virtual environment affects the throughput performance sufficiently to cause actual performance loss.

The obtained results shed light on how both framework's features can impact their performance. XDP brings packet processing inside the kernel, maintaining its security and isolation measures. This includes also the usage of the standard kernel device drivers, that are not developed with highest performance in mind, as the results of the NIC packet drop show, for example. The DPDK approach with user space packet processing and a custom device driver presents a higher performance with less packet loss. Also, user space processing brings less restrictions towards memory and CPU operations, which also leads to higher throughput performance in most cases.

However, XDP also has multiple advantages. The same fact that it processes packets inside the kernel brings higher security to the host system and network cards as a whole. Along with the standard kernel procedures, the eBPF binary compiler and verifier performs greater byte code checks and verification to ensure kernel and host safety. A misguided programmer can develop an application that uses DPDK that can crash the operating system and misconfigure the NIC, which cannot happen when using XDP, since the eBPF environment verifies the program before execution and the XDP API executes inside the kernel.

Also, DPDK requires by default the full CPU core usage with busy polling, even if it is processing no packets. XDP on the other hand, escalates the CPU usage according to the incoming packet rate, so if there is no packet to process or that there is a low incoming rate, XDP's CPU usage is also low. This consequently frees host resource usage, that grows accordingly with the processing need.

5 CONCLUSION

The network packet processing capacity on GNU/Linux operating systems became an important factor to deal with on data centers, especially for cloud computing environments, over the recent years. With the advance of network cards and CPU processing power, the Linux kernel turned into a notable bottleneck in network processing, especially considering the emerging technologies like TSN, SDN, 5G and 6G and their new requirements for high-throughput and low-latency packet processing. In this context, new software solutions surfaced in an attempt to enhance packet processing under these recent systems. DPDK and XDP are two of these solutions, and represent the main ones. Each one applies a different method to enhance packet processing performance. DPDK uses the kernel bypass, that consists of eliminating the kernel from the network path and processing packets in user space. XDP, on the other hand, processes packets inside the kernel, at an early point right after the NIC receives them.

These frameworks can provide a great performance boost to cloud computing data centers, as the servers used there are mainly composed of multiple virtual machines that demand equal high packet processing with minimal interference. However, the same VMs that would require such high packet processing could end up limiting this packet processing capacity depending on the load it imposes on the host. The current state-of-the-art literature lacks experiments that expose the usage of these frameworks in such an environment and evaluate this behaviour.

This dissertation presents experiments aiming to evaluate how a virtual environment may interfere with DPDK and XDP performance, considering the throughput and the dropped packets count as the two main metrics. It also provides an initial comparison of DPDK and XDP when subjected to different work loads with the presence of no virtual environment.

In cases when there is no virtual environment and the framework should process large sized IP packets, both XDP and DPDK can perform near line-rate throughput performance, exhibit a low packet loss, and are possible choices for implementation. If host CPU consumption is a constraint, XDP may be preferred since it escalates the CPU usage according to the incoming packet rate. However, if the framework should process mainly small sized packets, DPDK may be a better option since it can achieve near line-rate performance and low packet loss, with the trade-off of full CPU core consumption.

But, if a virtual environment is present when the framework should process mainly large size packets, the resource usage from the VMs degrades XDP's throughput performance. In this case DPDK stands out, especially if the packet processing tasks require intense host CPU usage.

As in the cases when the framework should process small packets, the packet processing task should receive attention. If the task requires intense host memory access and reading, both DPDK and XDP are not affected by it. However, DPDK should be preferred since the presence of the virtual environment does not affect its throughput performance as much as it affects that of XDP. On the other hand, if the task requires heavy CPU usage, the virtual environment affects both frameworks. In such case, XDP should be considered instead while using its feature to offload tasks to the network card and use the dedicated NIC hardware to process packets.

Heavier CPU tasks in combination with an intense virtual environment causes DPDK to loose more throughput performance than XDP, even though DPDK still achieves a higher throughput. The same goes to packet dropping by the network driver. But, increasing further the number of VMs may cause DPDK throughput performance to decrease even more, which turns the offloading option of XDP as a valuable one, for both freeing resources for the VMs and for increasing packet processing throughput. Another way to increase packet processing when the framework task requires heavier CPU usage is tuning more CPU cores to enhance its performance. However, as mentioned previously, this may not be an ideal option if it is running in a production environment dedicated for cloud applications, as the framework will use more CPU cores that could instead be used by the VMs.

If packet loss is a constraint, XDP should be avoided since it uses the standard Linux network device driver, that causes higher packet loss than when using DPDK.

5.1 CONTRIBUTIONS AND TAKEAWAYS

Taking into consideration the questions this research seeks to answer, referenced in Section 1.2, and the results achieved by the experiments, this work counts with the following contributions and takeaways:

- Is the performance of **in-kernel** processing approach proposed by frameworks like XDP surpassing the performance of **kernel bypass** technologies from frameworks like DPDK?
  - In most cases, this is not the case. When the framework should process big packets as in 1500 bytes in a bare-metal environment, both DPDK and XDP perform similarly.
- Is the **virtual environment** from cloud computing servers affecting the performance of fast packet processing frameworks?
  - When using XDP, then the answer is affirmative. With DPDK, it is only when processing tasks with high CPU usage and small packet sizes.
- Is a cloud computing data center with **high disk I/O** usage like database or storage VM servers affecting the performance of fast packet processing frameworks?

  - For both frameworks, no. However, DPDK can achieve higher throughput performance especially in environments with higher number of VMs.

- Is a data center with **high network and CPU** usage like VNFs servers (running services such as DDoS protection, routing, DPI, Load Balancing) affecting the performance of fast packet processing frameworks?
  - Yes, especially with high network load. A higher VM number results in higher throughput degradation in DPDK than in XDP, even though DPDK can still achieve higher throughput performance.

- Also, the PPVE architecture is available to the community, with a modular structure that allows the possibility of its expansion to other packet processors as well as inserting different packet processing tasks with different resource usage levels.

5.2 DIFFICULTIES AND LIMITATIONS OF WORK

The accomplishment of this study came with some difficulties and consequently limitations associated to it. The time necessary to complete each experiment as a whole is considerably long, regarding the available time to complete this work. The main reason is due to the virtual environment initialization and termination. To avoid disturbance from different VMs resource loads from one experiment to another that could potentially interfere in results from different experiments, the experiment destroys all VMs when it finishes and start fresh new VMs at the next run. This causes a significant delay before and after each run, especially with the 128 VMs scenarios. This created a time constraint that ended up limiting the amount of different variables and responses this work could implement in each experiment. A specific and direct example is higher stress levels like 100 checksums calculations and 512MiB list size for both frameworks, that could not be applicable in the available time.

Also, the development of the PPVE architecture required a significant amount of research and tests needed to familiarize with both frameworks development environment, as well as to ensure compatibility for comparison of the functions and processing tasks.

5.3 FUTURE WORKS

As future works, it is intended to further explore the frameworks performance, with different response metrics as well as resource usage. First, evaluating resource utilization by both frameworks, such as memory consumption and CPU usage. As stated before, XDP offers a scalable CPU usage, as opposed to the DPDK approach of busy polling. As a result, evaluating the throughput achieved by these frameworks for different incoming packet rates can show how the resource usage of each of these behave according to the VM resource load. Works considering new network metrics such as, mainly, packet latency are also considered in future works. Furthermore, to assess the possible performance gain, experiments whit frameworks tuning, like memory space reservation and CPU cores pinning are intended.

REFERENCES

BELAY, A.; PREKAS, G.; KLIMOVIC, A.; GROSSMAN, S.; KOZYRAKIS, C.; BUGNION, E. Ix: A protected dataplane operating system for high throughput and low latency. In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. USA: USENIX Association, 2014. (OSDI'14), p. 49–65. ISBN 9781931971164.

BRUIJN, W. D.; DUMAZET, E. sendmsg copy avoidance with msg_zerocopy. In: *NetDev, The Technical Conference on Linux Networking*. Montreal, Canada: netdev, 2017. (Netdev 2.1). Available at: <https://netdevconf.info/2.1/papers/debruijn-msgzerocopy-talk.pdf>.

CORBET, J.; RUBINI, A.; KROAH-HARTMAN, G. *Linux Device Drivers*. third. [S.l.]: O'Reilly Media, Inc., 2005. ISBN 9780596005900.

Dantas, R.; Sadok, D.; Flinta, C.; Johnsson, A. Kvm virtualization impact on active round-trip time measurements. In: *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. [S.l.: s.n.], 2015. p. 810–813.

EFRON, B.; TIBSHIRANI, R. *An Introduction to the Bootstrap*. [S.l.]: Taylor & Francis, 1994. (Chapman & Hall/CRC Monographs on Statistics & Applied Probability). ISBN 9780412042317.

EMMERICH, P.; GALLENMüLLER, S.; RAUMER, D.; WOHLFART, F.; CARLE, G. Moongen: A scriptable high-speed packet generator. In: *Proceedings of the 2015 Internet Measurement Conference*. New York, NY, USA: Association for Computing Machinery, 2015. (IMC '15), p. 275–287. ISBN 9781450338486.

EMMERICH, P.; RAUMER, D.; BEIFUß, A.; ERLACHER, L.; WOHLFART, F.; RUNGE, T. M.; GALLENMüLLER, S.; CARLE, G. Optimizing latency and cpu load in packet processing systems. In: *2015 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*. [S.l.: s.n.], 2015. p. 1–8.

FOUNDATION, L. *Data Plane Development Kit*. 2018. Available at: <https://www.dpdk.org/>.

GALLENMüLLER, S.; EMMERICH, P.; WOHLFART, F.; RAUMER, D.; CARLE, G. Comparison of frameworks for high-performance packet io. In: *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. USA: IEEE Computer Society, 2015. (ANCS '15), p. 29–38. ISBN 9781467366328.

GROUP, D. P. D. K. *DPDK Docs: Programmer's Guide, Overview*. 2017. Internet. Accessed: September, 2020. Available at: <http://doc.dpdk.org/guides/prog_guide/overview.html>.

GROUP, D. P. D. K. *DPDK Docs: Programmer's Guide, Ring Library*. 2017. Internet. Accessed: February, 2021. Available at: <https://doc.dpdk.org/guides/prog_guide/ring_lib.html>.

HAN, S.; JANG, K.; PARK, K.; MOON, S. Packetshader: A gpu-accelerated software router. In: *Proceedings of the ACM SIGCOMM 2010 Conference*. New York, NY, USA: Association for Computing Machinery, 2010. (SIGCOMM '10), p. 195–206. ISBN 9781450302012.

HESTERBERG, T. Bootstrap. *WIREs Computational Statistics*, v. 3, n. 6, p. 497–526, 2011.

Hohlfeld, O.; Krude, J.; Reelfs, J. H.; Rüth, J.; Wehrle, K. Demystifying the performance of xdp bpf. In: *2019 IEEE Conference on Network Softwarization (NetSoft)*. [S.l.: s.n.], 2019. p. 208–212.

HØILAND-JØRGENSEN, T.; BROUER, J. D.; BORKMANN, D.; FASTABEND, J.; HERBERT, T.; AHERN, D.; MILLER, D. 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*. New York, NY, USA: Association for Computing Machinery, 2018. (CoNEXT '18), p. 54–66. ISBN 9781450360807.

IDG. *Cloud Computing Survey*. [S.l.], 2020. Available at: <https://resources.idg.com/download/2020-cloud-computing-executive-summary-rl>.

JEONG, E.; WOOD, S.; JAMSHED, M.; JEONG, H.; IHM, S.; HAN, D.; PARK, K. mtcp: a highly scalable user-level TCP stack for multicore systems. In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, 2014. p. 489–502. ISBN 978-1-931971-09-6. Available at: <https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong>.

KOURTIS, M.-A.; XILOURIS, G.; RICCOBENE, V.; MCGRATH, M. J.; PETRALIA, G.; KOUMARAS, H.; GARDIKIS, G.; LIBERAL, F. Enhancing vnf performance by exploiting sr-iov and dpdk packet processing acceleration. In: *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*. [S.l.: s.n.], 2015. p. 74–78.

Liu, J. Evaluating standard-based self-virtualizing devices: A performance study on 10 gbe nics with sr-iov support. In: *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. [S.l.: s.n.], 2010. p. 1–12. ISSN 1530-2075.

LIU, M.; CUI, T.; SCHUH, H.; KRISHNAMURTHY, A.; PETER, S.; GUPTA, K. Offloading distributed applications onto smartnics using ipipe. In: *Proceedings of the ACM Special Interest Group on Data Communication*. New York, NY, USA: Association for Computing Machinery, 2019. (SIGCOMM '19), p. 318–333. ISBN 9781450359566.

Miano, S.; Doriguzzi-Corin, R.; Risso, F.; Siracusa, D.; Sommese, R. Introducing smartnics in server-based data plane processing: The ddos mitigation use case. *IEEE Access*, v. 7, p. 107161–107170, 2019. ISSN 2169-3536.

MONNET, Q. *Dive into BPF: a list of reading material*. 2016. Internet. Accessed: January, 2021. Available at: <https://qmonnet.github.io/whirl-offload/2016/09/01/dive-into-bpf/#what-is-bpf>.

NIST/SEMATECH. *e-Handbook of Statistical Methods*. 2012. Available at: <http://www.itl.nist.gov/div898/handbook/>.

NTOP. *PF_RING ZC (Zero Copy) Guide*. 2018. Available at: <https://www.ntop.org/guides/pf_ring/zc.html>.

RIZZO, L. netmap: A novel framework for fast packet i/o. In: *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association, 2012. p. 101–112. ISBN 978-931971-93-5. Available at: <https://www.usenix.org/conference/atc12/technical-sessions/presentation/rizzo>.

RYBCZYńSKA, M. *Bounded loops in BPF for the 5.3 kernel*. 2019. Internet. Accessed: January, 2021. Available at: <https://lwn.net/Articles/794934/>.

SCHOLZ, D.; RAUMER, D.; EMMERICH, P.; KURTZ, A.; LESIAK, K.; CARLE, G. Performance implications of packet filtering with linux ebpf. In: *2018 30th International Teletraffic Congress (ITC 30)*. [S.l.: s.n.], 2018. v. 01, p. 209–217.

SOARES, L.; STUMM, M. Flexsc: Flexible system call scheduling with exception-less system calls. In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. USA: USENIX Association, 2010. (OSDI'10), p. 33–46.

SURVEYS, W. W. W. W. T. *Usage of operating systems broken down by data center providers*. 2021. Internet. Accessed: Feb, 2021. Available at: <https://w3techs.com/technologies/cross/operating_system/data_center>.

TSUNA. *How long does it take to make a context switch?* 2010. Internet. Accessed: January, 2021. Available at: <https://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html>.

TU, N. V.; YOO, J.-H.; HONG, J. W.-K. evnf - hybrid virtual network functions with linux express data path. In: *2019 20th Asia-Pacific Network Operations and Management Symposium (APNOMS)*. [S.l.: s.n.], 2019. p. 1–6. ISSN 2576-8565.

VIEIRA, M. A. M.; CASTANHO, M. S.; PACÍFICO, R. D. G.; SANTOS, E. R. S.; JúNIOR, E. P. M. C.; VIEIRA, L. F. M. Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Comput. Surv.*, Association for Computing Machinery, New York, NY, USA, v. 53, n. 1, Feb. 2020. ISSN 0360-0300.

WU, W.; CRAWFORD, M.; BOWDEN, M. The performance analysis of linux networking – packet receiving. *Computer Communications*, v. 30, n. 5, p. 1044 – 1057, 2007. ISSN 0140-3664. Advances in Computer Communications Networks.