BRNO UNIVERSITY OF TECHNOLOGY

Faculty of Electrical Engineering
and Communication

BACHELOR'S THESIS

Brno, 2020 Marek Benc

BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF ELECTRICAL AND ELECTRONIC TECHNOLOGY

ÚSTAV ELEKTROTECHNOLOGIE

HARDWARE ACCELERATION OF PACKET CLASSIFICATION USING TC FLOWER

HARDWAROVÁ AKCELERACE KLASIFIKACE PAKETŮ S VYUŽITÍM ROZHRANÍ TC FLOWER

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR — **Marek Benc**
AUTOR PRÁCE

SUPERVISOR — **Ing. Jiří Libich, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2020

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

Bachelor's Thesis

Bachelor's study program **Microelectronics and Technology**

Department of Electrical and Electronic Technology

Student: Marek Benc ***ID:*** 203192

Year of study: 3 ***Academic year:*** 2019/20

TITLE OF THESIS:

Hardware acceleration of packet classification using TC Flower

INSTRUCTION:

Get familiar with the TC Flower interface and its capability of offloading packet classification rules into hardware. Get familiar with the NDK platform developed by the Liberouter project and their P4 language compiler. Design an extension of the NDK platform network card driver, to make it capable of offloading rules from the TC Flower interface into a COMBO acceleration card. Implement the driver design and verify its functionality on an appropriate example architecture described in the P4 language. Evaluate the achieved results and discuss the options for further continuation of the project.

RECOMMENDED LITERATURE:

Podle pokynů vedoucího práce.

Date of project specification: 3.2.2020 ***Deadline for submission:*** 8.6.2020

Supervisor: Ing. Jiří Libich, Ph.D.
Consultant: Ing. Tomáš Martínek, Ph.D.

doc. Ing. Jiří Háze, Ph.D.
Chair of study program board

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní program **Mikroelektronika a technologie**

Ústav elektrotechnologie

Student: Marek Benc ***ID:*** 203192
Ročník: 3 ***Akademický rok:*** 2019/20

NÁZEV TÉMATU:

Hardwarová akcelerace klasifikace paketů s využitím rozhraní TC Flower

POKYNY PRO VYPRACOVÁNÍ:

Seznamte se s rozhraním TC Flower a jeho možnostmi pro převod pravidel klasifikace paketů směrem do hardware. Seznamte se s platformou NDK vyvíjenou v rámci projektu Liberouter a překladačem jazyka P4. Navrhněte rozšíření ovladače síťové karty NDK platformy tak, aby byl schopen převádět pravidla z rozhraní TC Flower směrem do akcelerační karty COMBO. Proveďte implementaci navrženého ovladače a ověřte jeho funkci na vhodném příkladě architektury popsané v jazyce P4. Zhodnoťte dosažené výsledky a diskutujte možnosti dalšího pokračování projektu.

DOPORUČENÁ LITERATURA:

Podle pokynů vedoucího práce.

Termín zadání: 3.2.2020 ***Termín odevzdání:*** 8.6.2020

Vedoucí práce: Ing. Jiří Libich, Ph.D.
Konzultant: Ing. Tomáš Martínek, Ph.D.

doc. Ing. Jiří Háze, Ph.D.
předseda rady studijního programu

ABSTRACT

The CESNET association develops the COMBO line of high-performance (currently with up to two 100Gbps ports) programmable network adapters, focused on network data analysis and processing. These cards come with an FPGA chip, which allows users to define exactly how the network traffic should be processed. A possible use case for these cards is as network switches for virtual machines within a data center.

The focus of this thesis is on implementing TC Flower offloading support for the COMBO line of cards (software and FPGA firmware). It is a common interface for installing flow match+action rules into SmartNICs, and allows them to be used to manage network traffic between virtual machines and the outside world, saving CPU cycles in the host machine.

KEYWORDS

computer network, packet classification, flow rule, offloading, virtual network switch, device driver, virtualization, hardware acceleration

ABSTRAKT

Sdružení CESNET vyvíjí vysokorychlostní programovatelné síťové karty COMBO (aktuálně až s dvěma 100Gbps porty) zaměřené na analýzu a zpracování síťových dat. Karty obsahují FPGA čip, který dovoluje uživatelům přesně definovat způsob, jakým má být síťový provoz zpracován. Jedno z možných využití těchto karet je jako síťový přepínač pro virtuální stroje v data centru.

Tato práce je zaměřená na implementaci podpory TC Flower offloadu pro karty COMBO (software a FPGA firmware). Jedná se o všeobecné rozhraní pro instalaci flow pravidel typu shoda+akce do SmartNICů, a dovoluje nám použít je pro správu síťového provozu mezi virtuálními stroji a vnějším světem. Cílem je úspora procesorových cyklů hostitelského stroje.

KLÍČOVÁ SLOVA

počítačová síť, klasifikace paketů, flow pravidlo, převod do hardwaru, virtuální síťový přepínač, ovladač zařízení, virtualizace, hardwarová akcelerace

ROZŠÍŘENÝ ABSTRAKT

Sdružení CESNET vyvíjí vysokorychlostní programovatelné síťové karty COMBO (aktuálně až s dvěma 100Gbps porty) zaměřené na analýzu a zpracování síťových dat. Karty obsahují FPGA čip, který dovoluje uživatelům přesně definovat způsob, jakým má být síťový provoz zpracován. Nedávno byla pro COMBO karty implementována podpora technologie SR-IOV, která umožňuje rozdělení prostředků karty na několik tzv. virtuálních funkcí, které mohou být v rámci virtualizace předány virtuální strojům.

Typické využití technologie SR-IOV v kontextu síťových karet je poskytnutí přímého síťového přístupu virtuálním strojům, síťový provoz v tomto případě neprochází hostitelským operačním systémem. Tento přístup je výhodný hlavně u systémů s mnoha virtuálními stroji mezi kterými nepřetržitě protéká veliké množství dat, jako například u serverů v data centrech, kde jsou virtuální stroje pronajímány různým zákazníkům pro účely internetového hostingu. Čas, který u softwarového řešení musí procesor serveru na realizaci síťového provozu strávit, bývá v tomto případě výrazný, protože v rámci třídění paketů mezi jednotlivými virtuálními stroji a vnějším světem musí hostitelský operační systém analyzovat datové políčka hlaviček každého jednoho paketu.

Tato práce je zaměřena na vytvoření hybridního řešení pomocí karty COMBO, kde virtuální stroje jsou připojeny na SR-IOV virtuální funkce karty, ale karta sama o sobě pakety z virtuálních strojů předává přímo hostitelskému operačnímu systému, který je třídí stejně jako u softwarového řešení. Rozdílem od čistě softwarového řešení je, že hostitelský operační může do karty nainstalovat klasifikační pravidla, které pro pakety známého druhu provádí směrování přímo v samotné kartě. Využívá se tady konceptu síťových toků (angl. *network flows*), kde tok představuje skupinu paketů pocházející od stejné aplikace nebo služby, se stejnou destinací. Principem funkce hybridního řešení je, že do hostitelského operačního systému se vždy dostane jenom první paket síťového toku, který operační systém využije pro sestrojení klasifikačního pravidla pro daný tok, a zbylé pakety toku zpracuje již samotná síťová karta.

V hostitelském operačním systému je realizace síťového provozu pro virtuální stroje zajištěna virtuálním síťovým přepínačem (angl. *virtual switch*). Implementace přepínače je různá v závislosti od požadavků, např. jádro Linux obsahuje implementaci přepínače pracujícího na OSI úrovni 2, který je spravován pomocí nástroje *brctl*. Pro zapojení do složitějších vnějších sítí jsou výhodné programovatelné síťové přepínače, mezi které patří např. *Open vSwitch*, který podporuje konfigurační protokol *OpenFlow*.

Pro *Open vSwitch* bylo v jádru Linux vyvinuto rozhraní TC Flower, které dovoluje virtuálnímu přepínači převádět pravidla na zpracování síťových toků do hardwaru. Toto rozhraní existuje jako součást klasifikačně-akčního podsystému TC CA, který poskytuje celou řadu mechanizmů pro řízení síťového provozu. TC Flower, formálně TC *filtr* typu *flower*, byl navržen jako alternativa k existujícímu *Open vSwitch* jádrovému ovladači, jelikož dochází u zmíněného ovladače k značné duplicitě kódu se zbytkem jádra. Filtr *flower* využívá existujícího kódu jádra, jako například *flow dissector*, čímž zamezuje dvojímu výskytu stejných poruch, a dovoluje využití existujících nástrojů.

Technologie využita pro návrh samotného FPGA firmwaru je jazyk P4, konkrétně překladač z P4 na VHDL, který je na CESNETu vyvíjen v rámci platformy NDK (*NetCOPE Development Kit)*. Jazyk P4 je doménově specifický jazyk určen pro popis funkce síťového prvku, s možností konfigurace za běhu pomocí *P4Runtime* rozhraní. Konkrétně se v jazyku P4 popisuje rozkladač (angl. *parser*), který rozloží procházející pakety na hlavičky a náklad (angl. *headers* and *payload*); tabulky, které podle hlaviček a metadat mohou nad paketem vykonat libovolně složité úkony; a ovládací programy, které jsou spouštěné rozkladačem na základě obsahu hlaviček paketů, a mohou na pakety podmíněně aplikovat tabulky. Samotný obsah tabulek, ve formátu pravidel typu shoda+akce, je nahráván za běhu přes rozhraní *P4Runtime*.

V rámci NDK P4 implementace existuje knihovna *libp4dev*, která slouží na nízkoúrovňovou konfiguraci firmwaru karty vytvořeného pomocí NDK P4 kompilátoru. Tato knihovna je určena hlavně pro použití v *P4Runtime* implementaci, ale dá se použít i pro specifické aplikace, pokud nepřekáží nepřenositelnost na jiné P4 platformy. Knihovna využívá pro konfiguraci rozhraní MI32, které je zpřístupněno knihovnou *libnfb* pomocí mapování virtuální paměti mezi ovladačem karty v jádře a knihovnou *libnfb*.

Pro přenos síťových dat byla v rámci platformy NDK vyvinuta technologie NDP (*NetCOPE Data Plane*), která specializovaným aplikacím umožňuje zpracovávat pakety v dávkovém režimu, bez toho aby operační systém musel jednotlivé pakety analyzovat. Aplikacím jsou pomocí knihovny *libnfb* přímo zpřístupněny paměťové regiony DMA, do kterých karta nezávisle na procesoru ukládá příchozí pakety, a z kterých čte pakety na odeslání.

V rámci ovladače pro karty podporující NDK platformu, tzv. NFB karty (COMBO karty jsou příkladem NFB karet), existuje několik pod-ovladačů vhodných pro rozličné aplikace. Jeden z pod-ovladačů ovladače NFB je *ndp_netdev*, který vytváří nad technologií NDP standardní síťová rozhraní. Počet těchto rozhraní odpovídá počtu tzv. NDP kanálů karty, které představují hardwarové jednotky řešící DMA komunikaci. Ovladač *ndp_netdev* je všeobecný, a způsob jeho použití závisí na konkrétní aplikaci.

Ovladač *ndp_netdev* byl v této práci využit jako testovací platforma pro vývoj podpory převodu TC Flower pravidel do COMBO karty. Síťová rozhraní pro NDP kanály zastupovali funkci reprezentátorů virtuálních funkcí, a bylo využito vlastnosti NDK P4 implementace, podle které výchozí destinace paketů je místo původu, což nám dovolilo s jednoduchostí modelovat situaci, kde virtuální funkce vždy provádějí loopback.

Pro instalaci pravidel do P4 tabulek bylo zapotřebí přizpůsobit knihovnu *libp4dev* pro práci v prostředí jádra operačního systému Linux. Tento krok si vyžadoval přidání podmíněného vkládáni hlaviček standardní C knihovny, náhradu funkcionality standardní C knihovny ekvivalentními funkcemi a datovými typy dostupnými v prostředí jádra Linux, a vytvořením skriptu pro export souborů knihovny do archivu a pro rozbalení archivu s *libp4dev* zdrojovými kódy pro využití v jádru pro NFB ovladač. Taky bylo zapotřebí napsat lepidlový kód, který knihovně *libp4dev* dovolil uvnitř NFB ovladače komunikovat po MI32 sběrnici s COMBO kartou.

Protože se jednalo u knihovny *libp4dev* o port do nového prostředí, bylo zapotřebí ověřit funkci knihovny v tomto novém prostředí. Byl navržen jednoduchý pod-ovladač ovladače NFB zvaný *p4test*, který na principu stavového automatu zpracovává příkazy od uživatele a volá funkce knihovny *libp4dev*. Tento ovladač byl následně upraven pro

funkci mimo jádra, jako běžný program, a porovnáním funkce této verze *p4test* s verzí fungující v jádře byla ověřena funkčnost knihovny *libp4dev* v prostředí jádra.

U návrhu P4 programu byl kvůli jednoduchosti, a kvůli pozdější možnosti optimalizace samotného P4 kompilátoru, zvolen postup použití jediné tabulky s dlouhým seznamem čtených políček, kde součástí klíče je maska specifikující sledované políčka, což umožňuje jednoduchý převod klíčů pravidel z formátu TC Flower to formátu P4. Pro akce byla zvolena jedna univerzální akce, která přímá dlouhý seznam parametrů, který určuje její chování. Tento přístup plyne z rozdílů akcí u TC Flower a v P4, kde TC Flower pravidlo může mít neomezeně dlouhý seznam libovolných akcí, zatímco pravidla v jazyku P4 mají vždy jenom jednu akci, která v našem případě představuje libovolnou kombinaci TC akcí. V budoucnu se uvažuje nad vytvořením skritptu, který by na základě napsaného firmwaru vytvářel firmwary podporující více tabulek, které by pro lepší využití omezených zdrojů FPGA podporovali jenom zákazníkem definovaný formát pravidel.

Po úspěšném zprovoznění převodu TC Flower pravidel do P4 tabulek uvnitř karty byl navržen nový NFB pod-ovladač zvaný *sriov_netdev*, který pro karty s kompatibilním firmwarem vytváří reprezentátory virtuálních funkcí, které se dají pomocí rozhraní TC Flower využít s *Open vSwitch* pro realizaci hybridního řešení síťového přístupu. Hlavní rozdíl P4 firmwaru použitého s tímto pod-ovladačem je přítomnost směrovací tabulky, která určuje výchozí destinaci paketů. Tato tabulka je ovladačem s*riov_netdev* naplněna tak, aby se pakety odeslané virtuálními funkcemi objevili na reprezentátorech daných virtuálních funkcí, a naopak aby pakety odeslané přes reprezentátory se objevili ve virtuálních funkcích. Tímto způsobem je realizována tzv. pomalá cesta paketů, která slouží hlavně pro poskytnutí informací *Open vSwitch* o síťových tocích.

Následně byl napsán skript pro vytvoření virtuálních strojů připravených k využití SR-IOV virtuálních funkcí COMBO karty, buď pomocí *ndp_netdev*, nebo pomocí specializovaných nástrojů využívajících technologii NDP. Pomocí těchto virtuálních strojů jsme byli schopni ověřit funkci hybridního řešení.

Výstupem práce je verze knihovny *libp4dev*, která je schopna pracovat v prostředí jádra operačního systému Linux; testovací program a ovladač *p4test*, podpora převodu TC Flower pravidel do hardwaru pro ovladač *ndp_netdev*, ovladač *sriov_netdev* který využívá vytvořené podpory TC Flower spolu s *Open vSwitch* pro realizaci rychlé cesty u hybridního softwarově-hardwarového řešení síťového připojení pro virtuální stroje, FPGA firmware vytvořen v jazyku P4 pro použití s převodem TC Flower pravidel, a skript pro vytvoření virtuálních strojů pro použití s *sriov_netdev*.

Hlavní omezení výstupu práce představuje použitá verze NDK P4 překladače, která prozatím nepodporuje některé důležité součásti standardního P4 jazyka, jako jsou např. podpora hlaviček proměnlivé velikosti, a obecná podpora výpočtu kontrolních součtů zahrnujících data nákladu paketu. Další omezení přestavuje volba konkrétní COMBO karty, která byla sice ideální pro účely testování a vývoje, ale která nepodporuje externí paměti, které jsou kritické pro dosažení velkého množství pravidel toků.

Další omezení představuje způsob přidělení prostředků virtuálním funkcím, kde každá virtuální funkce dostane jeden NDP kanál. Rychlost přenosu je tak omezena schopností virtuálního stroje sériově zpracovat proud paketů, což představuje v případě serveru na kterém jsme testovali přenosovou rychlost zhruba 1Gbps. Toto omezení se dá ale jednoduše odstranit změnou topologie v syntézním systému platformy NDK.

BENC, M. *Hardwarová akcelerace klasifikace paketů s využitím rozhraní TC Flower.* Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav elektrotechnologie, 2020. 42 s. Bakalářská práce. Vedoucí práce: Ing. Jiří Libich, Ph.D.

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

Author's Declaration of Originality

Student's first and last name:	Marek Benc
Student's VUT ID:	203192
Thesis type:	Bachelor's Thesis
Academic year:	2019/20
Topic of the thesis:	Hardware acceleration of packet classification using TC Flower

I hereby declare that I have written this thesis independently, under the guidance of my supervisor and my consultant, using exclusively the technical references and other sources of information cited in the thesis and listed in the comprehensive references section at the end of the thesis.

As the author I furthermore declare that, with respect to the creation of this thesis, I have not infringed any copyright or violated anyone's personal and/or ownership rights. In this context, I am fully aware of the consequences of breaking Regulation § 11 of the Copyright Act No. 121/2000 Coll. of the Czech Republic, as amended, and of any breach of rights related to intellectual property or introduced within amendments to relevant Acts such as the Intellectual Property Act or the Criminal Code, Act No. 40/2009 Coll., Section 2, Head VI, Part 4.

Brno . . . .

author's signature

ACKNOWLEDGMENT

I would like to thank my consultant Ing. Tomáš Martínek, Ph.D. and the members of the Liberouter project (Cesnet TMC group) for introducing me to the concept of FPGA-based SmartNICs, for giving me access to their network cards and their powerful server machines, and for sharing valuable knowledge regarding the design of firmware for these cards and FPGA-based systems in general.

I would also like to thank my supervisor Ing. Jiří Libich, Ph.D. for helping me with the formal aspects of writing this bachelor’s thesis, for his professional guidance, and for all of the valuable advice he shared with me.

Additionally, I would like to sincerely thank Ing. Kristýna Jandová, Ph.D., the coordinator of the “Bachelor’s Thesis” class at the Department of Electrotechnology, FEEC, BUT, for her patience and help, during the difficult times of the COVID19 crisis, with the remaining questions about the formal aspects of this thesis and the state finals.

Brno . . . .

author’s signature

CONTENTS

INTRODUCTION

The concept of virtual machines traces its origins back into the 1960s. Like today, they provided a way to maximize the usage of expensive and often under-utilized mainframe (nowadays server) computers, as well as providing software backwards compatibility for customers migrating to new hardware. Additionally, the inherent isolation provided by virtual machines acts as an extra layer of security and fault tolerance [1].

Several fundamentally different approaches exist when it comes to implementing virtual machines. The approach that is most interesting to server administrators is the one described by Popek and Goldberg by their set of virtualization requirements [2], which describe a model known as classical virtualization.

Unlike with emulation, which involves explicitly parsing and interpreting the guest machine's instructions (potentially needing to go through hundreds of native instructions for a single guest instruction [3]), in classical virtualization, the guest's machine code is run directly on the host's processor, with privileged instructions being trapped. This technique allows virtual machines to fulfill the requirement that a statistically dominant fraction of machine instructions must be executed without VMM (*virtual machine manager*) intervention [2].

Historically, this has been difficult to achieve on x86 due to some of its design limitations, leading to workarounds like binary translation and CPU paravirtualization [3], but with the introduction of the AMD-V and Intel VT-x extensions in 2005–2006, classical virtualization has become possible on the x86 platform, allowing people to run VMs with unmodified OSes at near full speed [1].

One of the main bottlenecks with virtualization in general is I/O overhead. Traditionally, I/O is handled by the VMM (also known as the *hypervisor*), which can result in a lot of processing time being spent on explicitly copying and analyzing data, especially when dealing with VMs in data centers, which require a high, sustained data throughput from storage and networking devices.

A proposed solution is to give virtual machines direct access to hardware devices. The SR-IOV specification makes this more practical by allowing for a single physical PCI Express device to be shared by multiple virtual machines, providing "virtual functions" that behave like separate pieces of hardware. This model is especially useful for network adapters, where a virtual function more-or-less acts as a networking port that a virtual machine can be hooked up to [4].

The goal of this thesis is to utilize the recently introduced SR-IOV capabilities of CESNET's COMBO network cards, in combination with industry-standard tools like *Open vSwitch* and the P4 language, to create a managed, hardware-accelerated network switch for virtual machines. An implementation of the TC Flower interface is proposed, alongside FPGA firmware written in P4. This interface is intended for use by *Open vSwitch*, a software network switch implementation, which can use it to offload a large portion of its packet classification and processing operations into hardware, saving valuable CPU cycles and providing a fast virtual network switch configurable using the industry-standard *OpenFlow* protocol.

1 NETWORK SWITCHES

In the field of computer networking, bridging refers to a means of connecting several independent local area networks (LANs) to create a single larger, aggregate network (known as a *bridged LAN*), using a device known as a bridge. These devices allow machines connected to networks of different kinds (Ethernet, Token Ring, Wireless) to communicate with each other as if they were on a single physical network, as well as potentially providing a fallback connection in the case of a failure of network components [5].

They have also found use in increasing the efficiency of Ethernet networks by allowing for configurations where only two machines exist within a single physical network at a time. This is called a fully switched network, and it avoids the problem of transmission collisions, allowing for full-duplex communication over Ethernet cables. The type of bridge used in this scenario is known as a network switch [6].

1.1 Level 2 Switches

This type of switch works by learning the MAC addresses of all of the devices on the local network, and transparently passing MAC frames destined for them to the appropriate ports. In order to be addressable as an OSI Layer 2 end node, these switches have a MAC address assigned to them, while also transparently forwarding frames, generally without modification (except perhaps for IEEE 802.1Q VLAN tagging). They also come with spanning tree protocol support, in order to avoid loops in a network which contains more than one switch.

Level 2 switches are protocol-independent, since they operate at the OSI Layer 2. This does however means that they don't scale well due of broadcasts. To a certain extent, VLANs help to alleviate this problem, but they introduce their own set of problems as well, such as making communication between machines within different VLANs less efficient, since the traffic between these machines needs to be directed through an external router. This problem is addressed by multilayer switches [7].

1.2 Multilayer Switches

A multilayer switch, sometimes referred to as a Level 3 or Level 4+ switch (based on its capabilities), is a network device which in addition to performing MAC bridging also examines and utilizes information from packet fields that correspond to higher layers of the OSI model. This information can be used for IP routing, as well as for policy-based switching, where a priority may be assigned to traffic of different applications based on their importance [7], [8], [9].

Their model of operation generally involves using algorithms like *Open Shortest Path Firs*t (OSPF) or *Routing Information Protocol* (RIP) to communicate with other

Level 3 routers or switches to build routing tables, and unlike Level 2 switches, they tend to modify packet header fields, such as decrementing the *Time To Live* (TTL) field and recalculating header checksum fields. By performing Level 3 switching, they allow for communication between machines in different VLANs to be efficient [7].

1.3 Managed Switches

Switches that provide a configuration interface are known as managed switches. They allow for better control and management of the network traffic within a LAN, allowing to segment the network into VLANs and to set priorities on different types of network traffic, as well as providing diagnostic data by monitoring network health [10].

The *Simple Network Management Protocol* (SNMP) is an example of a protocol used to manage and configure these types of network switches. Support for the *OpenFlow* protocol is sometimes provided as well, serving as a vendor-independent way of providing *Software-Defined Networking* (SDN) support [11]. The *P4Runtime* specification fulfills a similar purpose for switches that support the P4 language for describing packet processing [12].

2 SOFTWARE-DEFINED NETWORKING

SDN is a network management approach where the control and forwarding functions of network management are decoupled. A centralized, global view of the entire network is provided, appearing to applications and policy engineers as a single logical switch. This allows for the underlying hardware to be abstracted away, and for network control to be directly programmable, allowing for dynamic adjustments to network-wide traffic flow based on changing needs [11].

2.1 The OpenFlow Specification

The OpenFlow Specification defines an abstract model of a network switch known as an *OpenFlow Logical Switch*, as well as a communication protocol that allows it to be configured from a remote machine known as an *OpenFlow Controller* (Figure 2.1).

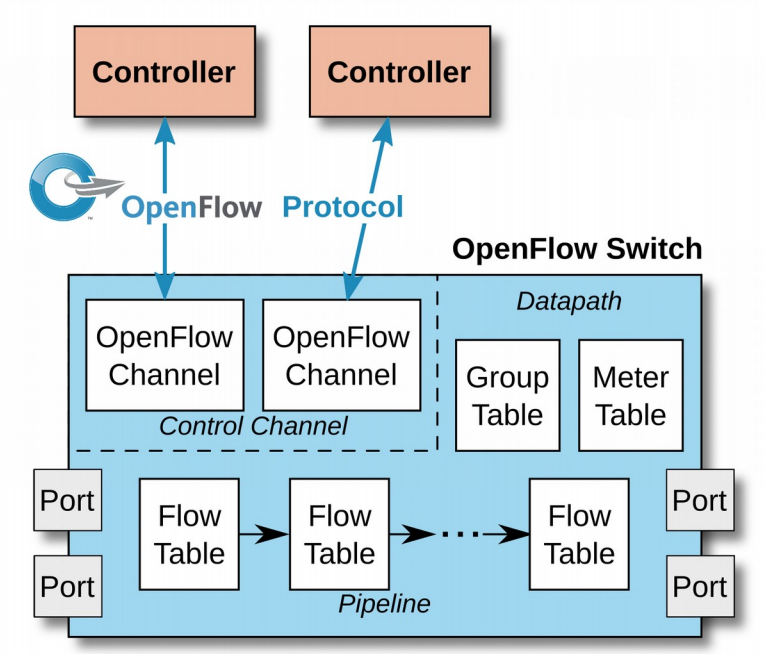

Fig. 2.1: Main components of an OpenFlow switch [13]

The basis for packet look-up and forwarding within an OpenFlow switch are *flow tables* and a *group table*. The flow tables contain *flow entries*, with *match fields* used for selecting packets based on the values of their header fields, *counters* to measure how many packets and bytes a given *flow entry* had selected, and a *list of actions* to perform on selection, potentially including classification into a group, which in turn results in actions described in the *group table* for the appropriate group to be performed.

The specification also describes several categories of ports that packets can be received from and ultimately redirected to. The most straight-forward are *physical*

ports, which either correspond to hardware interfaces of the switch, or potentially to virtual slices corresponding to network interfaces in a virtualized deployment.

Another category of ports are *logical ports*, which are a higher-level abstraction that may be implemented by non-OpenFlow means. For example, *logical ports* may correspond to *physical ports* with some extra implementation-defined processing added. Some of the possible uses are link aggregation groups, tunnels, and loopback interfaces.

When no flow entry manages to match on an incoming packet, the action of a special entry called the *table-miss flow entry* is invoked (if this entry is not present, the packets are discarded). A common use for this entry is to send unclassified packets to the controller for inspection. If the switch implements Level 2 switching or Level 3 routing, these unclassified packets may be sent into a *logical port* to be processed by these technologies.

Some of the supported actions that an OpenFlow Switch may perform on packets are redirection, grouping, discarding, queue id specification (to provide basic QoS support), metering, VLAN header manipulation, header field setting and copying, and TTL decrementation [13].

2.2 Open vSwitch: A Software Implementation

Open vSwitch is a multilayer software switch written in C, well suited to function as a virtual switch in VM environments. It supports multiple virtualization technologies, including Xen/XenServer, KVM, and VirtualBox, as well as deployment of a single instance across several physical servers.

Some of the features it provides are support for 802.1Q VLAN, NIC bonding with or without LACP, NetFlow, sFlow, QoS (Quality of Service) configuration, Geneve, GRE, VXLAN, STT, LISP tunneling, 802.1ag connectivity fault management, a transactional configuration database with C and Python bindings, high-performance forwarding using a Linux kernel module, and support for OpenFlow 1.0 with numerous extensions.

Additionally, the Open vSwitch project provides, among other useful tools, a utility for querying and controlling OpenFlow switches and controllers, *ovs-ofctl*. This can be used either with ovs-vswitchd (the switch daemon), or with any other OpenFlow compatible switch [14].

Because a pure-software approach for a network switch would involve a significant amount of overhead, there are options for providing hardware acceleration. The two officially supported approaches are implementing either an *ofproto provider* or a *dpif provider*. An *ofproto* provider can take full advantage of hardware with support for field masking, whereas a *dpif* provider is usually easier to implement, but it splits wildcard rules into exact-match entries, resulting in an inefficient use of TCAMs in hardware that support wildcarding [15].

Support in OVS for hardware rule offloading via the Linux TC Flower interface is currently implemented as a *dpif provider* [4].

2.3 The P4 Language

One of the limiting factors of *OpenFlow* is that the flow tables and group table are pre-defined by the standard and by hardware vendors. This means that a network administrator can't change the classification mechanisms and the actions that these tables support [13]. The P4 language project aims to resolve this issue by developing a domain-specific, declarative language that allows network administrators to describe how packets are to be classified and processed by the device's pipeline [16].

The language itself addresses the configuration of a forwarding element (definition of packet headers and parsing, tables, actions, and pipeline layout and control flow), which upon being configured may have its tables populated in a similar manner to a switch supporting *OpenFlow* [16]. The *P4Runtime* specification aims at creating a vendor-independent configuration interface for P4-based forwarding elements [12].

An abstract model consisting of a parser and a set of match+action tables split between ingress and egress is described by the specification. (Figure 2.2)

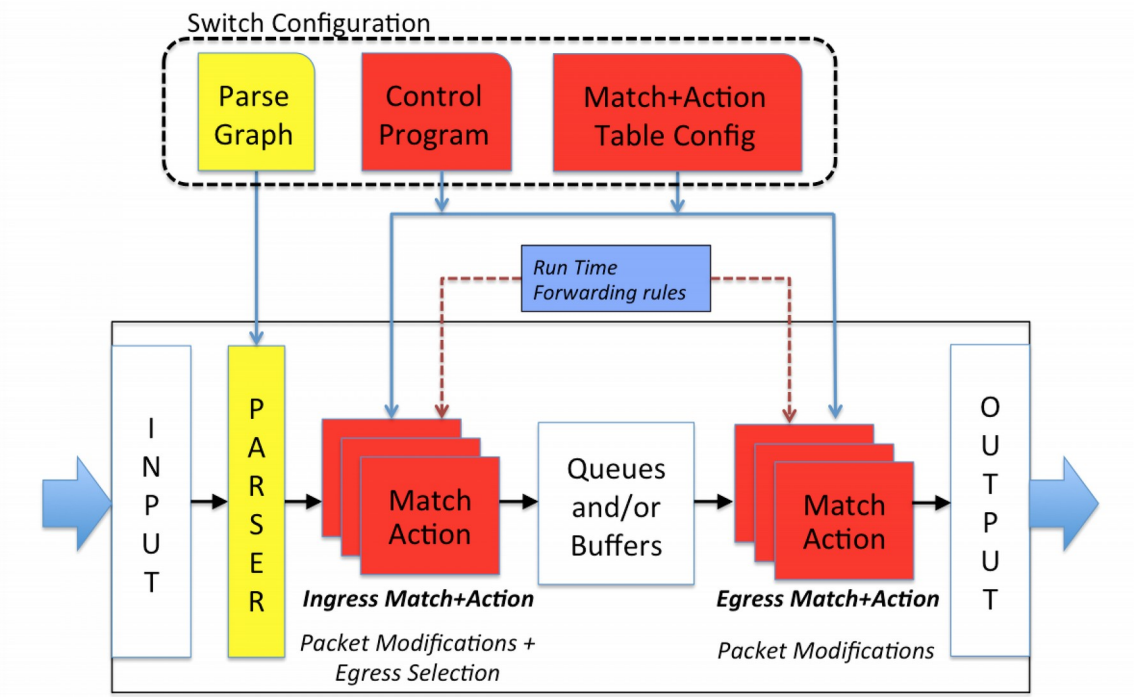

Fig. 2.2: P4 Abstract Forwarding Model [16]

The parser provides a *Parsed Interpretation* of each received packet, which is then used by the ingress pipeline match+action tables to select packets to perform actions on, and to specify the destination ports of the individual packets. This destination is decided upon before a packet enters the egress pipeline. Upon reaching the end of the egress pipeline, the packet is re-assembled from its potentially modified parsed interpretation and sent to the appropriate port [16].

Additionally, the P4 language provides support for packet cloning and recirculation, which is useful for monitoring purposes (port mirroring) and for implementing recursive packet processing policies.

Checksum verification and recalculation support is also provided. Generating hash or checksum values involves utilizing a selected algorithm to process a set of bytes from a packet, defined by a field list, to produce a fixed-width integer result. Checksum fields may be verified at ingress and updated at egress using a *calculated_field* declaration, preferably located just after the header instance declaration. Alternatively, the primitive action *modify_field_with_hash_based_offset*() may be used to calculate a checksum or hash value explicitly [16].

The language also provides support for memories which hold their values across multiple packets being processed, known as *stateful memories*, in the form of *meters*, *counters*, and *registers*. *Meters* provide a visual representation of a data rate, outputting a red, yellow or green color signal, with the metering algorithm being implementation-dependent. RFC 2697 and RFC 2698 provide examples of possible metering algorithms.

Counters provide an exact numeric amount of either packets or the bytes of packets that fulfill a certain criterion, such as being processed by a specific match+action rule, and *registers* are stateful memories that can be read and written by actions in a general manner, e.g. to verify that a "first packet" of a flow had already been encountered [16].

Actions, also known as *compound actions*, are imperative functions consisting of potentially multiple *primitive actions*, and are used to manipulate packets in a defined manner. A few notable examples of primitive actions are *modify_field*(), which is used for modifying header field values with optional bit-wise masking; *add_header*() and *remove_header*(), which are respectively used to add or remove fields from a packet (e.g. a VLAN tag); *drop*(), which used for dropping (ie. discarding) packets, and actions like *add*(), *subtract_from_field*(), *bit_xor*() or *shift_left*() which perform arithmetic and bit-wise operations on packet fields.

In addition to the previously mentioned *registers*, which are accessible using the primitive actions *register_read*() and *register_write*(), compound actions can optionally take parameters. The values passed to the actions as parameters are programmed using the run-time API as a part of the match+action entries which invoke the action [16].

The other part of a match+action entry in a table are the match criteria. P4 supports matching on *exact* values of fields, it supports *longest prefix matching* (lpm) where the rule with the longest common prefix with a field is selected (useful for subnets), *ternary matching* where a bitwise mask is applied to a field before comparison, *range matching* where a field's value has to be within a certain numeric range, and *valid* matching, which is used for checking whether the parser had extracted a certain header from a given packet or not.

Each table declaration consists of a set of matching criteria, the applicable actions on a match, and table properties like the size, and whether or not it should support time-outs. Whether or not a packet is subject to the match+action rules of a table is decided by a *control flow program*, which may conditionally apply tables to packets, and call other *control flow functions* [16].

3 TC FLOWER AND ITS USE WITH SR-IOV SWITCHES

The standard model for SR-IOV switches in Linux revolves around the concept of each virtual function being represented by a *netdev*. The term *netdev* refers to an abstract object which represents a network port, and within an SR-IOV setup, a *netdev* for a VF is known as a *virtual function representer* [4], [17].

These representers serve a role similar to TAP devices in a paravirtualized network setup. Packets sent through the egress of a VF representer appear at the ingress of the VF (thus are accessible to a virtual machine), and packets sent through the egress of the VF (e.g. by a virtual machine) arrive at the ingress of the appropriate representer.

Within this model, in order to create a network for virtual machines utilizing VFs, their representers are put into a software bridge. This approach is inherently inefficient, as all traffic ends up going through the host operating system, but with the use of bridge implementations that support hardware acceleration, such as *Open vSwitch*, it allows for a hybrid model where only some network traffic goes through the host operating system (e.g. packets needed for MAC address learning and flow identification), and the rest is handled by hardware, by traffic control rules installed by the software switch [4].

In the case of *Open vSwitch* on Linux, the *TC Classifier-Action* subsystem may be used for this purpose. In particular, the *flower* filter (referred to throughout this work as TC Flower) was designed as a flexible alternative to the official Open vSwitch kernel data-path implementation, with far less code duplication achieved by utilizing existing frameworks within the Linux kernel [18].

3.1 The TC CA Subsystem

The Linux kernel contains support for advanced network traffic control. Upon arrival at the ingress of a *netdev*, and before reaching the egress of a *netdev*, incoming and outgoing packets respectively are subject to the *Traffic Control Classifier-Action* (TC CA) subsystem.

This subsystem is configurable by user-space applications using the netlink API, and it revolves around the concept of *queuing disciplines* (qdiscs), which may reorder and limit the flow of packets, and use *filters* to sort the packets into various different *classes*, allowing them to treat different kinds of packets differently based on the criteria observed by the *filters*. A class may either be a FIFO or another *qdisc*. Using *qdiscs* as *classes* within *qdiscs* allows network administrators to build sophisticated, hierarchical chains of traffic control rules.

The *filters*, however, are not limited to merely sorting packets into *classes*. They have a wide and easily extensible selection of possible actions they may perform, which is the reason behind it being called a Classifier-Action subsystem [17].

3.2 Queuing disciplines on Ingress

The primary purpose of a *queuing discipline* is to schedule outgoing network traffic, and all *netdevs* are required to have one, even if it's merely a simple FIFO, attached on their egress. This is known as a *root qdisc* [19].

A *qdisc* can, however, also be attached to the ingress of a *netdev*. An *ingress qdisc* is a dummy queuing discipline which exists solely as an object onto which *filters* may be attached (figure 3.1). A typical use for an *ingress qdisc* is with a *policer* (a filter which performs one action when the data rate is above a specified value, and another when it's below) to limit the amount of traffic accepted on a network interface [19].

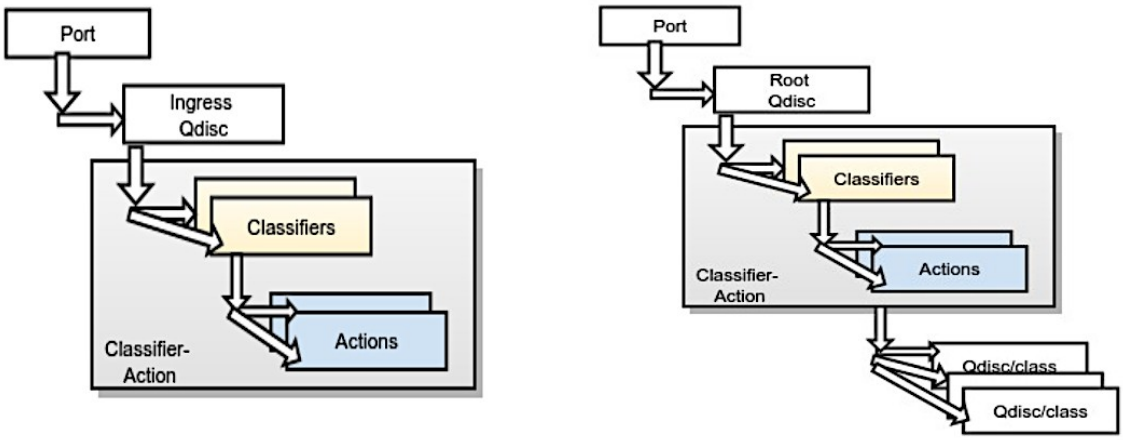

Fig. 3.1: Ingress and Egress traffic control in Linux [17]

The *ingress qdisc* has become especially useful with the introduction of support for SR-IOV virtual switches, where it can be used with filters which support hardware rule offloading (such as TC Flower) to handle switching rules [4].

Its usefulness as an anchor point for classifier-action rules eventually lead to the *ingress qdisc* being generalized into the *clsact qdisc*, which in addition to the ingress also works on the egress of a *netdev*. The *clsact qdisc* replaces and is fully backwards compatible with the *ingress qdisc*, but it does not replace the *root qdisc* on egress. Instead, it acts as an anchor point for classifier-action rules that should be applied before packets reach the *root qdisc* (for example, in the case of the *root qdisc* being *classless*, since *classless qdiscs* don't support having *filters* attached to them), and it provides a central egress counterpart to the ingress classifier-action rules, allowing for them to easily share their driver state [20].

3.3 Overview of TC Filters

Filters look at incoming packet data and/or metadata and perform a defined action when applicable. They are kept in a priority-ordered list for each protocol, where the priority values with a smaller numeric value represent higher importance. This list is traversed for each packet until an appropriate filter is found [17].

Originally, *qdiscs* used to support having only one such chain of filters, which conceptually corresponds to a single flat match+action table, with the option of being able to modify and then recirculate packets, making them go through the list of filters again to create hierarchical rules (e.g. for encapsulation). Since it's convenient to be able to create such rules using a hierarchy of tables instead, and because table hierarchies are how such rules are commonly implemented in hardware, multi-chain support was eventually introduced to TC CA [21].

3.3.1 The “basic” filter

The *basic* filter utilizes the extended match infrastructure of the Linux kernel, which allows for complex matching rules to be built out of a group of simple rules chained together using logic AND and OR operators, and parentheses [22], [23].

The currently supported rules in the extended match infrastructure are *cmp*, which performs arithmetic comparisons of packet data fields at specified offsets (with the ability to specify the OSI layer); *meta*, which does the same for supported metadata fields; *nbyte*, which matches on a sequence of bytes within a packet; *ipset*, which checks whether a packet is a member of an ipset; *ipt*, which checks for xtables matches; *canid*, which matches on CAN bus frames, and *u32*, which performs bit-wise masked matching on a specific packet field [23].

3.3.2 The “flow” filter

The *flow* filter, not to be confused with the *flower* filter, is a filter that serves to extend the hashing capabilities of a *Stochastic Fairness Queuing* (SFQ) *qdisc*, while avoiding the need of hard-coding new hashing functions into said *qdisc* [24].

The *SFQ qdisc* schedules the transmission of packets based on “flows”, trying to ensure fairness so that a single application doesn’t utilize the entire upload bandwidth by making *flows* transmit data in turns. The term *flow* refers to a distinct connection or conversation between two machines, such as the packets of a TCP session [19], [25].

The *flow* filter supports matching on a set of pre-defined fields, such as the source and destination address of the level 3 protocol, the specific level 3 and 4 protocols that are used, the priority key (DSCP/ECN value for IP packets), etc. In addition to these fields, *flow* can utilize the extended match infrastructure used by the *basic* filter [24].

3.3.3 The u32 filter

The universal 32bit filter, also known as the “ugly” 32bit filter, can be used to match on arbitrary bit fields in a packet. It uses values, masks, and offsets, with several abstraction directives that provide a higher level for defining matching rules.

In addition to being able to perform an action or assign packets to classes, the u32 filter can delegate packets to another filter, and when used with another u32 filter, this allows it to be used to build arbitrarily complex match+action policies.

The filter delegation is typically achieved by using hash tables, with the match field providing the key from which a hash value is computed [26].

3.3.4 The BPF filter

The BPF filter is a fully programmable filter which provides an implementation of the *Extended Berkeley Packet Filter* (eBPF) and the *Classic Berkeley Packet Filter* (cBPF) instruction set architecture. These are minimal instruction sets, which are designed for implementing small programs that can be safely executed in the kernel environment.

The eBPF instruction set is seen as a successor to the classic BPF instruction set, providing better run-time performance, with specific design considerations ensuring that it works well with JIT compilers.

Support for these instruction sets means that the user is not limited to any particular set of classifiers or actions, as they can write their own specialized classification and actions code in a subset of the C language, e.g. using the LLVM framework. Since the code is specialized, it can also be remarkably efficient, as it doesn't have to deal with unused features that could potentially slow down the classification process [27].

3.3.5 The flower filter

Similarly to the *flow* filter, the *flower* filter identifies packets based on them belonging to a "flow" by observing the values of well-known packet fields and metadata. Unlike the *flow* filter, which mainly serves to distinguish between packets of different flows, the *flower* filter aims at selecting packets of specific flows and performing specific sets of actions on them.

The *flower* filter utilizes the Linux *flow dissector* to extract the fields of interest from a packet, which then serve as the packet's matching key. This key is compared to the keys in the list of installed match+action rules, and upon a match, the corresponding action is performed [28].

Unlike the other previously mentioned filters, which allow for arbitrary fields to be matched, the *flower* filter operates with a fixed set of supported matching fields, although the selection of fields is rather comprehensive. This approach is similar to that of *OpenFlow*, which is one of the main inspirations for the *flower* filter [18].

3.4 Overview of TC Actions

In order to fulfill their classification purpose, *filters* within the TC CA subsystem have a *built-in action* that selects an appropriate *class* for matching packets. Filters may also have an arbitrary list of *programmed actions* attached to them, which allows for the passing network traffic to be manipulated in various ways, such as by dropping packets, by modifying packet contents, by having packets copied or redirected (stolen) to a different network interface, etc.

The *programmable actions* are designed in the spirit of the UNIX philosophy, with a single action doing just one thing and doing it well, and being able to be combined with other actions by attaching the input of one action to the output of another. It is also relatively easy to implement a new TC action in case the need arises [17].

3.4.1 The “generic action”

A core aspect of policy specification in the TC CA subsystem are its pipeline controls, known as *pipeline opcodes*, which are used by *programmable actions* to control packet flow through the pipeline, their existence serving as a form of separation of policy and mechanism.

The generic action, also known as *gact*, is a *programmable action* that doesn’t do anything on its own, existing solely as a vessel for *pipeline opcodes*, analogous to how the *ingress* and *clsact qdiscs* exist solely to have filters attached to them [17].

The *drop* opcode causes matching packets to be dropped, the *reclassify* opcode causes classification to restart by jumping to the first filter in the *filter chain* of a *qdisc*, the *continue* opcode causes classification to continue by jumping to the next filter in the current *filter chain*, the *pipe* opcode causes the next action in a list of actions to be performed, and the *pass* opcode causes packet classification to end for a given packet, with the *qdisc* being able to process it afterwards [29].

3.4.2 The pedit action

This action allows for arbitrary packet data to be changed. Either a numeric offset and field size (potentially with an offset value retrieved from a separate field of the packet), or the name of a commonly recognized header field, can be used to identify the section of the packet that should be modified.

The selected part of the packet may be cleared (set to zero), inverted bit-wise, set to a specific value, or have a numeric value added to it [30].

3.4.3 The “checksum” action

Modifying packet fields with *pedit* might cause the various checksum values stored in the packet to become invalidated. The *checksum* action serves to correct this, triggering checksum recalculation of specified header fields.

At the time of writing, checksum recalculation is available for the IPv4 header checksum, ICMP and IGMP header checksums, as well as TCP, UDP, UDP-Lite and SCTP checksums [31].

3.4.4 The mirred action

Packet mirroring and redirection, which is useful for network analysis, and in the case of SR-IOV switches with TC Flower offloading, for having the hardware itself redirect incoming packets to their correct destinations, is implement by the *mirred* action.

The *mirred* action additionally allows the network administrator to specify whether the copied or stolen packets are to appear on the *ingress* or *egress* of the specified destination network interface. *Ingress* mirroring is useful if a software application is listening on the given destination network interface, whereas *egress* mirroring is useful for sending the packets into external hardware [32].

3.4.5 The “vlan” action

Manipulation of IEEE 802.1Q and 802.1ad tags is provided by the *vlan* action. It allows for VLAN encapsulation and decapsulation, as well as for modifying existing 802.1Q tags. Each VLAN tag has a protocol ID, VLAN ID, and priority number, all of which can be set using this action [33].

3.4.6 The skbedit action

To modify a packet’s associated metadata, the *skbedit* action is provided. It can be used for changing the transmission queue used for a given packet, for changing the packet’s type (supporting the *host*, *other-host*, *broadcast* and *multicast* types), for changing firewall marks, as well as for overriding classification decisions by either specifying a priority or using the *Differentiated Services* field of IPv4/IPv6 headers [34].

4 THE NDK PLATFORM

The NetCOPE Development Kit, which is developed by CESNET and commercialized by Netcope Technologies a.s., is a platform that allows for the rapid development of hardware-accelerated network applications using FPGA network cards [35], [36], [37].

It consists of a collection of efficient implementations of components useful for network cards (network interfaces, timestamp generation, PCIe bus interface, fast DMA, etc.), a build system for synthesizing firmware for FPGA chips from both Xilinx and Intel, and drivers, libraries and utilities for interfacing with and manipulating with the cards. Figure 4.1 provides a general overview of the architecture [35].

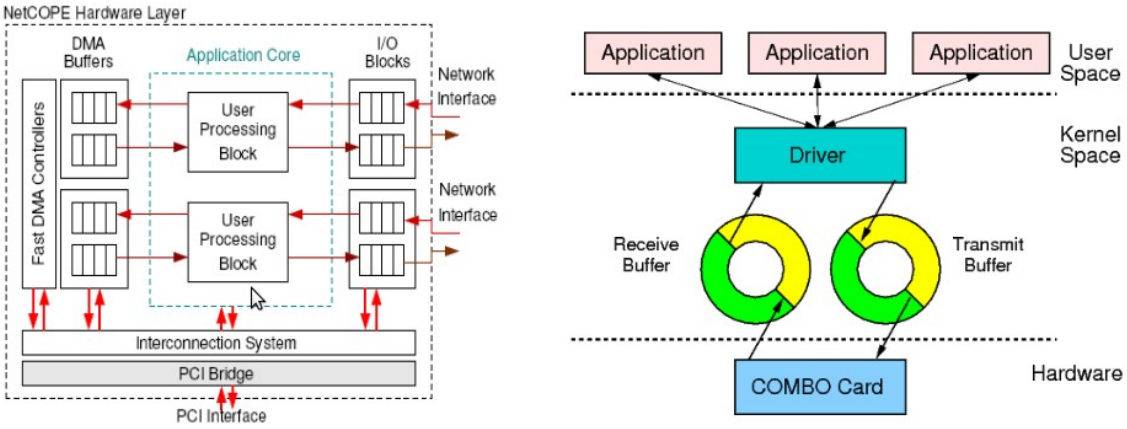

Fig. 4.1: NetCOPE Architecture [35]

4.1 NetCOPE FPGA Boards

The term *NetCOPE FPGA Board* (NFB) generally refers to any FPGA-based network card that the NDK platform has been ported to. In addition to CESNET's COMBO cards, parts of NetCOPE have been ported to Intel's programmable acceleration cards, such as the Intel FPGA PAC N3000 [38].

Within this work, we will restrict ourselves to CESNET's COMBO cards, although supporting Intel's cards with the proposed TC Flower extension is an eventual goal, given how it would allow for the use of unmodified guest OS images on VMs.

The specific card used throughout this work is the COMBO-200G2QL (Figure 4.2). The main factors that lead to its selection are its fully operational SR-IOV support, and the fact that it has two 100Gbps QSFP28 transceivers, which allows for easier testing by utilizing one of the transceivers to generate network traffic for the other to receive.

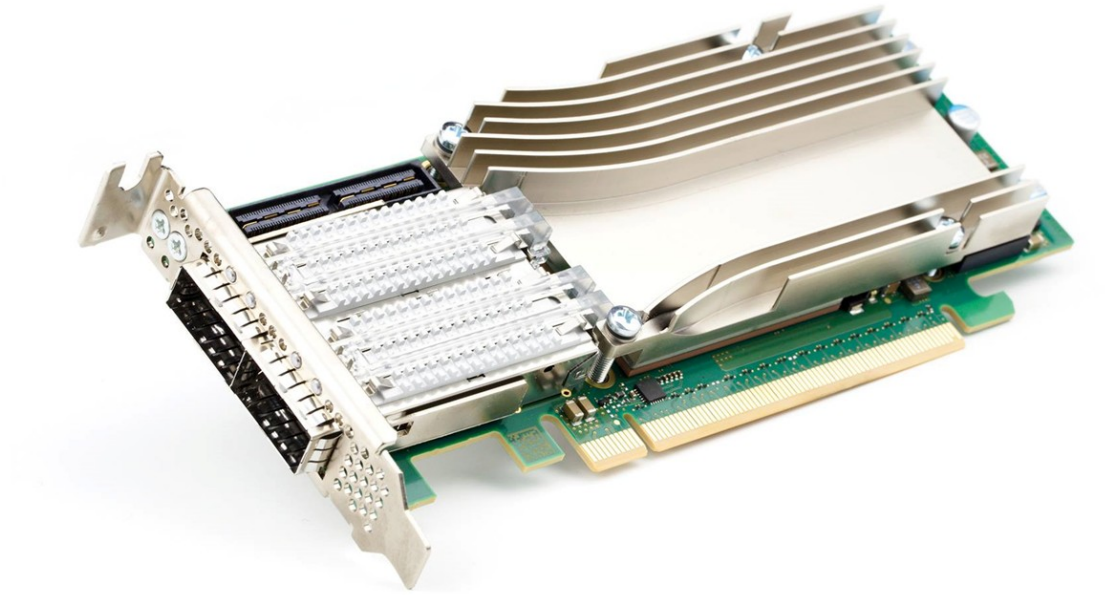

Fig. 4.2: The COMBO-200G2QL card

At the heart of the COMBO-200G2QL card is a Xilinx Virtex UltraScale+ FPGA. The card can be configured with either one or two Gen. 3 PCI-Express end-points, with 16 lanes each, allowing for a 200 Gbps data throughput to RAM [39].

4.2 NetCOPE Data Plane

Network traffic is conventionally processed a packet at a time. Since the overhead of processing each packet individually can be substantial when operating with a high data throughput, the *NetCOPE Data Plane* (NDP) provides an alternative that uses buffers containing many packets, which are transferred between software and hardware in bulk.

These buffers are directly accessible to applications via the *libnfb* library, and allow for more efficient implementations of specialized network data analysis and processing solutions [38].

The software driver communicates with so-called NDP controllers on an NFB card, which serve to implement DMA communication channels, with RX channels copying packet data into the computer's memory, and TX channels reading packet data from it. For each of these channels, the driver maintains a ring buffer, and allows applications to "subscribe" to the channels to gain access over the buffer, with it being mapped into the application's address space by the *mmap*() system call.

To read received packets, the *ndp_rx_burst_get*() function can be used to fill out an array of packet structures, with each member containing a pointer to the packet's data, metadata, and the length of these fields. These structures refer to data in the ring buffer, and thus, once the application no longer needs them, it needs to notify the NDP driver by invoking the *ndp_rx_burst_put*() function. This will move the read pointer for the given subscriber to after the set of packets which was read. The read pointer for the ring buffer is set to the value of the read pointer of the subscriber who's read pointer is the

farthest away from the write pointer, which means that in order for space to be freed in the ring buffer, all active subscribers need to read (or at least acknowledge) all received packets on a given RX channel [40].

As for transmitting packets, it involves reserving space on the ring buffer of a TX channel, copying packet data into it, and then notifying the driver that the data is ready to be transmitted. The *ndp_tx_burst_copy*() function works by copying data from the previously defined packet structures into the ring buffer, providing an easy, albeit slow way of transmitting packets.

The *ndp_tx_burst_get*() function allows for a more efficient approach, by utilizing a partially filled packet structure array to reserve space on the ring buffer for a given set of packets. It serves as an allocation function, taking the length parameter of each of the data and metadata fields, and filling out the buffer parameters with appropriate pointers into the ring buffer. The application can then use these pointers to populate the buffer, and use the *ndp_tx_burst_put*() function once it's done to publish the packets.

4.3 Command-line tools

In addition to the *libnfb* library, a set of command-line utilities is provided as a part of the NDK Platform [40].

nfb-boot is a utility that allows for FPGA firmware manipulation on an NFB card. It can be used to upload new firmware to a card, as well as to reboot the card and switch between which of the currently loaded firmware images is being used (the cards usually have two firmware images, with one of them serving for recovery purposes).

nfb-info is a tool that displays basic information about an NFB card, including the board type, currently loaded firmware and its capabilities, and PCI bus and NUMA node information.

nfb-dma is a tool for querying the status of the individual DMA channel controllers, showing information about how many packets were transmitted or received via a DMA channel, as well as the current state of its registers.

nfb-eth is a tool used for configuring and querying information about the network transceivers on an NFB card. Similarly to *nfb-dma*, it shows how many packets were transmitted or received via a transceiver and its status information, but it also allows the user to enable or disable a transceiver, configure the PCS/PMA, and configure the MAC filter of a transceiver.

nfb-bus is a tool that provides direct access to the MI32 bus of an NFB card. This bus is used for communicating with an NFB card outside of packet data transfers, and is used by all of the above-mentioned utilities. This makes *nfb-bus* especially useful when working on new hardware features for NFB cards, as it provides a convenient means of configuring the newly implemented feature, allowing hardware designers to fully flesh out the design before spending time on implementing a proper configuration utility.

ndp-tool is a utility that performs packet transfers over the *NetCOPE Data Plane*. It supports generating sets of packets and transmitting them, transmitting predefined sets of packets from .pcap files, receiving packets and either displaying information about

them or storing them into a .pcap file, and re-transmitting any received packets. These individual modes of operation can also be accessed via standalone executables, such as *ndp-generate*, *ndp-transmit*, *ndp-read*, *ndp-receive*, *ndp-loopback*, etc.

4.4 The P4 Compiler

The NDK Platform provides a compiler capable of generating VHDL architectures from P4 descriptions. At the time of writing, only the P4.14 revision of the P4 language was supported, although support for the P4.16 revision was close to being released [41].

4.4.1 Compiler description

The compiler works by creating an internal representation of a forwarding element described in the P4 language and mapping it to a network device architecture. It uses a Parser-Deparser model, where packets are broken down into individual header fields, potentially modified by a processing block, and then reconstructed by a deparser.

The main components of a match+action table in this design are the *search engine* and the *action engine* (figure 4.3). The purpose of the search engine is to pick the most appropriate action for a packet based on its headers and metadata [42].

At the time of writing, search engines of three different kinds were supported. The *ternary content-addressable memory* (TCAM) engine is the most versatile, as it is able to provide exact, LPM and ternary matching, but it also consumes the most resources out of the three. The b*inary search tree longest prefix match* (BST LPM) engine allows for a more efficient implementation of LPM matching, and the Cuckoo hashing engine is optimal for exact matching, using the least resources out of the three [41].

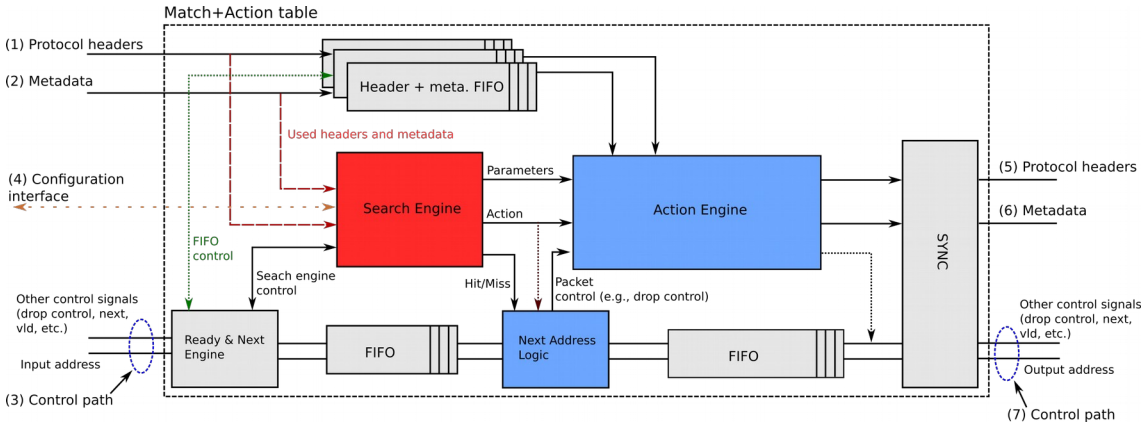

Fig. 4.3: Architecture of match+action tables in NDK P4 [42]

The *action engine* performs operations on packet header fields, metadata, as well as control information (e.g. drop control), with the output values dependent on the action selected by the *search engine*. Primitive actions are implemented as blocks in parallel with each other, with a distribution block assigning them inputs and synchronization tags, and a selection block deciding, based on the sync tags, what data to provide on the output in order to preserve the sequential logic of the action definition [42].

Within this implementation of P4, packet redirection is performed by modifying the *egress_port* field of the intrinsic metadata structure. This field is an 8-bit value, and it refers to network transceiver channels on the range of 0-127, and NDP channels on the range of 128-255. By default, it has the same value as the *ingress_port* field, which means that the default behavior is to return packets to where they came from.

Some of the notable missing features of the current implementation are support for variable-length headers, support for implicit checksum verification, and for recirculation and mirroring of packets. Additionally, checksum and hash-value generators can only use data from defined headers, use of the *payload* keyword within field lists for hash generators is not yet supported. To partially mitigate this limitation, an extension was introduced which calculates a 16-bit one's complement sum of the payload, which can be used in TCP and UDP checksum calculations [41], [43], [44].

4.4.2 Firmware configuration

The FPGA architecture generated by the compiler can be synthesized using the build system provided by NDK, and once loaded onto an NFB card, can be configured using the *libp4dev* library [41].

This library can either be used directly in the case of specialized applications, or it may be used as a back-end by a *P4Runtime* implementation to provide a standardized P4 configuration interface. Limited *P4Runtime* support using the *libp4dev* library is provided in a locally maintained branch of the PI repository [45].

The synthesized firmware may contain one or more P4 cores. The 200G2QL card contains two P4 cores by default, one for each of the two QSFP28 transceivers, where received packets are sent into the P4 core which corresponds to the transceiver. The P4 cores do, however, have the ability to send packets anywhere, and the mapping of packets transmitted from NDP DMA channels to P4 cores (or transceivers, if one wishes to skip all P4 pipelines) is configurable on run-time with the TX Mapper components.

Within the *libp4dev* library, a P4 core is represented by a *p4device_t* structure. This structure is meant to be allocated manually (useful for making it a part of a larger structure) and initialized to P4DEVICE_EMPTY before having *p4device_init*() invoked on it. This function accepts either the path to the card's NFB driver device node in /dev, or a device ID from which it can guess this path, as well as a component ID to specify which of the card's P4 cores to represent with the newly initialized *p4device_t* structure.

Match+action tables are accessed using the *p4device_get_table*() function, which accepts a *p4device_t* structure and a table name, returning a *p4table_t* structure. This structure can be used with functions like *p4table_insert_rule*(), *p4table_modify_rule*(), *p4table_delete_rule*(), and *p4table_insert_default_rule*() to maintain the matching and action rules of the given table.

Because *libp4dev* keeps its table information state in memory, and this state is lost upon termination, whenever *libp4dev* opens a P4 core, it does not know about the rules which were installed into its tables beforehand. Therefore, the *p4device_reset*() function is provided. It accepts a *p4device_t* structure, and it fully clears and zeroes out all tables and registers within the P4 core. This makes it useful as an extra initialization step for cases when knowledge of the exact state of the P4 core is important [41].

5 EXPERIMENTAL PART

The development process of implementing TC Flower offloading support on the NDK Platform was largely a learning experience. While it is open source, some parts of the Linux kernel source tree aren't particularly well documented, which means that the most effective way to get familiar with certain APIs is to study their code and the code that uses them, and to write your own test drivers that interacts with said APIs. This holds true for the TC Flower offloading API, at least at the time of writing. The Bootlin Elixir Cross Referencer prove to be especially useful for studying the kernel source tree [46].

In order to be able to offload TC Flower match+action rules, the driver behind a network interface needs to be able to access hardware resources which control packet classification in a card. Within the NDK Platform, this is provided by the *nfb* kernel module, which among other things exposes the MI32 bus of the card, which is used for configuring match+action rules with the P4 implementation developed by CESNET.

The *nfb* kernel module itself follows a modular design, where it consists of several sub-modules that fulfill different roles, such as a character device driver for granting user-space applications access to NFB cards, the NDP driver for packet transfers, the boot driver for flashing and booting FPGA designs, and a driver with network interfaces linked to NDP (DMA) channels. While the *nfb* kernel module does support SR-IOV, it does not come with a sub-module that creates *representer* network interfaces for *virtual functions*, which is ultimately implemented within this work for use with OVS [40].

5.1 Configuring P4 pipelines from within the Linux kernel

The first roadblock encountered on the journey of implementing TC Flower offloading using the NDK P4 implementation was the fact that *libp4dev*, the low-level library used for configuring P4 pipelines of FPGA architectures generated by the NDK P4 compiler, is a user-space library, not intended to be run in the Linux kernel environment.

A saving grace in this regard is the fact that the library is written in the standard C programming language, and it doesn't depend on any libraries that would be difficult to port to the kernel environment. Additionally, the *nfb* kernel module provides APIs that are very similar to the user-space APIs provided by the *libnfb* library that are ultimately used by *libp4dev*.

After some negotiations with the developers of *libp4dev* and the *nfb* kernel module, it was decided to introduce compatibility macros into the *libp4dev* source tree to provide functionality usually provided by the standard C library by the equivalent kernel calls for cases when the library is compiled in the kernel environment, as well as providing macros for logging messages which would either be backed by *printf*() or *printk*(), instead of invoking *printf*() function directly.

These compatibility macros would be introduced by the *config.h* header file, which is typically used to provide macros about detected system characteristics and requested features to enable during compilation. This header file is typically included by every

code file within the project, so that the macros are always available. A script would then be introduced that would create an archive with the *libp4dev* source tree, and a special *config.h* file that contains not only feature test macros, but also type definitions, and macros that replace standard C library calls like *malloc*() and *strdup*() with their Linux kernel environment counterparts. With this file available, all that a *libp4dev* code file needs to do to be able to function in the Linux kernel is to conditionally avoid including any standard C library headers, as is shown in listing 5.1.

```
#include "config.h"

#include "p4dev.h" // Include high-level contents
#include "firmware/fw.h" // Low-level device implementations
#include "firmware/dummy.h" // Test implementation
#include "p4dev_base.h"
#include "p4dev_msg_logging.h"

#ifndef __KERNEL__
    #include <string.h>
    #include <stdlib.h>
    #include <assert.h>
    #include <stdbool.h>
    #include <math.h>
#endif
```

Listing 5.1: Example of include guards introduced into libp4dev

The most significant changes took place in the error reporting code of the library, consisting previously of only the *p4dev_error.h* file, which contained both declarations of error types, as well as definitions of inline functions for copying and printing names of error codes, as well as a static definition of a table of names for the error codes.

This original design had a few deficiencies. First off, since the table of error code names was defined in a header file as a static array, it meant that a copy of the table was placed into every object file which used the headers of *libp4dev*. Not only was this approach inefficient, it also meant that when the library would be updated, applications wouldn't have access to the names of any new error codes without recompiling.

Additionally, a bigger concern for the kernel-space port were the inline functions, which required access to the compatibility macros provided by *config.h.* This header file is only meant to be included by code files, as including it in a header file would contaminate any application which tried to use is with *libp4dev* project constants, and more disconcertingly, it would contaminate drivers that would use the library with its compatibility macros, which could lead to compilation issues.

Thus, the error reporting header *p4dev_error.h* was split, creating a new code file, *p4dev_error.c,* which contains the table of names of error codes, as well as the functions which were previously defined as inline in the header file. Additionally, a new function, *p4dev_err_str*(), was introduced, which behaves very similar to the standard *strerror*() call in that it returns a pointer to an error code name that is not meant to be freed.

Another notable change was the removal of floating point arithmetic use, since the Linux kernel environment doesn't support floating point math. The library used floating point division, paired with the standard *ceil*() function, to perform integer division with upwards rounding. This was easy enough to replace with a macro that in addition to division would also check the remainder, which if non-zero would increment the result by 1, as is shown in listing 5.2.

```
/*!
 * \brief This macro performs integer division (unsigned 32-bit) with the
 * result being incremented by one if the remainder after division is not zero.
 *
 * \param [in] dividend Number to be divided
 * \param [in] divisor Number to divide by
 */
#define UINT32_DIV_CEIL(dividend, divisor) \
    ((uint32_t)(((dividend) / (divisor)) + ((((dividend) % (divisor)) > 0)? 1 : 0)))


/* Without the macro: */
new_transactions = (uint32_t)ceil((*bit_index + bitwidth)/32.0);

/* With the macro: */
new_transactions = UINT32_DIV_CEIL(*bit_index + bitwidth, 32);
```

Listing 5.2: The UINT32_DIV_CEIL() macro, with example usage

The *libp4dev* library was written in a modular way to allow it to easily work with various different drivers for FPGA cards running architectures produced by the NDK P4 compiler. It has several built-in "firmware backends" that provide the library with the means of communicating via the MI32 bus with the P4 application core. The backend used with the *nfb* kernel drivers is provided by *fw_nfb.c*, support for drivers that provide direct address space mapping is handled by *fw_map.c*, and *fw_generic.c* allows for the use of application-specific callbacks to access the MI32 bus. Which backend should be used is decided upon at compile time with the *--enable-device=<variant>* option of the build system's *configure* script.

In order to make the code easily usable within various different kernel drivers, the generic backend was selected as the backend that would be provided in source archives generated by the *create-kernel-archive.sh* script. Additionally, the script provides a list of bundled code files, and a directory with symbolic links to the header files that should be publicly available, for ease of integration into the source tree of a kernel driver.

Because P4-related code developed by CESNET is not freely distributable, unlike the device driver code, a copy of the *libp4dev* library is not enclosed in the electronic attachment. However, all changes made to *libp4dev* as a part of this thesis are provided in the electronic attachment, in patch form (generated by *git format-patch*), in the *source-code*/*benc-patches*/*p4base*/ directory. This directory contains most changes made to the *p4base* repository, of which *libp4dev* is a part of, found in the *sw*/*libp4dev*/ directory. A copy of the *create-kernel-archive.sh* script is also provided in *source-code*/ in the electronic attachment.

5.1.1 Gluing it all together

The libp4dev library does have a few dependencies, such as the flat device tree library *libfdt*, and the APIs required by firmware backends such as *libnfb* in the case of the *nfb* backend. The *nfb* kernel module contains within its source tree a copy of the *libfdt* library, and it also provides APIs that are very similar to the *libnfb* library's API, which makes incorporating *libp4dev* into the *nfb* kernel module a straight-forward process.

Within the source tree of the *nfb* kernel module, which is located in the *drivers/* directory of the *swbase* repository of the NDK platform, the code is located in the *kernel*/*drivers*/ sub-directory, categorized into further sub-directories such as *fdt* for the

libfdt library, *spi* for the Xillinx SPI controller driver, and *nfb* for the source code files of the *nfb* kernel module itself, which contains further sub-directories for sub-modules.

The natural place where to put *libp4dev* is into the *drivers*/*kernel*/*drivers*/ directory of the *swbase* repository, next to *fdt*. However, it is unwise to directly copy the source code files and add them into the repository, since these files are already version-controlled in the *p4base* repository. A better approach is to either use a *git submodule*, which allows for the nesting of git repositories, or to write a script that clones the *p4base* repository, uses *create-kernel-archive.sh* to create an archive of *libp4dev* kernel source code files and unpacks it into the *drivers*/*kernel*/*drivers*/*libp4dev*/ directory. The *create-kernel-archive.sh* script automatically creates a *.gitignore* file that lists all of the files within the archive, to avoid accidentally adding these files into other repositories.

Two scripts were introduced; *unpack-libp4dev.sh*, which takes an archive created by *create-kernel-archive.sh* and unpacks it into the *nfb* kernel module's source tree, and *retrieve-libp4dev.sh*, which performs the above-described steps to copy the *libp4dev* kernel source files from the *p4base* repository into the *nfb* kernel module's source tree.

In addition to *.gitignore*, a file called *libp4dev_csrc_files* is created by the *create-kernel-archive.sh* script. This file contains a list of all of the C code files (as opposed to header files) provided by the archive. This list is intended to be used by the kernel build system, as is shown in listing 5.3.

```
# Read the list of libp4dev source files from a separate file
libp4dev-csrc-list := $(DRIVER_TOPDIR)/kernel/drivers/libp4dev/libp4dev_csrc_files
libp4dev-csrc := $(shell cat $(libp4dev-csrc-list) 2>/dev/null || true)
$(foreach p4csrc, $(libp4dev-csrc), $(eval libp4dev-csrc-rel += ../libp4dev/$(p4csrc)))
libp4dev-objs := $(libp4dev-csrc-rel:%.c=%.o)

# Add glue code:
libp4dev-objs += ../libp4dev/p4lib_nfb_glue.o

# The library requires C99 to work
libp4dev-cflags := -std=gnu99 -Wno-declaration-after-statement -Wno-strict-prototypes \
                   -UPACKAGE_VERSION

# CFLAGS declarations understood by older kernel versions (e.g. Linux 5.3):
$(foreach p4obj, $(libp4dev-objs), $(eval CFLAGS_$(shell basename $(p4obj)) :=        \
        $(libp4dev-cflags)))

# CFLAGS declarations understood by newer kernel versions (e.g. Linux 5.5):
$(foreach p4obj, $(libp4dev-objs), $(eval CFLAGS_$(p4obj) := $(libp4dev-cflags)))

# Add the libp4dev global mutex (must not have libp4dev CFLAGS applied):
libp4dev-objs += ../libp4dev/p4lib_nfb_glue_mutex.o


nfb-$(CONFIG_NFB_LIBP4DEV) += $(libp4dev-objs)
```

Listing 5.3: Makefile snippet for compiling libp4dev into the nfb kernel module

In order to be able to use *libp4dev* within the *nfb* kernel module effectively, a small amount of glue code is necessary. First off, *libp4dev* is not thread safe, so it's important to provide a global mutex to ensure that no more than one kernel thread uses the library at a time. This mutex is defined in a separate file from the rest of the glue code, since the glue code uses inline functions from *libp4dev* headers and thus needs its CFLAGS, but whereas *libp4dev* is designed with the C99 standard in mind, code within the Linux

kernel is typically compiled with the *-std=gnu89* option of GCC (ANSI C89 with GNU extensions), and this mode is necessary for mutex definitions to work.

The second issue is that *libp4dev* expects to be passed the path to a device node (or possibly other type of directory entry) that corresponds to an FPGA accelerator card. Within the *nfb* kernel module, the *struct nfb* type is used to identify an accelerator card, and a pointer to an instance of this type needs to be passed to the library instead. The approach chosen to achieve this was to implement a token structure that would contain said pointer and a special text string, where a pointer to this embedded text could be passed to the library as a device path, and upon the text being recognized, it would know that it's safe to type-cast the embedded text pointer into a pointer of the token type and gain access to the *struct nfb* pointer for the acceleration card. With this mechanism implemented, it was trivial to adapt the code from the *libp4dev nfb* firmware backend, *fw_nfb.c*, to work within the *nfb* kernel module environment as callback routines to be used with *p4device_init_callbacks*() with the generic firmware backend selected.

5.1.2 Testing it with the p4test driver and user-space tool

While the previous steps have allowed for *libp4dev* to be compiled as a part of the *nfb* kernel module, it is not enough for a codebase to merely compile for it to be ready for use, its functionality needs to be evaluated first, especially when it has never been used in a given environment before.

In order to sufficiently evaluate the *libp4dev* library in kernel-space, a test driver that accepts commands written from user-space into a virtual file in the /*proc* directory was implemented. This driver, called *p4test*, creates a file for each P4 application core in the /*proc*/*p4test*/ directory, e.g. /*proc*/*p4test*/*nfb0-p4c1* for the second core of the first *nfb* accelerator card, where each of these files correspond to a *p4device_t* instance, and the commands written into the individual files cause the appropriate *libp4dev* functions to be invoked on the corresponding *p4device_t* instance.

Specifically, the *reset* command invokes the *p4device_reset*() function, the *enable* and *disable* commands invoke the *p4base_enable*() and *p4base_disable*() functions respectively, and the *table* command adds or removes match+action rules from tables using *p4table_insert_rule*() and *p4table_delete_rule*(), or similar functions for default rule manipulation and addition of rules at specific table indices.

The *table* commands accepts sub-commands, with *add-rule* and *del-rule* accepting a set of matching keys, where *add-rule* uses the keys, along with an action name and parameter list, to construct a *p4rule_t* instance and insert it into the selected table, whereas *del-rule* uses it to identify the rule to be deleted. The special keyword *default*, when used in place of the key list, makes the command refer to the default rule of a table. A simplified *del-rule-index* sub-command that accepts only the index of the rule within the table is also provided. A detailed description of the command syntax, as well as examples of usage, are available in the *p4test* source tree, which is enclosed in the electronic attachment in the *source-code*/*swbase-sriov-netdev-tc-flower*/*drivers*/*kernel*/*drivers*/*nfb*/*p4test*/ directory (see the *README* file in the directory and in the demo sub-directories for details). The *p4test* command interpreter code files are not included in the electronic attachment, since they could be seen as a contribution to the NDK P4 project (discussed later), which is currently not freely distributable.

Reading the contents of the virtual files produces information about the given P4 application core, such as the name of the firmware image, the list of tables, the rule format (keys, actions, parameters) accepted by tables, currently installed rules, and the state of counters.

In addition to the kernel driver, a user-space variant of *p4test* was implemented, which runs as a regular application, and accepts the same format of commands on its standard input, allowing for back-to-back testing of the kernel-space version of *libp4dev* with the user-space version, as is shown on figure 5.1. The user-space version of *p4test* has shown to be generally useful, so it's been integrated into the *p4base* repository, into the *sw*/*p4test*/ directory.

The user-space variant of *p4test* supports an additional command, *status*, which prints the information that the user would acquire from the kernel version by reading from the virtual files in /*proc*/*p4test*/ onto its standard output.

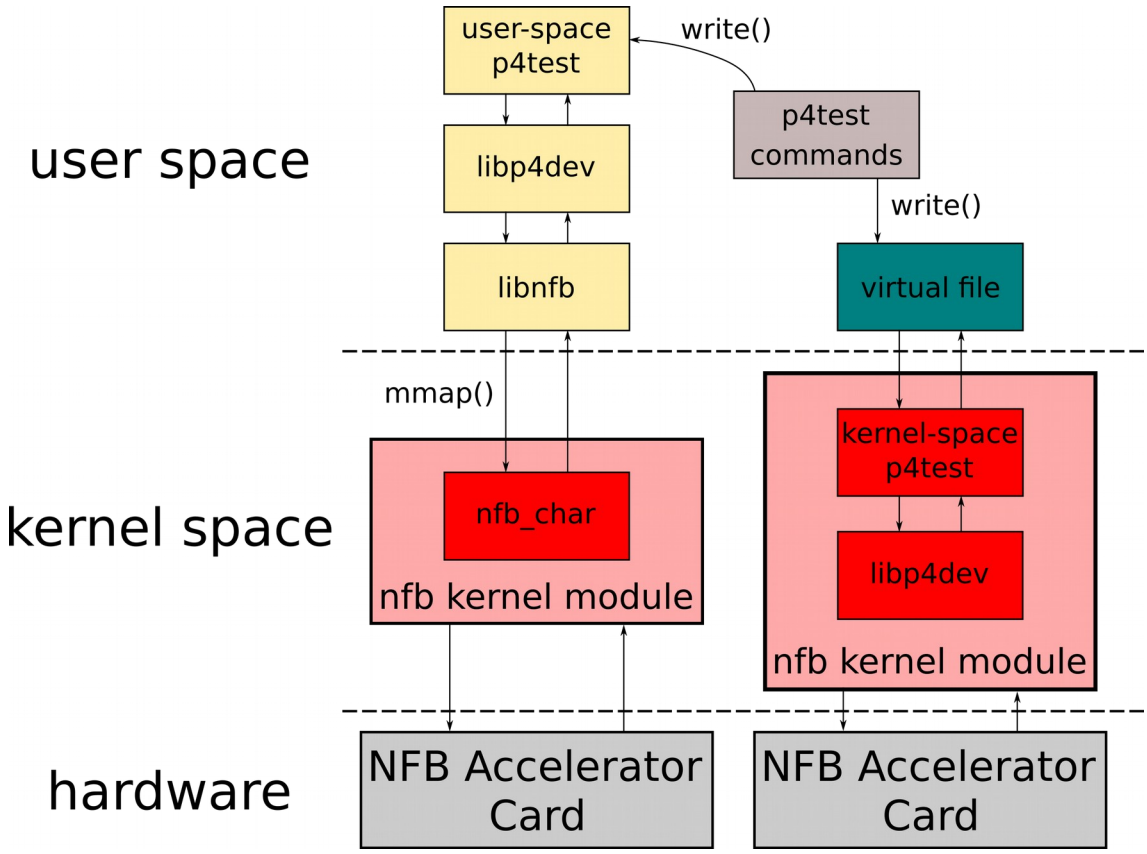

Fig. 5.1: Block diagram of the user-space and kernel-space p4test variants

To aid with the testing, a small python script that uses the *scapy* library to generate a set of packets with pre-defined header field values and stores them in the *.pcap* format was written. This script, *packet_generate.py*, is based on the hypothetical IPv7 packet generation script, but instead of the header configuration and field values being a part of the program text, they are stored in a structure, for ease of creating packets with specific properties without having to modify the code itself. The script supports generating IPv4

and IPv6 packets with up to two VLAN tags, utilizing either the TCP or UDP transport layer.

These *.pcap* packet sets can be sent through an NFB card's P4 pipeline using the *ndp-transmit* tool, and the processed packets can be collected using *ndp-receive*. The results can then be analyzed using either the companion script *packet_examine.py*, or a more sophisticated tool like *wireshark*. Both scripts are in the *p4test* source directory, in *examples*/*general-demo*/ for the user-space variant, and in *general-demo*/ for the kernel-space variant. This directory also contains the P4 program that was mainly used during this phase of testing, and a set of *p4test commands* to populate the tables of an NFB card running firmware generated from said P4 program.

Using *p4test*, several bugs in *libp4dev*, as well as the newly introduced glue code, were discovered and fixed. The electronic attachment contains most of the alterations made to the NDK git repositories as a part of this thesis in *source-code*/*benc-patches/*, with *p4test* code files redacted due to licensing concerns. The most significant bug fixed in *libp4dev* was one which prevented the use of the second P4 application core on NFB cards, caused by incorrect iteration over the card's device tree structure, and incorrect resource locking.

The evaluation process was ultimately a success, the library ended up operating in the same way in both the kernel environment and user-space environment. The act of writing the two *p4test* variants and the example P4 programs and their corresponding p4test command sets was useful in its own right as a learning experience in becoming more familiar with both the P4 language and the NDK tools and P4 compiler.

5.2 Extending the existing ndp_netdev driver

Within the *nfb* kernel module exists the *ndp_netdev* network interface driver. This driver creates a network interface (*netdev*) for each of the card's NDP channels, as is shown on figure 5.2. Packets sent through the egress of an *ndp_netdev* interface are sent into the card through the corresponding NDP channel, and a copy of all packets received from an NDP channel appear on the ingress of the corresponding *ndp_netdev* interface.

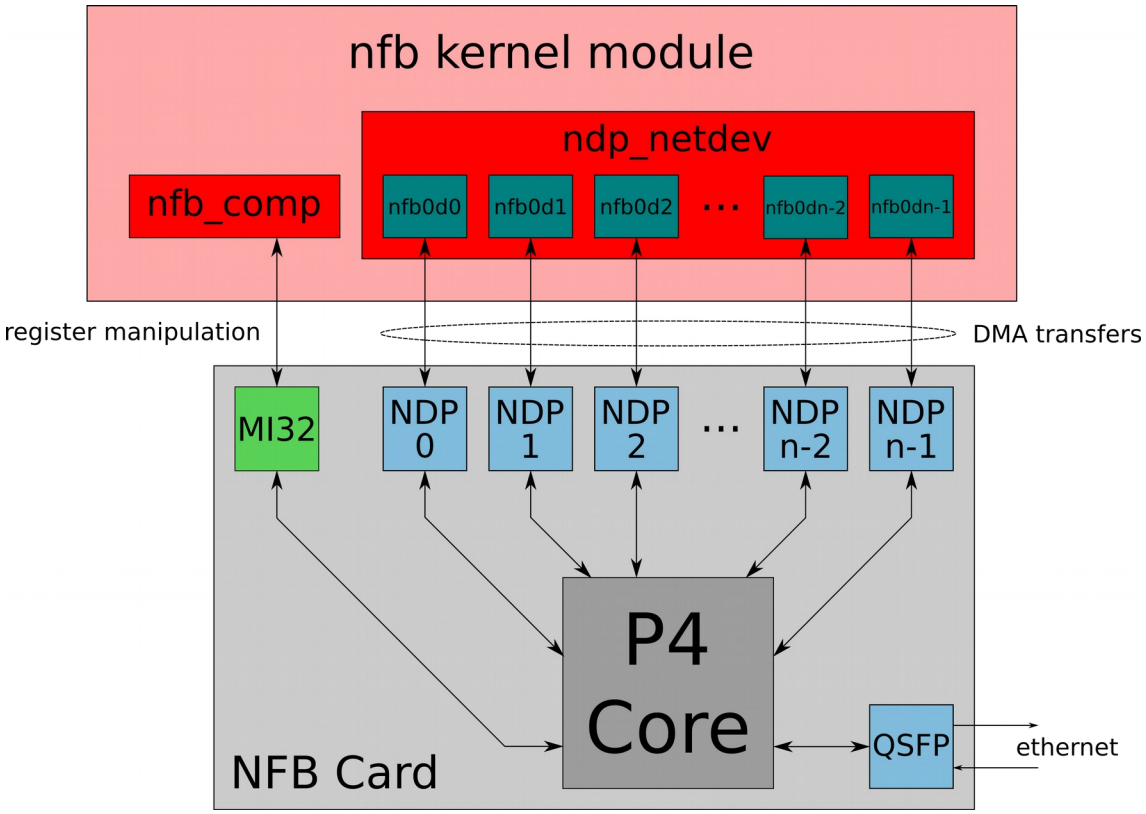

Fig. 5.2: Block diagram describing the ndp_netdev driver

This driver is generic, meaning that it can be used for various different purposes in different applications. We can use the *ndp_netdev* driver as a test bed for TC Flower offloading development by pretending that the individual NDP channels correspond to SR-IOV virtual functions and implementing TC Flower offloading on top of their *netdevs*, which act as *virtual function representers* in this model.

The default behavior of the NDK P4 implementation is to return packets to where they came from, as discussed in chapter 4.4.1. Using this default behavior, we have a test model where the mock VFs always perform packet loopback, e.g. any packets sent through the egress of *nfb0d2* will end up appearing back on its ingress. This behavior can be overridden for specific packets with a match+action rule that changes the *egress_port* in the *intrinsic_metadata*, allowing us to model the redirection of packets into a different mock VF.

In order to provide TC Flower offloading for a *netdev*, the NETIF_F_HW_TC bit needs to be set in its feature mask, and an *ndo_tc_setup*() callback routine that sets up *TC flow blocks*, configuring callbacks to handle the management of hardware offloaded filters attached to either an *ingress* or *clsact qdisc*, needs to be provided. The caller specifies whether to set up a *TC flow block* on the ingress or egress of the *netdev.* In our case, filter offloading on ingress has been implemented, attempting to perform filter offload on egress will result in an "operation not supported" error.

The *enum tc_setup_type* type is used to specify what kind of TC element is to be configured. The callback function that is installed during the *TC fow block* setup in our driver supports the TC_SETUP_CLSFLOWER variant when TC Flower offloading is enabled in the driver and supported by hardware, otherwise it returns an "operation not supported" error. In the case of TC Flower, the function handles the installation and removal of match+action rules from hardware, as well as querying information about how many packets and bytes a given rule in hardware had processed.

The individual offloaded rules are all identified by a unique numeric cookie value, which may be used as a hash table key for fast lookup. Additionally, a priority number and chain number are provided.

The software implementation of TC Flower uses the Linux *flow dissector* to extract the desired fields from packets, which it then compares with the fields of match+action rules, utilizing a hash table search to locate any matching rule and to perform the action associated with it. This means that the matching key data is stored in the *flow dissector* format as well, and it's passed in this format to hardware offload drivers as well.

This key format can be understood as a wide key with optional fields. An analogy in P4 would be a table with a long list of ternary matching keys. It was decided that for the sake of simplicity, the *flow dissector* key would be directly converted into a P4 key for such a table, which is performed by *nfb_tcfl_parse_keys*().

While this approach is simple, it is not ideal from a resource usage point of view, as with the current NDK P4 compiler implementation, it uses expensive TCAM memory, significantly restricting the number of rules that can be offloaded, especially when used in combination with OVS, for which the TC Flower support is implemented as a *dpif provider* (as discussed in chapter 2.2), meaning that it doesn't make good use of TCAM memory. This will eventually be addressed by introducing wide-key optimizations into the P4 compiler, and by using an acceleration card with external memories, but that is outside of the scope of this thesis.

With this approach, a set of supported matching keys had to be decided upon. It is an eventual goal to create a script that would generate a P4 firmware image that would support matching keys based on a description, so that users would be able to specify the kind of fields they want to match on to make better use of limited hardware resources. This script could work by deleting unwanted parsing steps and table keys from the P4 program, reducing the size of the key. Additionally, multiple tables could be specified, each supporting different matching criteria.

With this in mind, it was decided that the default P4 program would have a single table for match+action rule offloading, with its key being as wide as possible, since the key can always be shortened, and the TC Flower offload driver can simply provide a zero mask for any parts of the wide key that weren't specified in the rule. This way,

we could have a universal firmware image that supports all keys, but can't hold many rules. This image would not only be useful for developing the TC Flower offloading code, but also for examining the kinds of rules that get installed by solutions like OVS.

For the *ndp_netdev* TC Flower implementation, the P4 program used for generating such a firmware image is enclosed in *source-code*/*swbase-ndp-netdev-tc-flower*/*drivers*/*tc-flower-offload-p4-prog*/ in the electronic attachment, currently supporting matching on IPv4 and IPv6 packets with up to two VLAN tags and either a TCP, UDP, or SCTP transport layer. The firmware does not support fragmented IPv4 packets, or packets with optional IP header fields (both IPv4 and IPv6) due to limitations of the current NDK P4 compiler implementation.

A useful property of the TCAM search engine implementation used by the NDK P4 compiler is that the priority of rules within tables is well defined, with the rule index specifying the priority – rules with a higher index number have a higher priority. This property can be exploited to implement priority handling for offloaded rules, the driver keeps a priority-ordered linked list of offloaded match+action rules, and a new function was introduced into the *libp4dev* library, *p4table_insert_rule_next_to*(), which allows for the insertion of a rule in-between existing rules, pushing them to the side to free up space for the new rule if necessary, maintaining a desired ordering.

This list is also useful in error recovery. If an inconsistency is either suspected or detected between the information the driver keeps about the offloaded rules and the actual rules offloaded in hardware, e.g. as a consequence of a failed MI32 bus write, the driver temporarily shuts down the P4 pipeline and re-configures the tables, able to traverse the linked list of offloaded rules to quickly re-populate the match+action tables.

For the actions of match+action rules, a conceptual incompatibility exists between TC and P4. While a TC CA rule can have an arbitrary combination of actions, the P4 language allows only a single pre-defined *compound action* per match+action rule. This lead to the implementation of a universal action that performs a variety of optional sub-actions specified by its parameters, and the parameters are generated by interpreting the manner in which a set of TC actions would modify a packet. Analogous to the keys, there could later be a script that would cut this action down into less capable variants according to the needs of the user, with the driver then examining the actions available in hardware and picking the optimal variant for a particular rule.

The universal action consists of a set of *modify_field*() invocations, making use of the mask field to only perform the modifications on demand. In order to add or remove VLAN tags, a metadata bit is set, and the actual addition or removal is done by separate actions that are invoked by the *control flow program* when the corresponding metadata bits are set. In order to achieve this behavior in P4.14, helper tables that have only a default rule and always perform this single rule are provided. These tables can be applied conditionally from within the *flow control program*, and as long as the runtime environment ensures that the default rules are actually configured, the behavior of directly invoking actions is achieved. Within the *ndp_netdev* TC Flower driver, the default rule installation is handled by the *nfb_tcfl_set_default_rules*() function.

The action that required the most effort to implement was *pedit*, as it manipulates raw packet data at given offsets, either from the start of a packet or from headers. The header-based variant was implemented, interpreting the provided modification bitmask data and using it to construct the correct set of parameters for the universal action.

It's important to note that while this TC Flower implementation does support IPv4 header checksum recalculation, which is performed implicitly by the hardware after the *t_ipv4_output_csum_calc* helper table is configured, it does not support transport layer header checksum recalculation. The primary reason is that the NDK P4 compiler does not support the *payload* keyword for specifying that the data beyond the parsed headers should be included in the data set used to calculate a checksum, and while there is an extension that provides a limited alternative that can be used for TCP and UDP, it is non-standard, and for this test driver, the disadvantage of the P4 program not being conformant to the P4.14 standard is greater than the advantage of having the checksum recalculation, since the modified packets are not used by any actual applications for networking, they're merely looped back by the hardware for testing.

To keep track of the amount of bytes and packets that a match+action rule handled, a P4 counter is provided. This counter is bound to the offloading table statically, which means that each rule specifies the counter cell that it should be tracked by. This allows for greater flexibility compared to direct binding, for which the counters track the rules based on their indices, since the indices of existing rules can change as a consequence of new rule insertion. This does, however, mean that the driver itself needs to implement the allocation of counter cell identifiers for rules. Since the identifiers are all of the same size, a simple bitmap-based allocator was implemented. The counter ID bitmap is stored as a part of the *flow chain* information within the driver. Each rule is assigned a counter cell identifier, and it's passed to the card as an action parameter.

The actual testing of the *ndp_netdev* TC Flower driver was performed similarly to the testing of *p4test*, using the *packet_generate.py* script, *ndp-transmit* and *ndp-receive*. NDP channels can have multiple readers and writers (as described in chapter 4.2), so the regular NDP tools can be used as an alternative to the kernel network interface. The kernel network interface itself was tested as well, using *tcpdump* and *Bit-Twist*. In order to install TC Flower rules, the *tc* utility provided by the *iproute2* project can be used, as shown on listing 5.4.

```
# modprobe mdio
# insmod nfb.ko ndp_netdev_enable=yes ndp_netdev_carrier=yes \
                ndp_netdev_tc_flower_offload=yes

# ip link set dev nfb0d5 up
# tc qdisc add dev nfb0d5 ingress

# tc filter add dev nfb0d5         \
          protocol ip parent ffff: \
          flower skip_sw           \
          src_ip 192.168.2.0/24    \
          action drop

# tc filter show dev nfb0d5 ingress
filter protocol ip pref 49152 flower
filter protocol ip pref 49152 flower handle 0x1
  eth_type ipv4
  src_ip 192.168.2.0/24
  skip_sw
  in_hw
        action order 1: gact action drop
         random type none pass val 0
         index 1 ref 1 bind 1

# tc filter delete dev nfb0d5 ingress prio 49152
```

Listing 5.4: Example of using the ndp_netdev TC Flower driver

The code for this driver is available in the *feat-benc-tc-flower-demo* branch of the NDK *swbase* repository, in the *drivers*/*kernel*/*drivers*/*nfb*/*ndp_netdev*/ directory, a copy of the files from this branch is in *source-code*/*swbase-ndp-netdev-tc-flower*/ in the electronic attachment. In order to compile this driver, a Linux kernel of the version 5.3 or newer is necessary, *libp4dev* files need to be added as described in chapter 5.1.1, and TC Flower offloading needs to be enabled in the *drivers*/*Makefile.conf* file. The driver can then be compiled using either the autotools-based or cmake-based build system, and used as shown in listing 5.4.

5.3 The sriov_netdev driver and Open vSwitch

While the above described driver does provide a functional implementation of TC Flower hardware offloading using a COMBO card, it's not particularly useful, since the network interfaces that the rules may be attached to only ever perform packet loopback. The bigger picture in which TC Flower plays a major role is in the context of virtual network switches, such as *Open vSwitch*, for which it provides a means of accelerating their function by allowing for some of their operations to be performed by hardware.

In the context of the COMBO series of cards and the NDK, the most useful way to utilize TC Flower and *Open vSwitch* is with NVF (*network function virtualization*), of which the SR-IOV variant is currently supported. Figure 5.3 shows a conceptual layout of an NFB card being used together with *Open vSwitch* to act as a network switch for virtual machines, with TC Flower making it possible for *Open vSwitch* to alter the paths of packets in the P4 core.

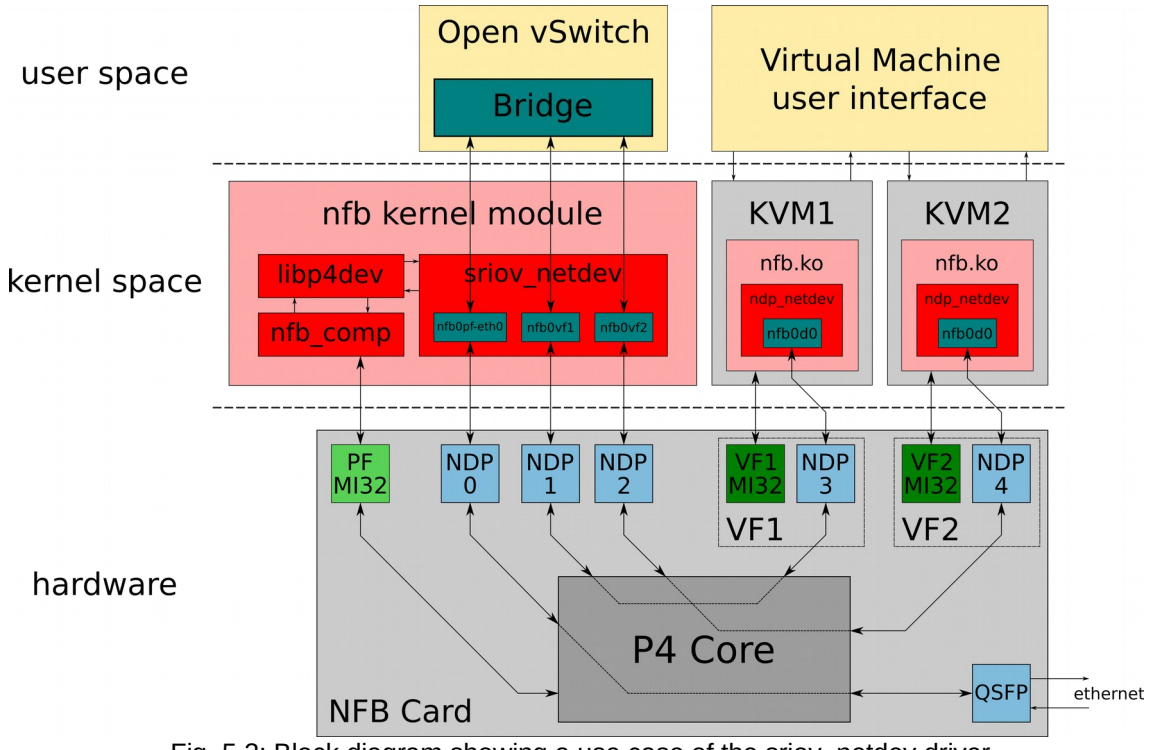

Fig. 5.3: Block diagram showing a use case of the sriov_netdev driver

The layout shown in figure 5.3 was chosen because it would be easy to implement using a modified version of the *ndp_netdev* driver. This modified driver would create network interfaces for communicating with individual *virtual functions* or with *ethernet* ports, configuring the P4 core to facilitate this communication, and allow for the default packet paths (the dotted lines in the P4 core) to be overridden by TC Flower using the *mirred* action (egress redirect mode), as described in chapter 3.4.4.

Originally, this modified driver, *sriov_netdev*, used only a single NDP channel for all packet communication with the card, since with *Open vSwitch*, only the first packets of a flow would reach the driver, and it would leave more NDP channels available for use with virtual functions. While this approach worked, it introduced an extra metadata header that would be difficult to work with in the eventual port to Intel acceleration cards, and it was therefore decided to set aside 2 NDP channels for each *virtual function* and *ethernet* port, and to set up a default mapping between the pairs.

This mapping is handled by the *t_redirect* P4 table, which is set up by the driver on initialization. The driver analyzes the card's device tree, checking how many SR-IOV virtual functions it supports and how many NDP channels it has set aside for them, and uses this information to populate the *t_redirect* P4 table and to create the corresponding *netdevs*. Since the amount of virtual functions that are currently enabled is configurable on runtime using *sysfs*, a notification mechanism was added into the *nfb* kernel module to allows drivers such as *sriov_netdev* to be notified about changes in the current *virtual function* count, which is used by the *sriov_netdev* driver to maintain the correct amount of *netdevs*, creating or destroying them as necessary.

The NDK P4 synthesis tools create two P4 cores on the COMBO-200G2QL cards by default, to help achieve better data throughput when both of its 100Gbps QSFP28 transceivers are used. Since we do not plan on using this card in the long term, due to its high manufacturing costs and lack of external memory support, we've decided not to fully exploit its design, and use only one P4 core for TC Flower offloading. It's possible to modify the NDK P4 synthesis tools to generate only one P4 core for this card to reclaim the resources that the unused second core takes up, but currently, it’s merely set up to drop all of the packets it receives. It can be used, however, together with the second PCIe end-point that the card offers, for testing, acting almost as a second NFB card that can generate a stream of packets to be processed by the first P4 core.

The P4 firmware adapted for the *sriov_netdev* driver additionally supports TCP and UDP checksum recalculation, implemented using non-standard P4 extensions, although it's easy to comment out the non-standard parts and uncomment the corresponding lines that adhere to the P4.14 standard in case the NDK P4 compiler eventually implements the *payload* keyword in *field_list* declarations. This feature is important since the data used for these checksum calculations includes not just the packet payload and the fields from the respective transport layer headers, but also from the IP header, particularly the source and destination IP addresses [43][44][47][48], and not being able to recalculate the level 4 checksum would disallow the hardware from changing IP addresses of packets, an operation that is commonly performed as a part of level 3 packet routing.

The code for this driver is available in the *feat-benc-sriov-ovs-demo* branch of the NDK *swbase* repository, in the *drivers*/*kernel*/*drivers*/*nfb*/sriov*_netdev*/ directory, with the *p4-program* sub-directory containing the firmware P4 description. A copy of the files from this branch is in *source-code*/*swbase-sriov-netdev-tc-flower*/ in the electronic

attachment. The P4 program requires a special version of the NDK P4 compiler, as per the instructions in the *README* file located in the *p4-program* directory.

Once the kernel module is built, the *sriov_netdev* driver can be enabled by passing the *sriov_netdev_enable=yes* argument to the module (e.g. using insmod). Once it starts up and creates the necessary *netdevs*, which can be verified using the *dmesg* and *ip -a* commands, *Open vSwitch* can be set up as per the instructions in [49], substituting the names of the network interfaces in that guide for the names used by *sriov_netdev*.

For the actual virtual machines themselves, they need a copy of the NFB drivers themselves, since the virtual functions appear as limited-use NFB cards, currently with just a single NDP channel for each. This NDP channel can be used either with the *libnfb* library, with tools such as *ndp-receive* and *ndp-transmit*, or the *ndp_netdev* driver can be used to provide a regular system network interface, allowing unmodified applications to use it for networking, as is shown in figure 5.4.

Because several identical VMs may be attached to the virtual switch, and there isn't currently a hardware solution in the NFB cards for passing MAC addresses from the host system to guest systems, which could lead to there being several end nodes with the same MAC address within a single level 2 network, the *ndp_netdev* driver had to be tweaked for use in such scenarios by implementing the *ndo_set_mac_address*() callback function. The implementation is identical to the one present in *sriov_netdev*, and it only sets the MAC address in software, which is sufficient for our purposes. This feature can be seen in use in figure 5.4.

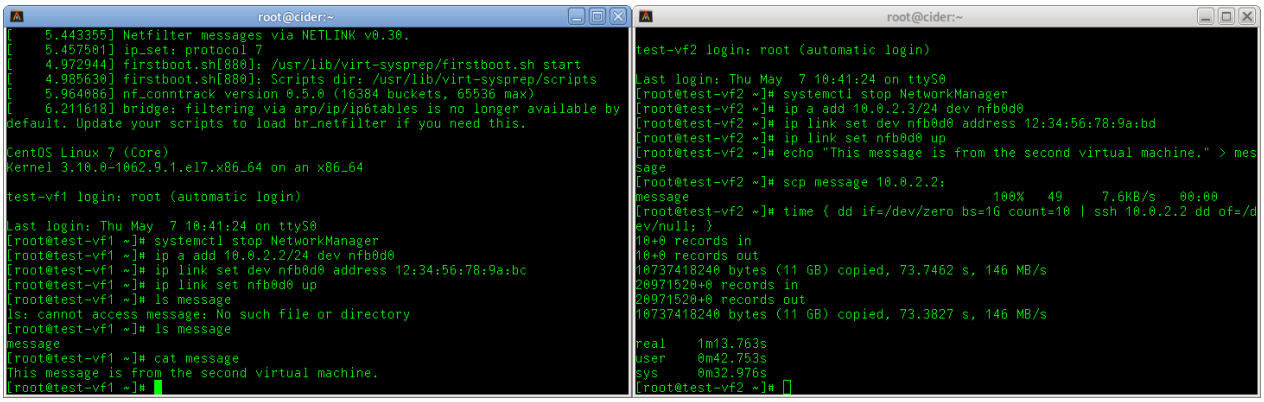

Fig. 5.4: Demonstration of communication between VMs on an NFB virtual switch

In order to automate the deployment process, a script that creates a *libvirt virtual machine* running CentOS with the NFB tools and drivers installed was written. This script, enclosed in the electronic attachment as *demo-sriov-tcfl-vm/vm-create.sh*, accepts a name for the new virtual machine and whether or not to enable *ndp_netdev* by default, and uses the *virt-builder* and *virt-customize* tools to create the virtual machine. A part of the routine is a full system package update, installation of kernel headers, configuring the grub boot loader to optionally enable *ndp_netdev*, enabling automatic root login on the serial console for quicker debugging, and installation of the NDP tools and drivers themselves, in the form of the *netcope-common* package. The *vm-data/* directory next to the script contains the needed helper files, including the modified *netcope-common* package, with source code included (the *netcope-common-6.6.1-1.tar.gz* file).

5.4 Evaluation of the achieved results

The primary purpose of a hardware-accelerated virtual switch is to reduce the amount of CPU time spent on sorting and distributing packets between virtual machines. We were able to achieve this goal by combining *Open vSwitch* with the NDK P4 technology with a custom device driver supporting the TC Flower interface.

During initial testing of the setup, with a Linux 5.2 kernel, there were some issues related to the PCI configuration space with this older kernel version, which resulted in only one of the virtual functions being able to transmit packet data over NDP. This made testing more difficult, but not impossible, since the representer for the QSFP28 transceiver was fully operational, and it allowed for the virtual machine to be connected using the virtual switch to a 100Gbps optical network.

We did not have access to a 100Gbps optical internet connection where the server that this virtual switch was developed on was located (at the CESNET offices in the IT faculty of BUT), but we did have access to another server with the same type of card installed, as well as an optical cable that could be used to connect the two machines. This allowed for one of the machines to act as a gateway to the internet for the other.

Using this configuration, we were able to successfully transfer packet data between the virtual machine and the gateway. Additionally, once the gateway was configured to perform IP forwarding, we were able to connect the virtual machine to the internet, and using the *ovs-ofctl* tool for administering OpenFlow switches, we were able to add rules to block various websites. By observing the kernel logs with *dmesg*, the *netdev* packet counters with *ip*, and the TC Flower rules with *tc*, we were able to see the classification offloading process in action. Only a handful of packets ever reached the kernel network interfaces on the host system, the rest were directly transferred between the gateway and the virtual machine as per the classification rules set by *Open vSwitch*. These rules are maintained by *dpif* (*datapath interface*), and by watching the debugging messages in the kernel log, we could see how *dpif* checks all of the offloaded rules twice a second to see if the individual rules had handled any packets, and when a rule hasn't handled any in 10 seconds, we could see it being being removed to make space for other rules.

The few packets that do end up reaching the kernel network interfaces are either the first packets of a flow, or packets of a protocol that currently isn't supported by the P4 firmware's parser, such as the ARP and ICMP protocols, which we were able to verify using the *tcpdump* utility. These packets are transferred via the “slow path” by the default redirection table. While this typically doesn't present much of an issue, as these particular protocols aren't used for the transfer of bulk data, a similar problem exists for packets that use a different transport layer protocol than TCP, UDP, or SCTP, or a different level 3 protocol such as IPX. In these cases, the P4 firmware and the TC Flower driver would need to be extended to ensure that the packets use the “fast path”.

While the TC Flower driver is easy to expand, the current NDK P4 implementation is rather constrained in what it can do. For example, SCTP checksum recalculation is not implemented, so while SCTP packets can be classified and redirected, they must not be modified in any way, since not only does the SCTP protocol use a CRC32c checksum, an algorithm not supported by the non-standard NetCOPE P4 checksum extensions, but

the SCTP checksum calculation additionally uses the entire packet (with the checksum field zeroed out) as its input data [50], which would result in a lot of separate *field_list* declarations being necessary in order to properly handle packets with VLAN tags, a lot of extra conditional steps would be required in in the *control program*, and the resulting firmware would be sub-optimal. A similar problem exists for implementing support for packets with optional IP header fields, since the NDK P4 implementation does not support variable-length fields, and while it's technically possible to write a parser in P4 that extracts the optional fields into separate header structures, this adds a lot of extra complexity into the P4 program and the corresponding generated firmware.

For this reason, and because of the GPL incompatibility of the *libp4dev* library (which currently prevents the drivers written as a part of this thesis from being deployed, since the Linux kernel is released under the terms of the GNU GPL license, although this issue might get resolved in the future if *libp4dev* becomes open source), it is planned to use a more flexible HLS approach for the firmware of the acceleration card. The architecture of the drivers wouldn't change, but *libp4dev* would need to be replaced with a runtime configuration library that would provide similar features to *libp4dev*, but would instead work with the new HLS firmware.

As for the actual network performance, on the hardware we were testing, we were able to achieve a transfer rate of roughly 1Gbps with *ndp_netdev* running on the virtual machines, as can be seen on figure 5.4. The main limiting factor is related to the CPU overhead of processing the packets in the virtual machines, the *ndp_netdev* driver processes packets going through an NDP channel serially, and if the overhead of packet processing reaches a point where the CPU core running *ndp_netdev* reaches full usage, the driver simply can't keep up. This problem can be mitigated by adding queue support into *ndp_netdev* and dedicating more than just a single NDP channel for VFs. A quick and dirty solution would be to pass several VFs to a single virtual machine, although the software configuration would need to be more complicated to fully make use of the extra *ndp_netdev* interfaces.

The biggest limitation of the current design is by far the small amount of rules that can be offloaded into the acceleration card, 64 rules are currently supported, but in order to be useful in a data center, thousands of rules need to be supported. Usage of external memories is currently the planned way to overcome this limitation.

6 CONCLUSION

This thesis provides an overview of some of the technologies that can be used for the purposes of providing efficient network access to virtual machines where a high data throughput is desired, and where conventional network driver para-virtualization would result in substantial amounts of overhead.

The COMBO line of acceleration cards are useful for a large number of different tasks thanks to their programmable FPGA chips. With support for SR-IOV having been recently introduced to the cards, it makes it possible for them to be used as network switches for virtual machines. The aim of this thesis is to develop a concrete application where a COMBO card fulfills this task.

The *flower* filter of the Linux TC CA subsystem was chosen as the mechanism that would be used for installing match+action rules onto the card, thanks to its similarity to *OpenFlow*, which allows projects like *Open vSwitch* to utilize it to manage the network traffic between virtual machines. Unlike the other filters in the TC CA subsystem, the *flower* filter (TC Flower) was specifically designed with hardware offloading of flow rules in mind, which makes it easy to implement support for it in network accelerators.

In the experimental part of the thesis, an extension of the NFB device drivers was proposed and implemented, using the P4 compiler developed at CESNET as a part of the NetCOPE Development Kit for generating HDL code from a high-level networking description for generating the COMBO card's firmware. This required that the *libp4dev* library would be ported into the Linux kernel environment, where it could be used by the extended driver.

This endeavor ended up being successful, allowing us to directly connect virtual machines to a 100Gbps optical network, with the vast majority of the data transferred between them and the outside network being handled by the card itself, with only a few packets ending up reaching the host operating system, mainly for the purpose of setting up the flow rules in the acceleration card's application core.

However, while the technology developed as a part of this thesis does work, it's not yet in a state where it can be commercially used. First off, the COMBO series of cards is simply way too expensive to be used for this purpose, a cheaper alternative where this technology could be ported to needs to be found, with the Intel PAC-N3000 being the most likely candidate. Additionally, the current firmware is rather inflexible, and scripts need to be introduced to customize it for a particular use case in order to make better use of limited FPGA resources. The underlying technology might also need to be changed, since the NDK P4 technology itself is quite inflexible, both in terms of its capabilities and in terms of licensing, with packet classification based on HLS (*high-level synthesis*) being the most likely replacement.

Over all, this project has been a success. While it didn't result in a commercially viable hardware-accelerated virtual switch being created, it is an important stepping stone on the journey of creating one.

REFERENCES

[1] FISJER-OGDEN, J. *Hardware Support for Efficient Virtualization* [online]. San Diego: University of California, San Diego, 2008, 12 pages. [cit. 2019-10-20] Retrieved from: http://www.cs.ucsd.edu/~jfisherogden/hardwareVirt.pdf

[2] POPEK, J. G.; GOLDBERG, R. P. Formal Requirements for Virtualizable Third Generation Architectures, *Communications of the ACM*. New York: ACM, 1974, vol. 17, 412-421. ISSN: 0001-0782. DOI: 10.1145/361011.361073

[3] ADAMS, K.; AGESEN, O. *A Comparison of Software and Hardware Techniques for x86 Virtualization* [online]. Palo Alto: VMware, 2006, 12 pages. [cit. 2019-10-20] Retrieved from: https://www.vmware.com/pdf/asplos235_adams.pdf

[4] EFRAIM, R.; GERLITZ, O. *Using SR-IOV offloads with Open-vSwitch and similar applications* [online]. Tokyo: The Technical Conference on Linux Networking, 2016, 6 pages. [cit. 2019-10-20] Retrieved from: https://netdevconf.info/1.2/papers/efraim-gerlitz-sriov-ovs-final.pdf

[5] IEEE Std 802.1D-2004. *Media Access Control (MAC) Bridges*. New York: IEEE, 2004, 269 pages.

[6] DORDAL, P. L. *An Introduction to Computer Networks, edition 1.9.19* [online]. Chicago: Loyola University Chicago, 2015. [cit. 2019-11-08] Retrieved from: http://intronetworks.cs.luc.edu/current/html/ethernet.html

[7] THAYUMANAVAN, S. Layer 2 and Layer 3 Switch Evolution, *The Internet Protocol Journal*. San Jose: Cisco Systems, Inc., 1998, vol. 1, nr. 2, 38-43. ISSN: 1944-1134.

[8] Technopedia: *Multilayer Switch Definition* [online]. [cit. 2019-11-08] Retrieved from: https://www.techopedia.com/definition/8465/multilayer-switch

[9] Technopedia: *Layer 4 Switch Definition* [online]. [cit. 2019-11-08] Retrieved from: https://www.techopedia.com/definition/8013/layer-4-switch

[10] BHATTACHARJEE, A. *Managed vs Unmanaged Switch – Which to Choose?* [online]. San Francisco: Udemy, Inc., 2014. [cit. 2019-11-09] Retrieved from: https://blog.udemy.com/managed-vs-unmanaged-switch

[11] Open Networking Foundation: *Software-Defined Networking (SDN) Definition* [online]. Menlo Park: 2019. [cit. 2019-11-09] Retrieved from: https://www.opennetworking.org/sdn-definition

[12] The P4 Language Consortium: *P4Runtime Specification, Version 1.0.0* [online]. 2019, 97 pages. [cit. 2019-11-09] Retrieved from: https://p4.org/p4runtime/spec/v1.0.0/P4Runtime-Spec.pdf

[13] Open Networking Foundation: *OpenFlow Switch Specification, Version 1.5.1 (Protocol version 0x06)* [online]. Menlo Park: 2015, 283 pages. [cit. 2019-11-09] Retrieved from: https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf

[14] Linux Foundation Collaborative Projects: *What Is Open vSwitch*? [online]. San Francisco: Linux Foundation, 2019. [cit. 2019-11-09] Retrieved from: https://docs.openvswitch.org/en/latest/intro/what-is-ovs

[15] Linux Foundation Collaborative Projects: *Porting Open vSwitch to New Software or Hardware* [online]. San Francisco: Linux Foundation, 2019. [cit. 2019-11-09] Retrieved from: http://docs.openvswitch.org/en/latest/topics/porting/

[16] The P4 Language Consortium: *The P4 Language Specification, Version 1.0.5* [online]. 2018, 96 pages. [cit. 2019-11-09] Retrieved from: https://p4.org/p4-spec/p4-14/v1.0.5/tex/p4.pdf

[17] SALIM, J. H. *Linux Traffic Control Classifier-Action Subsystem Architecture* [online]. Ottawa: Mojatatu Networks, 2015, 10 pages. [cit. 2019-12-02] Retrieved from: https://people.netfilter.org/pablo/netdev0.1/papers/Linux-Traffic-Control-Classifier-Action-Subsystem-Architecture.pdf

[18] PÍRKO, J. *Implementing Open vSwitch datapath using TC* [online]. Prague: Red Hat, 2015, 4 pages. [cit. 2019-12-02] Retrieved from: https://people.netfilter.org/pablo/netdev0.1/papers/Implementing-Open-vSwitch-datapath-using-TC.pdf

[19] BROWN, M. A. *Traffic Control HOWTO, Version 1.0.2*. [online]. Chapel Hill: ibiblio, 2006. [cit. 2019-12-09] Retrieved from: http://www.tldp.org/HOWTO/Traffic-Control-HOWTO

[20] BORKMANN, D. *net, sched: add clsact qdisc*. [online]. Boulder, CO: Eklektix, 2016. [cit. 2019-12-09] Retrieved from: https://lwn.net/Articles/671458/

[21] PÍRKO, J. *net: sched: introduce multichain support for filters.* [online]. Boulder, CO: Eklektix, 2017. [cit. 2019-12-09] Retrieved from: https://lwn.net/Articles/723067/

[22] GRAF, T. *Extended Generic Packet Classifier.* [online]. Boulder, CO: Eklektix, 2003. [cit. 2019-12-09] Retrieved from: https://lwn.net/Articles/41615/

[23] WESTPHAL, F.; BIRGER, E; MASHAK, R. *ematch - extended matches for use with "basic" or "flow" filters* [online]. San Francisco: The Linux Kernel Organization, 2019. [cit. 2019-12-09] Retrieved from: https://git.kernel.org/pub/scm/network/iproute2/iproute2.git/tree/man/man8/tc-ematch.8?id=cc08619c3c64c2a2705642cb89af177d6cb0fcfc

[24] SUTTER, P.; SOLTYS, M. *flow - flow based traffic control filter* [online]. San Francisco: The Linux Kernel Organization, 2006. [cit. 2019-12-09] Retrieved from: https://git.kernel.org/pub/scm/network/iproute2/iproute2.git/tree/man/man8/tc-flow.8?id=bdd6104f52ad10c67d37bf4c6c36efa137d5d0ec

[25] KUZNETSOV, A. N.; DUMAZET, E; HUBERT, B. *sfq - Stochastic Fairness Queueing* [online]. San Francisco: The Linux Kernel Organization, 2015. [cit. 2019-12-10] Retrieved from: https://git.kernel.org/pub/scm/network/iproute2/iproute2.git/tree/man/man8/tc-sfq.8?id=5699275b424c637edc60f46c89071d4e8f4edde3

[26] SUTTER, P.; MASHAK, R. *u32 - universal 32bit traffic control filter* [online]. San Francisco: The Linux Kernel Organization, 2019. [cit. 2019-12-10] Retrieved from: https://git.kernel.org/pub/scm/network/iproute2/iproute2.git/tree/man/man8/tc-u32.8?id=9ab56784a25ecc3763f48450aea98aca242da36f

[27] BORKMANN, D.; KICINSKI, J.; MASHAK, R. *BPF - BPF programmable classifier and actions for ingress/egress queueing disciplines* [online]. San Francisco: The Linux Kernel Organization, 2019. [cit. 2019-12-10] Retrieved from: https://git.kernel.org/pub/scm/network/iproute2/iproute2.git/tree/man/man8/tc-bpf.8?id=4d9e90f36b3f92644ed5c915758b403953727e40

[28] HORMAN, S. *TC Flower Offload* [online]. Seoul: The Technical Conference on Linux Networking, 2017, 3 pages. [cit. 2019-12-10] Retrieved from: https://netdevconf.info/2.2/papers/horman-tcflower-talk.pdf

[29] BATES, L.; SALIM., J. H.; BUSLOV, V. *Actions in TC* [online]. San Francisco: The Linux Kernel Organization, 2019. [cit. 2020-03-08] Retrieved from: https://git.kernel.org/pub/scm/network/iproute2/iproute2.git/tree/man/man8/tc-actions.8?id=fb2e033add073893dea71bb483353790fe8c5354

[30] SUTTER, P.; VADAI, A.; GERLITZ, O.; PIRKO, J. *pedit - generic packet editor action* [online]. San Francisco: The Linux Kernel Organization, 2017. [cit. 2020-03-08] Retrieved from: https://git.kernel.org/pub/scm/network/iproute2/iproute2.git/tree/man/man8/tc-pedit.8?id=d19f72f7898a78ef76628833c204afb96f9a05cd

[31] SUTTER, P.; CARATTI, D.; HEMMINGER, S.; CROCE, M. *csum - checksum update action* [online]. San Francisco: The Linux Kernel Organization, 2017. [cit. 2020-03-08] Retrieved from: https://git.kernel.org/pub/scm/network/iproute2/iproute2.git/tree/man/man8/tc-csum.8?id=6bf156415a588fa1c975be9a18a1579f63a936a2

[32] SUTTER, P.; LADKANI, S. *mirred - mirror/redirect action* [online]. San Francisco: The Linux Kernel Organization, 2016. [cit. 2020-03-08] Retrieved from: https://git.kernel.org/pub/scm/network/iproute2/iproute2.git/tree/man/man8/tc-mirred.8?id=5eca0a3701223619a513c7209f7d9335ca1b4cfa

[33] SUTTER, P.; ZION., H. H.; LADKANI, S; PIRKO, J. *vlan - vlan manipulation module* [online]. San Francisco: The Linux Kernel Organization, 2018. [cit. 2020-03-08] Retrieved from: https://git.kernel.org/pub/scm/network/iproute2/iproute2.git/tree/man/man8/tc-vlan.8?id=8ee38d833ccb1863f06634e12c5236b0ef7c2d76

[34] SUTTER, P.; SALIM, J. H.; CARATTI, D.; MASHAK, R. *skbedit - SKB editing action* [online]. San Francisco: The Linux Kernel Organization, 2019. [cit. 2020-03-08] Retrieved from: https://git.kernel.org/pub/scm/network/iproute2/iproute2.git/tree/man/man8/tc-skbedit.8?id=26a49de4dbb6b73e251c9256460ba6b0a8ac5201

[35] CESNET: *liberouter: NetCOPE* [online]. [cit. 2019-11-21] Retrieved from: https://www.liberouter.org/technologies/netcope/

[36] CESNET: *liberouter: Partners* [online]. [cit. 2019-11-21] Retrieved from: https://www.liberouter.org/partners/

[37] Netcope Technologies: *Netcope Development Kit* [online]. [cit. 2019-11-21] Retrieved from: https://www.netcope.com/en/products/fpga-development-kit

[38] CESNET: *NetCOPE project wiki* [online]. [cit. 2020-05-24] Retrieved from: https://redmine.liberouter.org/projects/tmc/wiki/Fwbase

[39] CESNET: *liberouter: Cards* [online]. [cit. 2019-11-21] Retrieved from: https://www.liberouter.org/technologies/cards/

[40] CESNET: *NetCOPE swbase (netcope-common) repository* [online]. [cit. 2020-05-24] Retrieved from: https://gitlab.liberouter.org/ndk/swbase

[41] CESNET: *NetCOPE P4 base repository* [online]. [cit. 2020-05-24] Retrieved from: https://gitlab.liberouter.org/p4/p4base

[42] BENÁČEK, P. *Generation of High-Speed Network Device from High-Level Description.* Dissertation thesis, Faculty of Information Technology, Czech Technical University in Prague, 2016.

[43] USC Information Sciences Institute: *RFC: 793 Transmission Control Protocol* [online]. IETF, 1981. [cit. 2020-05-07] Retrieved from: https://tools.ietf.org/html/rfc793

[44] POSTEL, J. *RFC: 768 User Datagram Protocol* [online]. IETF, 1980. [cit. 2020-05-07] Retrieved from: https://tools.ietf.org/html/rfc768

[45] CESNET: *NetCOPE P4Runtime repository* [online]. [cit. 2020-05-24] Retrieved from: https://gitlab.liberouter.org/p4/p4-runtime

[46] bootlin: *Elixir Cross Referencer – Linux source code (v5.3)* [online]. [cit. 2020-05-10] Available at: https://elixir.bootlin.com/linux/v5.3/source/

[47] USC Information Sciences Institute: *RFC: 791 Internet Protocol* [online]. IETF, 1981. [cit. 2020-05-07] Retrieved from: https://tools.ietf.org/html/rfc791

[48] DEERING, S.; HINDEN, R. *RFC: 8200 Internet Protocol, Version 6 (IPv6) Specification* [online]. IETF, 2017. [cit. 2020-05-07] Retrieved from: https://tools.ietf.org/html/rfc8200

[49] MINTZ, Y.; SCHIMMEL, I. *OVS* [online]. Yokneam Illit: Mellanox Technologies, 2019. [cit. 2020-05-07] Retrieved from: https://github.com/Mellanox/mlxsw/wiki/OVS

[50] STEWART, E. R. *RFC: 4960 Stream Control Transmission Protocol* [online]. IETF, 2007. [cit. 2020-05-07]. Retrieved from: https://tools.ietf.org/html/rfc4960

LIST OF SYMBOLS AND ABBREVIATIONS

API	Application Programming Interface
ARP	Address Resolution Protocol
BPF	Berkeley Packet Filter
BST	Binary Search Tree
CA	Classifier-Action
cBPF	Classic Berkeley Packet Filter
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DMA	Direct Memory Access
dpif	Datapath Interface
DSCP	Differentiated Services Code Point
eBPF	Extended Berkeley Packet Filter
ECN	Explicit Congestion Notification
FIFO	First in, First out
FPGA	Field-programmable gate array
Gpbs	Gigabits per second
GCC	GNU Compiler Collection / GNU C Compiler
GNU	GNU's Not Unix
GPL	GNU General Public License
HDL	Hardware Description Language
IEEE	Institute of Electrical and Electronics Engineers
IP	Internet Protocol, or Intellectual Property
IPv4	Internet Protocol, Version 4
IPv6	Internet Protocol, Version 6
JIT	Just-in-time (compiler)
KVM	Kernel virtual machine
LAN	Local Area Network
LLVM	Low Level Virtual Machine (a compiler back-end project)
LPM	Longest Prefix Match
MAC	Medium Access Control
MI32	Universal 32-bit Memory Interface

NDK	NetCOPE Development Kit
NDP	NetCOPE Data Plane
NFB	NetCOPE FPGA Board
NIC	Network Interface Card
NVF	Network Function Virtualization
OS	Operating System
OSI	Open Systems Interconnection
OSPF	Open Shortest Path First
PCS	Physical Coding Sublayer
PF	Physical function
PI	Program Independent
PMA	Physical Medium Attachment
Qdisc	Queuing discipline
QoS	Qality of Service
QSFP	Quad Small Form-factor Pluggable
RAM	Random Access Memory
RFC	Request for Comments
RIP	Routing Information Protocol
RX	Receive
SCTP	Stream Control Transmission Protocol
SDN	Software-Defined Networking
SFQ	Stochastic Fairness Queuing
SNMP	Simple Network Management Protocol
SPI	Serial Peripheral Interface
SR-IOV	Single-root input/output virtualization
TC	Traffic Control
TCAM	Ternary content-addressable memory
TTL	Time to live
TX	Transmit
VF	Virtual Function
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VLAN	Virtual Local Area Network
VM	Virtual Machine
VMM	Virtual Machine Monitor

TABLE OF FIGURES

TABLE OF CODE LISTINGS