BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

EVALUATION OF RTE_FLOW NETWORK INTERFACE CARDS SUPPORT

TESTOVÁNÍ PODPORY RTE_FLOW NA SÍŤOVÝCH KARTÁCH

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **JAKUB ŠURÁŇ**
AUTOR PRÁCE

SUPERVISOR **Ing. LUKÁŠ ŠIŠMIŠ**
VEDOUCÍ PRÁCE

BRNO 2022

Brno University of Technology
Faculty of Information Technology

Department of Computer Systems (DCSY) Academic year 2021/2022

Bachelor's Thesis Specification

Student:	**Šuráň Jakub**
Programme:	Information Technology
Title:	**Evaluation of rte_flow Network Interface Cards Support**
Category:	Networking

Assignment:

1. Study the possibilities of the DPDK library with an increased focus on the classifier interface rte_flow and learn how to manipulate with the application testpmd, which is part of the DPDK library.
2. Analyze real-world support and capabilities of rte_flow rules on network cards supported by the DPDK library.
3. Design a new tool capable of evaluating actual support and capabilities of rte_flow rules of the DPDK-supported network cards.
4. Implement the proposed tool and evaluate its functionality.
5. Conclude the achieved results and discuss possibilities of future work.

Recommended literature:

- According to the instructions provided by the supervisor.

Requirements for the first semester:

- Completion of items 1 to 3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Šišmiš Lukáš, Ing.**
Consultant:	Viktorin Jan, Ing., CESNET
Head of Department:	Sekanina Lukáš, prof. Ing., Ph.D.
Beginning of work:	November 1, 2021
Submission deadline:	May 11, 2022
Approval date:	October 29, 2021

Abstract

The support of the classifier interface `rte_flow` significantly differs across various network cards. This bachelor's thesis deals with the process of evaluation of this support. The main goal is to provide tools that can perform evaluation systematically and automatically. Two approaches are used for this purpose. The first one is based on the progressive loading of `rte_flow` rules into the network card and the collection of supported capabilities from the successful attempts. The results are used for the final summary generation. The second approach verifies that particular rules indeed have expected effects on the packets processed by card. Each of these approaches was then transformed into an executable tool. Both were applied and validated on several network cards by the Intel and NVIDIA manufacturers. Simultaneously, their produced outputs were utilized for the mutual comparison of `rte_flow` interface support between these network cards.

Abstrakt

Podpora klasifikačního rozhraní `rte_flow` se značně liší napříč různými síťovými kartami. Tato bakalářská práce se zabývá procesem testování této podpory. Hlavním cílem je vyvinout nástroje, které umožní provádět testování systematicky a automatizovaně. K tomuto účelu jsou využity dva přístupy. Ten první je založen na postupném nahrávání `rte_flow` pravidel do síťové karty a následném sbírání podporovaných vlastností z úspěšných pokusů. Ty jsou na konci využity k vytvoření závěrečného shrnutí. Druhý přístup naopak ověřuje, že jednotlivá pravidla opravdu mají očekávané efekty na pakety zpracovávané kartou. Každý z těchto přístupů byl následně transformován do podoby spustitelného nástroje. Oba byly aplikovány a otestovány na několika síťových kartách od společností Intel a NVIDIA. Zároveň byly výstupy obou z nich použity na vzájemné porovnání podpory `rte_flow` rozhraní na těchto síťových kartách.

Keywords

DPDK, rte_flow, rte_flow_checker, pytest, Scapy, MLX5, I40E, ICE, IXGBE

Klíčová slova

DPDK, rte_flow, rte_flow_checker, pytest, Scapy, MLX5, I40E, ICE, IXGBE

Reference

ŠURÁŇ, Jakub. *Evaluation of rte_flow Network Interface Cards support.* Brno, 2022. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Lukáš Šišmiš

Rozšířený abstrakt

V současné době se elektronická zařízení, jako například mobilní telefony, počítače nebo chytré hodinky, stala naprosto běžnou součástí životů nás všech. K rozšíření těchto technologických výdobytků značně přispěl rozvoj počítačových sítí v posledních desetiletích. Klíčovou událostí byla především masivní expanze Internetu. Díky tomu se k síti připojuje stále více všemožných zařízení. Tento rozvoj s sebou ale přináší i několik komplikací. Tou hlavní je stále se zvyšující objem přenášených dat a s tím související nároky na rychlost zpracovávání síťového provozu.

Částečné řešení těchto problémů nabízí knihovna DPDK. Ta dovoluje výrazně urychlit zpracovávání paketů, a tím pádem i tvorbu vysokorychlostních aplikací. Hlavní myšlenkou tohoto urychlení je přesun zpracovávání z jádra operačního systému na procesy běžící v uživatelském prostoru. DPDK nicméně poskytuje i další prostředky, pomocí kterých je možné zpracovávání síťového provozu dále zrychlit a zefektivnit. Jednou z možností je využití klasifikačního rozhraní `rte_flow`. Toto rozhraní umožňuje pomocí nahrávání speciálních pravidel přenést zpracovávání přímo na hardwarové komponenty síťových karet. Bohužel se však ukazuje, že míra podpory `rte_flow` se značně liší napříč různými síťovými kartami. Při vývoji reálných aplikací tak často vyvstává otázka, zda-li daná karta umožňuje skrze `rte_flow` optimalizovat určitou funkční logiku. Dostupná dokumentace je v těchto chvílích často nedostatečná. Z tohoto důvodu se jedinou možností stává explicitně ověřit podporu potřebných pravidel pomocí ručního testování. Toto je ale zdlouhavý a monotónní proces. Tato bakalářská práce se snaží tuto proceduru vylepšit. Hlavním cílem je poskytnout nástroje, které umožní provádět testování podpory `rte_flow` rozhraní systematicky a automatizovaně. Výsledným produktem těchto nástrojů je poté jednotné shrnutí popisující míru podpory `rte_flow` na dané kartě.

Začátek této práce je věnován knihovně DPDK. Úvodní sekce se zaměřuje na představení její architektury a nejdůležitějších částí. Dále je diskutována problematika ovladačů, které zprostředkovávají komunikaci mezi DPDK aplikací a síťovou kartou. Následující sekce podrobně rozebírá klasifikační rozhraní `rte_flow`. Důraz je kladen zejména na popis části, ze kterých se skládají `rte_flow` pravidla. Poslední sekce poté popisuje aplikaci testpmd, která je přímou součástí DPDK. Tato aplikace umožňuje manipulaci s funkcemi z DPDK knihovny skrze jednoduché textového rozhraní.

V úvodu následující kapitoly je vysvětleno, proč byl jako hlavní implementační jazyk nástrojů zvolen Python. Poté jsou v krátkosti představeny knihovny a nástroje tohoto jazyka, které jsou použity v rámci implementace. Prvním z nich je framework `pytest`, který umožňuje pohodlnou tvorbu automatizovaných testů. Další je knihovna `Scapy`. Ta poskytuje prostředky pro manipulaci s pakety, včetně jejich zasílání na vybraná síťová rozhraní. Kromě toho tato knihovna také umožňuje zpracování síťového provozu zachyceného do PCAP souborů.

V další kapitole jsou diskutovány přístupy k testování podpory `rte_flow` rozhraní. Ten první se zaměřuje na zautomatizování a vylepšení ručního testování. Podpora je zjišťována pomocí postupného nahrávání pravidel do karty a analyzováním úspěšných pravidel. Druhý přístup si naopak klade za cíl otestovat, že pravidla nahraná do karty opravdu mají očekávané efekty. V následující sekci je představen návrh DPDK aplikace, která je nezbytná pro realizaci zmíněných testovacích přístupů. Hlavním úkolem této aplikace je zajistit konfiguraci kartu prostřednictvím `rte_flow` rozhraní. Dalších dvě sekce se věnují návrhu nástrojů, pomocí kterých je možné realizovat oba uvedené testovací přístupy.

Následující kapitola popisuje implementační detaily. Úvodní sekce je věnována dříve zmíněné DPDK aplikaci. Ta byla nazvána `rte_flow_checker` a podporuje tři různé režimy.

V jednom z nich je schopná i zpracovávat síťový provoz. Díky tomu je zajištěno její použití v obou navržených nástrojích. Následující sekce se zaměřuje na implementaci prvního nástroje. Ten byl realizován pomocí skriptu v jazyce Python. Tento skript nejprve vygeneruje konfigurační soubor, který pokrývá různé kombinace vnitřních parametrů `rte_flow` pravidel. Soubor je následně poskytnut aplikaci `rte_flow_checker`, která ověří podporu každého z pravidel. Pomocí vyprodukovaných výsledků a analýzy původního konfiguračního souboru je následně seskládáno závěrečné shrnutí. Poslední sekce představuje implementaci druhého nástroje. Ten umožňuje prostřednictvím frameworku `pytest` testovat funkcionalitu jednotlivých pravidel. Testovací proces se skládá z několika kroků. Ze všeho nejdřív je aplikace `rte_flow_checker` použita na nahrání pravidel do karty. Následně je na kartu poslán vhodný testovací provoz. Očekávané výsledky jsou získány pomocí virtuální interpretace efektů nahraných pravidel na zaslané pakety. Ty jsou nakonec porovnány se skutečnými výsledky.

Závěr práce se věnuje testování implementovaných nástrojů. Oba byly aplikovány na několik síťových karet od společností Intel a NVIDIA. Díky tomu bylo možné je otestovat v různých prostředích a ověřit jejich funkčnost. K testování byla použita především aplikace testpmd. Výstupy z obou nástrojů byly porovnávány s výstupy analogických akcí prováděných pomocí testpmd. Kromě toho byly výsledky konfrontovány i se zkušenostmi získanými pomocí ručního testování `rte_flow`. Díky tomu bylo potvrzeno, že implementované nástroje jsou schopny správně prozkoumat a ověřit míru podpory `rte_flow` rozhraní. Vyprodukované výsledky proto byly posléze použity i k vzájemnému porovnání podpory `rte_flow` rozhraní na testovaných kartách. Ukázalo se, že největší míru podpory poskytují karty s `MLX5` ovladačem. Závěrečná sekce celé práce následně shrnuje dosažené výsledky a diskuje možnosti dalších rozšíření.

Evaluation of rte_flow Network Interface Cards support

Declaration

I hereby declare that this bachelor's thesis was prepared as an original work under the supervision of Ing. Lukáš Šišmiš. I have listed all the literary sources, publications, and other sources which were used during the preparation of this thesis.

.....
Jakub Šuráň
May 8, 2022

Acknowledgements

I would like to thank my supervisor, Ing. Lukáš Šišmiš, for his professional guidance, help, and constructive feedback whenever I needed it. My sincere gratitude also goes to my consultant, Mr. Ing. Jan Vikrorin, for his advice and countless consultations.

Contents

Chapter 1

Introduction

Nowadays, most people cannot imagine their life without the Internet. It spread over almost all areas of our society. As a result, it is natural that the number of devices connected to the Internet grows every year. That causes a gradual increase in the amount of transferred data. However, it also puts higher demands on hardware and software components of the network infrastructure. That leads to a steady effort for packet processing optimization.

Besides other options, the DPDK library can be used for high-speed network applications. It provides many possibilities for acceleration. One of them is to offload the part of the processing directly to hardware components in NIC. That can be achieved with the `rte_flow` interface. It allows the user to configure the NIC's offloads with special flow rules. However, the inner pieces of these rules can be combined in many ways. It leads to an innumerable number of possible flow rules. That brings some practical problems when developing real-world applications. The official documentation generally includes only a brief list of supported `rte_flow` features for each NIC. That is insufficient in many cases. In practice, the essential question often is whether NIC supports particular offloading logic (represented by a set of rules). DPDK does not provide many means which could answer this question. Because of that, the developer is left with brute-force manual testing. The typical workflow usually involves the repetitive loading of particular rules into NIC. From the result of every attempt, desired information can be determined. That might become monotonous and boring when dealing with several types of NICs.

This thesis tries to improve the described workflow. The goal is to provide tools that automate and extend the naive brute-force approach. These tools could simplify the work with `rte_flow` for many developers. E.g., by easily and quickly inspecting the `rte_flow` capabilities of any new NIC or verifying that a particular NIC supports the required set of specific rules.

The following chapter discusses the key features and tools from the DPDK library. It briefly introduces all relevant parts. The main focus is given to the `rte_flow` interface. The third chapter introduces the major Python libraries that have been used during development. The architectures of all proposed tools are presented in the fourth chapter. The fifth chapter is dedicated to the implementation details. It focuses on each tool as complex and all of its components. The sixth chapter presents the results produced by the actual application of implemented tools. They have been tested on various NICs, mainly from major vendors (Intel, NVIDIA, etc.). This chapter compares automatically generated results with the results gained from the manual testing. The final part is the conclusion, where the whole process described in this thesis is summarized. The possibilities for future work are discussed as well.

Chapter 2

DPDK

Data Plane Development Kit (DPDK) [14] is an open-source framework that enables faster and easier development of high-speed packet processing applications. It was created in 2010 by Intel and provided under the Open Source BSD Licence. 6WIND established an open-source community in 2013 to ensure and facilitate an expansion of this project. Another key breakthrough came in 2017 when DPDK was taken over by Linux Foundation. Development of the DPDK project, managed by the Linux Foundation, continues to the present day. At the beginning of 2022, versions up to 21.11 were available. This thesis uses the DPDK framework of version 20.11. A sign of DPDK's growing popularity is its integration into several significant open-source projects in recent years. DPDK is now core technology of projects such as TRex (traffic generator) [18], Open vSwitch (virtual multilayer switch) [13], or FD.io (software data plane based on vector packet processing) [16].

DPDK is implemented to be as platform-independent as possible. That is achieved through the creation of the Environment Abstraction Layer (EAL). EAL hides environment-specific details and provides a standard and unified programming interface. As a result, DPDK supports different processor architectures, such as Intel x64, ARM, or PowerPC. Apart from that, the framework can be used under various operating systems. Even though Linux and FreeBSD are currently the major target platforms, support for Windows has been added as well (although it still has some limitations[1]). Another significant advantage is that DPDK can be used in conjunction with NICs from different manufacturers. Major vendors, such as Intel or Mellanox, integrate native support of DPDK into the device driver of their cards. NICs of various vendors might require the installation of a custom device driver to work correctly with DPDK because the framework needs access to advanced parts of NIC.

The main goal of DPDK is an acceleration of packet processing. This acceleration is achieved by shifting data processing from kernel to processes running in user space. This kernel bypass allows an increase in packet throughput and computing efficiency (Section 2.1). However, this omission also brings some complications. One of them is the need to establish direct communication between the NIC and the DPDK application. For that reason, DPDK introduced Poll Mode Drivers (PMD), which are capable of arranging that. Generic PMD does not exist due to specific requirements and constraints of individual NICs. Therefore DPDK provides a variety of PMDs for particular types of NICs (Section 2.2). Besides that, it also offers many libraries. The `rte_flow`, which is one of them, is introduced in Section 2.3. Moreover, DPDK also includes several useful tools. Section 2.4 describes most widely used tool – testpmd application.

[1] https://doc.dpdk.org/guides-20.11/windows_gsg/intro.html

2.1 Framework overview

This section describes the architecture and main components of the DPDK framework. The motivation for its development has been an acceleration of network traffic processing. Traffic processing is relatively slow when performed classically (i.e., by the kernel). When a new packet arrives at the NIC, it has to go through a series of operations in kernel space (e.g., data copying, TCP/IP stack processing or socket operations) before it is made available to the application. A packet also needs to undergo a similar procedure when the application wants to send data. That is not a problem in most cases, as the user can see it as an advantage that the kernel processes data for him. However, in case of high-speed networks (10 Gbit/s and more), it becomes highly insufficient. DPDK tries to solve this problem by bypassing the kernel during packet processing. That means packets can be processed directly by DPDK libraries in user space applications. It significantly improves packet throughput and speed [7]. A comparison between packet processing directly with the kernel and with DPDK can be seen in Figure 2.1.

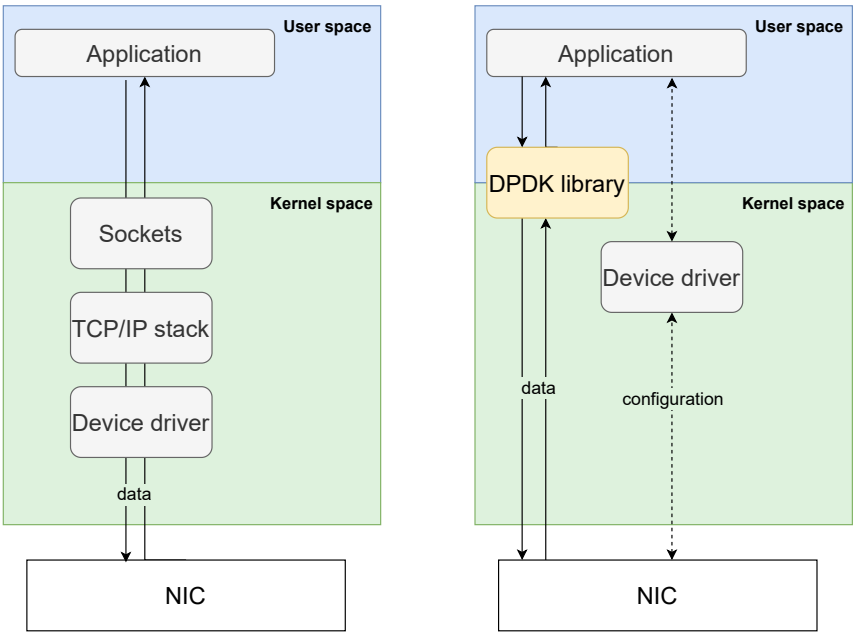

Figure 2.1: Comparison between packet processing without (left) and with DPDK (right).

When using DPDK, kernel space is used only for running the device driver with in-built support for DPDK. The device driver handles only the NIC's configuration, whereas the rest is done by DPDK in user space. DPDK provides Poll Mode Drivers (PMD) to ensure the transfer of packets from NIC directly to the application. Section 2.2 discusses PMDs in detail.

Environment Abstraction Layer

DPDK uses Environment Abstraction Layer (EAL) to hide specifics of a particular environment and make DPDK applications easily portable. EAL is mainly responsible for

initializing and gaining access to low-level resources (e.g., memory space). Apart from that, it also guarantees several important services, such as:

- Loading and launching of DPDK components,
- system memory reservation,
- interrupt handling,
- assignment of execution units to cores,
- utility functions (e.g., spinlocks or atomic counters).

EAL configuration can be modified by special command line parameters (EAL parameters). It is necessary to separate these parameters with `--` from parameters intended for the application itself. Every DPDK application should include EAL initialization at the beginning of its `main()` function. Function `rte_eal_init()` is responsible for this task. It allocates and initializes necessary resources (e.g., memory) at first. Then, it starts a special thread called logical core (lcore) for each configured core. Lcore is an instance of a POSIX thread (pthread) with some DPDK-specific metadata. Each core is usually assigned with exactly one lcore to prevent task switching. All initializations (e.g., EAL or shared components) should be performed in the MAIN lcore only. WORKER lcores are at WAIT state at that point. Once initialization is complete, the MAIN lcore starts a particular execution unit on each WORKER lcore. Then it waits until all lcores finish. At the end of the application run, function `rte_eal_cleanup()` should be used to release all allocated resources [15]. An example of described life-cycle, including initialization and cleanup, can be seen in Figure 2.2.

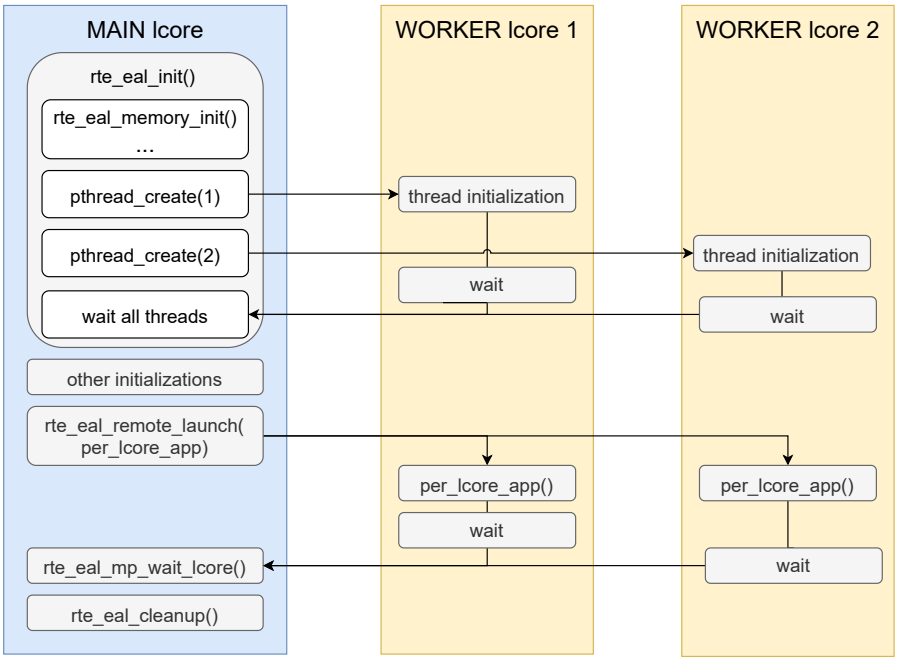

Figure 2.2: Life-cycle of DPDK application with one MAIN lcore and two WORKER lcores.

Memory in DPDK

As mentioned before, one of the important services provided by EAL is system memory reservation and management. It is necessary to perform the allocation of a large continuous physical memory block before running the DPDK application. Function `mmap()` is used for this task on Linux systems. EAL provides API to reserve zones in this block. Libraries for manipulation with this memory will be introduced later. It should be mentioned that on Linux systems, this memory block has to be allocated in *hugetlbfs*. *Hugetlbfs* is a special kernel filesystem that allows the usage of huge pages [12]. The reason why huge pages are used instead of normal pages is to increase processing speed even more. The concept of huge pages will be described in the following subsections.

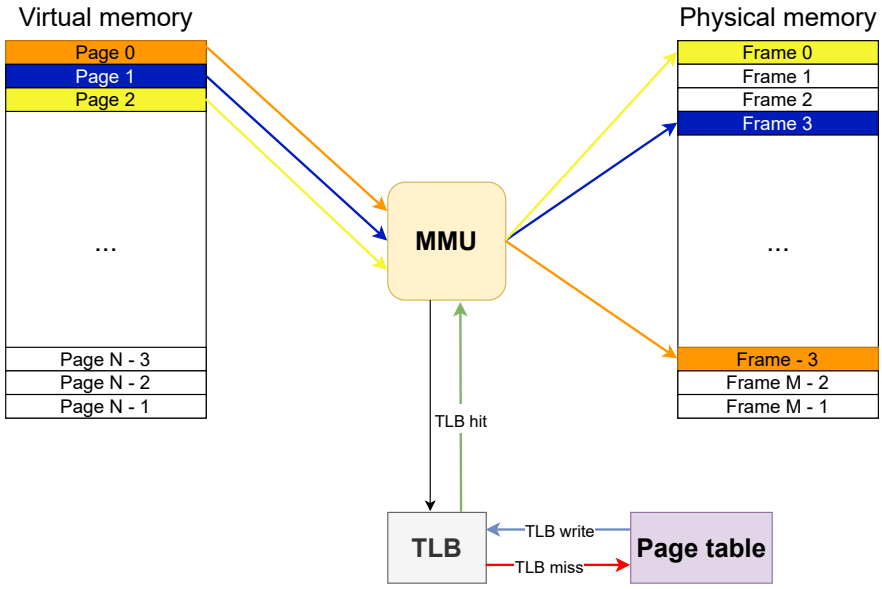

Figure 2.3: Translation of virtual addresses to the physical using the Memory Management Unit (MMU).

Memory virtualization

To fully appreciate the importance of huge pages, it is necessary to understand memory virtualization. An illustration of memory virtualization can be seen in Figure 2.3. Operating systems typically use this technique for RAM management. Besides physical address space, there is also virtual address space. Virtual address space is divided into blocks called pages. Page is the smallest fixed-length (usual size is 4 KiB) memory unit of virtual memory. In contrast to that, physical address space is divided into page frames. As a result, the process is given an impression that it is working with large and contiguous memory sections. But in reality, memory sections can be dispersed across different parts of the physical address space. Whenever a process needs to access data from memory, it is necessary to perform a translation of a virtual address to a physical address. The operating system is responsible for this action. The translation is usually performed by a hardware

component called Memory Management Unit (MMU). At first, MMU tries to look up the answer in the translation lookaside buffer (TLB). TLB is an associative cache of recently used translations. If the requested record is found (TLB hit), MMU returns particular physical memory. But if there is no match (TLB miss), MMU has to look the answer up in a page table. Found mapping is written back to TLB, and the translation is restarted. This time translation will succeed [4].

Benefits of huge pages

When using the usual size of pages (i.e., 4 KiB), the operating system is forced to work with a relatively large number of pages. That makes TLB less effective since TLB misses may occur very often. If the process works with a large amount of memory, it can significantly slow down the CPU. However, this problem can be eliminated by using larger pages. These larger pages are supported in most operating systems. They are called *huge pages* on Linux, *super pages* on BDS, and *large pages* on Windows. Their size usually varies from 2 MiB to 1 GiB [8].

Libraries for memory manipulation

DPDK offers a few different libraries which can be used to manipulate memory. One of them is the `rte_mallloc` library. This library provides API very similar to memory management found in standard C stdlib. Library `rte_malloc` re-implements all the functions from stdlib to allow allocation from huge pages memory region. Names of all these re-implemented functions are prefixed with `rte_`. However, this library is hardly used in DPDK applications. The reason is that allocations in most DPDK data processing applications can be managed much more efficiently by `Mempool` library. But in some use cases, it is necessary to use this library because it offers much more flexibility than the alternatives.

Significant libraries

DPDK framework offers a lot of libraries. Three important libraries will now be briefly introduced. Library called `rte_flow` is discussed in detail in Section 2.3.

Ring library

The `Ring` library provides a data structure called *ring* and operation for its management. The *ring* structure is a simple fixed-sized circular buffer. It is of type FIFO (First In First Out). It supports bulk enqueue/dequeue operations. That means it is possible to enqueue/dequeue multiple objects in one operation.

Mempool library

The core of the `Mempool` library is a data structure called a memory pool (*mempool*). *Mempool* is an allocator of fixed-sized objects. It uses a special handler for the management of its free objects. By default, this handler internally stores free objects in a *ring* structure. Allocation of the object from *mempool* can be divided into two steps. The handler first verifies that the *ring* isn't empty. Then it removes the first object from the *ring* and returns it. In case of the empty *ring*, the handler throws an error. Freeing the object is even simpler. The handler puts the freed object back at the end of the *ring*. The

demonstration of this process can be seen in Figure 2.4. Usage of a shared *mempool* in a multi-core application is a little problematic. Each time one of the cores wants to access *mempool*, it is necessary to use locks. That may cause performance losses. Fortunately, *mempool* offers a straightforward solution. During *mempool* creation, it is possible to request the initialization of per-core caches of free objects. Core does not have to use locks for manipulation with its cache. When it wants to interact with *mempool*, it first tries to use its cache to get a new object or free an old one. It needs to manipulate directly with the shared *mempool* only when its cache is full or empty. *Mempool*'s performance can be increased even more with appropriate memory alignment. For this reason, padding between objects is usually added.

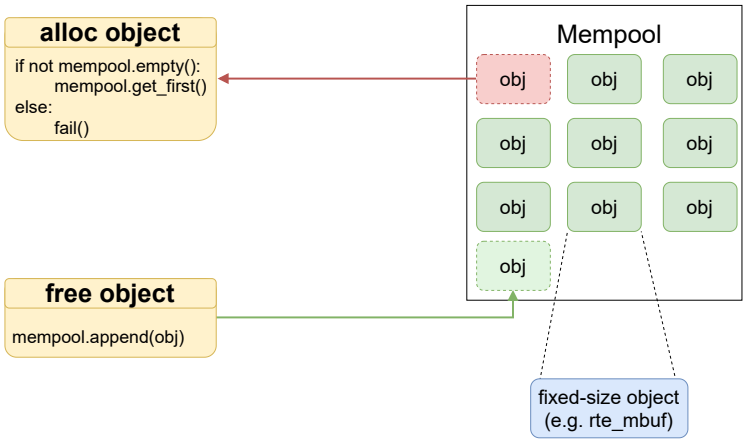

Figure 2.4: Manipulation with objects from memory pool (mempool).

Mbuf library

The `Mbuf` library has a close relationship with `Mempool` library. It provides special fixed-sized message buffers (*mbufs*). *Mbufs* are usually used to carry packet data and its metadata (e.g., length or type). For performance reasons, *mbufs* are stored in *mempools*. That means the API provided by the `Mempool` library is internally used for allocating *mbufs*. It is always necessary to specify the *mempool* from which new *mbuf* should be taken. Every *mbuf* includes a reference to the *mempool* from which it originated. Freeing a *mbuf* is equivalent to its return into the original *mempool*.

2.2 Poll Mode Drivers

NIC (similar to other I/O devices) requires a specialized device driver to manage communication with user applications. In modern operating systems, these drivers run in kernel space. They are based on asynchronous interrupt handling. Depending on the configuration of the driver, the received packet is stored either in NIC's internal buffer or ring buffer in kernel space. Besides that, NIC generates an interrupt request (IRQ) to inform the CPU

about a new packet. The CPU then delegates control to the particular interrupt service routine (ISR). ISR handles the packet processing – removes it from the buffer, processes it through the network stack, and passes it to the listening socket [2], [10]. However, interrupt handling slows down significantly the operating system. CPU has to stop the execution of a current task, call ISR, wait until ISR finish, and only then come back to the original job. This mechanism might become insufficient for a large number of incoming packets. The CPU might get overloaded by many IRQs and fail to handle all of them. This situation would result in a significant decrease in throughput.

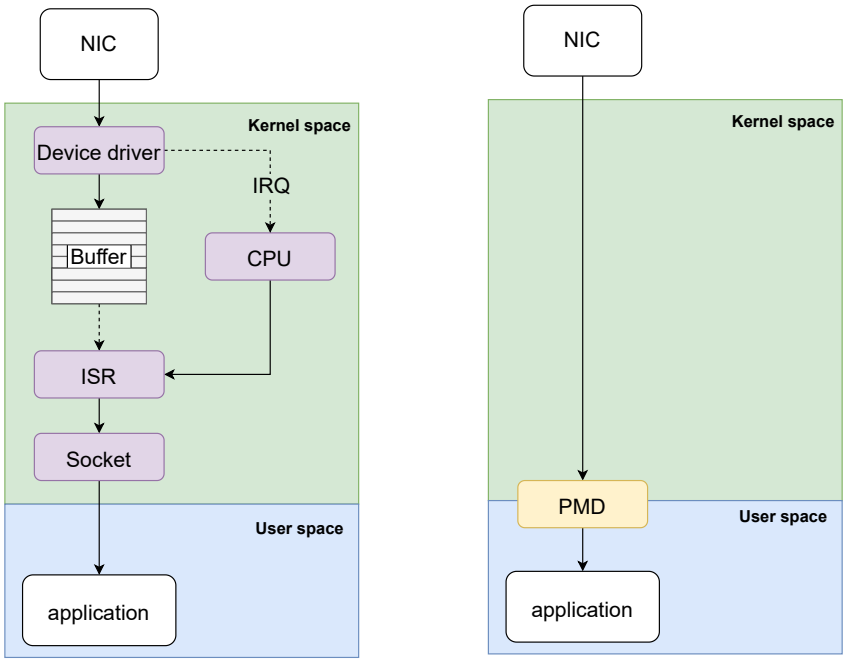

Figure 2.5: Comparison between interrupt handling (left) and polling (right).

Because of that, DPDK uses an alternative approach. Communication between application and NIC is managed by Poll Mode Drivers (PMDs). PMDs run in user space and directly access NIC's queues. But instead of interrupt handling, it uses polling. Polling in this context is a process when PMD actively samples the status of NIC. Status is changed when a packet arrives. In that case, received packets are immediately processed and delivered to the DPDK application [7]. This processing is much faster than interrupt handling. Packet reception with interrupts compared to polling mechanism is shown in Figure 2.5. However, the described technique also brings several negatives. Polling is a trade of faster processing and higher throughput for higher demand on resources (energy consumption and processor load). The polling process is also very time-consuming. For this reason, most of the DPDK applications dedicate several lcores purely for polling. All received packets are passed to other lcores via *ring* structure (discussed in Section 2.1) and further processed.

Besides polling, PMD also provides several other services. PMD offers API for DPDK applications to configure NIC and its receive (RX) and transmit (TX) queues. For this purpose, it internally uses routines from kernel space to achieve low-level communication with NIC. As mentioned before, the rest is done purely in user space.

Some types of NIC (e.g., most Intel NICs) require the usage of a special low-level kernel driver for their PMD to work properly. DPDK provides three different types of low-level kernel drivers that can be used for this purpose – `vfio-pci`, `uio_pci_generic` and `igb_uio`. Usage of these kernel drivers is required for NICs whose PMDs internally use `UIO` and `VFIO` drivers in their implementations [6]. In other words, for some NICs, it is necessary to load a special kernel driver and explicitly bound NIC's ports to it. This action (called binding) informs the operating system that a particular NIC won't be managed by the kernel but by DPDK. The binding consists of two steps. At first, it is necessary to unbind NIC from the kernel driver and then load the new one. It is possible to perform this operation manually by directly writing into particular files in the `/sys` folder. Code 2.1 shows example of unbinding from kernel driver `IXGBE` and binding to driver `vfio-pci` for device with PCI identifier 0000:82:00.0.

```
modprobe vfio-pci
echo -n "0000:82:00.0" > /sys/bus/pci/drivers/ixgbe/unbind
echo -n "0000:82:00.0" > /sys/bus/pci/drivers/vfio-pci/bind
```

Code 2.1: Manual binding of NIC to the `vfio-pci` driver.

To simplify this process, DPDK provides a special utility called `dpdk-devbind`. This utility is written in Python. An example of binding with `dpdk-devbind` is shown in Code 2.2.

```
modprobe vfio-pci
dpdk-devbind -b vfio-pci 0000:82:00.0
```

Code 2.2: Binding to the `vfio-pci` driver with `dpdk-devbind` utility.

However, several PMDs do not require these operations. It is because they internally use special bifurcated drivers. For this reason, they can co-exist with the device kernel driver. In this case, NIC is controlled by the kernel, whereas PMD manages only the data path. NICs by Mellanox are an example of adapters that use this model [3].

2.3 Generic flow API – `rte_flow`

One of the most powerful parts of DPDK is the generic flow API. This API is usually called `rte_flow` – named after the prefix used for all its symbols. Library `rte_flow` provides methods to offload packet processing and classification to hardware, thus gaining even greater acceleration. For example, it can be used to match specific traffic and perform various actions on it or configure and query hardware components, such as counters [7].

Flow rules are the core part of the `rte_flow`. These rules provide high-level means to describe intended functionality. They can be parsed and programmed directly into the hardware. The interface is uniform for all types of NICs. Conversion of flow rules into hardware is performed by PMD. Details of this conversion are highly dependent on NIC's hardware resources. That means PMD is responsible for handling all implementation details, such as conversion strategy, dealing with overlapping rules, etc. Due to different hardware limitations, it is not required to map all aspects of flow rules directly into the hardware. Instead of that, it is possible to implement certain functionality in software by PMD or leave it unsupported.

Flow rules

Flow rules consist of three main parts – attributes, matching pattern, and a list of actions. Visualization of a simple flow rule can be seen in Figure 2.6.

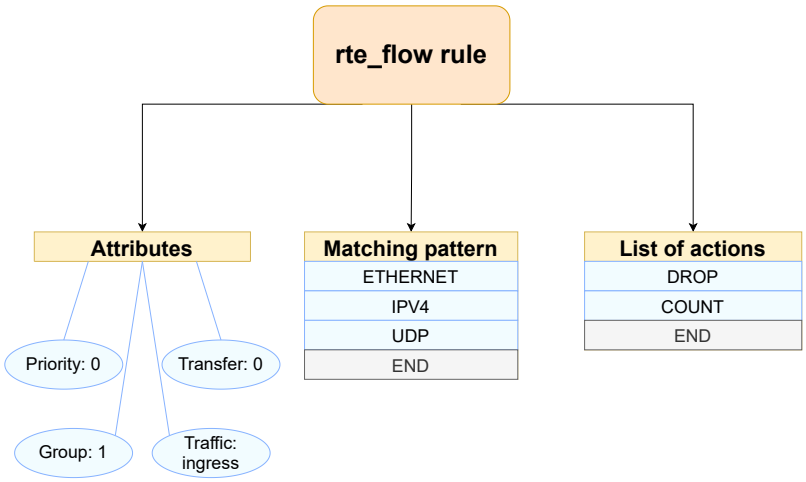

Figure 2.6: Flow rule with its three parts – attributes, matching pattern and list of actions.

There is a large number of combinations to build rules. To simplify work with them, `rte_flow` API provides a method for rules validation – `rte_flow_validate()`. This method checks if it is possible to program the rule into NIC with its current configuration. Similar to rule conversion, this check is in full control of PMD. The rule can be mapped to NIC with the call of function `rte_flow_create()`. Once the rule is created, it is internally represented by an opaque handle managed by PMD. This handle can be used for rules manipulation (e.g., updating of the rule or retrieving counter states) or for their destruction. All created rules should be destroyed by the application at the end of its run. Two function can be used for this purpose – `rte_flow_destroy()` (destroys only one particular rule) or `rte_flow_flush()` (destroys all rules).

Attributes

Every flow rule can be assigned with additional properties. These properties describe the rule's priority, group membership, etc. They are defined at the creation time and represented by structure `rte_flow_attr`.

Group

The first attribute is group number. It enables the division of the flow rules into a logical group hierarchy. That further extends possibilities in packet processing and filtering. Processing can be split into different levels, where each level is responsible for one task. The default group number is 0. All packets are processed by rules from this group at first. The rest of the groups can be reached only with a particular type of action – JUMP. An example of a simple group hierarchy can be seen in Figure 2.7. The hierarchy can be implemented

directly on the hardware level or virtually in PMD. However, not all NICs support more than one group.

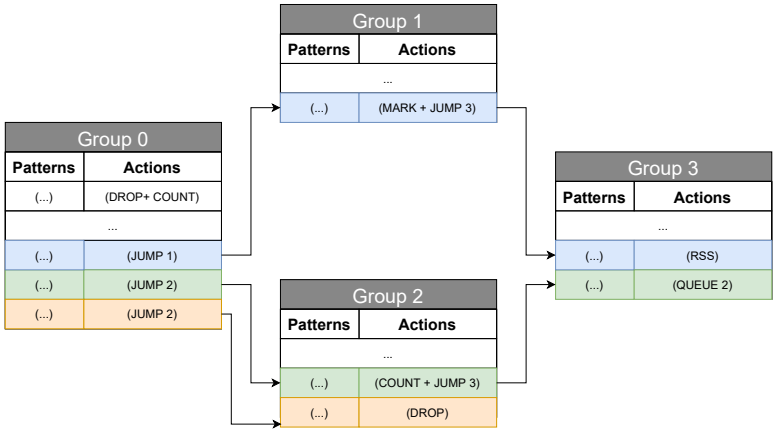

Figure 2.7: Hierarchy of the flow rules constructed by the JUMP action and group attribute.

Priority

Another attribute is the priority which relatively orders created rules. It expresses the priority level of a particular flow rule in a specific group. That means it only makes sense to compare priority levels inside a single group. The highest priority level is 0. If some packet matches multiple rules from one group, it is processed only by the flow rule with the highest priority. A situation with multiple matching rules with the same priority levels causes undefined behavior. Similar to groups, more priority levels are also not guaranteed on all types of NICs.

Traffic direction

The traffic direction attribute specifies to which type of traffic the rule should be applied. It is possible to match only inbound traffic, outbound traffic, or both. Most of the pattern items and actions can be used for traffic in both directions. At least one of them has to be specified in every rule. Specification of both directions at once is not recommended and should be used only in exceptional cases.

Transfer

The last attribute is called transfer. By default, the DPDK application can manipulate only traffic intended directly for it (i.e., going to its configured DPDK ports). However, enabling transfer attribute allows DPDK application to gain access to other traffic — e.g., addressed to other physical ports or applications. There are several pattern items designated for matching this kind of traffic. Flow rules also can reroute this traffic with particular types of actions. In conclusion, enabling the transfer attribute makes sense only for some pattern items and actions.

Matching pattern

The matching pattern controls to which packets the rule will be applied. In other words, it specifies properties to look for in network traffic. It is defined as a list of pattern items, where each item is represented by structure `rte_flow_item`. There are many types of pattern items. They can be divided into two basic categories:

1. The first category includes items that match the headers of network protocols. There is a pattern item for most of the common protocols, such as Ethernet, IPv4, TCP, ICMP, etc. These items have to respect the relation of a particular protocol to the network layer. For this reason, they cannot be combined arbitrarily in a matching pattern. They have to be stacked in the correct order, starting from the protocol of lower layers. Since network protocols usually consist of many different fields, these items have associated structures to match particular values in those fields. This category also includes one special item called ANY. It will be described more closely later.

2. The second category includes special meta items, such as END, VOID, INVERT, MARK, etc. Some of them affect pattern processing, whereas others are necessary for pattern definition itself. Most of them can be put anywhere in the matching pattern list. Some of the items also have associated structures for further specification.

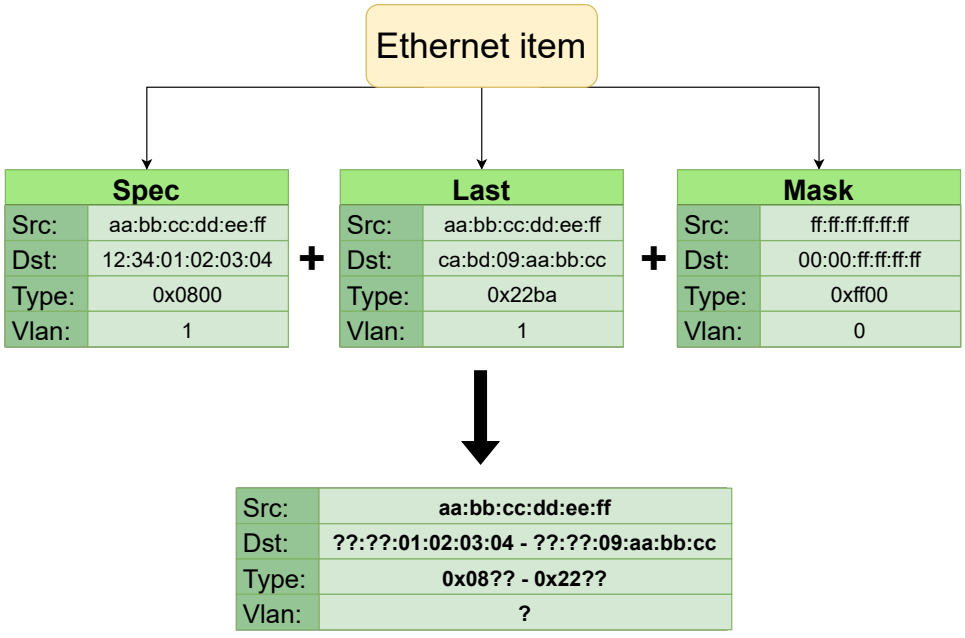

Figure 2.8: Specification of the matching properties for Ethernet pattern item.

Pattern items with associated structures can be further specified using three parameters – `spec`, `mask`, and `last`. An example of pattern item specification can be seen in Figure 2.8. In `spec` parameter it is possible to specify values to match, such as particular MAC addresses for Ethernet items. Parameter `mask` can be used to set bit-masks for particular fields. This way, it is possible to match values only in some fields or even part of the

fields. With `last` parameter, it is possible to set ranges for matching – values in `spec` is lower bound, whereas `last` is upper bound. Setting all 0 into the `last` parameter result in matching exact values specified in `spec`. Also, specification of `last` and `mask` parameters make sense only if `spec` parameter is set.

Some important pattern items that require closer explanation will be now briefly discussed:

END
: Used as an end marker in the matching pattern list.

VOID
: Is meant to be a simple placeholder and is skipped during parsing.

ANY
: Can be used to match any protocol of a particular layer.

INVERT
: Causes invert matching of pattern.

MARK
: Matches integer value previously set in the packet by MARK action.

PORT_ID
: Matches traffic going from or to given DPDK port (should be used in combination with transfer attribute).

Actions

The list of actions determines what will happen with the matching traffic. The list may consist of more single actions and needs to be terminated with a special end marker – the action of type END. Every action in the list is represented by the structure `rte_flow_action`. Actions fall into three categories:

1. The first category contains actions DROP, QUEUE, RSS, etc. These actions alter the fate of matching traffic. That includes dropping, redirecting, etc. Flow rule should contain at least one of the fate actions. Otherwise, the behavior is undefined.

2. The second category contains actions that modify packet content or assign new properties to it. E.g., by setting values in packet headers or adding marks. These effects can be achieved by actions SET_IPV4_SRC, SET_TP_DST, MARK, FLAG, etc.

3. The last category contains actions PASSTHRU, JUMP, COUNT, etc. They affect the rule itself. That includes switching to other rule groups or making the rule non-terminating.

Similar to pattern items, some actions have an associated configuration structure. This structure allows further specification of action's behavior. That is the case mainly for more complicated actions such as RSS.

Several important actions will be now briefly introduced:

VOID
: Can be used as a convenient placeholder.

QUEUE
Redirects packet to RX with index specified in configuration structure.

PASSTHRU
Makes the rule non-terminating. That means matching traffic can be processed by subsequent rules afterward.

JUMP
Redirects packets to a particular rule group. If there is no matching rule in that group, behavior is undefined. Only jumping from groups with a higher hierarchy to lower is usually supported. That is due to hardware restrictions and to avoid the creation of infinite loops.

MARK
Assigns integer to packet and sets special *mbuf* flags.

FLAG
It is very similar to MARK action. The only difference is that the FLAG action only sets the *mbuf* flags.

COUNT
Assigns counter to matched flow. This counter is incremented every time rule is triggered. The value of the counter can be retrieved by function `rte_flow_query`.

AGE
Adds timer to flow rule. If the timer passes without triggering the rule by matching traffic, a special event is reported.

RSS
RSS (receive side scaling) action is similar to QUEUE action. The difference is that the RSS action spreads packets among several RX queues. Because of that, the RSS action distributes the load between available resources. At first, it calculates the hash value from specified fields in the packet (hash type). Then, several least significant bits of the produced value serves as an index to the indirection table. Values in the indirection table provide mapping to particular queues [19]. A simple scheme of this process can be seen in Figure 2.9.

SET_IPV4_SRC
Sets a new IPV4 source address in the outermost IPV4 header found in the packet.

SET_TP_DST
Sets a new destination port number in the outermost TCP or UDP header found in the packet.

2.4 Testpmd application

Testpmd is an application shipped as a part of DPDK. Its primary purpose is to provide means to try out various DPDK features. That includes configuration of the NIC as well as testing packet processing with a particular configuration in practice. Apart from that, testpmd also serves as an example of how to build an advanced DPDK application. Since

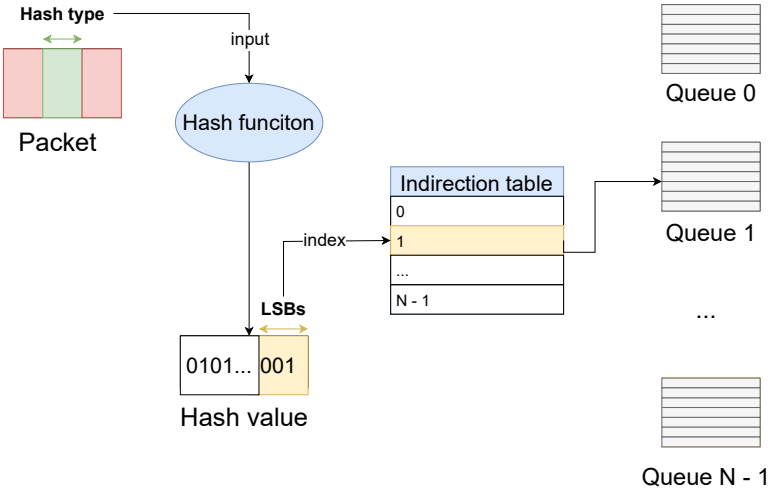

Figure 2.9: Scheme of receive side scaling.

testpmd supports interactive mode with easy-to-understand commands, it is also often used for demonstration purposes. E.g., most features in the documentation are explained in simple examples using testpmd commands. They are also commonly used in bug reports. Here they are used to describe the reproduction of problematic behavior.

There are several ways the user can interact with testpmd. First of all, it is possible to use one of many command-line options. Since testpmd is a regular DPDK application, it is possible to use EAL parameters (introduced in Section 2.1). Testpmd provides lots of command-line options itself. They can be used to request specific configurations (e.g., number of queues).

Testpmd can operate in two basic modes – non-interactive and interactive. Non-interactive mode is the default, whereas interactive mode needs to be requested by the command-line option (`-i`, `--interactive`).

In non-interactive mode, testpmd parses command-line options, performs necessary NIC configuration, and immediately starts packet forwarding. The user can terminate forwarding at any point. In the end, testpmd at prints statistics – the number of received and transmitted packets, etc.

In interactive mode, testpmd starts with the prompt. It allows the user to insert special text commands. Most of them are analogies of functions from DPDK API. Others provide convenient control of testpmd application (e.g., start packet forwarding). Even though testpmd supports various commands, for the scope of this thesis, only commands related to `rte_flow` are relevant. Basic `rte_flow` commands are introduced in Code 2.3. It is possible to write a sequence of commands in a text file. File with commands can later be loaded at the launch or run time. As a result, it is possible to avoid writing the same commands between multiple testpmd runs [5].

```
### validation of flow rule (analogy of rte_flow_validate())
flow validate 0 ingress \
    pattern eth / ipv4 src is 10.3.1.1 / end \
    action drop / end

### (analogy of rte_flow_create())
flow create 1 ingress \
    pattern eth / ipv4 src spec 10.3.0.0 src mask 255.255.0.0 / end \
    action queue index 3 / end

### (analogy of rte_flow_destroy())
flow destroy 0 0

### (analogy of rte_flow_flush())
flow flush 1
```

Code 2.3: Demonstration of testpmd commands related to the `rte_flow`.

Chapter 3

Python libraries enabling automated testing

The main objective of this thesis is to test the support and capabilities of `rte_flow` rules on NICs. Ideally, the testing process should be automated. For automation, Python has been chosen as the main implementation language. There are many reasons for this choice. First of all, Python has a very friendly syntax that is easy to read and understand. Secondly, it offers a large ecosystem of powerful libraries, frameworks, and other tools. That significantly simplifies and accelerates development. All these factors make Python one of the most popular languages for automation. This chapter briefly introduces the most important Python libraries for this thesis. The first part is dedicated to `pytest` (Section 3.1). The rest of the chapter is focused on `Scapy` (Section 3.2).

3.1 Pytest

`Pytest` [9] is a very popular third-party testing framework. Many Python packages use `pytest` as a base for their automated test management. Even some of the most popular packages rely on `pytest` – `Flask`[1], `Requests`[2], or `Pandas`[3]. Its huge advantage is flexibility. In comparison to other testing frameworks, it is not limited to a single development stage. `Pytest` can be used during the whole development process. It can manage unit tests, integration tests, or even more complex functional tests. Furthermore, `pytest` has a large community and a rich list of plugins to make it suitable for almost any use case.

Overview

`Pytest` offers a simple interface for test execution. The base of this interface is a powerful test runner. It can automatically discover tests and control their execution. The following process is applied to locate tests:

1. Recursively scan the specified directory or the current directory if no particular is specified.
2. Look for Python modules prefixed with `test_` or suffixed with `_test` in all found directories.

[1] Micro web framework – https://flask.palletsprojects.com/en/2.1.x/.
[2] Simple and elegant HTTP library – https://docs.python-requests.org/en/latest/.
[3] Data analysis and manipulation tool – https://pandas.pydata.org/.

3. Collect functions with `test_` prefix and `test` prefixed methods inside `Test` prefixed classes in all found modules.

The first two steps may be skipped by providing a specific directory or module name to the test runner.

In `pytest`, every test case usually follows the standard four-phase pattern known mainly from unit testing. It breaks test case execution into the following phases – `SETUP`, `EXERCISE`, `VERIFY`, and `TEARDOWN` [11]. The demonstration of the four-phase pattern is in Figure 3.1.

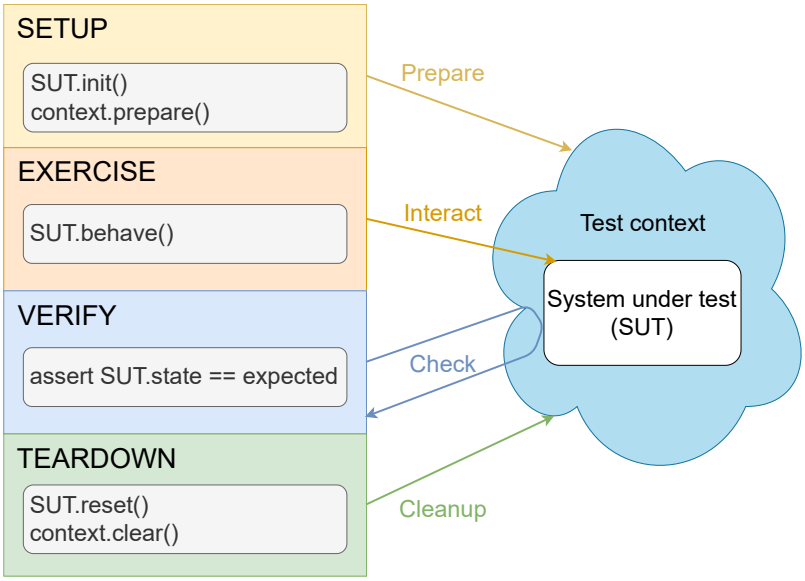

Figure 3.1: Four-phase testing pattern.

The `SETUP` phase typically includes the necessary preparation of the system under test (SUT) and all the other preconditions. These preconditions are called test context (or test fixture). It could mean objects initialization, adding entries into the database, generating configuration files, etc. In `pytest`, this preparation is normally achieved by special functions called fixtures (will be discussed in next subsection).

During `EXERCISE` phase, the main logic of the whole test is performed. It represents the behavior that is being tested. The behavior changes the state of the SUT and produces output that can be later inspected. This phase is usually equivalent to a function or method call.

The `VERIFY` phase analyses the results produced by the previous phase. It typically involves checking that SUT ended up in the expected state and testing behavior has expected effects. In other words, this phase determines whether the test passes or fails. It often takes the form of several assert statements.

The last phase is `TEARDOWN`. It is responsible for the clean-up of the SUT and test context. Besides that, it also releases all used resources. That allows the test to run in an isolated environment and not to be affected by previous tests. Similar to the `SETUP` phase, this step is often handled by fixtures in `pytest`.

Fixtures

The fixture sets up the SUT and test environment by providing reliable and consistent context. That allows tests to run repeatedly with the same results. In terms of the mentioned four-phase pattern, they are mainly applied during the `SETUP` and `TEARDOWN` phases.

In `pytest` terminology, fixtures are functions with a particular decorator. Test function can request execution of a fixture by adding the fixture name in its parameter list. That means that fixtures specified among parameters are executed prior test function run. The parameter represents the object or service initialized by the particular fixture. Because of this mechanism, the test function can access properties managed by fixtures. An example of fixture usage can be seen in Code 3.1. Fixtures themselves can also request execution of other fixtures in their parameter list.

```
# setup + teardown
@pytest.fixture
def prepared_calculator():
    calculator = Calculator()
    calculator.reset()
    return calculator

def test_calculator(prepared_calculator):
    # execute
    prepared_calculator.calculate('((50 + 41) * 6) / 13')

    #verify
    assert prepared_calculator.get_result() == 42
```

Code 3.1: Definition and usage of fixture in `pytest`.

Fixtures usually also include teardown logic. It is performed at the end of the fixture‘s life cycle. There are two possibilities for the definition of teardown logic. The first option is to define it as a special finalizer callback. The other option is to use the yield command. Both possibilities are demonstrated in Code 3.2.

```
@pytest.fixture
def seed_database_finalizer(database, request):
    entry = database.create_entry()

    def clear_database():
        database.remove_entry(entry)

    request.addfinalizer(clear_database)
    return entry

@pytest.fixture
def seed_database_yield(database):
    entry = database.create_entry()
    yield entry
    database.remove_entry(entry)
```

Code 3.2: Two possibilities for teardown logic definition.

Fixture can be defined on different scope levels – function, module, package, etc. Scope determines when the fixture is destroyed. It can be at the end of the test function run, when all test functions from the module have been executed, etc. That enables sharing of specific test data among more test functions or even test modules.

Parametrization

Because of the DRY principle, parametrization is a very practical feature. It allows one to generate multiple test cases from only one test function. `Pytest` natively provides API to achieve that. Even though test cases are generated from the same source, they execute independently.

Parametrization can be used on several levels – test functions parametrization or fixture parametrization.

Test function can be parametrized by using a particular decorator. Through this decorator, sets of argument values are passed to the test function. This action will result in the generation of a test case for each set of arguments. An example can be seen in Code 3.3.

```
@pytest.mark.parametrize('base', 'expected', [(-8, -7), (0, 1), (89, 90)])
def test_increment(base, expected):
    increment(base) == expected
```

Code 3.3: Test function parametrization.

It is possible to parametrize fixtures as well. That causes fixtures to execute multiple times. Each time with a different set of arguments. As a result, all test function, which depends on the parametrized fixture, will be executed for every fixture invocation. This behavior can be very useful sometimes. For example, when SUT allows configurations in multiple ways. A brief example is shown in Code 3.4.

```
@pytest.fixture(params=['Jack', 'Mike', 'Alec'])
def create_student(request, school):
    return Student(school=school, name=request.param)
```

Code 3.4: Fixture parametrization.

3.2 Scapy

`Scapy` [1] is a popular Python open-source project. It offers a very flexible means for the manipulation of network packets. It is possible to use `Scapy` in two different ways.

Firstly, `Scapy` provides an interactive console application. This application allows users to operate with network traffic in numerous forms – sending, sniffing, capturing, analyzing, etc. That makes `Scapy` a very flexible tool that can be applied in various situations. For this reason, it is frequently used as a replacement for other popular networking tools, such as `hping`[4], `arping`[5], or even `tcpdump`[6].

On the other hand, `Scapy` can also be used as a normal Python library. The user gains access to all API functions. This way, it is possible to extend `Scapy` without directly modifying its source code. As a result, one can build own tools, such as scalable packet generators, packets analyzers, etc.

[4] TCP/IP packet assembler/analyzer – http://www.hping.org/.
[5] Send ARP REQUEST to a neighbour host – https://www.root.cz/man/8/arping/
[6] Powerful command-line packet analyzer – https://www.tcpdump.org/

Packets manipulation

`Scapy` provides classes that represent standard network protocols. Packets with desired values are built by stacking those classes. `Scapy` automatically fills necessary fields of lower layers according to upper layers during this process (e.g., type in ethernet header). Constructed packets can be later sent to a particular interface with the `sendp()` function. A simple example is shown in Code 3.5. Unlike other similar tools, `Scapy` allows the user to build any invalid packet. That may be very useful in some cases (e.g., testing purposes).

```
packet = Ether() / IP('dst=195.168.0.1') / TCP(sport=80)
sendp(packet, iface='lo')
```

Code 3.5: Packet manipulation with `Scapy`.

Reading PCAP files

`Scapy` also offers an API for reading PCAP files. The user may choose from two approaches. The first one is the `rdpcap()` function. This function reads the whole PCAP file and loads it into memory. That might bring a few performance issues for large PCAP files. The alternative option is `PcapReader` class. This class reads the file sequentially and never completely loads the file into memory. It allows the user to iterate over captured packets and progressively process them. Because of this memory efficiency, `PcapReader` should be generally preferred when working with large datasets [17]. A brief demonstration of both approaches is in Code 3.6.

```
def parse_pcap_rdpcap(pcap_file):
    pcap_flow = rdpcap(pcap_file)
    sessions = pcap_flow.sessions()
    for session in sessions:
        for packet in sessions[session]:
            process_packet(packet)

def parse_pcap_Pcapreader(pcap_file):
    for packet in PcapReader(pcap_file):
        process_packet(packet)
```

Code 3.6: PCAP files processing with `Scapy`.

Chapter 4

Architecture of the proposed tools

Section 2.3 described the `rte_flow` library as a powerful instrument that is very easy to use. However, that is true only in theory. Its practical usage is a little bit problematic. There are many differences in support of `rte_flow` among various NICs. That complicates matters during the development of real-world applications. The official documentation usually provides only a brief overview of supported actions and pattern items for each PMD. But it does not answer other essential questions, such as which actions can be combined, is it possible to match a specific field of a particular pattern item with range and similar. Developers have to retrieve these pieces of information by themselves. Since no official tool for this purpose is provided, they are left with simple manual testing. The typical scenario usually is as follows. First, it tries to create various flow rules with tools such as testpmd (mentioned in Section 2.4). After that, the needed information can be determined from the obtained result of each creation.

The described manual method is not ideal. The main problem is that it is not systematic or scalable. For that reason, it would be much better to replace it with an automated alternative. The problem can be approached from two different perspectives – collecting NIS's capabilities and testing the actual functionality of the rules. Produced information by one approach completes information by the other and vice versa. Because of that, this thesis covers both of them. The main goal is to implement tools that automate both approaches and offer a pleasant user interface and access to synoptical information.

The first one tries to mimic and automate naive manual testing. That can be achieved through the creation of a simple pipeline. It is further discussed in Section 4.2.

On the other hand, the second approach tries to verify the functionality of flow rules. The possible problem of the first approach is that the support of the rule properties is determined only from the result of the API functions calls. But it does not inspect the rule's behavior. The second approach tries to improve that. Details are in Section 4.3.

Both approaches require a tool that can program the NIC through `rte_flow` API. It should accept the configuration file with desired flow rules. Section 4.1 focuses on the architecture of this tool.

4.1 Configuration tool

A simple configuration tool is an initial step on the road to automated `rte_flow` capabilities testing. Its primary responsibility is to program NIC through the `rte_flow` library. In other words, it serves the role of the access point to the main testing objective for other

components of the whole process. That is why the tool has to be a simple DPDK application. Apart from that, it should satisfy the following basic requirements:

1. Capability to use `rte_flow` API functions (mainly for rule validation and creation).
2. Support basic packet processing.
3. Generate information if a particular flow rule has been successfully validated/created.
4. Provide an interface for passing desired set of flow rules.
5. Generate necessary additional information for rule functionality verification(e.g., retrieve counter values or examine marks in packets).

The first candidate for this task is naturally testpmd application. However, there are several reasons why it is not a good choice in this situation. Let's go through the whole requirement list. The first three requirements are easily satisfied. Testpmd allows the user to call methods from the `rte_flow` API and use the forwarding mode. However, the subsequent requirement is already a little problematic. Testpmd does not provide commands for all types of rule properties. For example, even though the IGMP pattern item is a valid part `rte_flow` library, testpmd does not offer the equivalent command. Testpmd also does not give full control over the rule specification. For example, if the pattern item does not have an explicitly specified `mask`, testpmd uses its default values. That might sometimes result in unexpected results. It is also not trivial to generate all of the essential additional information. Testpmd is mainly designed for interactive usage by users. Because of that, commands are parsed in real-time. That causes delays in command processing which might result in unpleasant race condition artifacts in the produced output. That complicates the automated control and extraction of information. That makes testpmd unfortunately not very suitable for automation.

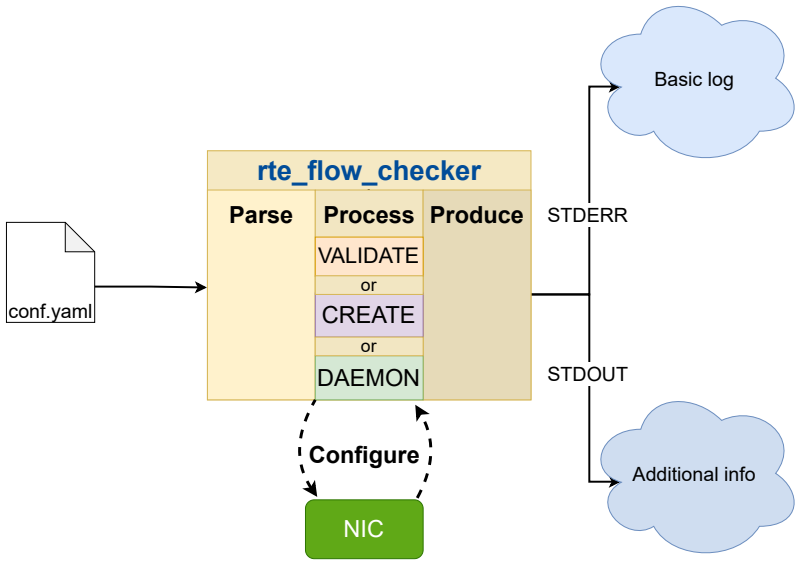

Figure 4.1: Architecture of the `rte_flow_checker`.

Testpmd and other existing tools turn out not to be the ideal candidates. For this reason, it is better to implement a new custom tool and optimize it to the intended use

case. From now, this tool will be referred to as `rte_flow_checker`. Its architecture can be seen in Figure 4.1. It parses the given configuration file with flow rules at first. The latter processing depends on the chosen run mode. Generally speaking, it interacts with a particular NIC and produces an appropriate report with obtained results.

Configuration file

The YAML is a markup language that is often used for configuration files. That is primarily because of its versatility. Furthermore, it is human-readable, and most programming languages provide libraries for YAML manipulation. For these reasons, it has also been chosen for `rte_flow_checker`. The proposed format of the configuration file tries to reflect the `rte_flow` API. Because of that, rules can be translated into the C code almost without changes. Code 4.1 demonstrates a brief example of the configuration file. It consists of the flow rules list. Each rule is represented by an associative array with three possible key-value pairs – flow rule parts (Section 2.3). Specification of attributes is optional and is realized by the associative array. The pattern and actions are represented by lists and are a compulsory part of every rule. Specifications of a particular pattern item and action mimic structures from the `rte_flow` API.

```
- actions:
  - drop: {}
  - end: {}
  attributes:
    direction: ingress
    group: '0'
    priority: '0'
    transfer: '0'
  pattern:
  - eth: {}
  - ipv4:
      mask:
        src_addr: 255.255.255.255
      spec:
        src_addr: 195.168.1.0
  - end: {}
```

Code 4.1: Example of configuration file for `rte_flow_checker`.

Run modes

Both testing approaches require slightly different behavior from the configuration tool. For this reason, `rte_flow_checher` needs to be a flexible tool. As already demonstrated in Figure 4.1, the `rte_flow checker` should support three run modes – `VALIDATE`, `CREATE`, and `DAEMON`. In the first run mode, all rules from the configuration file are validated with the `rte_flow_validate()` function. After each validation, `rte_flow_checker` generates a brief report describing the result. The second run mode is very similar. The difference is that rules are created with the function `rte_flow_create()`. The most complicated is the last run mode. At first, all rules from the configuration file are parsed and created. After that, `rte_flow_checker` starts simple packet processing in an infinite loop.

Processing continues until a SIGINT or SIGTERM signal is received. In this run mode, `rte_flow_checker` also generates a report with additional information. That includes the state of counters right before the termination, marks in received packets, etc.

4.2 Collection of `rte_flow` rule capabilities

The first approach to evaluation of `rte_flow` rules capabilities is the automation of naive manual testing. For this purpose, a simple pipeline can be constructed. Figure 4.2 shows its scheme. In the beginning, it is necessary to generate a set of appropriate flow rules. After that, the configuration file is passed to the `rte_flow_checker`. The `rte_flow_checker` then tries to validate or create each rule (depends on chosen run mode). The produced output and the original configuration file are then used for analysis. The pieces of information are gradually collected from successfully created/validated rules. As a result, a final summary is successively built in the end.

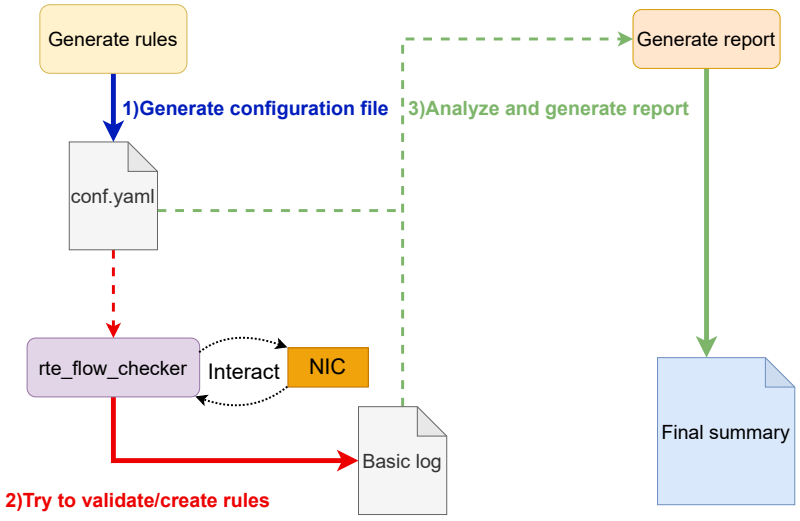

Figure 4.2: Pipeline for `rte_flow` capabilities collection.

Flow rule generation

The first step in the pipeline is a generation of the configuration file with flow rules. It should ideally cover properties from all three flow rule parts (Section 2.3). For that reason, generation should be split into three sections. Each section focuses on only one of the flow rule parts – attributes, pattern items, or actions. Generated rules in each section try to modify only the target part of flow rules. The remaining two parts should stay the same for all rules if possible. The main aim behind that is to isolate the main testing objective in each rule.

The attributes section is pretty straightforward. The priority and group can be tested by rules with various priority levels and group numbers. In case of direction and transfer, it is sufficient to cover all of their possible values. However, particular values of these attributes can be combined only with specific actions and pattern items. Generated rules have to respect those limitations.

The pattern items section is much more complicated. Most items have the associated structure with many fields. Furthermore, the matching pattern for each item is determined from a combination of three parameters – `spec`, `last`, and `mask`. For that reason, rules covering three following scenarios should be generated for each field:

1. Exact matching with a full bitmask.
2. Support of a partial bitmask.
3. Support of matching ranges defined with the `last` field.

In contrast to pattern items, most actions do not offer many configuration possibilities. The exception is only the some more sophisticated actions, such as RSS. Because of that, the most valuable piece of information is how actions can be combined.

Report generation

The main idea of report generation is extracting the information from successfully created/validated rules. Since each flow rule part requires a specific approach for extracting, the process should be again separated into three areas. In each section, the main component is internal context. This context holds the information that can be later used in summary generation. All contexts are progressively updated with relevant data from tested rules. For attributes, relevant data are the following things:

1. Supported group number.
2. Supported priority levels.
3. Supported directions (ingress, egress, and both at once).
4. Transfer attribute support.

For every pattern item:

1. Whether it is supported at all.
2. If it can be used alone in a pattern, or it is necessary to always specify the items from lower layers.
3. If it can be used empty – no `spec`, `mask`, or `last` specification provided.
4. Detailed information about support of each field (usage with full bitmask, partial bitmask, or matching range).

And finally, for each action:

1. Whether it is supported at all.
2. List of actions with which it can be combined.
3. Details about supported configuration (relevant only for actions such as RSS).

4.3 Testing of `rte_flow` rules functionality

The verification of the flow rule functionality is the alternative approach to testing. As apparent from Figure 4.3, the `pytest` framework (Section 3.1) serves the role of the primary director. As a result, fixtures are used for all necessary preparations during the `SETUP` phase. At first, a configuration file with flow rules is generated for each test case. Then, the `rte_flow_checker` is started in the `DAEMON` run mode with this configuration. However, the subsequent verification raises the need for capturing of process packet. Fortunately, the DPDK framework provides a tool designed precisely for this use case – utility called dpdk-pdump[1]. It has to be used with the other DPDK applications as a secondary process. Once started, it can be configured to capture packets from the primary DPDK application into PCAP files. After this preparation, the traffic generator component sends appropriate traffic to the NIC controlled by `rte_flow_checker`. Besides captured traffic and output from `rte_flow_checker`, the `VERIFICATION` phase also requires expected results. They are obtained from virtual flow rule interpretation in software. The final task is the check that all expected packets have been captured. The additional information is used to verify that captured packets include correct marks and NIC's internal components have expected values (e.g., counters).

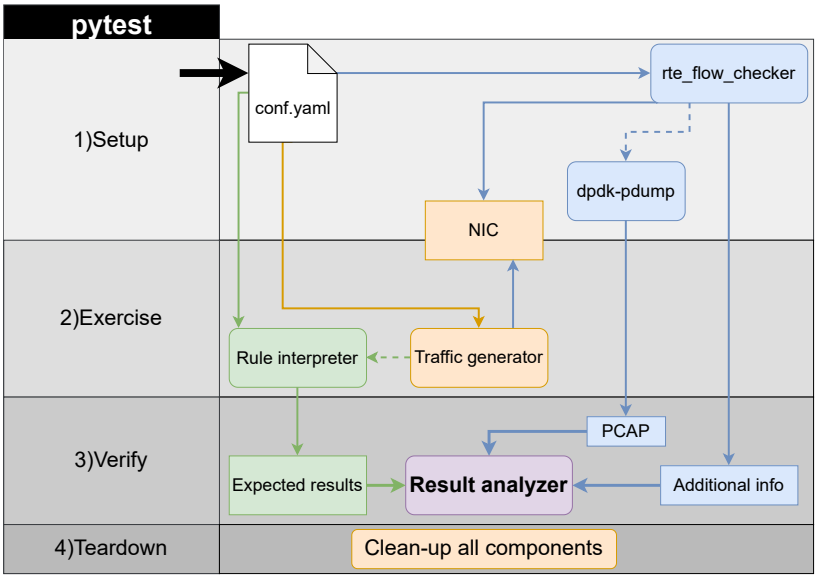

Figure 4.3: Testing architecture for verification of the flow rules functionality.

Traffic generator

The cardinal responsibility of the traffic generator is to construct and send the testing packets. It scans the configuration file and analyzes the pattern of each rule. Based on the analysis, a set of matching traffic can be generated. The non-matching packets are also generated since they are the necessary complement to the testing process.

[1]Tool that is capable of enabling packet capture on DPDK ports – https://doc.dpdk.org/guides-20.11/tools/pdump.html

Virtual flow rule interpretation

The expected results are the essential precondition of the evaluation for each test case. That includes the anticipated traffic for each queue and the additional information (e.g., state of counters). Because of that, all tested rules have to be interpreted in software to preserve maximum flexibility. For this purpose, the matching pattern and the list of actions from each flow rule are extracted. Derived information should be stored in an appropriate lookup table. After that, generated testing traffic can be processed. If a packet matches any of the rules, the effect of specific actions is simulated. That entails in gradual construction of the expected results.

Chapter 5

Implementation

This chapter focuses on the implementation details of the proposed testing approaches. The first one (Section 4.2) gradually collects supported capabilities of `rte_flow` rules on particular NIC. On the other hand, the second approach (Section 4.3) tries to verify that rules have expected effects. The most significant component for both is the `rte_flow_checker`. It is an application implemented in C which supports several run modes. Details can be found in Section 5.1. Python is the leading implementation language for all remaining parts. Section 5.2 is dedicated to the script, which loads flow rules into NIC and creates report based on information from successful attempts. It is a realization of the automated pipeline introduced in Section 4.2. The implementation of the second approach is based on the `pytest` framework (Section 3.1). It tests flow rule functionality by evaluating the effects of traffic sent to NIC. More information can be found in Section 5.3.

5.1 Application `rte_flow_checker`

The `rte_flow_checker` is a DPDK application which enables evaluation of `rte_flow` rule support on particular NIC. Various use cases requests different approaches to the evaluation. For that reason, the `rte_flow_checker` supports three run modes:

VALIDATE run mode
: It inspects the support of each flow rule from the configuration file with `rte_flow_validate()`. Every function call results in generation of appropriate log information. Since rules are not loaded into NIC, their mutual relationships cannot be tested.

CREATE run mode
: In contrast to `VALIDATE` run mode, it actually loads rules into NIC. Initially, specified burst of rules is extracted from the configuration file and successively processed with `rte_flow_create()`. After that, the application flushes out all create rules and repeat the process for another burst of rules.

DAEMON run mode
: At first, it tries to create given flow rules. If all creations succeeds, the application enters a packet processing mode. Otherwise it exits with appropriate error code. During the processing, it also collects the information necessary for the verification of flow rule effects.

As demonstrated in Figure 5.1, the `rte_flow_checker` consists of several collaborating components. The base component is the `parser`. It handles the processing of the input configuration file and the conversion of extracted values to the equivalent `rte_flow` structures. Every rule consists of three different parts as described in Section 2.3. Because of that, each of these parts has its handler. Converted rules are then utilized according to the configuration given to `parser`. For `VALIDATE` and `CREATE` run modes, those are the only performed operations. But when `DAEMON` run mode is used, the execution continues with a traffic processing. During this process, information about flags in received packets is collected by the `packet_flags` component. After the processing finishes, the `query_rule_properties` component comes to action. It performs all necessary queries of properties influenced by particular rules.

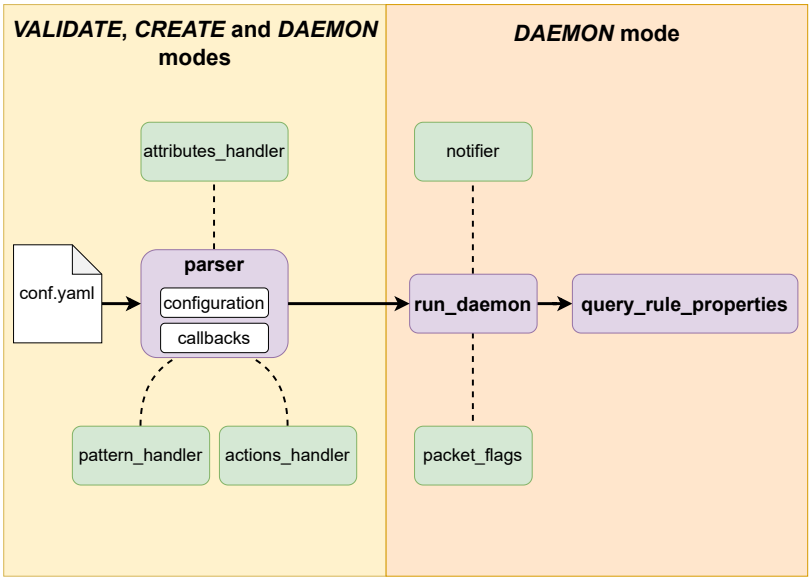

Figure 5.1: Components of the `rte_flow_checker`.

Parser

The `parser` component accepts the configuration structure and two callback functions – the first one for the processing of extracted rules and the second for potential clean-up. Both callbacks are also responsible for appropriate report generation. Because of this flexibility, each run mode can configure the `parser` according to its specific needs. The cardinal obligation of the `parser` is to verify the structure of the given configuration file and translate its content to the actual `rte_flow` rules. For the comfortable manipulation of the YAML file, the event-based API[1] from `libyaml` library is used. The sequence of processed events is controlled by a state automaton which is the realization of the configuration file format from Section 4.1. Figure A.1 shows the scheme of this automaton. The extracted SCALAR values are then passed to the appropriate handler, which reflects those values into proper `rte_flow` structures. When the end of the flow rule is reached, the `parser` applies the first of the given callbacks. The configuration file can also include several overlapping bursts of rules. That is implemented by the multiple documents inside a single data stream of the

[1] https://libyaml.docsforge.com/master/documentation/#events

YAML file. Each burst of rules can be stored inside a separate document. When the `parser` processes the entire rule group, it uses the clean-up callback to ensure that groups do not affect each other. However, it can be configured to skip clean-up after the last document of the YAML file. This behavior is utilized by the `DAEMON` run mode. The `parser` can also measure the total and average processing time for rules from each burst.

Handlers for `rte_flow` structures manipulation

The `parser` component works in close conjunction with three convenient handlers. For simplicity, there is a dedicated handler for every rule part. Their primary responsibility is to provide the abstraction for `rte_flow` structures management. Handlers accept the values extracted from the configuration file. These values are validated and converted to the proper data type. After this preparation, the converted value is set to the appropriate field in particular `rte_flow` structure.

Packet processing

In `DAEMON` run mode, there is a need for basic packet processing. It is managed primarily by the `run_daemon` component. Apart from that, this component also handles the necessary preparation before the start of processing. At first, it initializes the packet capture framework by calling the function `rte_pdump_init()`. That is necessary to enable the usage of dpdk-pdump in conjunction with `rte_flow_checker`. The `run_daemon` component is also capable of notifying the start and end of the processing. SIGUSER1 or SIGUSER2 can be used for this action. The parameter `--pid` allows specification of the PID of the receiving process. The binding of a signal to the specific notification can be done with `--initialized` and `--finished` parameters.

Once the preparation stage is done, packet processing can be started. Each RX and TX queues pair is handled by a separate lcore. The number of queues can be specified by a parameter `--queue-cnt`. The upper allowed value is the number of available lcores. The count of lcores can be configured by the EAL parameter `--lcores`. The MAIN lcore starts the packet processing loop on each WORKER lcore with function `rte_eal_remote_launch()`. After that, it waits until it is interrupted by SIGINT or SIGTERM signals. Received packets are retrieved as *mbufs* with the function `rte_eth_rx_burst()`. After that, component `packet_flags` processes them. It inspects the `ol_flags` field of each *mbuf* and tries to locate `RTE_MBUF_F_RX_FDIR` or `RTE_MBUF_F_RX_FDIR_ID` flags. They can be set by actions FLAG or MARK. Found flags are then stored into the structure shared among all lcores. All processed *mbufs* are freed by `rte_pktmbuf_free()` afterwards to make space for incoming packets.

After packet processing terminates, the control is passed to the `query_rule_properties` component. It relies on the close collaboration with the `parser` component. During rule processing, the `parser` stores the handles of rules with actions that can be queried. That is the case for actions COUNT and AGE. In the end, the `query_rule_properties` calls function `rte_flow_query()` for each of these rules and generates appropriate log information.

5.2 Automated inspection of `rte_flow` support

The first application of `rte_flow_checker` (Section 5.1) comes in the pipeline for collecting capabilities of `rte_flow` rules. This pipeline (Section 4.2) is realized by the `generate_report` script. It inspects the support of `rte_flow` on a particular NIC and generates an appropriate report. As can be seen in its detailed scheme in Figure 5.2, it heavily makes use of the products from external tools. That requires the means for spawning and controlling new child processes. Interface from the internal Python test suite of the Liberouter project is used for this purpose. Besides other things, it provides convenience wrappers for the `subprocess`[2] module.

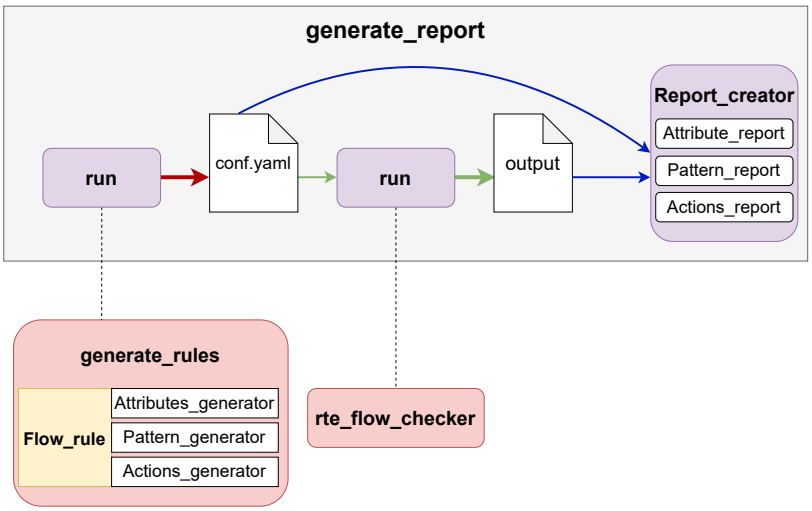

Figure 5.2: Scheme of the `generate_report` script.

The initial step of the whole procedure in Figure 5.2 is the application of the `generate_rules` script. It manages the generation of the configuration file with flow rules. Since this functionality might also be practical for other use cases, it has been extracted into the individual script to sustain adaptability. Moreover, it also contributes to the separation of independent functional units. Once the configuration file is successfully produced, `rte_flow_checker` comes to action. It employs generated file and applies it to the specific NIC. For the scope of the pipeline, `VALIDATE` and `CREATE` run modes are sufficient. The produced report is gathered for the needs of subsequent analysis. The final step is the utilization of obtained outputs and the creation of the overall summary. Both of these operations are performed directly by the `generate_report` script.

The `generate_report` script also accepts custom configuration files to simulate manual testing. That results in omitting the rules generation step. Instead of that, provided file is passed to the `rte_flow_checker`. The user might request this behavior with the `--conf` parameter. That might be effective when the user is not interested in the total summary of `rte_flow` capabilities on a particular NIC. But instead of that, he only requires verification of specific features.

[2] https://docs.python.org/3/library/subprocess.html

The user can get access to all rules used for inspection. To sustain direct compatibility with the DPDK framework, they are presented as testpmd commands. The `--success` parameter serves for the specification of the file with successfully validated/created rules. The rest of the rules might be collected into the file specified by the `--failed` parameter.

Besides the automated pipeline, the `generate_report` script also offers more functionality. It can further investigate the support of the `rte_flow` on a particular NIC. That involves inspection of the maximum rules capacity or their insertion time. Both information is measured in separate invocations of the `rte_flow_checker` independent on the pipeline. These measurements are performed only if the user provides appropriate configuration files through dedicated parameters (`--extra` and `--time-perf`).

Configuration files production

The configuration files can be automatically produced by the `generate_rules` script. As demonstrated in Figure 5.2, the `Flow_rule` class is its key element. It provides means for manipulating flow rules. Because of that, the representation of rules can be easily constructed and later expressed in a particular output format. The first supported format is naturally the YAML file for `rte_flow_checker`. For comfortable YAML handling, the `pyyaml`[3] framework is used. Apart from that, it is also possible to convert rules to the equivalent testpmd commands.

Because of colliding requirements, each rule part has its dedicated generator. That allows maximizing the coverage of tested features for actions, pattern items, and attributes.

The `Actions_generator` makes use of several helper generators. Each one focuses on one group of actions with similar features. That makes possible to use parametrization to cover all possible configurations. Apart from that, the possible action combinations are tested as well. At first, each one is tested alone. All non-fate actions are subsequently combined with every fate action.

The `Pattern_generator` applies a similar approach. It utilizes two helper generators specialized in diverse pattern item groups – with and without associated structure. The first one is responsible for generating rules that cover requirements from Section 4.2 for each item's field. The possible combinations with other pattern items are also tested for every item. The following process is applied for this purpose:

1. Generate a pattern where the tested item is situated in the last position, and the rest is filled by items from lower network layers.
2. Replace the first non-void item with the VOID item.
3. If the tested item is still present, generate a new pattern and repeat step two.

The `generate_rules` script can also produce rules with gradually increased IP addresses. They are utilized mainly for testing the maximum capacity on a particular NIC or insertion times.

Final analysis

The final analysis is managed by the instance of `Report_creator` class. Initially, it utilizes the output produced by `rte_flow_checker` to obtain the results of all validations/cre-

[3] https://pyyaml.org/

ations. Then, it iterates over the configuration file with flow rules to inspect the support of particular `rte_flow` capabilities.

The information extraction is again realized by three modules – one for each rule part (Section 2.3). But they all share common main logic. The extracted information is stored in a lookup table represented by the dictionary. Each item holds a piece of information about a single `rte_flow` property. Because of that, it is identified by the name of an attribute, pattern item, or action. A tuple is used to represent the value of the items. The detailed structure differs for each module to satisfy requirements from Section 4.2. But in general, a tuple consists of a couple of flags and arrays of strings. In case of pattern items, it is necessary to keep information about each field in nested tuples. Two separate tables are used for actions – the first for ingress direction and the second for egress. That is required because their support is usually strongly influenced by the traffic direction in flow rules.

Every processed flow rule is inspected by all three modules. If a particular module encounters an uncovered property, it adds a new entry to its lookup table. This operation can be triggered by rules regardless of the result of their validation/creation. However, the entries already inside the lookup table are updated only by the successfully validated/created rules.

The `Create_report` class is also able to find out more sophisticated information. That includes a maximum number of successfully created rules at the same moment. Once the configuration file is completely processed, it is possible to use lookup tables to construct the final summary. The summary takes the form of a reStructuredText[4] file. This format is human-readable and can be easily converted to a very synoptical HTML documentation with the help of the `Sphinx`[5]. Figure 5.3 shows an example of documentation with reports for several NICs. The column on the left includes the table of content, whereas the remaining part shows the beginning of the generated report for a particular NIC.

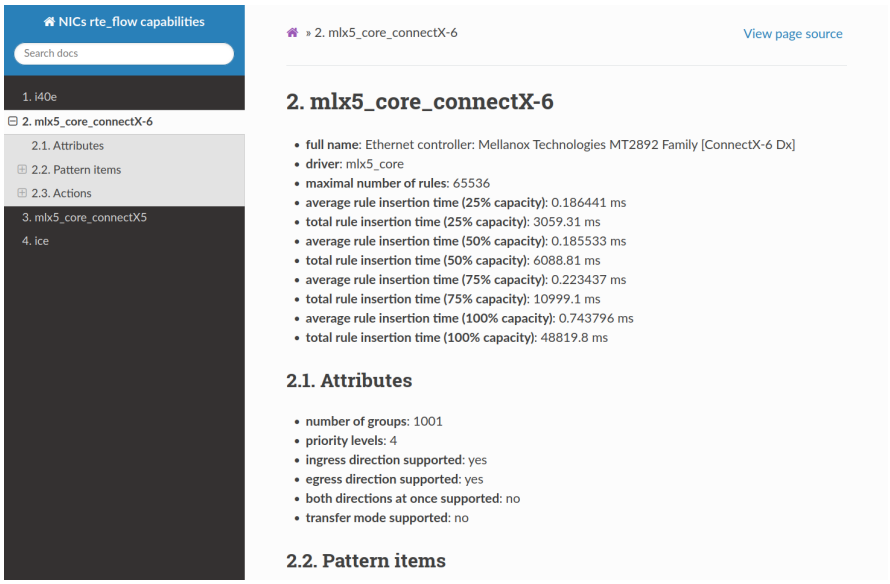

Figure 5.3: Example of HTML documentation that shows `rte_flow` capabilities support.

[4] https://docutils.sourceforge.io/rst.html
[5] https://sphinx-tutorial.readthedocs.io/

5.3 Rules verification with `pytest`

The actual functionality of the `rte_flow` rules is verified using a test suite based on the `pytest` framework (Section 3.1). It is a realization of testing architecture proposed in Section 4.3. Each test case starts with the necessary environment preparation. This step involves the creation of a file with flow rules and the initialization of `rte_flow_checker` (Section 5.1). If this step is not completed successfully (e.g., some rules from configuration files cannot be created), the test case is skipped. Otherwise, the testing traffic is sent to configured NIC by the instance of `Traffic_generator` class. Packets received by `rte_flow_checker` are captured by dpdk-pdump into PCAP files. Simultaneously, class `Rule_interpreter` produces the expected outcome that can be compared with the actual result. The final check includes inspection of captured packets and validation of other side-effects caused by rules.

Environment preparation

The initial phase takes full advantage of the `pytest` framework. As a result, all preparation steps are managed by the set of fixtures. They also include logic for a final tear-down – e.g., stopping `rte_flow_checker`. Figure 5.4 shows the sequence of performed operations. The first fixture generates flow rules. They are stored in two formats – YAML configuration and testpmd commands. The file with testpmd commands is not further used by the test case. It can serve for recreation of the particular test with the testpmd. The YAML file is then utilized to start `rte_flow_checker` in `DAEMON` run mode. Besides that, it is configured to emit a particular signal after the rules creation. When it is successfully received, the `rte_flow_checker` is considered initialized and ready to process packets. After that, the dpdk-pdump can be started to capture traffic from the `rte_flow_checker` application. Packets from each RX queue are stored in a separate PCAP file.

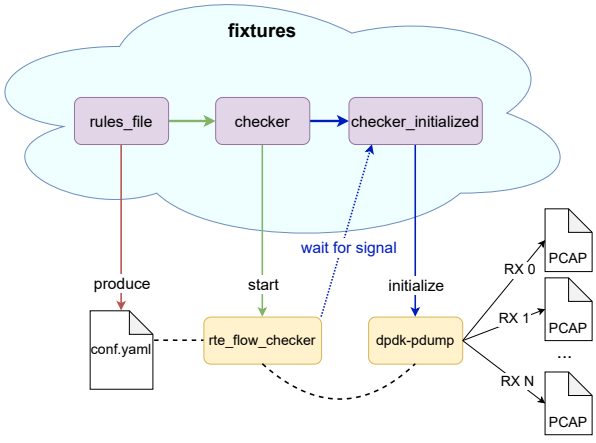

Figure 5.4: Preparation process performed before every test case.

The fixture, which generates the configuration file, is usually re-implemented for each test group to reflect specific requirements. Every group utilizes fixture parametrization to produce rules according to the needs of a particular test case. On the other hand, fixtures handling initialization of `rte_flow_checker` are the same for all test groups.

Generation of testing traffic

The implementation of traffic generation is primarily designed for NICs with at least two ports. The main idea is to leave control over one port to `rte_flow_checker`, whereas the other port serves as the interface for sending the testing traffic. The internal test suite of the Liberouter project provides an elemental set of fixtures for setting up this topology.

The `Traffic_generator` parses the configuration file and inspects the pattern of every rule. The result is the extraction of matching properties from each pattern item. They are later used to create matching as well as unmatching packets. Parts not specified by the pattern are filled with appropriate random values. The generator also tries to diversify the produced traffic by alternating used protocols in generated packets.

Because of simplicity, generated packets are not manipulated directly but through a couple of tree structures (Figure 5.5). Each path from the root to the leaf represents a single packet. A tree node symbolizes one protocol header. They are identified by values of the most significant fields of specific protocol – e.g., IP addresses or port numbers. Structure of the tree guarantees that nodes at the same depth are all from the same network layer. Besides that, `Traffic_generator` also keeps track of the nodes that belong to the currently processed rule. As a result, new nodes are added only to the right places in all appropriate trees. Node, which represents the L2 protocol, is used as the root of each tree. That means a request for the addition of the L2 node always causes the creation of the new tree. For the other type of nodes, the outcome is the addition of the new child to all relevant nodes from the lower layer. In the end, the pre-order traversal is applied to all constructed trees. As a result, each packet can be built with the `Scapy` library (Section 3.2) and sent to the tested NIC. The `Traffic_generator` might be forced to wait a couple of seconds before or after sending. That is necessary for a proper test of some rules, such as those that include AGE action.

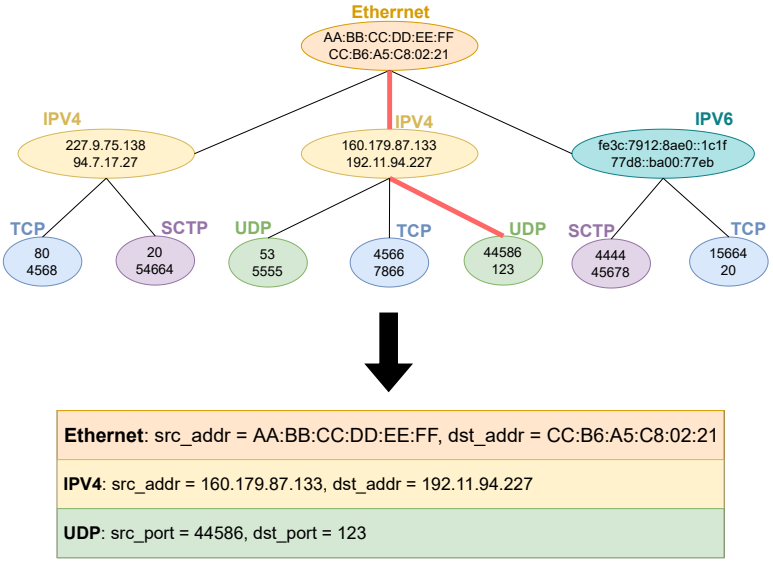

Figure 5.5: Tree structure used for packets representation.

Rule interpreter

The `Rule_interpreter` works in close collaboration with `Traffic_generator`. It utilizes an almost identical set of trees. The key difference is it actively uses only subtrees that represent matching packets. Because of that, it is possible to express the pattern of all rules. Every packet representation additionally has an appropriate list of actions in its leaf node. That allows the interpretation of all flow rules together with traffic sending. The `Rule_interpreter` utilizes matching patterns representation to determine whether the particular packet matches some rule. If a matching rule is found, it is possible to mimic the effects of its actions. As a result, trees representing expected packets are produced for each RX queue. Besides that, other products of flow rules are simulated as well. That includes the setting of flags and the state of counters. All information obtained from the simulation is later used in the final analysis.

Results verification

The essential part of the final verification is the captured traffic examination. This process is performed separately for each RX queue. It involves reading the particular PCAP file and processing captured packets with functions from the `Scapy` library. The extracted values are then used for traversing the expected packet representations obtained from `Rule_interpreter`. As a result, it is possible to determine whether all expected packets were captured

However, the verification of the captured traffic is not enough for rules with some actions. The additional information by `rte_flow_checker` and by `Rule_interpreter` are employed for extra checks. In case of MARK or FLAG actions, it involves checks that correct flags were set in particular packets. The final values of all configured counters are verified for the COUNT action. The AGE action is verified by checking the aged-out status of the associated rule. The expected value is determined by comparing the timeout interval used in the flow rule and the waiting time before and after sending testing traffic.

Chapter 6

Application of implemented tools

This chapter is dedicated to the evaluation of implemented tools. On the one hand, it includes the script for automated collection of `rte_flow` capabilities (Section 5.2). And on the other, a test suite that verifies the correct functionality of the rules (Section 5.3). At the same time, evaluation was used to analyze and compare the support of `rte_flow` across different NICs.

Both tools were applied to the multiple DPDK-supported NICs to test their behavior in different environments, thus proving their universality. Support of `rte_flow` needs to be guaranteed by the poll mode driver (PMD) of a particular NIC. For that reason, all chosen tested NICs are from major vendors, such as Intel or Mellanox (from 2019 part of NVIDIA[1]). Each of them ensures usage of the `rte_flow` according to the documentation. The list of used NICs and their brief specification is shown in Table 6.1. Since the support of capabilities mainly depends on the PMD, cards with as many different PMDs as possible were chosen. The first two cards are produced by Mellanox, whereas the remaining by Intel. Even though Mellanox cards both use the `MLX5` PMD, they belong to the diverse families of adapters and differ on the hardware level. In contrast, all Intel cards use different PMDs. Moreover, all of them also vary in available hardware resources.

Network card	Poll Mode Driver	Kernel driver version
ConnectX-5 MT27800 Family	MLX5	OFED 5.4
ConnectX-6 MT2892 Family	MLX5	OFED 5.4
Ethernet Controller E810-C	ICE	1.3.2
Ethernet Controller X710	I40E	2.8.20
Ethernet Connection X552	IXGBE	5.1.10

Table 6.1: List of network cards used for testing.

Details from the environment setup as well as the testing procedure are discussed in Section 6.1. Results for all tested NICs are presented in Section 6.2 which is split into four subsections. Each of them sums up results only for NICs with the same PMD. The final summary and comparison of results between all PMDs can be found in Section 6.3.

[1]NVIDIA's acquisition of Mellanox – https://nvidianews.nvidia.com/news/nvidia-completes-acquisition-of-mellanox-creating-major-force-driving-next-gen-data-centers

6.1 Environment setup and testing procedure

As already mentioned, testing was split into four sections, where each one focused only on NICs with particular PMD. Nevertheless, all testing sections followed an almost identical process. Since NICs are situated on different servers, the first step always was the environment preparation. And only after that, the implemented tools could be applied. Testpmd (Section 2.4) was used as a referential application to verify the correctness of produced results. That was possible because both tools can convert the processed flow rules into testpmd commands. However, the functionality cannot be confirmed only by testpmd. Testpmd can only check the assessment of support for a particular flow rule. For that reason, I also utilized the experience with the `rte_flow` support gained from the manual testing with testpmd. As a result, automatically extracted results were compared with this experience.

Environment setup for all tests

At the beginning of each testing section, the NIC and the environment had to be prepared. Some of the preparation steps were required by all NICs, whereas others were performed only in specific cases.

All common steps were extracted into the `prepare.sh` script. It manages the allocation of huge pages with the dpdk-hugepages[2] utility and compilation of the `rte_flow_checker` application (Section 5.1). Apart from that, all of the Python requirements are installed as well. Since the `rte_flow_checker` requires root privileges (e.g., due to huge pages manipulation), both tools also have to be run by the root user. For that reason, Python packages are installed globally.

The Intel NICs need to be bound to a special low-level kernel driver (Section 2.2). The `vfio-pci` driver was used for this purpose.

Testing procedure

Once necessary preparation for a particular section was completed, the implemented tools could be utilized. Simultaneously, the support of `rte_flow` on NIC was inspected. At first, the `generate_report` script (Section 5.2) was applied. To properly evaluate its functionality, the `verify_testpmd_commands` script was prepared. It accepts a file with testpmd commands that represent flow rules. The file is then passed to the testpmd application for processing. Meanwhile, the script calculates the number of successful and failed attempts.

The evaluation of the `generate_report` script consisted of several phases:

1. Use the `generate_report` script to collect `rte_flow` capabilities on a particular NIC. Capture all flow rules assessed as supported in one file and unsupported in another. Utilize the `verify_testpmd_commands` to verify that testpmd evaluates all flow rules the same way.

2. Test the maximum capacity of different types of rules with `generate_report`. After that, replicate the measurements with testpmd using the `verify_testpmd_commands` script.

[2] https://doc.dpdk.org/guides-20.11/tools/hugepages.html

3. Calculate the insertion times for several different bursts of rules. The burst sizes are chosen according to the information found during the previous phase. As a result, small bursts are used as well as bursts near the maximum capacity.

4. Optionally perform further examination according to discovered `rte_flow` support level (e.g., the capacity of rules in not default group).

All configuration files used in the second, third, and optionally fourth phases were produced by the `generate_rules` script (Section 5.2). For the time insertion tests, only rules with DROP action were utilized. On the other hand, different actions were used during the capacity testing. As a result, it was possible to determine if capacity is influenced by this factor. All flow rules had the default priority level and also the group number. Sets with the following actions were used:

- DROP,
- MARK + QUEUE,
- COUNT + QUEUE.

After that, a test suite (Section 5.3), which investigates the functionality of flow rules, was employed. The implemented test cases cover mainly correct matching of various pattern items and the functionality of basic actions (e.g., QUEUE, MARK, FLAG, COUNT, AGE, JUMP or SET_IPV4_SRC). The application of the test suite was the following:

1. Apply all tests from a particular group.

2. For the failed test cases, collect the used YAML configuration file as well as its analogy with testpmd commands. Use the latter file to configure the testpmd. Pass the former file to the script, which mimics the functionality of the traffic generator. As a result, it is possible to replicate the conditions of the failed test. The problematic behavior is then manually inspected.

6.2 Results for selected cards

This section discusses the results of the evaluation process. Besides that, it also provides a brief overview of the `rte_flow` support on each of the NICs. Since the testing process was split into four groups according to the used PMD on NICs, obtained results are also presented in separate subsections. The testing procedure described in Section 6.1 was applied to every group. At first, the `generate_report` script (Section 5.2) was evaluated. Its functionality was successfully proven by the testpmd application in all cases. The following subsections focus mainly on the presentation of supported `rte_flow` capabilities. The capabilities description can be found in Section 2.3. After that, the test suite for verification of the rules functionality (Section 5.3) was also utilized. The results slightly differ for each group of NICs.

`MLX5`

Two of the chosen tested NICs utilize the `MLX5` PMD. Testing on both of them produced the same results for capabilities collection and tests of maximum capacity. They only slightly differ in insertion times of rules.

At first, `rte_flow` capabilities were automatically collected by the `generate_report` script. The results showed the support for four priority levels. The actual number of supported groups was not fully revealed. The `generate_report` script only proved NICs support all groups used for capabilities collection (1001). For that reason, group attribute was later further separately investigated. For the direction attribute, the results were much clearer. Cards support usage of both directions – ingress and egress. But single flow rule can include only the specification of one of them, usage of both is not allowed. The `generate_report` evaluated the transfer attribute is unsupported. The manual tests proved that `MlX5` PMD allows usage of transfer flag only with virtual functions created using SR-IOV[3] technology, but not directly by a physical device.

The `generate_report` script discovered the support of the following actions:

- DROP – ingress rules,
- QUEUE – ingress rules,
- RSS – ingress rules,
- JUMP – ingress or egress rules,
- FLAG – ingress or egress rules,
- MARK – ingress or egress rules,
- COUNT – ingress or egress rules,
- AGE – ingress or egress rules,
- modify packet field (e.g., SET_IPV4_SRC) – ingress or egress rules.

Code B.1 includes part of flow rules, which `generate_report` script used during processing. Some of the collected capabilities will now be briefly described. In the ingress direction, flow rules need to include exactly one of the fate actions (Section 2.3). That means it is not possible to build a rule only with actions MARK, FLAG, COUNT, AGE, or modify packet field. However, this requirement is not requested in egress rules. The only supported fate action in egress is JUMP action.

Most of the items, which represent network protocol headers, are supported. The meta items, such as ANY or INVERT, cannot be used in rules. The specification of the L2 item can be omitted in the matching pattern. NIC then looks for it anywhere in packets. The pattern can consist of only items with no further specification of the matching properties. The other option is to specify matching properties with full or partial masks. Usage of matching ranges is available only for source and destination addresses of the IPV4 item. However, later application of the test suite showed that these rules are matched only by packets from the lower bound, not the full range. This behavior was later replicated also with testpmd. The rest of the tests confirmed the correct behavior of the other features.

The `generate_report` script was also utilized to inspect the maximum allowed amount of flow rules. For all rule types (Section 6.1), the discovered capacity is **65536** on these cards. After that, the `generate_report` script was used for the insertion times measurements. They were performed during a single invocation of the `rte_flow_checker` (Section 5.1) using the configuration file with several rule bursts. The number of rules in used

[3]Single Root IO Virtualization (SR-IOV) – https://docs.nvidia.com/networking/pages/viewpage.action?pageId=52011161

bursts was progressively increasing. As a result, two different values were captured for each burst – the average insertion time of a single rule and the total time. The results are presented in Figure 6.1. The left graph shows the average times. The horizontal axis displays the size of bursts expressed as the portion of the maximum rules capacity, which was discovered in the previous step. The vertical axis represents the average times in milliseconds. The right graph displays the total insertion times. The X-axis represents the sizes of bursts expressed with the number of rules, whereas the Y-axis displays the total insertion times in milliseconds. The first graph shows that the average time fluctuates around a constant value. But when the burst sizes reach 85% of the maximum capacity, the insertion time begins to grow exponentially for both NICs. This behavior is also reflected in the second graph. The original linear growth changes into the exponential near the maximum capacity.

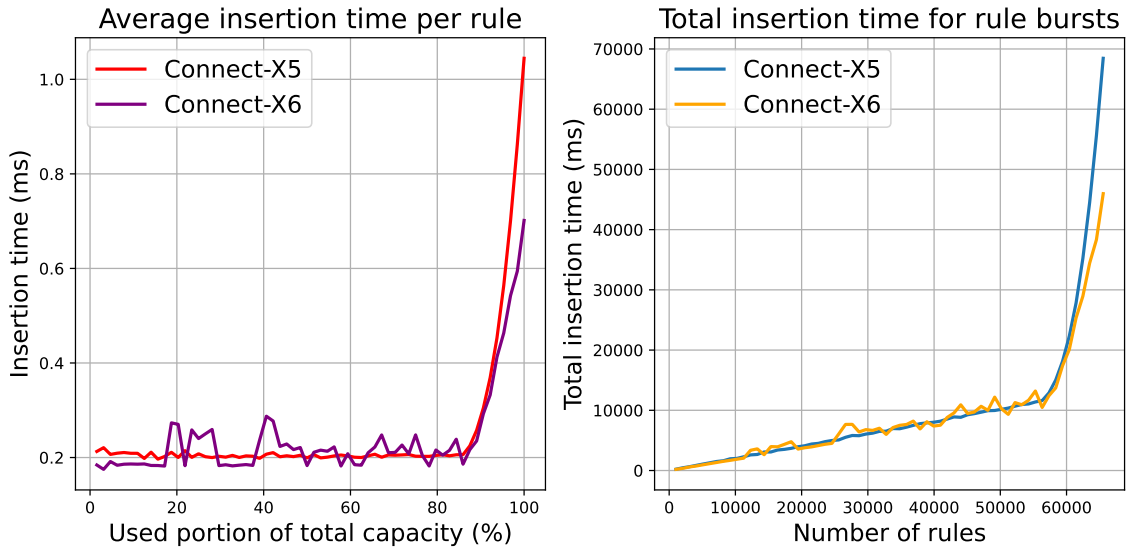

Figure 6.1: Average and total insertion times for rule bursts on NICs with MLX5 PMD.

Since prior evaluation proved the support of multiple groups, the capacity measurements were also performed for the non-default group. Additional tests showed that it is possible to create significantly more rules in these groups. The only limitation is the available resources, mainly allocated memory. The results from the measurements are in Table 6.2.

Allocated memory	*DROP*	*MARK + QUEUE*	*COUNT + QUEUE*
250 MB	2487612	2487612	1623356
500 MB	5047612	5047612	3360060
750 MB	7607612	7607612	5109052
1000 MB	10167612	10167612	6651904

Table 6.2: Dependency of rules capacity in non-default groups on the allocated space.

The number of rules without COUNT action constantly increases with more allocated memory. Even though the amount of rules with COUNT action also grows, it is always lower than for the other types. That signals that `MLX5` PMD implements most of the flow rule logic in software. That also includes counters. This also answers the question about

the number of groups. Any 32-bit number can be used as a group number. There is the only limitation on a maximum number of rules.

ICE

The application of the `generate_report` script produces the following result for NIC with `ICE` PMD. The card support usage of only one group and one priority level. All rules have to be created in the ingress direction. The transfer attribute is not supported. The additional manual testing showed that the card allows the creation of rules with different priority values. However, this behavior can be used only in pipeline mode, which needs to be explicitly requested by the extra argument for `-a` EAL parameter[4]. However, the interpretation of the priority is very specific. Rules with priority 0 are applied first and typically filter out the unwanted packets. The rest is later used for classification and further processing. That resembles the rule groups rather than the priority levels.

Support of following actions was proven by the `generate_report` script:

- DROP,
- QUEUE,
- RSS,
- PASSTHRU,
- MARK,
- COUNT.

Some of the rules, which were processed by the `generate_report` script, are demonstrated in Code C.1. The significant features will now be briefly summarized. Each action can be used alone in the flow rule. However, there are a few limitations to their mutual combination. It is allowed to combine one fate action (DROP, PASSTHRU, QUEUE) with MARK or COUNT actions. Meta pattern items (e.g., INVERT) are not supported. Patterns, which contain only items with no inner specification, can be combined exclusively with RSS action. Otherwise, it is necessary to always specify matching properties for at least one of them. That can be done with full or partial masks. Pattern also needs to always start with the L2 item. Because of that, patterns cannot omit items from lower network layers. Correct behavior of the rules was later confirmed by the test suite.

After that, the maximum capacity of rules was inspected. Results show that the capacity differs for different types of rules:

- DROP – **15360**,
- QUEUE + MARK – **15360**,
- QUEUE + COUNT – **256**.

The limitation for COUNT action is caused by the available hardware counters. The insertion times were measured afterward using the same approach as the previous subsection. For this card, the measurements discovered an anomaly. The very first insertion takes

[4] https://doc.dpdk.org/guides/nics/ice.html#runtime-config-options

significantly longer in comparison to the rest. The additional tests confirmed that this effect always happens only for the initial insertion since the launching of the DPDK application. When one rule is successfully loaded into the card, the insertion times considerably drop. It signals that the card performs an initialization routine before the initial flow rule insertion. For that reason, the insertion time of the initial rule was excluded from the measurements. Graphs in Figure 6.2 shows the results . From the left graph, it is apparent that the insertion time is almost constant. It fluctuates in a very narrow range of values near 0,01 ms. This behavior is also translated in the second graph. The total insertion times linearly increase.

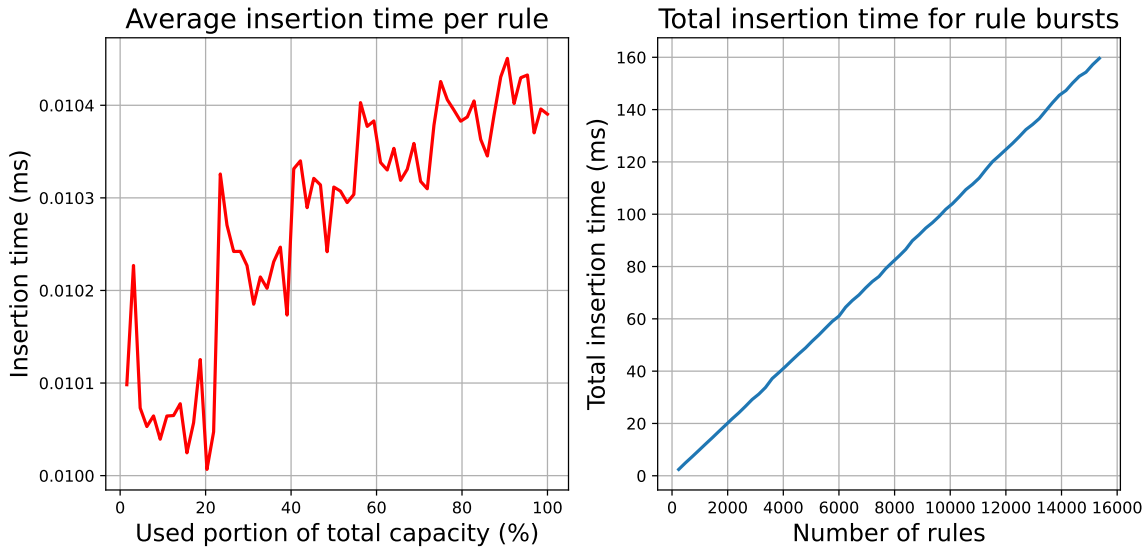

Figure 6.2: Average and total insertion times for rule bursts on NICs with ICE PMD.

`I40E`

The `generate_report` script was then applied to the NIC with `I40E` PMD. For attributes, the results were very straightforward. The script discovered support of only one group and one priority level. Furthermore, it proved that it is possible to create only ingress rules, and the transfer attribute is not supported. These facts are also in agreement with the experience from manual testing.

Support of the following actions was confirmed:

- DROP,
- QUEUE,
- RSS,
- PASSTHRU,
- FLAG,
- MARK.

Part of the rules used for capabilities collection by the `generate_report` script can be seen in Code D.1. This paragraph sums up the most relevant features. All of the supported actions, except for FLAG, can be used alone in flow rules. They can also be used in combinations. It is required to combine one fate action with exactly one non-fate (MARK or FLAG). However, the fate action needs to be on the lower index in the actions list. The reverse order is not supported. Usage of meta pattern items is not allowed. It is also possible to use a pattern filled only with items with no inner specification. The matching properties can be defined with full or partial masks. The omission of items from the bottom of the network stack in patterns is not supported. The ETH item cannot be used alone.

However, the application of test suite raised some concerns – part of the tests failed. Further inspection showed that `I40E` PMD interprets the matching pattern differently from the previously tested PMDs. For example, a pattern with just L2 and L3 items matches only fragmented packets without an L4 header. But in other PMDs, this pattern can be matched without inspecting the presence or type of L4 header. For that reason, the option to define a custom test by providing the configuration file through command line arguments was added to the test suite. The tests were later repeated with files that respect the interpretation of patterns by the `I40E` PMD. They prove the correct functionality of the rules also for this NIC.

Due to found capabilities, the maximum capacity was tested only for rules without the COUNT action. The discovered maximum amount was **7680** in all cases. After that, the insertion times measurements were performed. They again detected the anomaly described in the previous subsection. The insertion of the very first rule takes much longer than for the others. Because of that, the time of the first insertion was not included in the measurements. Outcomes are presented in Figure 6.3. The average insertion time soon starts to grow extensively. That changes when the value reaches 6 ms. At this moment, it sustains the increasing trend, but at a much slower pace. In the other graph, the value is significantly low at the start. But when the burst sizes reach approximately 500 rules, it begins to increase linearly.

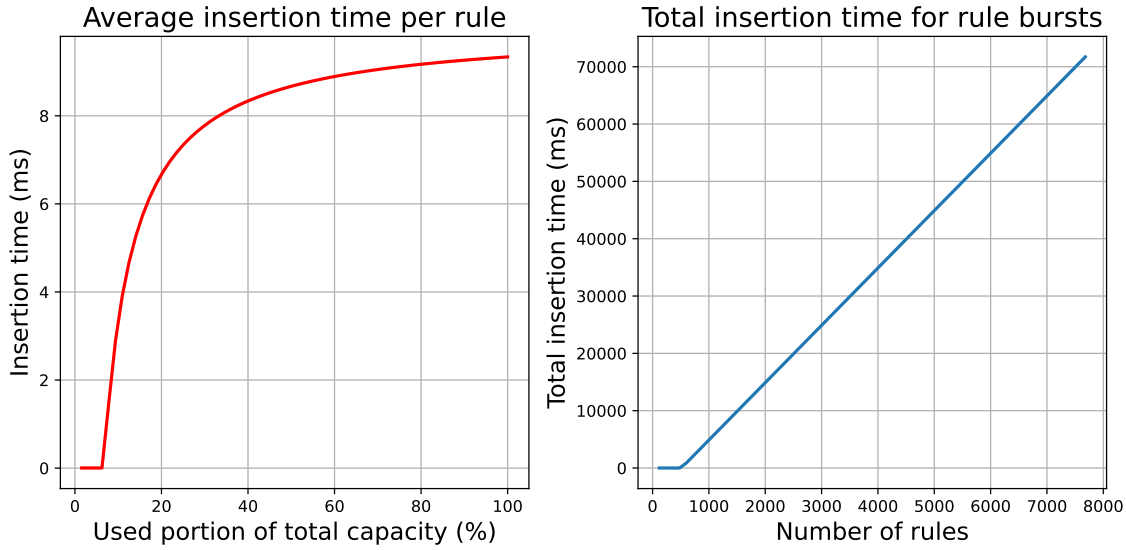

Figure 6.3: Average and total insertion times for rule bursts on NICs with I40E PMD.

IXGBE

Testing on the card with IXGBE PMD was the shortest one. The level of rte_flow support on this card turned out to be very limited. The only supported action is RSS. Rules also need to have the default group value, default priority level, and ingress direction. For these reasons, testing of the maximum capacity and measurement of insertion times were not performed. The test suite was also not utilized due to insufficient support of pattern items and actions.

6.3 Results summary and evaluation

This section summarizes the results produced by the testing process. As a result, it evaluates the functionality of both implemented tools – the generate_report script for automated collection of rte_flow capabilities (Section 5.2) and a test suite for verification of rules functionality (Section 5.3). Apart from that, the discovered supports of the rte_flow interface across all tested NICs are compared.

Testing was split into four sections. Each of them focused only on NICs with particular PMD. The detailed course of the testing procedure is described in Section 6.1. At first, the collection of capabilities by the generated_report script was tested. The evaluation utilizes the script's ability to convert most of the processed rules into equivalent testpmd commands. As a result, support assessment for all rules could be verified by testpmd. Correct functionality was successfully confirmed in all sections. The second step was the inspection of the maximum flow rules capacity. Testpmd was used to replicate the obtained results. For all cases, the results were successfully approved. In another step, the generate_report script was utilized to measure the insertion times of rules on tested NICs. The results were visualized with appropriate graphs.

After all that, a test suite was employed to verify the functionality of the flow rules. These tests helped to reveal several problematic behaviors (details in Section 6.2). All of the failed tests were later reproduced with testpmd, thus proving the results. The first was found on cards with MLX5 cards and is connected with matching ranges for IPV4 addresses. These rules are matched only by the packets from the lower bound of the range. Another inconsistency was found in NIC with I40E PMD. Unlike other PMDs, I40E interprets the pattern explicitly. That means that the pattern with L2 and L3 items matches only fragmented IP packets. The last three steps were skipped for NIC with IXGBE PMD due to insufficient support of rte_flow capabilities.

The application of implemented tools showed that cards with MLX5 PMD offer the most extensive support among all tested NICs. They enable the usage of multiple groups and priorities, whereas the other NICs support only the default group and priority level. Moreover, MLX5 cards support the creation of ingress but also egress rules. These cards also provide the largest capacity for the flow rules. Unlike other cards, they support numerous actions and allow the omission of pattern items from the bottom layers of the network stack. On the other hand, the lowest level of rte_flow support was discovered on the card with IXGBE PMD. This card allows only usage of RSS action.

The measurements of the insertion time provided the following results. The average insertion time on the ICE card remains constant for different burst sizes. On the other hand, the average time increases as the burst are closer to the maximum capacity for the MLX5 and I40E cards. The values are by far the highest for the I40E card.

Chapter 7

Conclusion and future work

The goal of his bachelor's thesis was to provide a method that enables the comfortable evaluation of `rte_flow` interface support on network cards. Previously, this data could be retrieved only by monotonous and time-consuming manual testing. For that reason, I designed and implemented a pair of tools that automate the evaluation process. As a result, all the supported `rte_flow` capabilities on a particular card can be inspected and collected systematically.

At the beginning of the solution, I had to get familiar with the DPDK framework. This process included a detailed study of all relevant tools (mainly the testpmd application) and libraries. All the gained knowledge was continuously tested in practice using the testpmd application. The increased focus was given to the main objective of the thesis – classifier interface `rte_flow`. As a result, I learned how to configure hardware offloads in the network card with the special flow rules.

After that, I spent a significant amount of time manually analyzing the `rte_flow` capabilities on different network cards with the help of the testpmd application. That helped me to identify the possibilities for the automation and optimization of the manual process. I decided to utilize Python as the major language for the realization. For that reason, I needed to find and get familiar with suitable libraries/tools which would help me with the latter implementation. Among others, the testing framework `pytest` was studied during this phase. Apart from that, I also learned how to manipulate network traffic with the `Scapy` library.

During the creation of design proposals, I realized that two approaches could be utilized for the `rte_flow` support evaluation. Both require the DPDK application, which uses the given flow rules to configure the NIC through the `rte_flow` interface. I decided to design my own DPDK application precisely for this use case. The first testing approach repetitively loads various flow rules into NIC and extracts the supported capabilities from the successful attempts. That was utilized in the proposal of an automated pipeline for capabilities collection. The second one focuses on the verification of the flow rules effect. To achieve that, I proposed the testing architecture based on the `pytest` framework, which verifies the correct functionality of the rules.

At first, it was necessary to implement the proposed DPDK application. It supports three different run modes, which allow its usage by both mentioned approaches. The implementation of an automated pipeline resulted in the production of the first tool. It systematically generates appropriate rules, assesses their support on particular NIC, and creates a report with supported features. It can also inspect the maximum capacity of rules or their insertion times. The realization of the mentioned testing architecture led to the

second tool – a test suite for rules functionality verification. It tests the effects of rules by sending the testing traffic to the configured NIC and evaluating the results.

Both implemented tools were then applied to various network cards. Produced results were validated using several testing scripts and the testpmd as the referential application. Moreover, they were also compared with the experience I gained during manual testing of `rte_flow` support. As a result, the functionality of both tools was successfully confirmed. The application of both tools also showed that cards with the `MLX5` driver provide the most extensive support for the `rte_flow`. They allow the usage of numerous capabilities and provide the largest capacity for the rules.

The main goal of this thesis, to provide tools for testing support of the `rte_flow` interface on network cards, was fulfilled. But there is still room for several future improvements. The test suite can currently verify the functionality of only ingress rules. That proved to be sufficient since experiments have shown that only a few cards support egress rules. Despite that, the addition of egress rules could be a potential improvement. Another improvement might be an upgrade to the new stable version of the DPDK framework, which was recently released. This version introduced several new features, such as INDIRECT action, that deserve to be included in the inspection process. It might also be interesting to measure the insertion time when the traffic flows through the NIC. All these and possibly also other further improvements will be the objective of my future collaboration with CESNET.

Bibliography

[1] Biondi, P. *Scapy* [online]. 2022 [cit. 2022-4-3]. Available at: https://scapy.readthedocs.io/en/latest/index.html.

[2] Chimata, A. K. *Path of a packet in the Linux kernel stack* [online]. 2005 [cit. 2021-11-18]. Available at: https://www.cs.dartmouth.edu/~sergey/netreads/path-of-packet/Network_stack.pdf.

[3] Efraim, R. *Mellanox Bifurcated DPDK PMD* [online]. 2017 [cit. 2021-11-18]. Available at: https://www.dpdk.org/wp-content/uploads/sites/35/2018/06/Mellanox-bifurcated-driver-model.pdf.

[4] Gorman, M. *Understanding the Linux: Virtual Memory Manager*. 1st ed. Pearson Education, Inc., 2004. ISBN 0131453483. Available at: https://pdos.csail.mit.edu/~sbw/links/gorman_book.pdf.

[5] Intel Corporation. *Testpmd Application User Guide* [online]. 2014 [cit. 2021-11-18]. Available at: https://doc.dpdk.org/guides-20.11/testpmd_app_ug/.

[6] Intel Corporation. *Network Interface Controller Drivers* [online]. 2015 [cit. 2021-11-18]. Available at: https://www.intel.com/content/dam/www/public/us/en/documents/guides/dpdk-network-interface-controller-drivers-guide.pdf.

[7] Intel Corporation. *Programmer's Guide* [online]. 2017 [cit. 2021-12-25]. Available at: https://doc.dpdk.org/guides-20.11/prog_guide/.

[8] Jayathirtha, B. *WDatabase Administrator's Reference* [online]. 2021 [cit. 2021-11-18]. Available at: https://docs.oracle.com/database/121/UNXAR/toc.htm.

[9] Krekel, H. *Pytest* [online]. 2015 [cit. 2022-4-3]. Available at: https://docs.pytest.org/en/7.1.x/index.html.

[10] Marek Suchánek, Milan Navratil, Don Domingo and Laura Bailey. *Performance Tuning Guide* [online]. 2017 [cit. 2021-11-18]. Available at: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/index.

[11] Meszaros, G. *XUnit Test Patterns: Refactoring Test Code*. 1st ed. Pearson Education, Inc., 2007. ISBN 9780131495050. Available at: https://web.archive.org/web/20160923114822/http://buhoz.net/public/books/computacion/desarrollo/metodologias/agiles/xUnit.Test.Patterns.Refactoring.Test.Code.2007.pdf.

[12] More authors. *HugeTLB Pages* [online]. 2021 [cit. 2021-11-18]. Available at: https://www.kernel.org/doc/html/latest/admin-guide/mm/hugetlbpage.html.

[13] Projects Linux Foundation. *Open vSwitch* [online]. 2016 [cit. 2022-01-18]. Available at: https://www.openvswitch.org/.

[14] Projects Linux Foundation. *DPDK* [online]. 2021 [cit. 2021-11-18]. Available at: https://www.dpdk.org/.

[15] Projects Linux Foundation. *DPDK API* [online]. 2021 [cit. 2021-12-25]. Available at: https://doc.dpdk.org/api-20.11/index.html.

[16] Projects Linux Foundation. *The Fast Data Project* [online]. 2022 [cit. 2022-01-18]. Available at: https://fd.io/.

[17] Ramos, P. *Memory Profiling in Python* [online]. 2021 [cit. 2022-4-3]. Available at: https://medium.com/a-bit-off/memory-profiling-python-b67b73c75951.

[18] TRex Team. *TRex Realistic Traffic Generator* [online]. 2022 [cit. 2022-01-18]. Available at: https://trex-tgn.cisco.com/.

[19] Viviano, A. *Introduction to Receive Side Scaling* [online]. 2021 [cit. 2021-12-25]. Available at: https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling.

Appendix A

State automaton

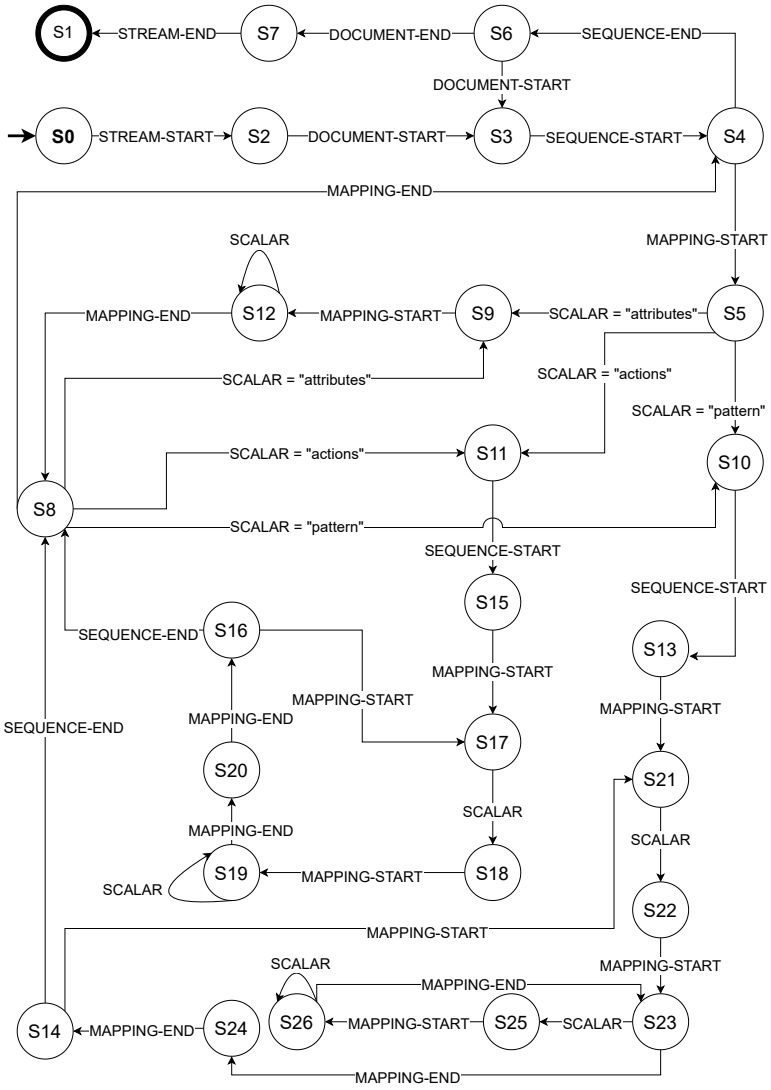

Figure A.1: Scheme of state automaton for configuration file parsing. The events from `libyaml`'s event-based API serve as an inputs.

Appendix B

Flow rules log for MLX5 cards

```
### supported
flow create 0 priority 3 ingress \
  pattern eth / ipv4 dst is 159.58.1.0 src is 195.168.1.0 / end \
  actions drop / end
flow create 0 group 5 ingress \
  pattern eth / ipv4 dst is 159.58.1.0 src is 195.168.1.0 / end \
  actions jump group 6 / end
flow create 0 ingress \
  pattern eth / ipv4 dst is 159.58.1.0 src is 195.168.1.0 / end \
    actions age timeout 100 / drop / end
flow create 0 egress \
    pattern eth / ipv4 dst is 159.58.1.0 src is 195.168.1.0 / end \
    actions jump group 3 / end
flow create 0 ingress \
    pattern eth / ipv4 / end \
    actions drop / end
flow create 0 ingress \
  pattern eth src spec aa:bb:cc:dd:ee:ff src mask ff:ff:ff:00:00:00 / end \
  actions drop / end
flow create 0 ingress \
  pattern eth / ipv4 src is 195.168.0.1 src last 195.168.5.0 / end \
  actions drop / end
flow create 0 ingress \
    pattern ipv4 src is 195.168.0.1 / end \
    actions drop / end

### unsupported
flow create 0 egress \
    pattern eth / ipv4 dst is 159.58.1.0 src is 195.168.1.0 / end \
    actions queue index 1 / end
flow create 0 ingress \
    pattern eth / ipv4 dst is 159.58.1.0 src is 195.168.1.0 / end \
    actions mark id 42 / end
```

Code B.1: Snippet of the processed rules for MLX5 NICs.

Appendix C

Flow rules log for ICE cards

```
### supported
flow create 0 ingress \
    pattern eth / ipv4 dst is 159.58.1.0 src is 195.168.1.0 / end \
    actions mark id 42 / end
flow create 0 ingress \
    pattern eth / ipv4 dst is 159.58.1.0 src is 195.168.1.0 / end \
    actions queue index 1 / end
flow create 0 ingress \
    pattern eth / ipv4 dst is 159.58.1.0 src is 195.168.1.0 / end \
    actions count identifier 128 / drop / end
flow create 0 ingress \
    pattern eth / ipv4 dst is 159.58.1.0 src is 195.168.1.0 / end \
    actions drop / count identifier 128 / end
flow create 0 ingress \
    pattern eth / ipv4 src spec 195.168.0.1 src mask 255.255.255.0 / end \
    actions drop / end

### unsupported
flow create 0 priority 1 ingress \
  pattern eth / ipv4 dst is 159.58.1.0 src is 195.168.1.0 / end \
  actions drop / end
flow create 0 group 1 ingress \
  pattern eth / ipv4 dst is 159.58.1.0 src is 195.168.1.0 / end \
  actions jump group 2 / end
flow create 0 egress \
    pattern eth / ipv4 dst is 159.58.1.0 src is 195.168.1.0 / end \
    actions count identifier 128 / end
flow create 0 ingress \
    pattern eth / ipv4 / end \
    actions drop / end
flow create 0 ingress \
    pattern ipv4 src is 195.168.0.1 / end \
    actions drop / end
```

Code C.1: Snippet of the processed rules for ICE NICs.

Appendix D

Flow rules log for I40E cards

```
### supported
flow create 0 ingress \
    pattern eth / ipv4 / end \
    actions drop / end
flow create 0 ingress \
    pattern eth / ipv4 dst is 159.58.1.0 src is 195.168.1.0 / end \
    actions mark id 42 / end
flow create 0 ingress \
    pattern eth / ipv4 dst is 159.58.1.0 src is 195.168.1.0 / end \
    actions queue index 1 / flag / end
flow create 0 ingress \
    pattern eth / ipv6 src spec ffee::1 src mask ffff:: / end \
    actions drop / end

### unsupported
flow create 0 priority 1 ingress \
  pattern eth / ipv4 dst is 159.58.1.0 src is 195.168.1.0 / end \
  actions drop / end
flow create 0 group 1 ingress \
  pattern eth / ipv4 dst is 159.58.1.0 src is 195.168.1.0 / end \
  actions jump group 2 / end
flow create 0 ingress \
    pattern eth / end \
    actions drop / end
flow create 0 ingress \
    pattern ipv4 src is 195.168.0.1 / end \
    actions drop / end
flow create 0 ingress \
    pattern eth / ipv4 dst is 159.58.1.0 src is 195.168.1.0 / end \
    actions flag / end
flow create 0 ingress \
    pattern eth / ipv4 dst is 159.58.1.0 src is 195.168.1.0 / end \
    actions flag / queue index 1 / end
```

Code D.1: Snippet of the processed rules for I40E NICs.