WARSAW UNIVERSITY OF TECHNOLOGY

DISCIPLINE OF SCIENCE INFORMATION AND COMMUNICATION TECHNOLOGY
FIELD OF SCIENCE ENGINEERING AND TECHNOLOGY

Ph.D. Thesis

Tomasz Osiński, M.Sc.

Data plane programmability for software datapaths in a virtualized network infrastructure

Supervisor
Halina Tarasiuk, D.Sc., Ph.D.

WARSAW 2023

Data plane programmability for software datapaths in a virtualized network infrastructure

ABSTRACT

The advent of Software-Defined Networking radically changed the approach to building modern network systems such as 5G/6G by enabling programmability in the network. Recent innovations around data plane programmability allow to express packet processing algorithms using high-level programming abstractions. Nevertheless, the data plane programmability has mainly been leveraged for hardware switches and NICs so far, with limited applications to software switches. At the same time, software switches have become a vital component of modern Network Function Virtualization (NFV) systems, playing a role of the primary provider of network connectivity and advanced network services. However, novel applications like 5G put new demands on software switches. Therefore network owners may want to introduce new data plane techniques in a timely manner.

This dissertation aims at applying a data plane programmability to software switches running in the virtualized network infrastructure. We present two novel systems implementing data plane programmability for software switches, namely P4rt-OVS and NIKSS. These systems enable expressing how packets should be processed in the software switch using P4, a high-level language for data plane programming. Therefore, both P4rt-OVS and NIKSS allow to quickly prototype new network protocols, implement novel packet processing algorithms or customize existing packet processing pipelines without an intimate knowledge about the internals of a packet processing engine on general-purpose CPUs. Moreover, when integrated into an end-to-end programmable network, P4rt-OVS and NIKSS enable novel network applications that would not have been feasible or would have been difficult to implement in a timely manner. This dissertation also presents several novel use cases taking advantage of P4rt-OVS and NIKSS in the context of end-to-end programmable networks.

P4rt-OVS derives from Open vSwitch, a fixed-function SDN software switch commonly used for multi-tenant network virtualization. P4rt-OVS is an original extension to Open vSwitch that enables runtime programming of protocol-independent and stateful packet pro-

cessing pipelines. The P4rt-OVS design results in a hybrid approach that provides data plane programmability without sacrificing well-known features of Open vSwitch. Performance evaluations show that P4rt-OVS does not introduce significant packet processing overhead compared to the classic OVS forwarding model, yet enables runtime protocol extensions and stateful processing.

NIKSS is a novel P4-programmable software datapath for Software-Defined Networking that has been designed around the following principles: high-level and feature-rich programming abstraction, performance, runtime programmability and operability. NIKSS leverages P4 Portable Switch Architecture as feature-rich programming abstraction and eBPF as a packet processing engine to provide runtime programmability and operability. High performance has been achieved by combining eBPF with the PSA to eBPF compiler, an original extension to the open-source P4 compiler that generates efficient packet processing pipeline for NIKSS. As part of NIKSS we proposed a unique ternary classiciation algorithm based on eBPF primitives and number of performance optimizations, including table caching and static pipeline-aware optimizations. We demonstrate that NIKSS can be successfully used as a software switch for modern NFV systems. We also provide an extensive performance evaluation, proving that NIKSS might be a viable alternative to existing software switches.

Both P4rt-OVS and NIKSS have been released as open-source projects and are available for researchers and system developers to build upon.

Key words: *Software-Defined Networking, programmable data plane, P4, eBPF*

Programowalna warstwa przekazu danych dla funkcji sieciowych w środowisku zwirtualizowanej infrastruktury telekomunikacyjnej

STRESZCZENIE

Technika sieci programowalnych (ang. Software-Defined Networking - SDN) radykalnie zmieniła podejście do budowania nowoczesnych systemów telekomunikacyjnych takich jak sieci 5G i 6G. Jednocześnie ostatnie innowacje w obszarze programowalnej warstwy przekazu danych dla sieci SDN pozwalają na implementację algorytmów przetwarzania pakietów w urządzeniach sieciowych wykorzystując wysokopoziomowe abstrakcje programistyczne takie jak język P4. Jednakże techniki programowalnej warstwy przekazu danych są obecnie wykorzystywane głównie dla sprzętowych urządzeń sieciowych takich jak programowalne switche czy karty sieciowe, natomiast zastosowania dla funkcji sieciowych w środowisku zwirtualizowanej infrastruktury sieciowej są ograniczone. Jednocześnie switche programowe zrealizowane jako moduły oprogramowania (ang. software switches) stały się niezbędnym komponentem nowoczesnym systemów wirtualizacji funkcji sieciowych NFV (ang. Network Function Virtualization) umożliwiając komunikację sieciową oraz implementację zaawansowanych usług sieciowych w centrach danych. Mimo to, nowe rozwiązania takie jak sieci 5G narzucają nowe wymagania (np. nowe protokoły sieciowe) na funkcje sieciowe w środowisku zwirtualizowanej infrastruktury sieciowej, a operatorzy sieci chcą mieć możliwość wdrażania nowych technik (m.in. obsługi protokołów sieciowych) w warstwie przekazu danych w krótkim czasie.

Celem niniejszej rozprawy jest zastosowanie techniki programowalnej warstwy przekazu danych do switchy programowych, tak aby ułatwić implementację oraz skrócić czas wdrażania nowych technik sieciowych w środowisku zwirtualizowanej infrastruktury telekomunikacyjnej. W niniejszej rozprawie zaprezentowane zostały dwa oryginalne rozwiązania: P4rt-OVS oraz NIKSS. Oba rozwiązania pozwalają definiować sposób przetwarzania pakietów w sieci wykorzystując technikę P4, wysokopoziomowy język programowania dla warstwy przekazu danych. Dlatego też systemy P4rt-OVS oraz NIKSS pozwalają w krótkim czasie i bez specjalistycznej wiedzy o mechanizmach przetwarzania pakietów tworzyć prototypy protokołów sieciowych, wdrażać nowatorskie algorytmy przetwarzania pakietów, czy też dostosowywać

obecnie działające mechanizmy przetwarzania pakietów. Ponadto, rozwiązania P4rt-OVS oraz NIKSS zintegrowane z siecią programowalną umożliwiają implementację nowatorskich usług sieciowych, które nie byłyby inaczej możliwe do zaimplementowania lub wymagałyby znaczących nakładów. W niniejszej rozprawie zaprezentowano również oryginalne zastosowania wykorzystujące systemy P4rt-OVS oraz NIKSS w kontekście sieci programowalnych od końca do końca.

Rozwiązanie P4rt-OVS jest oryginalnym rozszerzeniem systemu Open vSwitch, który jest wykorzystywany w nowoczesnych systemach wirtualizacji sieci. Architektura P4rt-OVS pozwala wykorzystywać dotychczasowe możliwości platformy Open vSwitch, a jednocześnie rozszerza ją o programowalną warstwę przekazu danych. Badania wydajnościowe pokazały, że P4rt-OVS nie wprowadza znaczących narzutów wydajnościowych w porównaniu do obecnego modelu przetwarzania pakietów na platformie Open vSwitch, a jednocześnie dostarcza możliwość rozszerzania tej platformy w czasie działania o dowolne protokoły sieciowe, czy też stanowe algorytmy przetwarzania pakietów.

Rozwiązanie NIKSS jest oryginalnym switchem zrealizowanym jako moduł oprogramowania dla sieci SDN. NIKSS cechuje się wysokopoziomowym i w pełni funkcjonalnym językiem programowania, wysoką wydajnością, możliwością programowania w czasie działania oraz łatwością w zarządzaniu. Rozwiązanie NIKSS wykorzystuje architekturę PSA (ang. Portable Switch Architecture) jako abstrakcyjny model przetwarzania pakietów oraz technikę eBPF jako platformę przetwarzania pakietów zapewniającą programowalność w czasie działania oraz łatwość w zarządzaniu. Wysoka wydajność przetwarzania pakietów na sekundę została osiągnięta poprzez połączenie możliwości techniki eBPF oraz kompilatora PSA-eBPF, autorskiego rozszerzenia do kompilatora języka P4. Jednym z osiągnięć w ramach pracy nad rozwiązaniem NIKSS jest realizacja algorytmu klasyfikacji pakietów typu "ternary" dla platformy eBPF oraz optymalizacje wydajnościowe. W niniejszej rozprawie udowodniono, że rozwiązanie NIKSS może być z powodzeniem używane w nowoczesnych systemach NFV. Ponadto, badania wydajnościowe pokazały, że NIKSS jest realną alternatywą dla obecnie istniejących rozwiązań.

Rozwiązania P4rt-OVS oraz NIKSS zostały udostępnione publicznie jako otwarte oprogramowanie i mogą być wykorzystywane do prowadzenia badań lub tworzenia systemów telekomunikacyjnych.

Słowa kluczowe: *sieci programowalne, programowalna warstwa przekazu danych, P4, eBPF*

Contents

1 Introduction **9**

2 Background **19**

2.1 Software-based packet processing 19

2.1.1 Packet processing in the multi-core CPU environment 21

2.1.2 Packet processing frameworks in software 26

2.1.3 Packet classification algorithms 29

2.1.4 Performance metrics for software-based packet processing 32

2.2 Abstractions for programmable data planes 34

2.2.1 Data plane programmability and P4 36

2.2.2 Programming end-host network stack with eBPF 46

2.3 Conclusions 50

3 Need for a programmable SDN software switch **52**

3.1 Use cases 57

3.2 Requirements for programmable software switches 59

3.3 Existing solutions 60

3.4 Conclusions 62

4 P4rt-OVS: Programming Protocol-Independent, Runtime Extensions for Open vSwitch with P4 **64**

4.1 Motivation 65

4.2 Design and implementation 66

4.2.1 Open vSwitch design 66

4.2.2 P4rt-OVS overview 67

4.2.3 Modifications to OVS 69

4.2.4 P4 to uBPF compiler . . . 71
4.2.5 P4Runtime-based control plane . . . 73
4.3 Performance evaluation . . . 74
4.3.1 Evaluation environment . . . 74
4.3.2 End-to-end performance . . . 75
4.3.3 Microbenchmarks . . . 77
4.3.4 Evaluation of an exemplary network function . . . 82
4.4 Conclusions . . . 85

5 NIKSS: A novel programmable software datapath for Software-Defined Networking 87
5.1 Design . . . 89
5.1.1 General-purpose design . . . 90
5.1.2 Specialized XDP-based design . . . 92
5.2 PSA-eBPF compiler . . . 93
5.2.1 P4 pipeline to eBPF translation . . . 94
5.2.2 XDP helper program . . . 96
5.2.3 Packet Replication Engine . . . 96
5.2.4 PSA externs . . . 98
5.2.5 Ternary matching algorithm . . . 101
5.2.6 Compiler optimizations . . . 102
5.3 Performance evaluation . . . 104
5.3.1 Environment setup . . . 104
5.3.2 Packet forwarding rate . . . 105
5.3.3 Microbenchmarks . . . 109
5.3.4 Comparison with software datapaths . . . 111
5.4 Limitations . . . 113
5.5 Conclusions . . . 114

6 Summary 116
6.1 Research outcomes . . . 120
6.2 Future work . . . 122

A Methodology to measure per-component CPU cycles for NIKSS 124

B Online DDoS mitigator with P4rt-OVS **127**

Chapter 1

Introduction

Over the last decade, the approach to building the computer and telecommunication networks has been radically changed. In the era of next-generation mobile networks (5G/6G) [8, 89], we can observe an evolution towards the Network Functions Virtualization (NFV) paradigm [120, 192], from network systems composed of specialized hardware appliances towards open, programmable and fully-softwarized network infrastructure built on top of a high-bandwidth optical transport layer and a common compute and network infrastructure composed of physical servers and network switches co-located in geographically distributed data centers, often referred to as *clouds*. The NFV approach assumes that network functions, which has so far been offered as hardware appliances, are now implemented as software modules (Virtual Network Functions - VNFs) running in a virtual execution environment (e.g., virtual machines or containers), which are connected to software switches running on compute servers. Software datapaths (also referred to as software switches or virtual switches and we use these terms interchangeably) are a key component of modern virtualized network infrastructure [97]. They play a role of a hypervisor switch forwarding packets to and from Virtual Machines (VMs) or containers, and enable multi-tenant network virtualization through logical (virtual) overlay networks.

This evolution would not be possible without Software-Defined Networking (SDN) [98, 185] - a concept of decoupling the control plane from the forwarding devices' data plane. The data plane is the plane where network packets are processed and forwarded between ports (or dropped) based on forwarding or routing tables, while the control plane implements the network-level intelligence, i.e. decides an end-to-end packet path

throughout a network and determines an output port within a device (router, switch) to reach a next hop in the network. SDN moves the control plane from distributed network nodes to a logically centralized module (called SDN controller), making the network devices' functionality limited just to data forwarding. This gives the SDN controller a global, abstract network view enabling a paradigm shift and enhanced network programmability - the network system can be now defined and implemented as a software application. Thus, in the next-generation network infrastructure, the SDN technology is used to program hardware and software forwarding devices and dynamically create underlay and overlay networks to provide connectivity between Virtual Network Functions (VNFs) and, also, between VNFs and external wide area or enterprise networks.

However, the control and data planes differ in operative requirements. The control plane does not need to operate at high packet processing speeds, but it needs to perform relatively complex computations, e.g., perform path computation for routing protocols such as BGP (Border Gateway Protocol) or OSPF (Open Shortest Path First). Consequently, control plane implementations have favored general purpose CPUs (Central Processing Units) and high-level languages such as Java, Python or Go. In contrast, the data plane should process data units as fast as possible. Thus, it has been commonly implemented using specialized, low-level programming languages based on dedicated hardware, such as ASIC (Application-Specific Integrated Circuit), NPU (Network Processing Unit) or FPGA (Field-Programmable Gate Array). Recently, due to the development of more powerful modern CPUs and more efficient packet processing techniques, such as Data Plane Development Kit (DPDK) [83], AF_XDP [177] or XDP (eXpress Data Path) [75], data planes are written in C/C++ and re-architected to be run on general-purpose CPUs as well.

In general, network programmability refers to the ability to define entire network systems as a set of software applications implementing data and control plane mechanisms. Moreover, it enables changing the networks' design and implementation over time by only modifying the software applications, without purchasing a new hardware. The diverse requirements for control and data planes affect how network programmability is enabled. Initially, SDN only enabled programmability of the control plane. In fact, SDN abstracts data plane devices by means of a programming API (Application

Programming Interface) that provides an interface for SDN controllers to communicate with forwarding devices and instruct them how to process packets. OpenFlow [110] has been a prominent instance of such an API to forwarding devices, as it provides a well-defined API to program a device and abstract Match-Action forwarding model enabling flow-based packet processing. Nonetheless, the control plane programmability is only a half of the path towards fully programmable networks. Indeed, the OpenFlow technology introduced a programming API to forwarding devices and made switch vendors open their products and offer "white-box" hardware switches that can be controlled by the third-party software. It is worth noticing that the SDN technology is not only applied to hardware switches. At the same time, the OpenFlow-compliant software switches were also brought to the market, with Open vSwitch (OVS) [148] as a prominent solution used in production [51, 167]. However, OpenFlow assumes that switches have a fixed and pre-defined set of supported data plane protocols and actions to be performed on packets. This assumption significantly limits applications, because the control plane logic is limited to what functionality a given OpenFlow device offers. This leads to long time to market for new features and slower release and innovation cycles. Any new feature at the data plane level (e.g., new tunneling protocol or header extension) would need to be first defined in the OpenFlow specification and, then, designed and implemented by silicon switching vendors, before it could be used in production. A similar problem exists in case of software datapaths. Developing a new network feature requires domain-specific knowledge of network protocol's design, low-level programming skills and familiarity with the large and complex codebase of the software switch or TCP/IP stack implementation in the OS kernel. Next, the feature must be reviewed, accepted by the community and distributed as a software package.

Data plane programmability and the P4 technology [23] have been proposed to address these challenges and unleash full network programmability. P4 is a high-level language for describing how packets should be processed in the network data plane in the form of a P4 program. P4 was designed around three principles [23]:

1. **Reconfigurability in the field.** Programmers should be able to change data plane algorithms that switches use to process packets once they are deployed.

2. **Protocol independence.** Switches should not be tied to any specific network protocol and a P4 programmer should be able to implement any data plane protocol.

3. **Target independence.** Programmers should be able to describe packet processing functionality independently of the specifics of the underlying platform.

The P4 language, along with re-programmable platforms (e.g., Tofino ASIC, NPUs) and SDN controllers, allows network owners to take a full control over their networks - both data plane and control plane can be described as a software program, enabling custom use cases and fostering network innovation. Importantly, P4 works in conjunction with P4Runtime [174], a control plane interface to forwarding devices that auto-generates control plane APIs for an SDN controller based on a P4 program. Thus, a P4 program also plays a role of a "contract" between data and control planes. Over recent years, programmable network devices have emerged as a promising alternative to traditional switches and Network Interface Cards (NIC) that provides flexibility to defining packet processing pipelines [72, 106]. Programmable data planes and P4 allow network operators to define custom protocols and packet processing algorithms and have become widely used by network operators, software developers and researchers [72]. BlueBird [12] and SD-Fabric [124] are examples of benefits that programmable devices and P4 bring to the networking area. Both solutions customize the standard data center fabric network to implement network virtualization system for the bare-metal cloud service (BlueBird) or to enhance the underlay network with 5G functionality and in-depth network visibility (SD-Fabric). Moreover, P4 brings the added value to many areas of network systems including network monitoring (e.g, flow monitoring, heavy hitter detection, path tracking), traffic management and congestion control, routing and forwarding (e.g., source routing, data plane resilience), network security (firewalls, DDoS mitigation) or advanced networking techniques such as Time-Sensitive Networking or Internet of Things [72, 106].

However, P4 has so far been used for programmable ASICs and SmartNICs and has not yet been successfully adopted to software datapaths. Even though a few P4-programmable software switches have been developed [160, 136, 138, 156], they are not performant or fully-featured enough to implement all data plane functions of a hypervisor switch or Virtual Network Function. Having said that, there is no doubt that with the emergence of NFV, 5G, network slicing and a general *network softwarization* trend, software datapaths have become a vital component of the network infrastructure, playing more and more important role in the network transformation. To

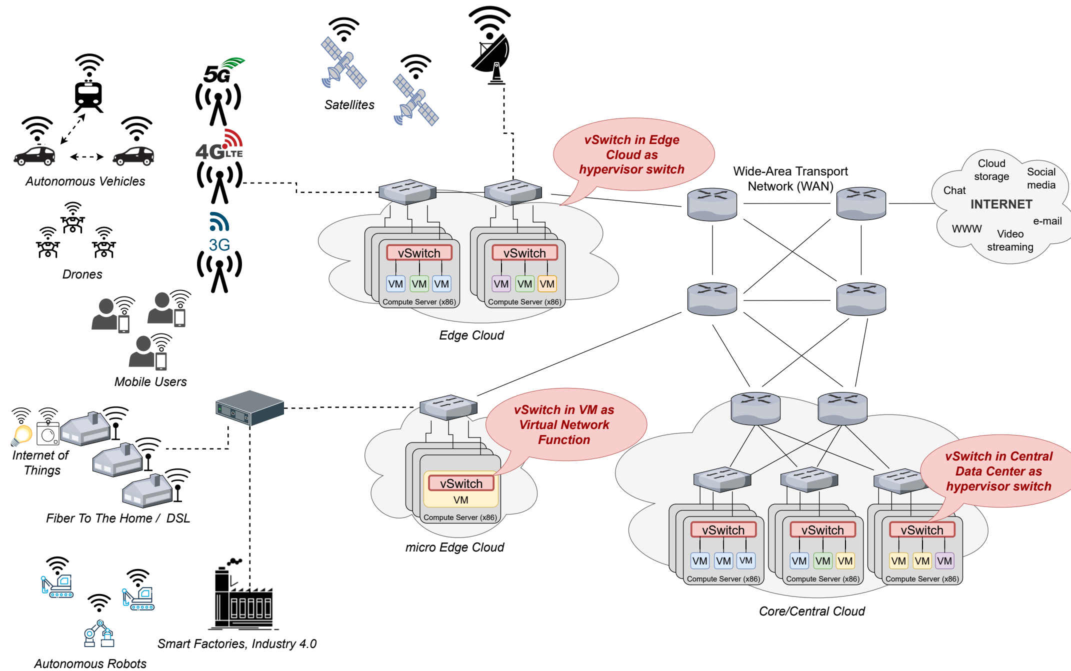


Figure 1.1: The role of software switches (marked by red callouts) in the end-to-end 5G/6G mobile network.

give an example, the role of software switches in the context of 5G/6G mobile networks is depicted in Figure 1.1. 5G/6G mobile networks are built from heterogeneous mobile devices and radio access technologies. The compute and network infrastructure is geographically distributed and composed of many Edge Clouds and a few Core/Central Clouds. Software switches play a role of hypervisor switches running in both Edge and Central Clouds to provide virtual networks for network tenants. Software switches can also run inside Virtual Machines or Containers and implement a data plane of Virtual Network Functions.

As more network services are migrated from hardware appliances to the cloud, virtual machine or container density on end hosts increases and the network infrastructure owners are pushed to adopt high-speed (10/40 or even 100 Gbps) NICs [55]. As a consequence, software switches and VNFs are expected to process higher volume of diverse network traffic, being generated by hundred of thousands traffic endpoints. Thus, software datapaths must primarily meet the demands of next-generation networks for high throughput packet processing, but, at the same time, provide a high degree of programmability, so that network owners can smoothly introduce new features, fix network issues, perform software upgrades to the data plane devices or customize

their network protocols to meet customers' requirements.

A sphere of interests of this dissertation is packet processing techniques for a data plane of software switches. In particular, the aim of the research summarized in this dissertation is to apply data plane programmability to software switches. Software switches might require frequent updates to support new protocols or encapsulation techniques for overlay networking, new network telemetry mechanisms or security applications. Moreover, network infrastructure owners may need to introduce a custom protocol extension to optimize their system. It should be possible to use a high-level programming language to customize a packet processing pipeline without an intimate knowledge about software switching technology, as it brings similar benefits to data plane programmability for hardware-based packet processors. Furthermore, expanding the programmable SDN domain to the end-host networking makes network platform programmable in an end-to-end fashion and enables new use cases. To name a few, a programmable software switch could provide an early classification for end-to-end slicing and Quality of Service (QoS) at the container or VM interface that could be further used by NICs or hardware switches to enforce network isolation or QoS policies. Moreover, the coordination between all programmable network devices could lead to enhanced, end-to-end network visibility (e.g., with In-Band Network Telemetry [141, 127]) to pinpoint a network element contributing to increased packet latency or detect packet drops and other anomalies. The end-to-end network visibility also makes a concept of verifiable networks feasible [57]. Finally, both fabric and end-host networking can be controlled by the same SDN controller giving an opportunity to eliminate network overheads and simplify network design by flattening underlay and overlay networks in virtualized data centers.

We further motivate the work in Chapter 3, but, in a nutshell, fully-programmable but performant software datapaths would bring a number of benefits that can be categorized as follows:

1. **Building extensible software switch.** This dissertation aims at proving that a P4-programmable software datapath is a viable alternative to already-existing kernel-based or kernel-bypass software datapaths. The benefits of using a reconfigurable software switch includes the ease of customizing packet processing pipelines for a specific use case.

2. **Data plane programming framework for VNFs.** P4 was initially designed to dynamically program hardware switches. However, with the emergence of NFV, 5G and edge computing, there are places (typically small edge sites), where it is more cost-effective to deploy a network function as a software module on an x86 server, instead of buying a specialized, programmable switch or NIC. Moreover, developing high-performance Virtual Network Functions on general purpose CPUs is a challenge [143] as programmers must know underlying specifics of the packet processing nature in the multi-core CPU environment to write an efficient code. A fully-programmable software datapath can become a framework for developing a software-based data plane of network functions, allowing to abstract the low-level details from programmers. Hence, they can use the high-level data plane programming language to implement Virtual Network Functions and let the P4 compiler perform micro-optimizations and ensure that the compiled code runs efficiently on the general-purpose CPUs.

3. **Towards end-to-end programmable network.** With the emergence of the end-to-end, programmable network platform a fully-programmable software switch might become a vital component enabling programming end-host datapaths in a portable way. This might enable using P4 as a common language for programming diverse network devices (from hardware switches up to end-host networking) and the P4Runtime API [174] as a unified control plane protocol, making such concepts as end-to-end verifiable networks [57] feasible.

An SDN software switch is expected to provide high performance, while maintaining a high degree of programmability, flexibility and general-purpose usage. Therefore, the main research challenge addressed in this dissertation is to design and implement a programmable software switch that provides enhanced programmability at runtime without sacrificing performance and operability. To address this challenge we need to create novel architectures and programming models for software switches as well as to design new, or adapt existing, packet processing techniques and algorithms. Therefore, this dissertation presents two novel systems for high performance, yet programmable, software-based packet processing: P4rt-OVS [132] and NIKSS (Native In-Kernel SDN Software Switch) [130]. Both solutions have been designed and implemented in cooperation with industrial partners: Orange Labs (Poland/France) and Open Networking

Foundation (USA). Therefore, both P4rt-OVS and NIKSS have practical applications and implementation potential - both solutions have been used to build systems described in [133], [145], [127] and [131]. The evaluation of the exemplary use case for P4rt-OVS described in [145] is also attached as Appendix B.

P4rt-OVS [132] derives from Open vSwitch (OVS), a fixed-function SDN software switch commonly used for network virtualization [148]. P4rt-OVS is an original extension of OVS that enables runtime programming of protocol-independent and stateful packet processing pipelines. It extends the OVS forwarding model with Berkeley Packet Filter (BPF) [85], bringing a new extensibility mechanism. The P4rt-OVS design results in a hybrid approach that provides P4 programmability without sacrificing the well-known features of OVS. We make the following unique contributions with P4rt-OVS:

- Design and implementation of runtime data plane programmability for Open vSwitch using P4. P4rt-OVS has been designed to be programmable at runtime. Therefore, we leverage userspace Berkeley Packet Filter (BPF) [109, 102] to provide a runtime extensibility mechanism for OVS.

- Enabling enhanced data plane programmability for OVS without sacrificing the performance.

- New programming model and control plane API for OVS. P4rt-OVS proposes extensions to the OVS architecture and programming model that enable extending the packet processing pipeline at runtime with P4.

- Design and implementation of a novel P4 to userspace BPF compiler that allows developers to write data plane programs in the high-level P4 language and run them in userspace BPF VM on general-purpose CPUs. The P4-to-uBPF compiler is not only limited to P4rt-OVS. On contrary, it is re-usable and can be leveraged by other solutions. It is currently open-source and publicly available as part of the open-source P4 compiler [139].

- Performance evaluation showing that P4rt-OVS does not introduce significant packet processing overhead comparing to classic OVS forwarding model, yet enables runtime protocol extensions and stateful packet processing.

- The P4rt-OVS solution is open-source and publicly available on Github [125].

NIKSS [130] is a novel programmable software datapath for Software-Defined Networking that overcomes the limitations of P4rt-OVS. NIKSS has been designed around the following principles: high-level and feature-rich programming abstraction, performance, runtime programmability and operability. It leverages P4 as a high-level programming abstraction, Portable Switch Architecture (PSA) as a fully-featured P4 packet forwarding model and eBPF [85] as an extensible, kernel-based packet processing engine. According to our knowledge, NIKSS is the first kernel-based and fully-functional P4-programmable software switch. The NIKSS solution achieves the following improvements over the state of the art systems:

- Design and implementation of the NIKSS's P4-programmable packet processing model. NIKSS supports two alternative designs of a packet processing model: a general-purpose design able to implement *any* P4/PSA program and a more limited specialized design that provides better performance.

- Design and implementation of the ternary packet classification algorithm for eBPF.

- Design and implementation of the PSA to eBPF compiler, an original extension to the P4 compiler that implements the PSA model for NIKSS. Moreover, NIKSS implements several compiler optimizations to maximize generated eBPF code performance. The PSA to eBPF compiler has been contributed to the official, open-source P4 compiler [140].

- A thorough performance evaluation of NIKSS, including microbenchmarks and a comparison with alternative software datapaths. The specialized NIKSS design can outperform the kernel-based Open vSwitch [148] and P4-DPDK, a main alternative to NIKSS, for most test programs.

- The NIKSS implementation is also open-source [129].

Finally, this dissertation contributes the added value to the state of the art by addressing the following research questions:

- *Can programmable data plane and P4 be successfully used for software datapaths without sacrificing performance?*
- *Given the hardware focus, is P4 flexible enough to express all software capabilities needed to implement most common features of a hypervisor switch or Virtual Network Function?*
- *Is the P4 Portable Switch Architecture (PSA) the right architecture for software datapaths? What is the price of portability?*

This disseration is organized as follows. Chapter 2 provides a necessary background information on software-based packet processing and data plane programmability. Chapter 3 motivates the work, defines a set of requirements for a programmable SDN software switch and discusses the state of the art of software datapaths. In Chapter 4, we present P4rt-OVS, a framework for programming protocol-independent, runtime extensions for Open vSwitch using P4. In Chapter 5, we present NIKSS (Native In-Kernel SDN Switch), a novel programmable software datapath for Software-Defined Networking. Finally, Chapter 6 concludes this dissertation, answers the research questions and discusses directions for future work. Additionally, Appendix A describes a methodology to measure per-component CPU cycles for NIKSS, while Appendix B presents an implementation and evaluation of the DDoS mitigator use case for P4rt-OVS.

Chapter 2

Background

This chapter presents background on software-based packet processing. We first introduce a general-purpose software-based packet processing environment, on which both P4rt-OVS and NIKSS are based. Next, we present and compare modern packet processing frameworks in software: DPDK, eBPF and AF_XDP. P4rt-OVS runs on DPDK, while NIKSS leverages eBPF. We also present algorithms and data structures used in software-based packet processing, focusing mainly on packet classification. Selected algorithms and data structures are used by P4rt-OVS and NIKSS. Finally, we provide details about performance metrics that are used to evaluate P4rt-OVS and NIKSS.

Next, we introduce the programming models (often referred to as abstractions) for programmable data planes. We present data plane programming and P4 in detail as the P4 language is a foundation of P4rt-OVS and NIKSS. Last but not least, we describe the eBPF framework as it is a technology that is leveraged by both P4rt-OVS and NIKSS switches.

2.1 Software-based packet processing

The nature of packet processing is common to both hardware-based and software-based switch implementations. Packet processing includes the following basic functional steps: *parsing*, *classification*, *modification*, *deparsing* and *forwarding*. Moreover, most packet processing systems can provide additional functionalities, such as *scheduling*, *filtering*, *metering* (often referred to as *rate-limiting* or *traffic policing*), or *traffic shaping* [115]. All these functional blocks (depicted in Figure 2.1) of packet processing are often

Figure 2.1: Functional blocks of packet processing pipelines.

referred to as *data plane functions*.

Parsing is a process of reading packet headers from the packet buffer and extracting the relevant protocol fields into a packet descriptor - a set of parsed values that are used as metadata through a packet processing pipeline. *Packet classification* is a process of matching a packet with a corresponding rule (stored in classification tables) determining a forwarding policy that defines processing actions to be applied to a packet. Actions may define a packet forwarding behavior (e.g., whether to drop the packet or which output port(s) to use) or required packet modifications (e.g., modify destination IPv4 address, decrement IPv4 Time-To-Live or add MPLS header). The *packet modification* steps applies the modification actions retrieved from the classification tables. This step may also include the update of some per-pipeline internal state (e.g., increase a packet counter). The combination of packet classification and subsequent processing based on matched rules is commonly referred to as *Match-Action processing* [115]. As a next step, a packet may be re-generated from the packet descriptor (*deparsing*) and, finally, the *forwarding* step is performed. The forwarding step may either send a packet to a single output port (*unicast forwarding*) or copy a packet and send packet replicas to multiple ports (referred to as *multicast forwarding* or *packet cloning*). Before a packet leaves the packet processing pipeline, additional steps may also be performed. For example, *packet scheduling* enforces network-level QoS policies, *metering* limits a rate of outgoing traffic by dropping packets above a given policing threshold, while *traffic shaping* manages output packet queues and buffers to achieve an assumed traffic profile.

Nevertheless, packet processing on general-purpose CPUs has specific nature. A software datapath typically makes use of a packet I/O framework and implements packet processing algorithms on top of it. In this section, we introduce packet processing mechanisms, frameworks and algorithms that are relevant to this dissertation and are used to implement the aforementioned data plane functions in P4rt-OVS and NIKSS.

2.1.1 Packet processing in the multi-core CPU environment

Software datapaths typically run on commodity servers and leverage general-purpose CPUs to perform data plane functions. The higher clock speed of a CPU core, the more instructions the CPU core can execute and, thus, more packets can be processed by a software datapath in a given period of time. In the early days of software-based packet processing the performance had been increasing due to the Moore's law [157] - CPUs had been becoming more and more efficient. However, the industry nowadays observes the slowdown of Moore's law and the end of Dennard's scaling [90]. As a consequence, the performance of a single CPU core has stopped to increase rapidly. Therefore, commodity servers started adopting the multi-core architecture, i.e. instead of executing CPU instructions on a single core, the multi-core environment incorporates multiple CPU cores to perform different tasks in parallel and, so, increase processing power. This approach has also been adopted by end hosts as a way to increase packet processing speed, forcing the re-design of packet processing frameworks. Furthermore, it is worth highlighting that the CPU frequency is not the only CPU-related factor affecting performance. In particular, the CPU cache plays an important role. The CPU memory is typically organized in the hierarchical structure composed of multiple levels (usually 2 or 3) of local-to-CPU memory called CPU caches. This architecture reduces the average cost to access data from the main memory by *caching* some data in a local CPU cache. Therefore, in case of large data and/or sub-optimally written code a packet processing application might spend more time accessing data if it is missing in the higher level CPU caches. Therefore, the larger CPU cache size, the higher CPU cache hit ratio is and, thus, a CPU core may process packets faster.

In the modern multi-core CPU environment, there are generally two *processing models* (a way to organize and coordinate between CPUs for a data plane processing): *the Pipeline model* and *the Run-to-Completion model* [49]. The former refers to a model in which each stage of the packet processing pipeline is mapped to a different core (or thread). A packet is being sent from one stage to the next one in the pipeline. Each CPU core running a given stage implements a specific functionality (e.g., core A performs parsing, while core B classifies packets). The Pipeline model is often used if more processing power is required than a single CPU core can provide. Also, the advantage of this model is that it does not require sharing state between CPU cores,

minimizing the need for shared access synchronization between different CPU cores. However, the Pipeline model often leads to a sub-optimal resource utilization due to over-provisioning (most likely some of CPU cores are not fully utilized) and, thus, waste of computing resources. On the other hand, the Run-to-Completion model assumes using a single CPU core for a given packet, i.e. a single CPU core handles a packet from the moment it enters the packet processing pipeline until it is forwarded or discarded. Typically, software systems implementing the Run-to-Completion model assign a group of CPU cores for packet processing and distribute incoming packets over available CPU cores. In this way, more packets can be processed by a system. The ability to increase packet processing speed by assigning more CPU cores is often referred to as *CPU scaling*. The Run-to-Completion model features a better resource utilization and minimized overhead related to context switching between CPU cores, but it requires synchronization between CPU cores to access the shared resources (e.g., classification tables). It is worth noticing that the hybrid model, combining both pipeline and run-to-completion models, is also feasible.

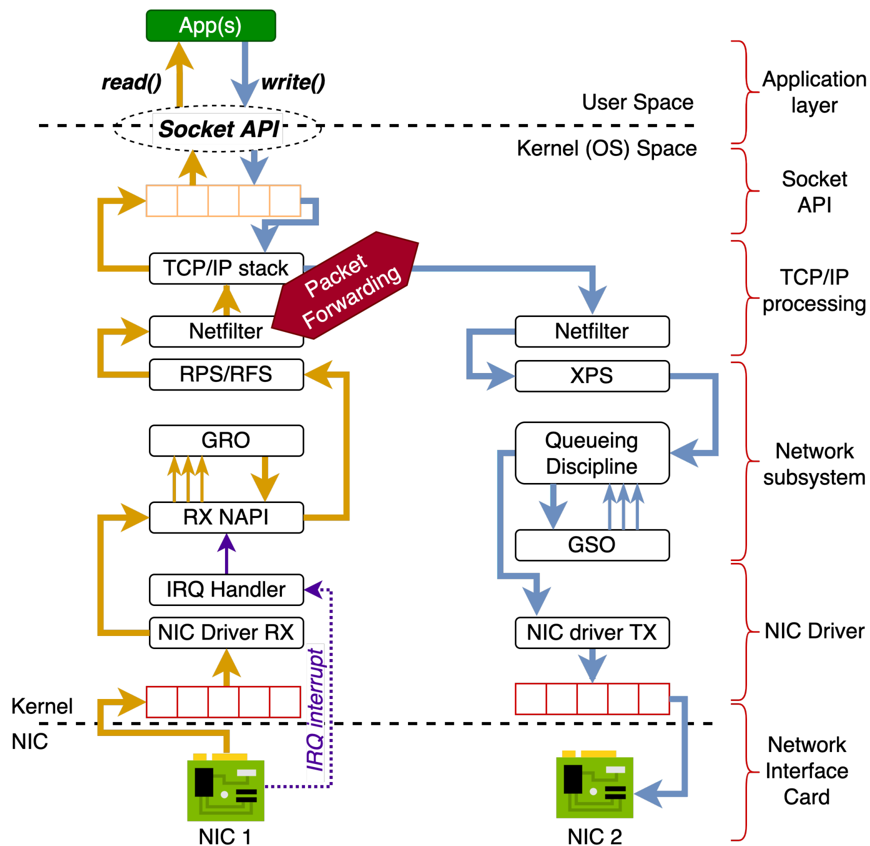


Figure 2.2: The end-to-end packet path between two NICs within a single Linux server.

Over years, the research and networking community have developed common mechanisms and components that compose a reference framework for multi-core packet processing. Such a standard multi-core packet processing framework is the Linux network stack. Figure 2.2 explains the Linux network stack based on the packet path between two NICs of the same server. A packet becomes its journey in the Linux network stack when it is received by NIC. In the diagram NIC-1 is a receiving (RX) NIC, while NIC-2 is a transmitting (TX) NIC. Upon receiving a new frame, the NIC uses the Direct Memory Access (DMA) mechanism to copy a received frame from NIC to the OS Random Memory Access (RAM) and stored into a DMA ring buffer. The memory address, where the frame is copied to, is determined by an RX descriptor (each NIC is configured with a number of RX descriptors). In parallel, the NIC triggers a hardware Interrupt ReQuest (IRQ). This event informs the NIC driver that a new frame has arrived and must be processed. The CPU core that handles the IRQ is statically configured (*CPU affinity*) or is dynamically selected by the NIC using the Reciever Side Scaling (RSS) mechanism. This is the first place, where the multi-core processing comes into play. Most modern NICs have the ability to write incoming frames to several different regions of RAM simultaneously and each region constitutes a separate queue. This allows to use multiple CPU cores to process incoming packet in parallel. RSS is a process of distributing incoming frames between queues (CPU cores) based on the hash function calculated from a packet data (typically a 5-tuple composed of source IP address, destination IP address, IP protocol, source UDP/TCP port and destination UDP/TCP port).

Upon receiving an IRQ on a given CPU core, the NIC driver triggers NAPI (New API) polling. The NAPI is a Linux mechanism that has been created to reduce the number of IRQs generated by network devices on packet arrival. Once a first frame is received, the NAPI starts polling on incoming frames, eliminating the need for subsequent IRQs, until a certain number of frames are received or a timer expires. This allows to handle multiple packets at once. While busy polling, the NIC driver allocates an `skb` (socket buffer), a packet descriptor within the Linux network stack. Each `skb` keeps a reference to the kernel memory address where the frame has been copied by DMA, so no additional copy of a packet is made at this point. Next, the network stack

invokes the GRO[1] (Generic Receive Offload) mechanism. GRO attempts to reduce the number of allocated `skb`'s by aggregating multiple packets from a single stream into a large buffer before they are passed higher up to the TCP/IP processing. The purpose of GRO is to increase inbound throughput by reducing CPU overhead as the TCP/IP stack must process packet headers only once for a large, aggregated chunk of data. Once a packet leaves, the RPS (Receive Packet Steering)/RFS mechanisms schedule a TCP/IP processing on one of the CPU cores. RPS is a software version of RSS. In case of RPS, there is a single CPU that process the hardware IRQs and distributed the load across multiple CPU cores based on a calculated hash value. Note that RPS consumes additional CPU resources in comparison to RSS. RFS (Receive Flow Steering) extends RPS to increase the CPU cache hit rate by steering the TCP/IP processing to the CPU where the application thread consuming the packet is running on. Importantly, if aRFS (accelerated RFS) - a hardware version of RFS - is enabled, all the tasks (the IRQ handling, TCP/IP and application processing) are performed on the same CPU in the Run-to-Completion model.

Next, a packet enters the TCP/IP stack, where the data plane functions such as parsing, classification, etc. are implemented. The TCP/IP processing starts from the Netfilter framework, which is basically a packet filtering mechanism in the Linux network stack. The Netfilter framework provides so-called kernel hooks, where the packet can be filtered at. The most commonly used implementation of the Netfilter framework is *iptables*. In the Netfilter subsystem, an IP header is firstly parsed and passed to the iptables to perform packet classification based on the user-provided iptables rules. Then, the packet, if allowed by filtering rules, is passed to the Linux routing engine, which performs classification and forwarding based on the destination IP address. The routing engine decides whether a packet is destined to a local application or should be forwarded to another NIC (either physical or virtual). Depending on this decision, the TCP/IP stack performs either the local delivery via the Socket API or packet forwarding to another NIC. If the packet is to be delivered locally, it is passed up to the transport layer (TCP/UDP) processing, where, for instance, the TCP state machine is running and, then, the packet's `skb` descriptor is appended to the socket's receive queue. The application thread receives an OS signal and performs data copy of the packet payload

[1] There also exists a hardware optimization of GRO that is known as Large Receive Offloading (LRO)

from `skb` in the socket receive queue to the user space buffer. It is worth noticing that this is the only step (when the data is transferred between userspace and kernel space), in which a packet buffer is copied. All other operations within the kernel space exploit a reference to `skb` and do not require data copy.

However, if the routing engine decides to forward a packet based on the routing table entries, the packet forwarding is performed. First, the packet is processed by the post-routing packet filtering (Netfilter) and enters Transmit Packet Steering (XPS). XPS is a mechanism for selecting which transmit queue to choose when transmitting a packet on a multi-queue device by recording a mapping from CPU to hardware queues. Note that XPS may be disabled and, then, packets are distributed over transmit queues based on the hash value calculated from packet fields. Next, the packet is sent to the Queueing Discipline (*qdisc*) subsystem. The *qdisc* is a subsystem that implements the QoS policies in the Linux OS. It can be configured to enforce advanced QoS mechanisms/behaviors (e.g., Hierarchical Token Bucket) or various queueing policies. At this point, the Linux kernel can perform traffic scheduling, shaping or policing. Then, the *qdisc* is responsible for enqueueing `skbs` in the NIC driver TX queue. Finally, the NIC driver processes TX queues and leverages the DMA to copy the packet data from the kernel buffer referenced by `skb` to the NIC.

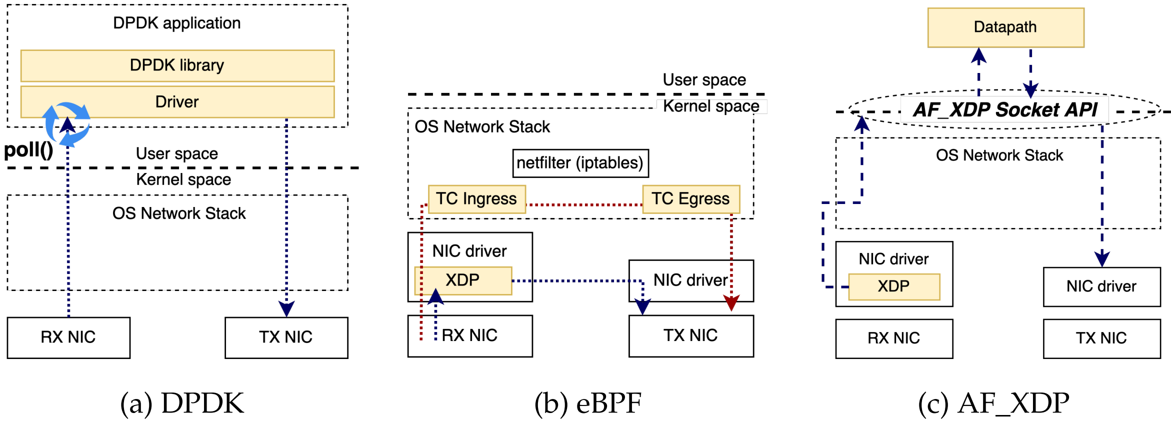
The described end-to-end data path shows different mechanisms and components that are involved in the packet processing on general-purpose CPUs. Most software datapaths rely, to some extent, on the Linux packet processing framework to take advantage of multi-core environment and, thus, increase packet processing speed, receive/transmit packets or perform data plane functions (including parsing, classification, modification, forwarding and QoS enforcement). Importantly, software datapaths shall be able to perform local delivery to the application thread via the Socket API, but their main task is to perform packet forwarding.

Nonetheless, despite the presented general-purpose Linux packet processing pipeline provides a reference framework for packet processing, its default flavor based on iptables and the Linux routing engine suffers from inefficiency and can hardly support multiple 10Gbps processing at line rate [29, 32]. This led to alternative packet processing frameworks, which are introduced in section 2.1.2. It is also worth outlining that the Linux network stack does not also provide a sufficient degree of programmability

- each new network feature must be incorporated into the kernel code first and, then, the OS kernel must be re-compiled and re-installed.

2.1.2 Packet processing frameworks in software

In this section, we introduce packet processing frameworks that have been lately proposed to improve packet processing speed on general-purpose CPUs. In [32] authors provides an extensive survey on different packet processing frameworks. RouteBricks [46] runs in the kernel space and extends the Linux Ethernet NIC driver to exploit parallelism of packet processing among multiple cores of a single server, or on multiple interconnected servers. This ability enables scaling performance of a software router up to 1 Tbps, but it requires multiple (100) servers with a speed of 20Gbps each. Moreover, several user-space I/O frameworks [83, 63, 50, 154, 123, 22] have been recently proposed to reduce packet processing overheads by bypassing the kernel and receiving packets directly in a userspace process. All these user space frameworks constantly poll the NIC for new packets (instead of waiting for the NIC to raise a hardware interrupt to the CPU), pre-allocate memory to minimize the packet buffer allocation overhead and process packets in batches to reduce CPU cache misses. They can be categorized into two groups. The first group including [154], [123] and [22] uses the default NIC driver and additional kernel component that provides a fast interface based on memory mapping for the user space process. The packet I/O from/to the NIC is still handled by the NIC driver, but the driver is directly connected to the user space application instead of the Linux network stack. Netmap [154], however, does provide a way to integrate with the Linux network stack and even to leverage the TCP/IP processing by using hardware filters that can steer packets to receive queues either managed by the netmap or kernel. Another category includes drivers that process packets completely in user space and do not require any kernel module. Nonetheless, since ixy [50] has only been created for educational purposes and Snabb [63] is not widely supported, only DPDK [83] is worthy of noting. In fact, DPDK is widely adopted by industry, has mature code base and large community, supports a wide range of offloading and filtering features and integrates with most of NICs available in the market. We elaborate more on DPDK in the next paragraph. Finally, other solutions uses GPUs (Graphics Processing Units) [69, 172, 180, 62] or FPGAs [105, 118, 10] to offload packet processing, but requires



(a) DPDK (b) eBPF (c) AF_XDP

Figure 2.3: Packet processing frameworks.

additional hardware and do not run purely on general-purpose CPUs.

Figure 2.3 shows the comparison of three packet processing frameworks that are most relevant from the perspective of our work as they provide the best trade-off between performance, maintainability, programmability and resource consumption: extended Berkeley Packet Filter (eBPF) with eXpress Data Path (XDP), DPDK [83] and AF_XDP.

DPDK implements a so-called Poll Mode Driver (PMD) that periodically polls the NIC for new packets. As mentioned before, DPDK processes packets completely in user space. While DPDK provides an unarguably high performance [58], it suffers from other issues that are well-summarized in [177]. First of all, DPDK bypasses the Linux network stack and uses its own network drivers. It causes not only the need for re-implementing the TCP/IP network stack in userspace, but, also, well-known tools to configure and manage NICs do not work with DPDK. This affects maintainability. According to [177], the additional maintenance of the tools and operational cost make this approach unattractive in large-scale deployments. Moreover, DPDK requires dedicated the entire CPU (or set of CPUs) to software switching, making this approach far from cost-effective solution due to per-core pricing models applied by public cloud providers. Nevertheless, DPDK is the most efficient packet processing framework for software datapaths, but the high performance comes at the cost of an increased resource consumption. Finally, DPDK provides not only the packet I/O framework, but also a set of libraries and packet processing engine that can be used to implement software datapaths or VNFs. It is not easily programmable though, as DPDK users cannot use a high-level, declarative language to define a new network feature (DPDK applications

are written in C/C++).

eBPF was originally a BSD (Berkeley Software Distribution) technology providing a minimal instruction set as well as an in-kernel infrastructure for packet filtering [109]. Over time, it evolved as a Linux kernel technology [85] into a general-purpose virtual machine in the Linux kernel, designed to allow user space processes to update the kernel's behavior at runtime by injecting so-called eBPF programs. It provides both the execution environment inside the Linux kernel as well as a programming model to develop new eBPF programs (we elaborate more on eBPF internals and the programming model in section 2.2.2). Despite eBPF is not only limited to packet processing, it has been widely adopted by networking industry [121, 11, 37]. In fact, eBPF extends the general-purpose Linux packet processing framework described in section 2.1.1 and allows to inject packet processing modules into the OS network stack. The Linux kernel provides a few *BPF hooks* - packet interception points in the Linux network stack, where an eBPF program can be attached to (Figure 2.3b). The TC (Traffic Control) hook is integrated into the Linux network subsystem. It handles incoming packets before they reach Netfilter (the TC Ingress hook), or after they are processed by Netfilter and send out to an output port (the TC Egress hook). The XDP (eXpress Data Path) hook [75, 181] runs directly in the NIC driver and receives packets immediately after they are received by the NIC driver, even before the `skb` descriptor is allocated. This allows to avoid wasting CPU cycles on other operations (e.g., memory allocation) and, thus, eBPF programs attached to XDP are able to reach higher throughput values than programs on TC [181]. The performance of XDP is still lower than of DPDK [75], but eBPF-based packet processing is highly extensible, does not consume additional CPU resources and, as it is well-integrated with the Linux kernel, enables using standard network configuration tools (it means better operability). Note that eBPF/XDP relies on the standard Linux mechanisms such as IRQ handling, NAPI, RSS, RPS/RFS, XPS, etc.

Finally, Figure 2.3c depicts the AF_XDP technology [86], which heavily relies on XDP, but moves the packet processing to the user space. AF_XDP is essentially a new socket API type that leverages an in-kernel eBPF program attached to XDP to bypass the kernel and send packets directly to the user space. Importantly, AF_XDP can operate in a zero-copy mode (if supported by a NIC driver), in which the userspace-

provided memory is directly used for DMA delivery. The choice between in-kernel XDP processing vs. AF_XDP depends on the use case. According to [147], their performance is comparable, with a little advantage of AF_XDP over XDP for packet forwarding. Since both are integrated with the Linux kernel, they provide a sufficient level of maintainability. Also, both depend on the Linux mechanisms to handle the multi-core processing. Thus, the CPU usage characteristics is roughly comparable (and the CPU usage is much lower than for DPDK [75]). The main difference between eBPF/XDP and AF_XDP is in programmability and packet processing capabilities. The AF_XDP socket is, in fact, just a packet I/O framework and requires implementation of packet processing engine in user space. The user space gives more flexibility in what algorithms and data structures are feasible, but requires re-implementation of the TCP/IP stack (e.g., ARP and ICMP handling). Also, a level of data plane programmability depends on the choice of a packet processing engine. AF_XDP itself (as a packet I/O framework) does not provide a way to develop packet processing pipeline. On the other hand, eBPF/XDP provides both a packet I/O framework and a highly extensible packet processing engine as well as a programming model. Nevertheless, eBPF/XDP may not always be the best choice, because in-kernel packet processing is constrained and some network features may not be feasible there.

2.1.3 Packet classification algorithms

The previous section focuses on the packet I/O frameworks - a way to deliver packets to a software application implementing a packet processing pipeline. In this section, we describe algorithms and data structures that are used to implement data plane functions, once a packet is delivered to a packet processing pipeline.

The main challenge for software-based packet processing is an efficient packet classification. *Parsing* is simply done by mapping a packet buffer referenced by a packet descriptor to protocol headers' format. *Deparsing* is a reverse operation that either modifies a packet buffer or copies data from a packet descriptor to the packet buffer. *Packet modification* typically modifies just a packet descriptor or a packet buffer directly, but it does not require significant computing cost. Finally, *forwarding* is performed by means of a packet I/O framework.

SDN software datapaths requires an online packet classification system. This means

that a classification table may be frequently updated, i.e., forwarding rules may be often inserted, modified or removed - depending on the intelligence implemented in an SDN controller. Therefore, packet classification algorithms for software datapaths must not only offer an efficient classification in the data plane, but, also, fast table updates. Moreover, software systems are typically provisioned with a large number of forwarding rules. Hence, packet classification mechanisms must scale, i.e. a growing number of forwarding rules should not degrade classification performance and, in consequence, packet forwarding rate (throughput).

We can distinguish four main types of classification: *exact*, *Longest-Prefix Match (LPM)*, *ternary* (or *wildcard*) and *range*. The *exact* classification looks for a one-to-one mapping between a packet header's field and forwarding rule and can be implemented with *hash tables* [158]. Hash tables provide O(1) average complexity for lookups, updates and deletions, regardless of the number of entries[2]. For instance, a cuckoo hashing has been proven to process packets at high rates [195].

The drawback of the exact classification is that it requires a huge number of entries in a large-scale system to cover all network policies, routes and network endpoints. Therefore, more scalable classification schemes have been proposed: Longest-Prefix Match, ternary (also known as wildcard) and range matching. All these algorithms can be easily implemented in hardware with Ternary Content-Addressable Memory (TCAM) at the cost of memory resources. However, general-purpose CPUs do not have an efficient way to perform this kind of classification. Thus, some algorithmic techniques for software are used.

The *LPM* classification aggregates network flows into subsets. The primary use case for Longest-Prefix Match is IP routing, in which a forwarding decision is made based on a destination IP address. In case of LPM, table entries matching on a destination address are aggregated per IP subnet and selected according to a prefix (the number of leading address bits of the destination address that matches a table entry). The longest prefix has always the highest priority. The LPM classification is typically based on a *trie*, a variant of a tree data structure. Many trie-based algorithms has been proposed for LPM including unibit tries, multibit tries, level-compressed tries, Lulea-Compressed tries and tree bitmap [179]. Also, binary search techniques have been

[2]The worst case complexity of hash tables is O(n) due to hash collisions that might occur.

adapted to implement Longest-Prefix Match [101, 179]. The range-based classification checks if a packet's field is within the specified value range. The range matching is typically implemented using range-to-prefix conversion [65], where a W-bit range can be represented by at most 2W - 2 prefixes, and implemented using LPM tables. Over years, some optimizations to range-to-prefix conversion have been proposed [52].

Finally, the ternary (wildcard) matching is a general classification scheme that can match on arbitrary packet fields, where a particular field (or even individual bits) can be set as "don't care" in a flow rule. The ternary classification provides a full flexibility to defining flow rules and it is often used for Access Control Lists (ACLs). Software-based packet processing systems have leveraged different algorithms for the wildcard classification. One solution is Tuple Space Search (TSS) algorithms [169, 40]. TSS [169] decomposes a ternary lookup into a sequence of lookups to hash tables, where each hash table represents a set of wildcard fields being matched. Thus, a lookup to such a ternary table is implemented as a linear search over each hash table. TupleMerge [40] optimizes the TSS algorithm by merging some hash tables to reduce number of lookups. The TSS algorithms are marked by fast updates and memory resources linear in the number of rules. Another group of ternary classification algorithms are decision-tree based classifiers such as HiCuts [66], HyperCuts [164] or EffiCuts [178]. These algorithms generally provide faster lookups than TSS, but consume memory polynomial in the number of rules. Moreover, decision trees must be usually re-constructed when inserting or updating a flow rule. Thus, they require complex computations making the update time slower comparing to TSS. Yet another approach are cross-producting [170] and solution based on binary logic [14, 100]. However, both approaches have prohibitive memory costs. A final choice of a wildcard classification algorithm should depend on throughput requirements, memory limitations and expected forwarding rule patterns. Nevertheless, the TSS algorithms have so far been preferable for SDN software datapaths (e.g., OVS [148]) mainly due to linear memory requirements and faster updates that are frequently performed in SDN systems. Therefore, TSS has also been adopted by NIKSS.

LPM, range or ternary matching on general-purpose CPUs is basically a costly operation that consumes significant amount of CPU cycles. Hence, it is inefficient to apply complex packet classification for each incoming packet. Flow caching is a

known technique for reducing the average cost of complex packet classification [95, 163, 193]. The concept of flow caching is straightforward. The simplest design assumes an existence of exact-match cache (typically based on hash tables) before the classification table. If a packet misses the cache, it is further processed by the packet classification engine. After the packet has been classified, a rule is installed in the cache, so that all subsequent packets of the same network flow will hit the cache and skip the costly packet classification. Some papers [163, 94, 148] also propose second-level caches to optimize for the case of short-lived connections that may be handled inefficiently with just a simple single-level cache. In our work the flow caching technique is applied to optimize the performance of lookups to P4 tables in the NIKSS solution.

2.1.4 Performance metrics for software-based packet processing

The following key performance metrics have been defined [26] to evaluate the performance of any kind of a data plane device:

- **Frame loss rate** (or **Packet loss ratio**) - percentage of frames that should have been forwarded by a network device under constant load that were not forwarded due to lack of resources.
- **Throughput** - defines a maximum rate at which a given device is able to process packets, assuming a certain maximum frame loss rate. Throughput is typically reported in packets per second (pps) or bits per second (bps). The former does not depend on a packet size of generated traffic stream and, thus, provides a clear metric about how fast a given system processes packets. Given a packet size in bytes, it is straightforward to calculate the throughput value in bits per second. Finally, the throughput is often measured in the Non-Drop Rate (NDR) scenario, which assumes 0% packet loss. On the contrary, a Partial-Drop Rate (PDR) is a throughput value measured assuming a certain level of packet loss ratio.
- **Latency** - for store-and-forward devices (such as software datapaths), it is defined as the time interval starting when the last bit of the input frame reaches the input port and ending when the first bit of the output frame is seen on the output port. Naturally, the latency is reported in seconds. However, the most meaningful metrics are the minimum and maximum latency, and the latency

distribution that shows different percentiles (e.g., 50%, 90%, 99%) from a set of latency measurement results.

In case of software datapaths, these metrics depend on execution efficiency of packet processing on general-purpose CPUs. Therefore, the major meaningful performance indicator for packet processing on general-purpuse CPUs is a number of CPU cycles per packet (CPP), which is defined as follows:

$$CPP = \frac{\#instructions}{packet} * \frac{\#cycles}{instruction} = \frac{IPP}{IPC}$$

The above equation binds two other important metrics: IPP (Instructions per Packet) and IPC (Instructions per Cycle). IPP depends on the packet processing program structure and can be improved by minimizing the number of CPU instructions needed to perform a packet processing task. IPC defines a number of instructions executed by a CPU per cycle. Software developers may optimize IPC by writing a more CPU-friendly code. Nevertheless, CPP is the most common performance metric for software packet processing and is experimentally measured using CPU profiling tools (e.g., Linx perf [126]). Importantly, the throughput rate can be bound with CPU cycles per packet as follows:

$$Throughput[pps] = \frac{1}{CPP * cycle_time[sec]} = \frac{CPU_{freq}[Hz]}{CPP}$$

In practice, it is extremely hard to mathematically predict the target performance of a software system. First of all, there are many components consuming CPU cycles and contributing to the performance overhead, including packet processing on CPUs, memory bandwidth, I/O bandwidth and inter-socket transactions. Moreover, there are many OS-level mechanisms that have significant impact on performance (e.g., scheduling, DCA, IOMMU, NUMA node allocation) [29]. Therefore, the software datapath benchmarking typically measures CPU cycles per packet and/or Instructions per packet taken by a program implementing a packet processing pipeline, in isolation from other performance-affecting factors and under a given, fixed OS configuration.

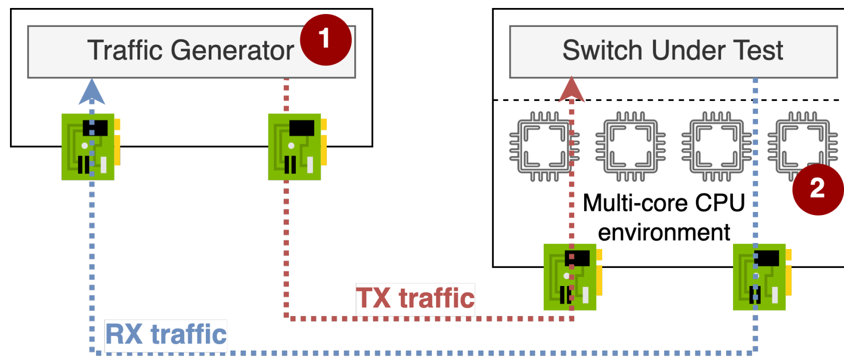


Figure 2.4: Typical performance evaluation framework for software switches with measurement points marked by red circles.

Figure 2.4 shows a typical performance evaluation framework for software switches. A Switch Under Test (SUT) is usually running on a multi-core CPU environment and receives traffic from high-speed NICs. At the other end, a traffic generator is connected to those NICs. The traffic generator sends a stream of packets to the SUT machine, the SUT processes packets and sends them back to the traffic generator. Figure also shows the measurement points. Packet loss ratio, throughput and latency is measured by the traffic generator (measurement point 1.), while CPU Cycles per Packet (CPP), Instructions per Packet (IPP) and Instructions per Cycle (IPC) are registered by polling CPU statistics on the SUT machine (measurement point 2.).

2.2 Abstractions for programmable data planes

Programming abstractions (or programming models) are essential to enable users, software developers and researchers a way to design and implement flexible packet processing pipelines, without an intimate knowledge about the underlying hardware (e.g., ASIC or CPU) architecture. There are two main programming models that have so far been proposed for data plane programming: *data flow graphs* and *Match-Action processing*.

Data flow graphs have been initially used in the early designs for packet processing systems. This model assumes that a packet processing pipeline is abstracted as a data flow graph, with the nodes representing packet processing stages and edges representing transitions from one processing stage to another. Most packet processing

engines based on the data flow graphs enable users to dynamically define and insert processing nodes and modify transitions. This leads to high flexibility and extensibility, but hinders the ability to implement platform-specific performance optimizations as a packet processing pipeline might be highly dynamic. A precursor of the data flow graph abstraction was the Click modular router proposed in 2000 by E. Kohler et al. [96]. Nowadays, the data flow graph abstraction is still used by the state-of-the-art software switches such as ClickOS [108], Vector Packet Processing (VPP) [15] or Berkeley Extensible Software Switch (BESS) [68].

Match-Action processing describes data plane programs using a sequence of classification tables (flow or Match-Action tables) organized into a hierarchical structure [115]. Each Match-Action table defines a match key, a subset of the packet header fields that are used to perform a table lookup (classification) to find a corresponding action. Network engineers or administrators configure the Match-Action tables with a set of rules (match key + action) defining packet processing behavior. The Match-Action processing is sometimes referred to as flow-based packet processing, as packets matching a given Match-Action table are represented as a *flow*. A prominent example of the Match-Action processing abstraction is OpenFlow [110], which had been an evolution of Ethane [30]. OpenFlow provides an abstract forwarding model based on Match-Action processing for open and programmable data plane devices and the OpenFlow protocol, an API to control such devices. With OpenFlow, SDN controllers or network administrators can instruct OpenFlow-compliant devices how to forward packets by installing the rules in the devices' Match-Action tables.

OpenFlow is a stateless data plane abstraction though, i.e. the data plane cannot modify a forwarding state itself (e.g., depending on packets seen by a switch). Over time, a number of stateful data plane abstractions extending the Match-Action processing model has been proposed [20, 21, 151]. Moreover, OpenFlow suffers from two major limitations [24]: 1) the Match-Action forwaring model allows processing on only a fixed, pre-defined set of packet fields, and 2) the OpenFlow specification only defines a limited repertoire of packet processing actions. In [24], authors propose the RMT (Reconfigurable Match Tables) model enabling more flexibility in how packet processing pipelines are defined. RMT allows defining custom protocol headers, matching packets on arbitrary header fields and applying packet modifications in a programmable

manner. dRMT [24] is an extension of RMT that relaxes some constraints on sequential processing and the forwarding state sharing between the pipeline stages. In 2014, a group of researchers borrowed from such techniques as RMT and programmable parsers [61] to overcome OpenFlow limitations and proposed the P4 technology [23], starting an industry move towards data plane programmability [72].

2.2.1 Data plane programmability and P4

Data plane programmability means that the network data plane realized by packet processors (e.g., ASICs, NPUs, general-purpose CPUs) can be re-programmed and a packet processing pipeline can defined as a software program. With data plane programmability everyone can build a custom network protocol or packet processing algorithm on a re-programmable packet processor. This provides full flexibility to designing network systems. Authors in [72] and [78] list the following benefits of data plane programmability:

- **New features.** Data plane programmability provides full flexibility to network packet processing. Novel data plane algorithms, protocols and techniques can be invented by students, researchers, network administrators or network programmers and quickly implemented. This can foster network innovation and produce many new network services.

- **Reduced complexity and efficient use of resources.** Typically, platform vendors provide all-in-one appliances that implement a wide spectrum of network protocols. It is up to a network administrator to choose and configure required protocols. With data plane programmability, a network owner can just implement protocols or mechanisms that are required for a particular use case and remove others. This decreases the overall complexity of a data plane implementation, improves security as a hacker cannot leverage unused protocols to attack a system and implies more efficient use of resources as unmaintained protocols do not occupy limited memory resources.

- **Fast prototyping and short time to market.** Network system or protocol designers can quickly experiment with novel protocols or packet processing techniques and build prototypes to validate their ideas. Compared to long development cycles of

new hardware-based solutions, new data plane algorithms can be implemented and tested in a matter of days.

- **Software style development.** Data plane programmers can apply development tools that are used for typical software development, e.g., Continuous Integration and Deployment, Agile development, bug fixing, automated tests, etc. All these tools and techniques allows to build more reliable software.

- **Intellectual property.** Data plane programmers keeps their own ideas and do not need to share know-how with silicon/ASIC vendors. This opens the market for smaller companies and can break a vendor lock-in.

In 2014, Bosshart et al. [23] proposed P4, a framework for programming protocol-independent packet processors. P4 defines both a high-level, domain-specific programming language to define packet processing behavior and an abstract forwarding model of a P4-compliant network device. The P4 framework was designed around three principles:

- **Runtime re-configurability.** A programmer or SDN controller should be able to replace or modify the way switches process packets once they are deployed.

- **Protocol independence.** Packet processing pipelines should not be tied to any specific network protocol. Instead, a P4 programmer should be able to define and implement *any* packet header that can be used in the packet processing pipeline.

- **Target independence.** P4 should be able to describe packet processing pipeline without the in-depth knowledge about the architecture and specifics of an underlying platform (called *P4 target*).

P4 is an open-source technology and all specifications are being maintained by the P4.org consortium. The first version of the P4 language, $P4_{14}$, was proposed in 2015. $P4_{14}$ was significantly based on the PISA (Protocol-Independent Switch Architecture) switch and consisted of the basic programming abstractions such as Parser, a Control flow, including Match-Acton tables and stateful objects (e.g., packet counter, register), and Deparser. Since 2015 the P4 language has evolved and currently the up-to-date language version is $P4_{16}$ [28].

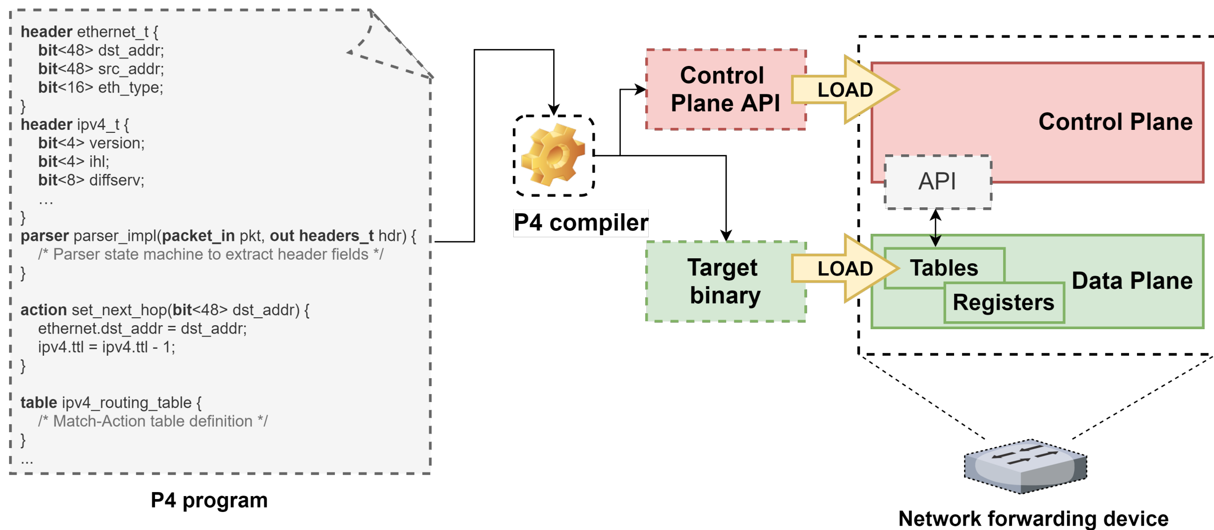


Figure 2.5: The P4 workflow.

P4 is designed to be implementable on a large variety of targets, including programmable network interface cards, FPGAs, software switches and hardware ASICs. The basic P4 workflow is depicted in Figure 2.5. The P4 architecture assumes that a network forwarding device is fully programmable, i.e. its packet processing pipeline can be dynamically changed at runtime. Firstly, a P4 programmer writes a P4 program, a high-level description of a packet processing pipeline, using programming abstractions provided by the P4 language. Moreover, the P4 program must be written according to an abstract forwarding model defined by P4. Next, the P4 program is translated to the target-specific microcode by the *P4 compiler*. The P4 compiler turns a target-independent representation of packet processing into a target-dependent binary that is loaded onto a device. In this way, the P4 compiler abstracts low-level platform details from a programmer and can optimize a target binary for a given platform. Finally, the target binary is injected into the network forwarding device. Additionally, the P4 compiler generates a control plane API, a meta-description of a forwarding pipeline, that can be used by an SDN controller to manage P4-capable devices (e.g., to insert table entries). Currently, P4Runtime API [174] is a standard control plane protocol for P4-programmable devices.

A basic P4 program is composed of at least three programmable blocks: Parser, Control and Deparser. A standard information flow between them is depicted in Figure 2.6. A Parser reads incoming packets and extracts header fields into header data and metadata. Since P4 does not include pre-defined packet header definitions,

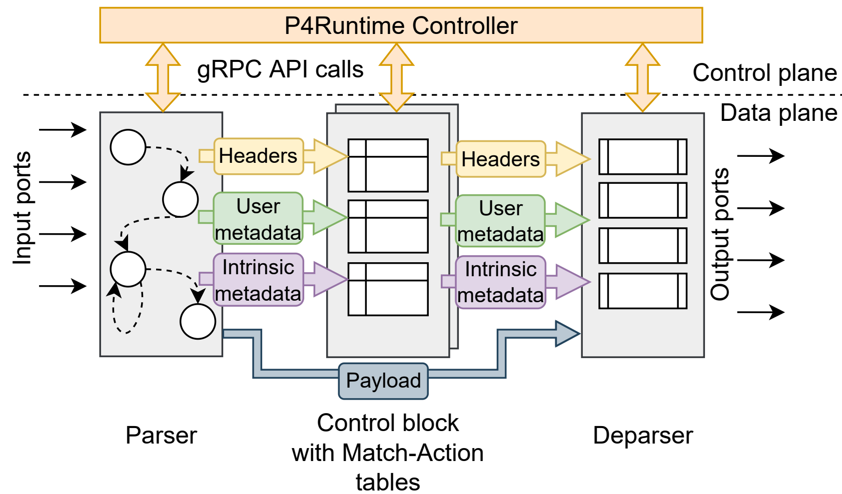


Figure 2.6: P4 information flow in the PISA model.

a programmer must define all required header formats and make sure that they are parsed correctly. Parsers are abstracted as a Finite State Machine (FSM) with an explicit `start`, `reject` and `accept` states and other custom states. Deparser performs a reverse operation to Parser - it serializes a packet from header data and metadata and reassembles the packet header and payload back into a byte stream. A Control block is placed in between Parser and Deparser and it implements data plane algorithms using Match-Action processing. The P4 language provides general-purpose statements such as *if-else*, *switch* or *return* to express the program flow within a control block. Moreover, it allows, among others, to perform assignments between header data, metadata and arbitrary constant values, to lookup Match-Action tables (called P4 tables) and to invoke external, platform-specific functions (called P4 externs) that might involve additional state preserved between program run. All these programmable blocks operate on shared data types such as `header` (defining a packet header) or `struct` that defines a custom metadata container.

```
header ethernet_t {
    bit<48>  dst_addr;
    bit<48>  src_addr;
    bit<16>  etherType;
}
header mpls_t {
    bit<20> label;
    bit<3> tc;
    bit<1> bos;
    bit<8> ttl;
}
header ipv4_t {
    bit<4>   version;
    bit<4>   ihl;
    bit<6>   dscp;
    bit<2>   ecn;
    bit<16>  totalLen;
    bit<16>  identification;
    bit<3>   flags;
    bit<13>  fragOffset;
    bit<8>   ttl;
    bit<8>   protocol;
    bit<16>  hdrChecksum;
    bit<32>  srcAddr;
    bit<32>  dstAddr;
}
[... TCP, UDP definitions
    ...]
struct headers {
    ethernet_t ethernet;
    mpls_t     mpls;
    ipv4_t     ipv4;
    tcp_t      tcp;
    udp_t      udp;
}
```

Listing 2.1: Sample declaration of packet headers supported by

```
parser IngressParserImpl(packet_in packet,
  out headers hdr, ...) {

    state start {
        transition parse_ethernet;
    }
    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition
            select(hdr.ethernet.etherType) {
            TYPE_IPV4: parse_ipv4;
            TYPE_MPLS: parse_mpls;
            default: accept;
        }
    }
    state parse_mpls {
        packet.extract(hdr.mpls);
        transition parse_ipv4;
    }
    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition select(hdr.ipv4.protocol) {
            PROTO_UDP: parse_udp;
            PROTO_TCP: parse_tcp;
            default: accept;
        }
    }
    state parse_udp {
        packet.extract(hdr.udp);
        transition accept;
    }
    state parse_tcp {
        packet.extract(hdr.tcp);
        transition accept;
    }
```

Listing 2.2: Sample Parser implementation extracting Ethernet, MPLS, IPv4, UDP and TCP

```
control IngressControl(inout headers_t hdr, inout metadata user_meta, in
    psa_ingress_input_metadata_t std_meta) {

    Counter<bit<32>, bit<32>>(100, PSA_CounterType_t.PACKETS) in_pkts;

    [... declarations of tbl_ingress_vlan, tbl_routable, tbl_routing ...]

    action mac_learn() {
        user_meta.send_mac_learn_msg = true;
        user_meta.mac_learn_msg.mac_addr = hdr.ethernet.src_addr;
        user_meta.mac_learn_msg.port = std_meta.ingress_port;
        user_meta.mac_learn_msg.vlan_id = hdr.vlan_tag.vlan_id;
    }

    table tbl_mac_learning {
        key = {
            hdr.ethernet.src_addr : exact;
        }
        actions = {
            mac_learn;
            NoAction;
        }
        const default_action = mac_learn();
    }

    action forward(PortId_t output_port) {
        send_to_port(ostd, output_port);
    }

    action broadcast(MulticastGroup_t grp_id) {
        multicast(ostd, grp_id);
    }

    table tbl_switching {
```

```
        key = {
            hdr.ethernet.dst_addr : exact;
            hdr.vlan_tag.vlan_id : ternary;
        }
        actions = {
            forward;
            broadcast;
        }
    }

    apply {
        in_pkts.count((bit<32>)std_meta.ingress_port);

        tbl_ingress_vlan.apply();
        tbl_mac_learning.apply();
        if (tbl_routable.apply().hit) {
            switch (tbl_routing.apply().action_run) {
                set_nexthop: {
                    if (headers.ipv4.ttl == 0) {
                        drop();
                        exit;
                    }
                }
            }
        }
        tbl_switching.apply();
    }
}
```

Listing 2.3: Sample Control block implementing basic L2/L3 processing.

```
control DeparserImpl(packet_out packet, ..., inout headers_t hdr, in metadata
    user_meta, ...) {
    Digest<mac_learn_digest_t>() mac_learn_digest;
```

```
    apply {
        if (user_meta.send_mac_learn_msg) {
            mac_learn_digest.pack(user_meta.mac_learn_msg);
        }

        packet.emit(hdr.ethernet);
        packet.emit(hdr.vlan_tag);
        packet.emit(hdr.mpls);
        packet.emit(hdr.ipv4);
        packet.emit(hdr.tcp);
        packet.emit(hdr.udp);
    }
}
```

Listing 2.4: Sample Deparser implementation emitting a combination of Ethernet, MPLS, VLAN, IPv4, TCP and UDP headers.

Listings 2.1-2.4 show sample implementations of different sections of a P4 program. Listing 2.1 shows declaration of packet headers supported by the sample P4 program - all headers should be implemented as they are defined in specifications. It is worth noting that a user has full flexibility in defining packet headers, so any custom fields are also allowed. Listing 2.2 depicts sample Parser implementation that uses packet headers defined in Listing 2.1. The process starts in the *Start* state and switches to the user-defined *parse_ethernet* state. Next, the following headers (MPLS, IPv4, TCP, UDP) are parsed according to the defined conditions. A packet header is parsed by calling the `extract()` function on a packet. The header data is then filled with data copied from the packet buffer. Once a packet is parsed, the processing moves to the Control block, which is shown in Listing 2.3. This sample implements a basic L2/L3 processing. Thus, it defines Match-Action tables and the control flow that performs such operations as ingress VLAN filtering, MAC learning, routing and switching. Each P4 table typically defines a set of match keys (the `key` keyword), a list of actions, a default action if no match is found and optionally a size of the table (maximum number of allowed table entries). A match key can be a packet field or user-defined metadata. The control plane software can install P4 table entries that maps a given set of match keys' values

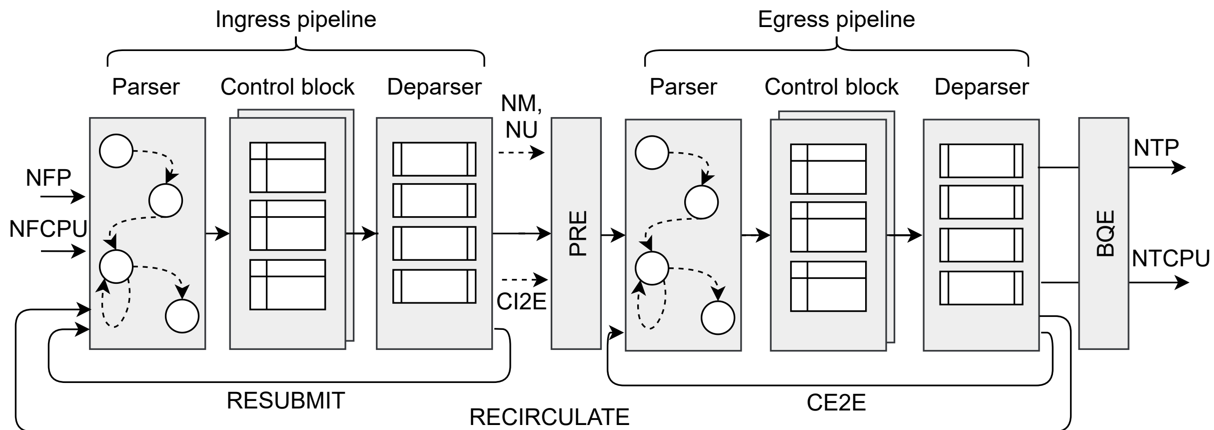


Figure 2.7: Portable Switch Architecture (PSA) for the P4 language

to an action. A P4 action can perform direct operations on packet headers (e.g., modify Ethernet destination address or decrement IPv4 TTL), assign some runtime-provided value to a metadata field or invoke a P4 extern. The control flow typically contains P4 tables' invocations (by calling `apply()` on a P4 table instance) and/or calls to P4 externs, all of them guarded by conditional statements. Finally, Listing 2.4 shows a sample Deparser implementation. A packet is reassembled by calling `emit()` functions on pre-defined packet headers in a user-defined order.

There are two significant extensions that were brought by $P4_{16}$. First, it introduces a concept of an *architecture model*, a platform-specific description of a forwarding model that can be changed by target manufacturers. As a consequence, a P4 target vendor should provide the target-specific P4 compiler along with a P4 architecture model that a P4 programmers can follow to implement packet processing pipelines. The P4 architecture typically defines P4-programmable blocks (e.g., parser, control flow, deparser). Another important extension to the P4 language are so-called *P4 externs* that should be part of the P4 architecture model definition. P4 externs are architecture-specific functions that can be used to describe a packet processing behavior. Examples of them are: a Register used for stateful packet processing, a Counter to count packets or bytes or a Meter to limit a packet rate. It is worth noticing that the P4 architecture model describes an extern functionality and API that externs expose to the control plane, but their implementation is still target-specific.

Over time, many P4 architecture models have been proposed [72], but none of them provide the portability of P4 programs between different P4 targets. To address

this challenge, the P4 Architecture Working Group has developed Portable Switch Architecture (PSA) [175]. PSA defines a set of P4 externs and APIs that every target mapping PSA should support. Also, PSA brings a fully-featured architecture model for network switching devices. PSA is illustrated in Figure 2.7. Since PSA provides a general overview of P4 capabilities (and it is also used by the NIKSS solution presented in this dissertation) we provide a detailed description of it.

The packet processing pipeline of PSA is divided into an ingress and egress pipeline, and each is composed of three P4-programmable blocks: a *Parser* that extracts headers, a *Control* block that specifies the control flow among Match-Action tables, and a *Deparser*, which writes the header fields back onto the packet before sending it further. PSA also defines configurable, platform-dependent blocks: the Packet buffer and Replication Engine (PRE) is responsible for packet buffering and replication, while the Buffer Queuing Engine (BQE) is responsible for packet queueing at egress. Furthermore, PSA specifies basic primitives (e.g., to drop or forward a packet) as well as a set of possible packet paths within a P4 target. At ingress, a packet may be received from a port (normal packet from port - NFP) or from a CPU port (NFCPU). The ingress pipeline determines the output port for a packet. Then, it can be unicasted (normal unicast - NU), sent back to the beginning of the ingress pipeline (RESUBMIT) or replicated - either by using a multicast (normal multicast, NM) or a clone session (CI2E). At the end of the egress pipeline, a packet may be sent to the normal port (NTP) or CPU port (NTCPU). Additionally, it can be duplicated at the end of the egress pipeline (CE2E) - in this case, the cloned packet starts processing at the beginning of the egress pipeline. Finally, a packet may be recirculated and sent from the end of the egress pipeline back to the beginning of the ingress pipeline (RECIRCULATE). PSA defines a set of externs that implement built-in packet processing primitives. *Counters* are used to count packets or bytes. *Registers* allow for stateful packet processing. *Meters* enable rate limiting, while *Checksum*, *InternetChecksum* and *Hash* are used to retrieve a checksum or hash value from a piece of data. *Packet Digest* can send user-defined data from the data plane to a control plane. *Action Profile* introduces a level of indirection to P4 tables and, thus, can reduce the size of the tables or the time to update them. Finally, *Action Selector* enables to dynamically select an action from a user-defined set, based on a hash value calculated from packet fields.

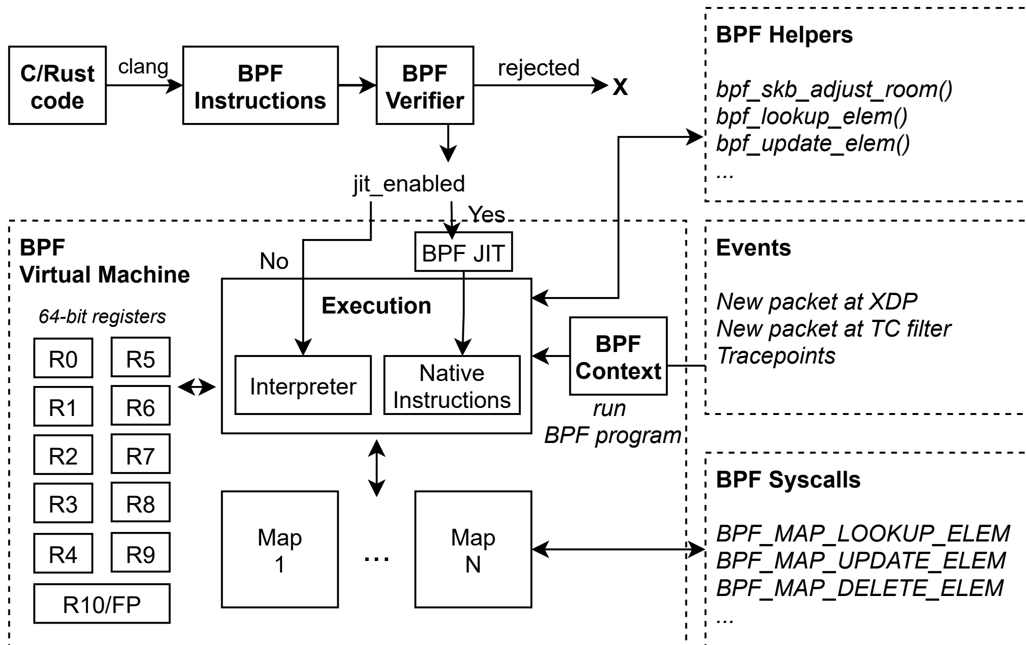


Figure 2.8: The eBPF subsystem.

At last, it is also worth mentioning NPLang [122], a main alternative to P4 supported by Broadcom switches. NPLang, however, have not been widely adopted and is only limited to Broadcom products. More importantly, NPLang does not provide a related control interface such as P4Runtime for P4.

2.2.2 Programming end-host network stack with eBPF

Unlike P4, eBPF [85] is a technology that has been designed for end hosts (commodity servers with general-purpose CPUs) and is not only limited to packet processing - it is also used for tracing, observability or profiling. Nonethelees, for the sake of this thesis, we focus on eBPF applications in the networking space. Moreover, eBPF as a programming abstraction is neither tightly coupled to data flow graphs nor Match-Action processing. It provides flexibility to users on how to design and implement a packet processing pipeline providing an assembly-like, Turing-complete language, but both data flow graphs and Match-Action processing are feasible.

eBPF can be seen as a framework for extending operating system's capabilities at runtime. It provides a general-purpose in-kernel virtual machine that can safely run eBPF programs (a set of eBPF instructions) within the kernel context on the operating system. The high-level overview of the eBPF subsystem is depicted in Figure 2.8. The eBPF virtual machine uses ten 64-bit, general-purpose registers and a single frame

pointer register to run eBPF instructions. Furthermore, eBPF provides two execution modes: eBPF instructions are either interpreted or Just-In-Time (JIT) compiled. eBPF instructions are typically compiled from a higher-level language such as C or Rust. eBPF instructions can be further loaded into the in-kernel eBPF virtual machine. Before that happens, instructions are first analyzed by the eBPF verifier - an in-kernel static analyzer that ensures the safe execution of an eBPF program and its successful termination within the kernel context. An eBPF program runs in the Run-to-Completion mode within a given BPF context and is triggered by a specific event (e.g., a packet reception). During its execution, an eBPF program can perform various operations in a BPF context. However, some complex tasks cannot just be defined by a set of eBPF basic instructions (e.g., they usually require privileged access to kernel data). To overcome this limitation, eBPF programs can use *BPF helpers* that implement more complex operations, such as checksum computation, packet buffer resizing or access to protected kernel memory areas. Each eBPF program runs in the per-BPF context scope. This means that the program's stack is not maintained across subsequent runs. To persist and share a state between multiple executions, *BPF maps* can be used. A BPF map is a key-value store that is declared at compile time and loaded into the kernel memory when an eBPF program is injected into the eBPF virtual machine. It can be accessed either from userspace, by invoking system calls, or from within an eBPF program, by calling BPF helpers. Among others, eBPF provides the *BPF hash map* that implements a hash table and provides constant lookup and update times, if no collision occurs, and a *BPF array map* that is organized as a fixed-size array providing the fastest possible lookup time.

Listing 2.5 shows a sample eBPF program written in C, with its corresponding bytecode form in Listing 2.6. The eBPF program implements simple packet forwarding based on destination IPv4 address. It firstly parses Ethernet and IPv4 headers, reads destination IPv4 address and performs lookup to the `routing` BPF map that is declared at the top of the snippet. A control plane is responsible for inserting BPF map entries that map a destination IPv4 address to an output port. The result of the BPF map lookup is an output port. Then, a packet is forwarded by calling `bpf_redirect()` BPF helper. Listing 2.6 depicts the eBPF program compiled to eBPF bytecode. As you can see eBPF executes assembly-like instructions as defined by eBPF Instruction Set. The eBPF bytecode can make use of all ten general-purpose registers and BPF helpers to

implement a packet processing algorithm.

```c
/* eBPF map declaration */
struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __type(key, int);
    __type(value, int);
    __uint(max_entries, 100);
    __uint(pinning, LIBBPF_PIN_BY_NAME);
} routing SEC(".maps");

/* eBPF program */
SEC("classifier/tc-ingress")
int tc_main(struct __sk_buff *ctx) {
    void *data_end = (void *)(long)ctx->data_end;
    void *data = (void *)(long)ctx->data;

    struct ethhdr *eth = data;
    if (eth + 1 > data_end) {
        return TC_ACT_SHOT;
    }

    struct iphdr *iph = data + sizeof(*eth);
    if (iph + 1 > data_end) {
        return TC_ACT_SHOT;
    }
    __be32 dst_ip = iph->daddr;

    int *out_port = bpf_map_lookup_elem(&routing, &dst_ip);
    if (!out_port) {
        return TC_ACT_SHOT;
    }
    return bpf_redirect(*out_port, 0);
}
```

Listing 2.5: Snippet of a BPF program implementing simple packet forwarding based on destination IPv4 address.

```
r0 = 2
r2 = *(u64 *)(r1 +80)
r1 = *(u64 *)(r1 +200)
r3 = r1
r3 += 14
if r3 > r2 goto pc+18
r3 = r1
r3 += 34
if r3 > r2 goto pc+15
r1 = *(u32 *)(r1 +30)
*(u32 *)(r10 -4) = r1
r2 = r10
r2 += -4
r1 = map[id:1]
call __htab_map_lookup_elem#141104
if r0 == 0x0 goto pc+1
r0 += 56
r1 = r0
r0 = 2
if r1 == 0x0 goto pc+3
r1 = *(u32 *)(r1 +0)
r2 = 0
call bpf_redirect#8525776
```

Listing 2.6: BPF bytecode for the C program in Listing 2.5.

As mentioned before, eBPF programs can run within various BPF contexts. For the sake of this thesis, we focus on TC and XDP hooks, BPF contexts related to packet processing within the Linux kernel, which has already been introduced in Section 2.1.2 and we extend their description in this section. The eXpress Data Path (XDP) hook is the lowest packet handling point in the network stack, where an eBPF program can be

triggered by the arrival of a packet. In fact, XDP is implemented by the NIC driver and the eBPF program is executed there before the allocation of the Linux socket buffer (*skb*) - a packet descriptor within the Linux network stack. Another network related BPF hook is Traffic Control (TC). On the RX side, eBPF programs attached to the TC hook are executed after the *skb* is allocated but before a packet reaches the upper layers of the network stack (the IP layer and higher). There are major differences between XDP and TC. First of all, XDP runs eBPF programs immediately after a packet is received by the NIC driver and avoids wasting cycles on some operations done by higher layers (e.g., *skb*'s allocation). Therefore, XDP usually offers higher throughput than TC. However, the capabilities of XDP are more limited [112]. TC and XDP also differ in the set of BPF helpers they support. Finally, the XDP hook is only available on ingress, while the TC layer provides both ingress (RX) and egress (TX) hook points.

The in-kernel implementation is the most adopted flavor of eBPF. It has also been selected as a packet processing engine for NIKSS. However, since P4rt-OVS processes packets in userspace with DPDK, it cannot use the in-kernel eBPF. Therefore, P4rt-OVS relies on a userspace BPF (uBPF) [102, 34], another flavor eBPF that implements eBPF VM and can execute eBPF programs in userspace. The uBPF implementation follows the same model as the in-kernel eBPF flavor and supports BPF maps, similar set of BPF helpers, can JIT-compile programs to x86-64, and includes a limited BPF verifier. Finally, it is worth noticing that there were also attempts to use eBPF for hardware-based packets processors. The hXDP project presents a solution to run XDP programs written in eBPF on FPGA cards using only a fraction of the available hardware resources while matching the performance of high-end CPUs [27].

2.3 Conclusions

P4rt-OVS and NIKSS take advantage of the multi-core CPU environment to scale throughput. P4rt-OVS leverages DPDK as a packet processing framework, while NIKSS makes use of eBPF/XDP. For both solutions, we measure throughput, latency distribution and CPU cycles per packet as described in Section 2.1.4. Both P4rt-OVS and NIKSS implement the exact and LPM packet classification. Moreover, NIKSS additionally implements ternary classification using Tuple Space Search algorithm and also employs

flow caching to optimize performance.

Furthermore, both P4rt-OVS and NIKSS use P4 over eBPF (or uBPF in case of P4rt-OVS) as a programming abstraction. P4 is a high-level language that can largely simplify the development of software switches, but it is less flexible than eBPF. On the other hand, eBPF provides a Turing-complete language for packet processing, but, as a low-level technology, it requires a strong development expertise and domain-specific knowledge about the OS network stack. This dissertation argue that the P4 technology can bring significant benefits to software switches. First and foremost, P4 provides an auto-generated control plane interface (P4Runtime [174]) and, thus, it provides interoperability with existing SDN controllers (such as ONOS [19]). This enables integrating software switches into the end-to-end programmable network and can lead to novel network applications. Second, P4 can lower a development complexity for network developers - they no longer need to satisfy the eBPF verifier, as it is a role of the P4 compiler to generate a code that is compatible with the underlying packet processing framework. Moreover, as shown in [160], the P4 language can significantly simplify the development of new network applications by reducing the lines of code compared to an equivalent C code.

Chapter 3

Need for a programmable SDN software switch

The main motivation for this thesis is to apply an approach of programmable data planes to software switches that are vital component of modern network virtualization systems [97]. In network virtualization, software switches play a role of the primary provider of virtual (overlay) network services for VMs or containers, while data center networks (often referred to as underlay networks) only provide IP connectivity between servers, where VMs/containers are running on. This approach allows overlay networks to be decoupled from their underlying networks, and by leveraging the flexibility of general purpose packet processors, virtual switches can provide VMs, their tenants, and administrators with logical network abstractions like Security Groups or ACLs, services (e.g., Firewall, NAT).

Network virtualization demands a capable software switch. The increasing complexity of virtual networking, emergence of network virtualization, and limitations of existing virtual switches led to novel solutions such as Open vSwitch (OVS) [148] that provide a certain degree of flexibility in defining packet processing pipelines using OpenFlow [110]. In fact, such a general-purpose software switches as OVS are currently widely deployed [182, 51, 167]. However, the emergence of 5G/6G networks, machine learning, artificial intelligence, augmented and virtual reality puts new demands on network virtualization platforms. First of all, software switches might require frequent updates to support new protocols or encapsulation techniques for overlay networking, new network telemetry mechanisms or security applications. Moreover, a network

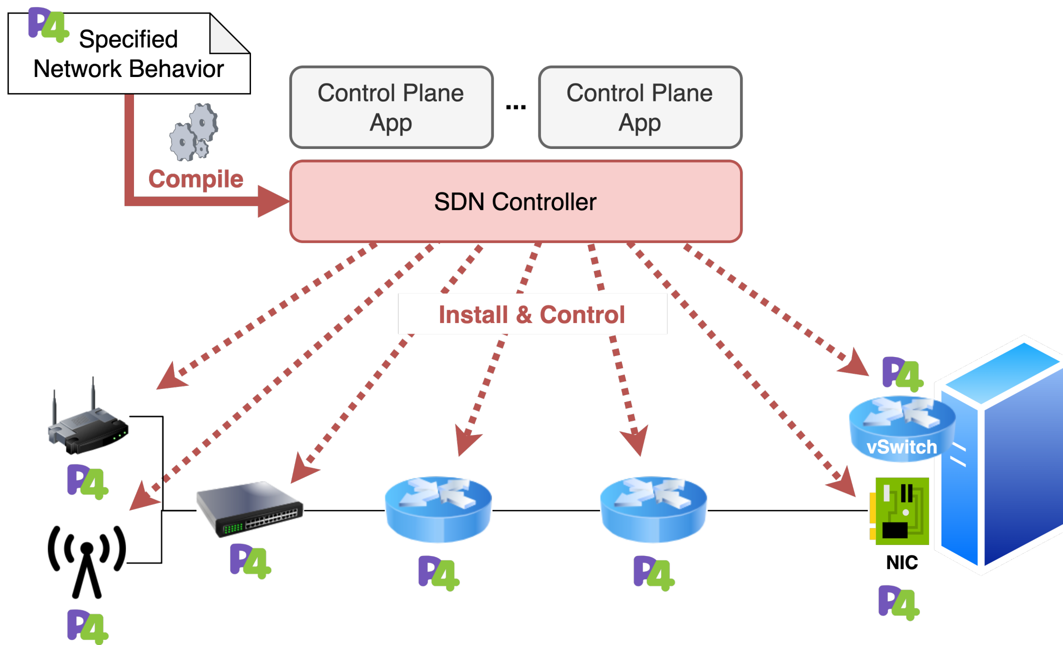


Figure 3.1: Vision of end-to-end programmable network.

infrastructure owner may need to introduce a custom protocol extensions to optimize their systems. It should be possible to use a high-level programming language such as P4 to dynamically customize a packet processing pipeline without an intimate knowledge about software switching technology, so that network owners can smoothly introduce new features, fix network issues, perform software upgrades to the data plane devices or customize their network protocols to meet customers' requirements.

Furthermore, real benefits of programmable software switches are more demonstrable once we put them in the context of an end-to-end programmable network (Figure 3.1), where each device on the packet path (including WiFi access points, radio base stations, aggregation switches, core routers, NICs and software swithes) exposes programmable data plane and the network behavior is entirely defined in software. Expanding a programmable SDN domain to end hosts makes networks programmable in an end-to-end fashion using a common programming language (P4) and a unified control plane interface (P4Runtime). This approach enables new use cases that would not have been possible (or would be difficult to implement in a timely manner) without P4-programmable software switches. To name a few, a P4-programmable software switch could provide an early classification for end-to-end network slicing and QoS at the container or VM interface that could be further used by each network element on

the packet path to enforce isolation or QoS policy. Moreover, coordination between all programmable network devices could lead to enhanced, end-to-end network visibility (e.g., with In-Band Network Telemetry [141]) to pinpoint a network element contributing to increased latency or detect packet drops or other anomalies. The end-to-end network visibility also makes a concept of verifiable networks feasible [57]. Looking ahead, the end-to-end programmable network could also enable deploying completely new networking paradigms such as Named Data Networking [116]. Finally, both underlay and overlay networking can be controlled by the same SDN controller giving an opportunity to eliminate network overheads and simplify network design by flattening underlay and overlay networks.

It is worth motivating a P4-programmable software switch through case studies that prove usability of the P4-capable software datapath.

Case study #1: End-to-end network visibility. Let us start from already-mentioned example - end-to-end network visibility with In-Band Network Telemetry (INT). INT [141] is a promising technology that has been enabled by P4-programmable switches. It defines an open framework that uses data plane elements to produce per-packet fine-grained telemetry information such as input and output ports, per-hop packet latency, queue congestion status or, if a packet is dropped, a reason code (e.g., TTL zero, table miss, etc.). It is achieved without any intervention of network control plane - telemetry reports are generated and collected by network data plane. In the INT architecture, each element on the packet path collects information about how packet has been processed internally (e.g., processing latency, input/output port, etc.). Then, depending on the INT mode, the telemetry information is either inserted to a packet as an INT header or sent to an INT collector via an out-of-band channel. So far, INT has been mainly implemented for P4 programmable hardware switches and integrated into data center underlay networks or wide-area networks. However, observing packets as they travel through data center switches is knowing only half of the story and it does not provide a full network visibility. Nowadays, a journey of a packet starts inside VM or container at end hosts. Therefore, observing hardware switches only is not enough. Forwarding scenarios in modern virtualized data centers are complex. Packets are encapsulated using different tunneling protocols for different portions of the path, they travel through end hosts multiple times because of the network services

such as load balancers, and they are transformed using NAT in the middle of the path. Many things can go wrong on the end hosts too. For example, packets can be dropped because of table miss in the packet processing pipeline of a software switch, because of an explicit network policy applied by a network administrator, or because of a bug in a packet processing pipeline of a software datapath. Similarly, packets can be delayed because of a congested end-host queue, or because the end-host CPU is overloaded with other tasks. INT provides a way to quickly detect and react to such anomalies. To obtain visibility on these and other anomalies, INT must be supported by software switches as well. However, none of existing software datapaths implement INT. There are research projects investigating an approach to add the INT support into the OVS pipeline [64], but it has not so far been accepted and merged by OVS maintainers. This is where P4-programmable software switches bring the added value. INT extensions can be implemented in a P4 program and compiled to the on-CPU packet processing pipeline. Network owners can implement INT on their own, without waiting for the INT framework to be implemented by software switches. Once implemented, the end-to-end network visibility with INT can be leveraged to validate Service Level Agreements (SLA) for QoS metrics and to provide faster network troubleshooting. More on the Host-INT use case can also be found in our paper [127].

Case study #2: Scalable large flows' detection. Real-world measurements [92] show that a large fraction of datacenter traffic is carried in a small fraction of flows. These flows that transfer significant amount of data are called large (or elephant) flows. Elephant flow detection is a significant research problem addressed by many papers e.g., [17, 67, 186]. This problem may be important due to several reasons. For example, it can aid QoS-aware traffic engineering decisions. Moreover, limited resources on hardware switches cause the need for hybrid software/hardware flow tables [184], where a subset of traffic is handled by switch CPU instead of hardware packets processor (e.g., ASIC). In such a case, it is important to handle elephant flows directly in the ASIC. Otherwise, the switch CPU would not manage to process such a volume of traffic, leading to packet drops and overall performance loss. Most of proposed detection mechanisms incur high monitoring overheads, consume significant switch resources, and/or have long detection times. These methods typically use periodic polling of statistics from switches, streaming techniques like sampling, or even L7

application-level modifications. Both per-flow statistics-based and sampling-based approaches can be implemented either on the switch CPU or in the controller - both methods have notable drawbacks. The switch CPU is typically not powerful enough to perform complex flows' analysis. On the other hand, sending packet samples or statistics to a remote controller causes longer detection times. Furthermore, an approach based on per-flow statistics is considered to be not scalable enough for large networks, while sampling-based methods introduce overheads to the detection process. In [39] authors presented a novel, scalable solution to large flows' detection problem. Instead of performing detection process in the network, they propose to run detection jobs at the end hosts, before a packet leaves a server. They implement a custom OS kernel extension that monitors socket buffers to detect elephant flows leaving a host. Once an elephant flow is identified, information is signalled to network devices by marking IP packets with a custom DSCP field value. It turns out that this brilliant idea is difficult to implement in real-world deployments. First of all, implementators are rarely willing to use custom OS extensions that have not been verified and accepted by the Linux community. It would be optimal to run the detection process inside a packet processing pipeline of a software switch. However, none of software switches provide elephant flows' detection mechanisms as it is an advanced feature, which is not typical for software datapaths. Furthermore, the DSCP field is usually used for QoS signalling. Therefore, it is not always possible to use it and it is hard to find other field to signal elephant flow existence to the network. However, an end-to-end programmable network with P4-programmable software switches at end hosts can change the status quo. With a P4-programmable software switch at end hosts, a network owner can extend the packet processing pipeline of the software switch and implement a detection method. A naive approach would be to inspect outgoing packets, check their size (packet length) and persist aggregated statistics in a stateful object such as a P4 Register. This is straightforward to implement with a P4-programmable software switch. Moreover, a P4 software switch provides a way to insert custom packet headers into a packet. This can be used to carry information about elephant flows. As far as other devices in a network are also P4-programmable, all elements on the packet path are able to recognize and read the custom header. Moreover, P4-programmable network devices can be managed by a single SDN controller making deployment of

the elephant flows' detection framework more smooth. Finally, the custom header may also carry some additional meta-information about a flow to provide an input to the decision algorithm for QoS-aware traffic engineering, hybrid software/hardware flow table offloading or other network services.

The above case studies clearly show benefits that a P4-programmable datapath can bring, particularly when deployed as part of an end-to-end programmable SDN domain. There are a lot more practical P4 applications for P4 software switches that we briefly present in the next section.

3.1 Use cases

Both P4rt-OVS and NIKSS have been built as the enablers of various P4 use cases that are well-summarized in [106] and [72]. We also briefly present some of these P4 use cases to show the full spectrum of P4 applications for which P4rt-OVS and NIKSS can be used. All the undermentioned examples show a great potential of the P4 technology and we believe that P4rt-OVS and NIKSS can successfully become enablers for future innovations in programmable networks.

Advanced network monitoring. An efficient network monitoring solution usually requires an advanced mechanisms to count flows, detect anomalies or sample packets. P4rt-OVS and NIKSS can be used on hypervisor servers to implement the state-of-the-art solutions that rely on stateful packet processing using P4 registers: SpreadSketch [173] to identify hosts with a large number of distinct connections using P4 registers, TurboFlow [168] to generate flow records using microflow aggregation, or ElasticSketch [188] to implement the measurement solution that is adaptive to available bandwidth, packet rate and flow size distribution.

Stateful firewall in data plane. Recent firewalls perform stateful analysis of packets to keep track of network connections. Existing software switches such as OVS already provide an integration with the Linux connection tracking module that enables stateful processing of network connections. However, P4rt-OVS and NIKSS can enable more advanced or customized solutions for stateful firewalls such as firewall for 5G networks [153] or P4Guard [42], a P4-based, configurable firewall.

Heavy hitter detection. Heavy hitters (traffic that consumes a lot of bandwidth

for a short period of time) often remain undetected by conventional solutions due to their specific traffic characteristics. P4rt-OVS and NIKSS could be used at end hosts to implement novel P4-based solutions such as network-wide heavy hitter detection [71, 45] or HashPipe [166], a heavy-hitter detection entirely in the data plane.

In-network DDoS mitigation. The P4 language can implement various strategies for DDoS detection and mitigation [7, 145]. Therefore, P4rt-OVS and NIKSS can be used to extend the end-host networking with those strategies. Moreover, offloading anti-DDoS applications from separate boxes (virtual machines) to the virtual switches inside the data center may also significantly improve the overall performance of network security functions [87, 33].

Traffic management and congestion control. First of all, P4 switches enable novel load-balancing schemes that improve traditional mechanisms. To name a few, SHELL [149] combines IPv6 Segment Routing with stateless flow recording to implement application-aware load balancer. HULA [93] and MP-HULA [18] propose scalable load-balancing with the mutlipath TCP support using programmable data planes. Other papers [189, 76, 150] leverages P4 switches to improve a classic ECMP-based multipath routing. Furthermore, P4 switches can improve congestion control in the networks. For instance, QoSTCP [35], TCP-INT [81] and [103] take advantage of measurement reports generated by P4 switches to optimize the TCP performance. Finally, P4 can improve traffic scheduling mechanisms. Authors in [31, 162] show how to use different P4 constructs such as Counters, Meters or Registers to provide fair queuing or bandwidth sharing.

Custom routing and switching schemes. P4 also enables innovation in routing and switching domains. From event-based packet processing in P4 [79] up to P4-enabled source routing [104] and multicast [161]. Importantly, P4 enables to prototype and deploy novel paradigms for data forwarding such as Named Data Networks [116] and routing with packet subscriptions [88].

VNF offloading. P4 switches can be used to offload packet processing tasks from virtual machines and, consequently, accelerate Virtual Network Functions. For instance, researchers presented P4-based prototypes of Broadband Network Gateway (BNG) [99, 128] or 4G/5G gateways [107, 159]. Moreover, several orchestration frameworks for P4-based network functions have been proposed [133, 117, 73].

In-network computations. One of remarkable P4 applications is offloading of computations, that were typically performed by endpoints, to programmable network switches. P4xos [41] accelerates the Paxos consensus protocol by implementing it directly in the forwarding plane with P4. In [187] authors show how to leverage P4-programmable switches to offload machine learning classification. P4rt-OVS and NIKSS can be certainly used in these use cases as well. Furthermore, BMC [60] uses eBPF as an in-kernel cache to accelerate Memcached, an in-memory key-value store. However, systems like BMC can be successfully implemented with NIKSS too.

3.2 Requirements for programmable software switches

Based on the analysis of the state of the art, we have selected the following requirements that a programmable software datapath should fulfill:

- **High-level and fully-fledged programming abstraction.** Developing new network features (e.g., adding new protocol extensions, packet headers or packet processing mechanism) should not require a deep knowledge of the underlying CPU architecture, system design or packet processing framework being used. A programmable software datapath should provide a high-level programming abstraction. At the time of writing this dissertation, P4 [23] is the best candidate providing high-level and declarative language to define packet processing behavior.

- **Performance.** A programmable software datapath should provide high enough performance to consider it as a reasonable choice from the cost-per-bit[1] perspective. It can be assumed that the throughput higher than 1 MPPS per a single CPU core for real-world network functions (such as IPv4 router or 5G User Plane Function) should be sufficient for a programmable software datapath to compete with other existing solutions.

- **Runtime programmability.** A highly programmable software datapath should offer the possibility to change the packet processing pipeline at runtime without need for recompilation or system reboot.

[1] A cost in terms of CPU/memory resources that has to be paid to handle a given volume of traffic.

- **Operability.** It is defined as a general ease of use of a solution. This includes the configuration overhead to deploy and keep running a software switch, resource overhead (e.g., extra CPU usage) as well as the ability to use well-known and proven in practice management tools (e.g., `ethtool`, `ip link`, `tcpdump`, etc.) and APIs to configure a software datapath. A programmable software datapath should ideally be integrated with the Linux configuration tools and require no additional resources or dependencies.

Combining all these design principles in a single solution is a key challenge addressed in this dissertation. In particular, an SDN software switch is expected to provide high performance, while keeping a high degree of programmability, flexibility and general-purpose usage. Therefore, the main goal of the research work presented in this dissertation is to obtain enhanced programmability at runtime for software switches, without sacrificing performance and operability.

3.3 Existing solutions

There have been many software switching solutions proposed so far. Several Kubernetes plugins [1, 5, 4] base on *iptables* to implement the Kubernetes network model. However, *iptables* is not programmable and efficient enough [29]. A few solutions are based on eBPF. bpf-iptables [113] is a re-implementation of *iptables* in eBPF. It improves the *iptables* performance but it suffers from the same programmability issues as the classic implementation. Both Calico [2] and Cilium [3] eBPF datapaths provide a high performance, programmable and highly operable Kubernetes datapath. However, eBPF programming is not easy as it requires an intimate knowledge of a restricted eBPF programming model, eBPF internals, and low-level details about the Linux network stack.

Over the years, researchers and engineers have proposed various software datapaths. VALE [155] is a L2 software switch based on the Netmap packet I/O framework, but it provides neither a high-level programming abstraction nor runtime programmability. Cuckoo Switch [195], which is based on DPDK and implements an efficient hash-based classification algorithm, falls into the same category. Some solutions are specialized for NFV such as Netbricks [143] and NetVM [77]. Furthermore, Snabb

[63] is a modular, re-programmable software switch using a custom vhost-user packet I/O framework, but LuaJIT as its programming abstraction is not high-level enough. Many of software switches use data flow graph abstraction, with Click [96] being a precursor. ClickOS [108] is an extension of Click for virtualized environments (NFV). FastClick [16] integrates both DPDK and Netmap in Click. ClickNF [59] extends Click with the support for complex L2 to L7 network functions. Both BESS [68] and VPP [15] are modern, DPDK-based software switches that borrow from the Click principles and apply batch packet processing. However, according to [194] VPP lacks modular design and, thus, it is less programmable than BESS. It is worth noticing that all DPDK-based solutions may suffer from worse operability. Moreover, writing modules for BESS still requires low-level programming skills in the C language as well as an internal BESS architecture, but the data flow graph abstraction allows to easily stitch them together and dynamically manage. There are also several OpenFlow software switches such as Lagopus [152] (based on DPDK) and ofsoftswitch13 (often referred to as CPqD or BOFUSS) [53]. Most likely the most popular software switch, that is also used in production [167, 182, 51], is OVS [148]. It implements the OpenFlow model and uses the Tuple Space Search algorithm with the multi-level flow table caching architecture. Its performance depends on the datapath choice among three possible options: 1) an in-kernel datapath using custom Linux kernel module and a userspace datapath based on 2) DPDK or 3) AF_XDP. The two last flavors are able to provide high performance. Nevertheless, OVS, as an OpenFlow switch, suffers from the limitations of the OpenFlow model mentioned in Chapter 2. Linux bridge has been considered as an OVS alternative, but it is a simple Ethernet software switch and does not meet requirements for neither performance nor runtime programmability and programming abstractions. Three other OVS extensions [34, 87, 176] try to increase a degree of programmability, but none of them meets the aforementioned requirements. SoftFlow [87] allows to execute arbitrary network functions as OVS actions, but is not programmable at runtime. Oko [34] integrates BPF with OVS and can be extended at runtime, but it is limited only to programmable packet filters. OVS-eBPF [176] applies eBPF to OVS to provide runtime extensibility, but eBPF turned out to be too limited to implement the OpenFlow model that fully relies on the wildcard packet classification [177]. Anyway, OVS-eBPF would still suffer from OpenFlow limitations. Finally, the Virtual Filtering Platform (VFP) [54]

goes beyond the OVS principles and proposes a kernel-based software switch leveraging a specialized slowpath classifier with a single exact-match flow cache. Nonetheless, VFP is a protocol-dependent software switch.

There is also a group of state-of-the-art P4-programmable switches. BMv2 [136] is a reference implementation of a P4 software switch, but it lacks performance [135] and it is not meant to be a production-grade solution. P4-eBPF [138] and p4c-xdp [156] target eBPF with TC and XDP hooks, respectively. These P4-to-eBPF compilers come with their own custom P4 architecture model with limited capabilities. [138] only supports packet filtering while [156] has no support for both ingress and egress pipelines. Other P4 compiler backends are based on DPDK [160, 137, 183]. In [160], PISCES has been proposed as a protocol-independent software switch based on OVS, whose behavior can be easily customized. PISCES enables custom protocol specification in the P4 language with negligible performance overhead and without the need for direct modifications to the switch codebase. PISCES, however, has two main drawbacks. First, it requires re-compilation every time the P4 program is changed. Such a design does not allow injecting custom, vendor-specific data plane applications at runtime. Second, PISCES implements a limited version of P4 (e.g., does not provide mechanisms to implement stateful data plane programs) and, thus, it does not provide a fully-fledged programming abstraction. Worthy of noting is P4-DPDK [137] that aims at implementing PSA for DPDK and makes use of the DPDK SWX pipeline to dynamically load P4 programs [48]. It provides a rich programming abstraction due to the use of P4 and the PSA model, enough performance and runtime programmability. The use of DPDK might affect the operability though.

3.4 Conclusions

A programmable software switch should fulfil the following requirements: high-level programming abstraction, performance, runtime programmability and operability. Table 3.1 presents a comparison of the most relevant state-of-the-art programmable switches with emphasis on how they meet requirements defined in Section 3.2. Among all the aforementioned software switches there are five solutions that offer a reasonable compromise between programming abstractions, performance, runtime programma-

Software switch	High-level programming abstraction	Performance	Runtime programmability	Operability
BMv2	Yes	Low	Yes	Good enough
Native eBPF/XDP	No	High (but worse than DPDK)	Yes	High
PISCES	Yes (but uses the depracated version of P4)	High	No (needs re-compilation)	Low (due to DPDK)
P4-DPDK	Yes	High	Yes	Low (due to DPDK)
BESS	No (requires C expertise to develop new packet processing modules)	High	Yes	Low (due to DPDK)
VPP	No (requires C expertise to develop new packet processing modules)	Very High	Yes	Low (due to DPDK)
Open vSwitch	No (OpenFlow)	Depends on datapath choice (Medium for kernel datapath, High for AF_XDP and DPDK)	No	Depends on datapath choice (High for kernel and AF_XDP datapath, Low for DPDK)
P4rt-OVS	Yes (OpenFlow + P4)	High	Yes	Low (due to DPDK)
NIKSS	Yes	High with XDP acceleration, Medium with TC-based datapath	Yes	High

Table 3.1: Comparison of the most relevant state-of-the-art programmable software switches

bility and operability: native eBPF/XDP, OVS, BESS, VPP and P4-DPDK. It is worth noticing that the solutions proposed in this dissertation, P4rt-OVS and NIKSS, are not expected to replace them, but both P4rt-OVS and NIKSS should be treated as complementary projects to these projects with different trade-offs between programming abstractions, performance, runtime programmability and operability. The final choice between these packet processing systems should depend on a use case and functional or performance requirements.

Chapter 4

P4rt-OVS: Programming Protocol-Independent, Runtime Extensions for Open vSwitch with P4

A software switch, such as Open vSwitch (OVS) [148], plays the role of a hypervisor switch forwarding packets to and from VMs or containers. Hypervisor switches implement a set of network protocols to enable, among others, multi-tenant network virtualization through overlay tunneling, ACL, and QoS [97]. Moreover, as network virtualization systems are evolving, more complex middlebox functions (such as stateful firewalls or NATs with connection tracking) are being implemented inside software switches to offload VM-based network functions, while preserving their flexibility and performance [87, 111].

Although OVS provides some degree of programmability through the use of the OpenFlow forwarding model [110], it is still difficult to extend its packet processing pipeline. Developing a new network feature requires domain-specific knowledge of network protocol's design, low-level C skills, and familiarity with the large codebase of the software switch. Moreover, OVS adopts the stateless forwarding model of OpenFlow, which prevents the implementation of stateful use cases, including many security services.

In this chapter, we present the design and implementation of P4rt-OVS, which allows programming protocol-independent, runtime extensions for a software switch with P4. P4rt-OVS [125] is an original extension of OVS, designed around the following

design principles:

Enable runtime programmability. We design our solution to be programmable at runtime. Therefore, we leverage the Berkeley Packet Filter (BPF) [85] to provide a runtime extensibility mechanism for OVS.

Provide performance for NFV. OVS can be used as a virtual switch in NFV systems. To meet the performance requirements of NFV, the OVS datapath has been ported to DPDK [83]. Therefore, to achieve high performance, we have built P4rt-OVS on top of OVS-DPDK. This decision also implies the use of the userspace BPF implementation.

Support stateful operations. Many network functions require access to the state of connections to fulfill their goal. As the P4 language provides a way to save custom data structures in the switch's memory, we treat the support for stateful operations as an added value for OVS.

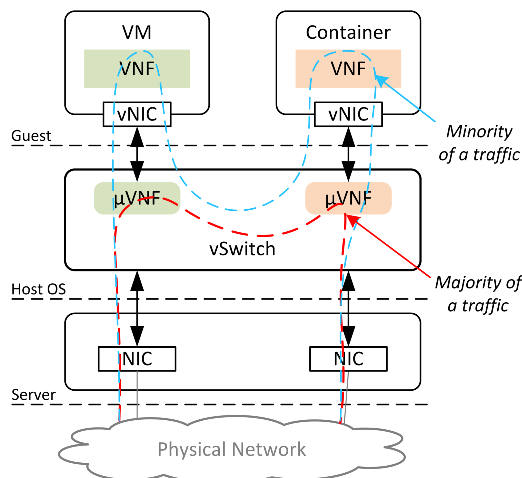


Figure 4.1: The concept of offloading packet processing tasks from the Virtual Network Function (VNF) to a virtual switch.

4.1 Motivation

The primary motivation to build P4rt-OVS is to enable offloading packet processing tasks from a Virtual Network Function (VNF) to a virtual switch in the NFV environment. The high-level concept of the VNF offloading is depicted in Figure 4.1. In such a scenario, a VNF intercepts packets destined to it, but it may install a dedicated packet processing module in a virtual switch, so that the packet processing may be offloaded

and subsequent packets will be handled directly by the virtual switch without involving the VNF. It may result in a noticeable increase of the VNF's performance. Moreover, P4rt-OVS can be used to implement various P4 use cases described in Chapter 3.

4.2 Design and implementation

This section provides an overview of the design of P4rt-OVS and description of internal mechanisms used to implement the design, including modifications to the OVS architecture, design of P4 to userspace BPF compiler and P4runtime-based control plane.

4.2.1 Open vSwitch design

The P4rt-OVS prototype enables the upgrade and customization of the OVS network protocol stack at runtime. This is achieved by integrating P4, as a high-level data plane programming language, and BPF, which provides the runtime extensibility mechanism, with OVS.

In OVS, two major components participate in packet processing. The *datapath* is the main component responsible for packet forwarding and is also referred to as the *fastpath*. In the case of OVS-DPDK, the fastpath component is implemented in userspace. The second component is *ovs-vswitchd*, a userspace daemon, also called the *slowpath*. It tells the fastpath how to forward incoming packets based on flow rules in Match-Action tables. Finally, *ovs-vswitchd* also exposes the OpenFlow interface to external SDN controllers. OVS provides a wide range of OpenFlow actions to modify packets in the fastpath.

When it comes to packet tunneling, OVS uses a concept of *packet's recirculation*. When a packet arrives at the switch, only the outer header is extracted and known to the datapath. Therefore, if there are nested packet's headers, OVS needs to recirculate the packet, i.e. send it back to the beginning of the datapath processing to extract inner headers and allow for further processing.

OVS implements flow caching mechanisms to prevent sending packets to the slowpath for every packet in a flow. As illustrated in Figure 4.2, whereas the slowpath implements a Tuple Space Search classifier [169] for each OpenFlow table, the fastpath only implements caches. There are two levels of caches. The first level consists of

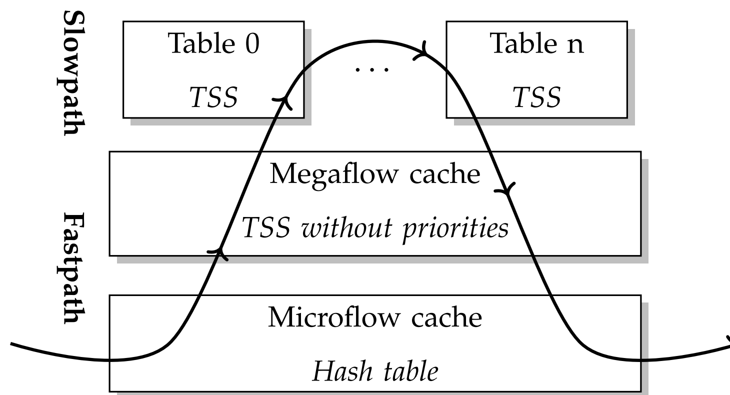


Figure 4.2: Open vSwitch's two-level caching architecture.

a simple hash table with exact-match rules for recent flows. The second-level cache (so-called megaflow cache) stores wildcarded rules in a simplified Tuple Space Search classifier. When a first packet is received, it will go through the OpenFlow tables in the slowpath. After the first packet has been processed, the corresponding entry is installed in the megaflow cache. Then, subsequent packets matching the megaflow cache will not need to go through the slowpath again. This second-level cache is therefore more general than the exact-match cache since it can match on all subsequent packets with the same packet fields and not only subsequent packets with the exact same headers (i.e. other packet headers are wildcarded).

4.2.2 P4rt-OVS overview

Figure 4.3 depicts the proposed extensions to the OVS architecture that enable programming the packet processing pipeline at runtime. It also shows the P4rt-OVS programming workflow. First of all, the userspace datapath of OVS has been extended with an additional BPF subsystem. This enables the injection of packet forwarding programs at runtime and their integration with the OVS forwarding pipeline. The BPF subsystem consumes bytecode that implements the packet processing model. In the framework, the P4-to-uBPF compiler is used to generate bytecode from the P4 program. Moreover, apart from OpenFlow, a P4Runtime abstraction layer (P4RT-AL) has been built to enable the integration of P4Runtime-compliant SDN controllers with P4rt-OVS. It results in a hybrid approach, which can be controlled by both OpenFlow and P4Runtime control protocols.

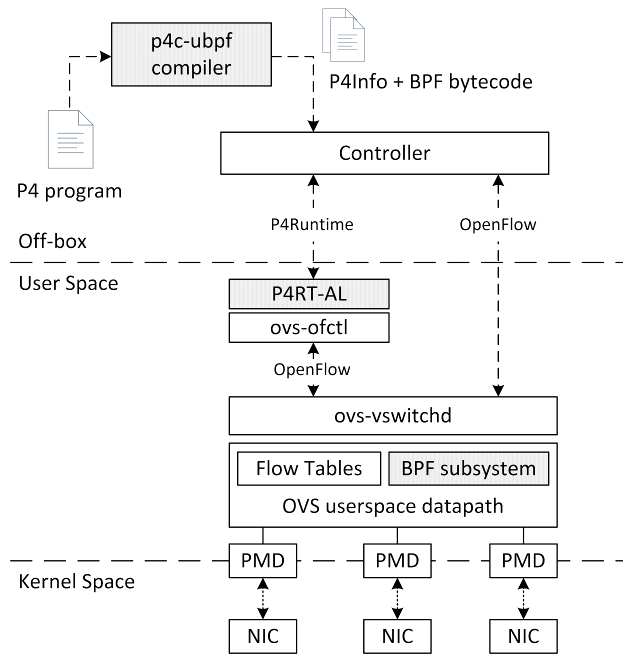


Figure 4.3: The overall architecture of P4rt-OVS

The P4rt-OVS programming workflow assumes that P4rt-OVS has been compiled and run beforehand and is as follows. In the first step, a programmer designs and creates a P4 program implementing specific network features. According to the $P4_{16}$ specification, P4rt-OVS also provides a P4 architecture model, which guides the programmer on how to write a data plane program for P4rt-OVS. Next, the user generates BPF bytecode using the P4-to-uBPF compiler and, optionally, the P4Info metadata file to be used by the control plane as a contract describing the data plane implementation. Then, the BPF bytecode is injected in the OVS forwarding pipeline by either the SDN controller using OpenFlow or P4Runtime, or is injected through a local CLI. The data plane program appears in the switch as a BPF program with a new identifier. The last step for the user is to define an OpenFlow flow rule that will invoke the BPF program. The user can also configure BPF map entries for the BPF program before configuring a flow rule or when the BPF program is already in action. To modify a data plane program a user can create a new BPF program, inject it with a new identifier and modify flow rules to point to the new BPF program. P4rt-OVS does not replace the entire datapath, while installing a new P4 program. In fact, the OpenFlow datapath is not affected. This feature of P4rt-OVS allows the user to inject new data plane features without traffic interruption if a seamless modification strategy is used.

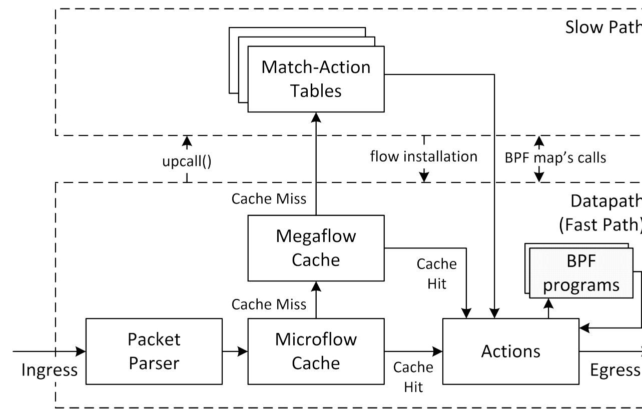


Figure 4.4: The BPF subsystem and forwarding model of OVS

4.2.3 Modifications to OVS

Four modifications to OVS have been made to enable programming protocol-independent, runtime extensions using P4.

The BPF subsystem for OVS. The first design principle was to provide the runtime extensibility mechanism to OVS. We extended OVS with a new subsystem based on the userspace BPF (uBPF) implementation (see Chapter 2). P4rt-OVS retains the BPF infrastructure (abstract machine, BPF verifier, and set of external functions) of Oko [34], but it also introduces several modifications needed to implement certain P4 capabilities. Unlike the Oko [34] approach, we propose to pass the whole `dp_packet` structure, which represents a packet inside the userspace datapath. The `dp_packet` structure contains various information about a packet and not all of them are needed by the BPF program. However, such a design is necessary to implement arbitrary packet tunneling, which we will explain further in this subsection. The BPF program takes the `dp_packet` structure as an argument that does not represent a packet data directly. Hence, for packet data to be processed by the BPF program, a new uBPF helper, `ubpf_packet_data()`, has been implemented to retrieve a packet's data from the `dp_packet` structure. Such a design requires a modification to the uBPF verifier to prohibit illegitimate accesses to the `dp_packet` structure.

Programmable actions. The implementation of the Oko switch [34] assumes that BPF programs are used as an enhanced filtering program to match packets. With the design proposed by Oko, a user cannot modify packets, neither to write a packet field nor to encapsulate them inside a uBPF program. In contrast, P4rt-OVS is de-

signed to allow a uBPF program to modify a packet. Therefore, P4rt-OVS executes BPF programs as an OpenFlow action by implementing the new action type *OVS_ACTION_ATTR_EXECUTE_PROG*. Figure 4.4 presents this new design for the OVS forwarding model. In addition, contrary to Oko, because we integrate BPF programs as OpenFlow actions, we do not need to extend the flow caching architecture. Therefore, BPF can be integrated without significant modifications to the OVS forwarding model, and all the actions defined in the P4 program are executed in the fastpath.

However, with the introduction of programmable read-write actions, we face a new problem when dealing with multi-tables pipelines. If the action of the first table is now a BPF program, the packet may have its destination IP address rewritten to any value when it moves from table 1 to table 2. The rule matched in the second table will therefore depend on the actions of the BPF programs. This dependency prevents from aggregation rules from multiple tables in a single rule in the second-level cache.

One, rather inefficient, solution is then to recirculate the packet after each BPF program execution. Because of recirculations, rules are not aggregated in the cache, thus ensuring correct behavior, at the cost of significant additional overhead.

Another solution, implemented in [87], requires the BPF program's developer to request recirculations when appropriate, that is when a packet is modified in such a way that it impacts forwarding through the OpenFlow pipeline. This approach however requires that BPF developers maintain a detailed understanding of the OpenFlow pipeline of the switch at any time. Failure to consider interactions between the BPF program and the pipeline may result in incorrect actions being executed on packets.

P4rt-OVS requires that BPF programs are always executed as the last action of the last table, thereby avoiding the drawbacks of previous solutions. This constraint is what allows P4rt-OVS to retain the high-performance caching mechanism of OVS while enabling programmable read-write actions. On the other hand, it increases the complexity of the development of the P4rt-OVS pipeline as a developer must take this constraint into account during the design phase. It does not limit the number of feasible use cases though.

Support for tunneling. An inseparable feature of the P4 language is support for arbitrary packet encapsulation. To implement this support, the BPF programs must be able to change the length of packets, to add and remove headers. The packet

adjustment requires access to the `dp_packet` structure from BPF. Therefore, to support arbitrary tunneling, we added a new uBPF helper function, `ubpf_adjust_head(int offset)`, which has access to `dp_packet` and adjusts the packet's length according to the *offset* value.

Exposing the interface to the BPF subsystem. To use our new BPF subsystem, it has to be exposed to the control plane. To that end, additional OpenFlow messages have been implemented so that they can be used to manage BPF programs and their maps. These messages are as follows.

- *LOAD_BPF_PROG* to install a new BPF program.
- *UNLOAD_BPF_PROG* to remove an existing BPF program with a given identifier.
- *SHOW_BPF_PROG* to list all BPF programs or show the information about a given BPF program.
- *UPDATE_BPF_MAP* to add or update an existing entry of the BPF map.
- *DUMP_BPF_MAP* to dump the content of the BPF map of a given BPF program.
- *DELETE_BPF_MAP* to remove an entry with a given key from the BPF map.

4.2.4 P4 to uBPF compiler

In compliance with the $P4_{16}$ compiler's design [28] the P4-to-uBPF compiler implements a new backend, uBPF, for the compiler's frontend [139]. The P4-to-uBPF compiler generates a target-specific, restricted C code, which is compatible with uBPF and can be further compiled to the BPF bytecode using the *Clang* compiler. This one intermediate stage allows to leverage the existing compiler optimizations implemented by *Clang*.

Along with the implementation of the P4-to-uBPF compiler, we have designed the architecture model for P4rt-OVS, which in particular, describes the forwarding model of the uBPF program. The forwarding model is tailored to the architecture of OVS and is depicted in Figure 4.5. In the architecture model, P4rt-OVS prevents P4 programs from forwarding packets themselves; it is therefore still the responsibility of OVS to determine an output port for a packet. As a result, the P4/BPF program can filter, inspect or modify a packet, but the only forwarding decision it can make is to decide

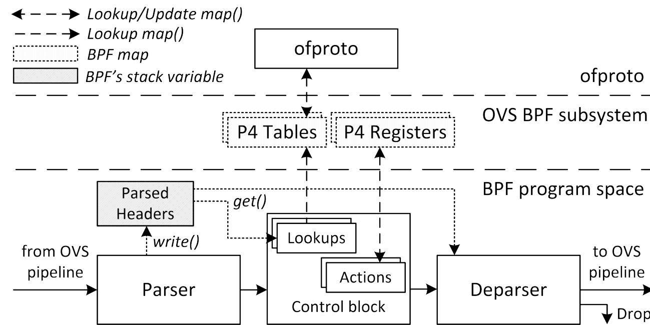


Figure 4.5: The forwarding model of the BPF program generated from the P4 language

whether to drop a packet or send it back to the OVS forwarding pipeline. This decision is due to the usage model of P4 in P4rt-OVS. Originally, P4 is designed to describe the whole functionality of a switch. In our case, P4 is used only to describe a specific part of a switch, a custom action. Thus, other OVS actions can still be used to perform packet forwarding and the P4 language is just utilized to play its role: providing a high-level language to process packets' headers.

The forwarding model of the BPF program generated from the P4 language consists of three packet processing blocks: a parser, control block, and deparser.

Parser. A parser is responsible for reading a packet's headers and copying them to the `Headers_t` structure. The parser reads each header field by field by loading bits and shifting or masking bits, if necessary. The output of a parsing stage is the `Headers_t` structure filled with a packet headers' data. Additionally, for each header in the `Headers_t` structure, a validity bit is associated. According to [142], if a header has been parsed correctly, the validity bit is set. The validity bit is further used to perform operations on headers (encapsulation or decapsulation) in a control block and deparser.

Control block. A P4 control block is composed of a set of Match-Action tables implementing a packet processing pipeline. In the P4rt-OVS design, to implement a packet processing pipeline, a programmer can use read-only Match-Action tables (for stateless network functions) or registers with read-write permissions to implement stateful network applications. P4rt-OVS provides two uBPF helpers, `ubpf_map_lookup()` and `ubpf_map_update()`, to read from the BPF map (table or register) and write to the BPF map (only registers), respectively. Since P4 tables and registers are implemented as BPF hashmaps, both `ubpf_map_lookup()` and `ubpf_map_update()` have an

average-case complexity of O(1). In particular, in a control block, a P4 program can encapsulate or decapsulate a packet by validating (`setValid()` operation) or invalidating (`setInvalid()` operation) the validity bit of a packet's header, respectively. The validity bit is further used in a deparser to define the order of headers for an outgoing packet.

Deparser. Its function is to prepare a packet to be sent back to the OVS pipeline. In particular, it is responsible for modifying the packet's headers and performing arbitrary packet encapsulations. The current design of the P4-to-uBPF compiler uses *post-pipeline editing*. It means that all modifications of the packet's headers are made in the deparser. The intermediary Match-Action tables modify the header's metadata structure, which is further used to generate an outgoing packet.

The deparser makes use of the `ubpf_adjust_head()` helper to adjust the length of a packet and perform an arbitrary encapsulation. Before adjusting a packet's head, the offset is calculated. If it is negative, bytes are removed from the head of a packet. Otherwise, zero bytes are added to the front of a packet. In comparison to previous versions of the language, $P4_{16}$ requires an explicit definition of the deparser. Thus, the programmer has to define the order of headers for the outgoing packet in the P4 code. Then, the deparser fills in the packet's payload with data from the `Headers_t` structure. As described above, the deparser decides to append a particular header based on the validity bit associated with each packet's header.

As is the case with other P4 compilers, the P4-to-uBPF compiler also generates the P4Info metadata, which can be used by the P4Runtime-based control plane to interface with Match-Action tables.

4.2.5 P4Runtime-based control plane

To effectively use P4rt-OVS via an external SDN controller, a user needs to leverage the OpenFlow and P4Runtime protocols in conjunction. It is the result of the hybrid design that P4rt-OVS follows. P4rt-OVS extends the OpenFlow protocol to support a new OpenFlow action, *prog*, in the FLOW_MOD message. The *prog* action invokes a given BPF program for packets matching the corresponding flow rule. The current implementation also provides all the OpenFlow messages listed in the last paragraph of subsection 4.2.3.

The P4Runtime protocol has also been extended. P4rt-OVS introduces a new usage model for P4 devices: P4rt-OVS may be configured with multiple P4 programs, each of them describing a separate forwarding element, with OpenFlow rules to dispatch between the P4 programs. Therefore, in order to support multiple P4 programs, the P4Runtime protocol has been customized by introducing a new field for the P4Runtime messages, *pipeline_id*. The *pipeline_id* field defines the P4 pipeline inside the P4 target. Thus, if the P4 target (identified by the *device_id*) supports multiple P4 pipelines running simultaneously, the P4Runtime controller can use the *pipeline_id* field to refer to a given P4 pipeline. The P4Runtime abstraction layer (P4RT-AL) has been implemented as a proof-of-concept Python application to control BPF programs using the P4 semantics.

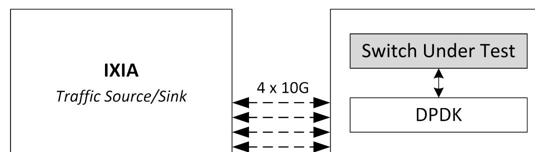


Figure 4.6: The test topology for P4rt-OVS

4.3 Performance evaluation

In this section, we compare the packet processing performance of P4rt-OVS, OVS [148] (the reference implementation), and PISCES [160] (a P4-capable OVS). The goal of the evaluation is to check 1) whether P4rt-OVS introduces any performance overhead and 2) how P4rt-OVS performs in comparison to other solutions. We also measure the overhead and efficiency of P4 extensions to OVS through a set of microbenchmarks.

4.3.1 Evaluation environment

Figure 4.6 shows the test topology. The experiments are conducted in the PL-LAB environment[1]. The Switch Under Test (SUT) running on top of the DPDK framework is installed on the HP ProLiant DL380 Gen9 server equipped with 2x Intel(R) Xeon(R) CPU E5-2690 v3 running at 2.60GHz, with 128 GB RAM and two dual-port Intel 82599ES 10GE NICs. We use an IXIA hardware traffic generator connected directly to the ports

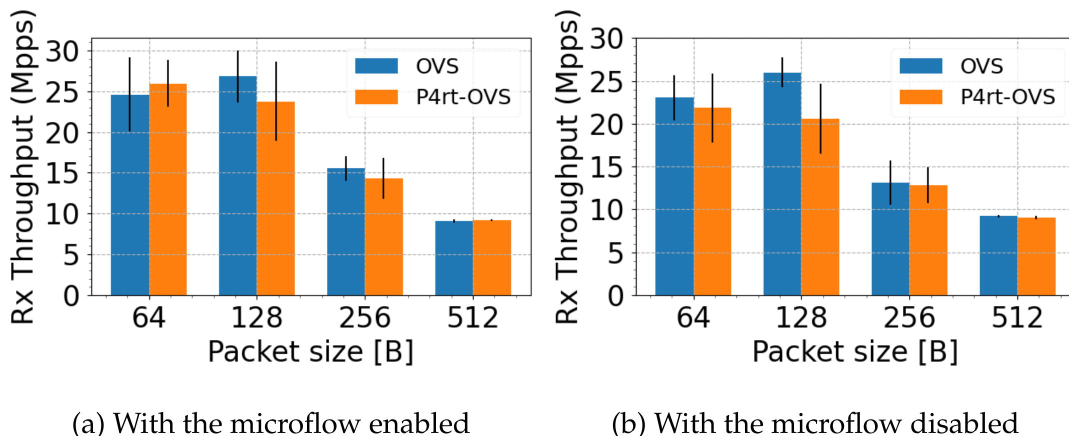
[1] www.pllab.pl

of the server, so that the SUT handles a total of 40 Gbps of traffic. For all experiments, Linux (Ubuntu 16.04) is configured to isolate DPDK cores from the Linux scheduler and disable frequency scaling. The DPDK framework has been configured with 4 receiving queues per port running on 2 physical (4 logical) CPU cores. In all experiments, the IXIA tool generates four traffic flows per port to test SUT with multiple flows.

Three software switches are compared: Open vSwitch (version 2.13), PISCES (based on OVS v2.5) and P4rt-OVS (based on OVS v2.13). To measure the results, the methodology from RFC2544 [25] is used (we assume a 0.002 % packet loss). In end-to-end comparisons, each experiment lasts 60 seconds and we report the mean throughput in millions of packets per second (Mpps). For a mean throughput, we calculate the 95% confidence interval over 10 runs[2]. In the microbenchmarks, we measure CPU cycles per packet using the machine's time-stamp counter (TSC).

4.3.2 End-to-end performance

We first measure the end-to-end performance of example network functions to illustrate the cost of the P4 programmability in a near-realistic scenario.



(a) With the microflow enabled

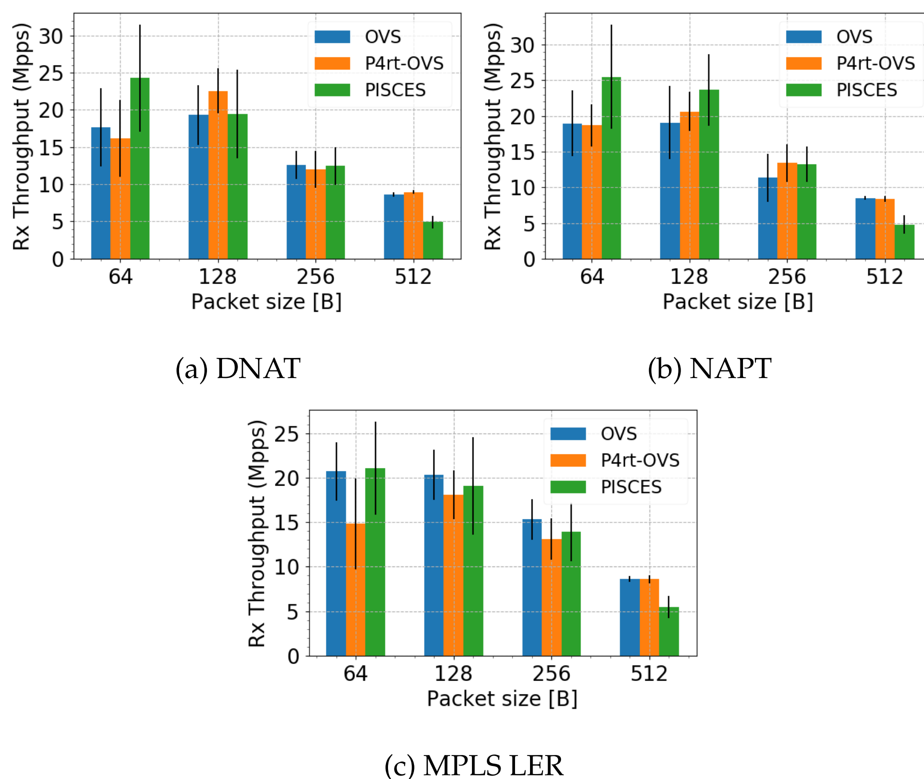
(b) With the microflow disabled

Figure 4.7: L2 Forwarding performance in Mpps for input traffic of 40 Gbps with and without the microflow.

Overhead evaluation. First, we conduct the experiment to evaluate the overhead of introducing P4 extensions to OVS. For this purpose, we compare the simple L2 forwarding performance of OVS and P4rt-OVS with the baseline P4 program performing

[2]We assume that the sampled values follow a normal distribution to be able to compute the confidence interval from estimated means and standard deviations.

no operations on packets. Thus, the experiment shows just the overhead introduced by a new OVS action, which invokes the BPF program to handle packets. The throughput rate is measured in two scenarios: with the microflow cache enabled and disabled. The results are shown in Figure 4.7. The most important observation is that invoking the BPF program to process a packet introduces a negligible overhead. Nevertheless, a more complex packet processing pipeline of the P4 program could increase the overhead. That impact is analyzed in the next paragraph.



(a) DNAT (b) NAPT

(c) MPLS LER

Figure 4.8: Comparison of OVS, P4rt-OVS, and PISCES's throughputs with three network functions: DNAT, NAPT, and MPLS LER.

Packet processing evaluation. We selected a few scenarios, in which different packet operations (complex packet parsing, tunneling, packet modifications) are executed to compare the performance in a near-realistic environment. We compare the performance of three solutions: P4rt-OVS, OVS, and PISCES. We measure the performance of the following stateless network functions. Note that this comparison does not involve any stateful network functions as OVS and PISCES do not support them; We evaluate the stateful capabilities of P4rt-OVS in Section 5.3.3.

- **Static Destination NAT (DNAT)** that matches a destination IPv4 address and

translates it based on static flow rules.

- **Static Network Address Port Translation (NAPT)** that matches source and destination IPv4 addresses and source and destination UDP ports and translates them based on static flow rules.

- **MPLS Label Edge Router (LER)** that matches an ingress port and destination IPv4 address and appends the MPLS label to a packet.

Figure 4.8a shows the performance results for DNAT, MPLS LER and NAPT.. Based on these results, we can conclude that the performance of P4rt-OVS is comparable to both OVS and PISCES for packet sizes of 64, 128, and 256 bytes. The mean throughput with 64B packets for MPLS LER in P4rt-OVS is lower than for OVS or PISCES; it may be caused by the overhead introduced by the `ubpf_adjust_head()` helper, which adds zero bytes to the head of a packet before sending it to the wire. For larger packets (512 bytes) the performance of P4rt-OVS is comparable to OVS and higher than the performance of PISCES. The results show that the overhead of more realistic P4 extensions to OVS is negligible in terms of the obtained throughput rate.

4.3.3 Microbenchmarks

As a next step, we evaluate each component of the BPF program (parser, deparser, Match-Action pipeline) separately. Moreover, we measure how much overhead the P4 programmability introduces in comparison to writing the BPF program in the C language.

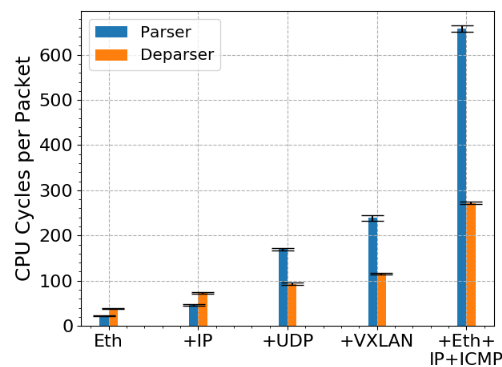


Figure 4.9: The performance of parser and deparser as more protocols are handled.

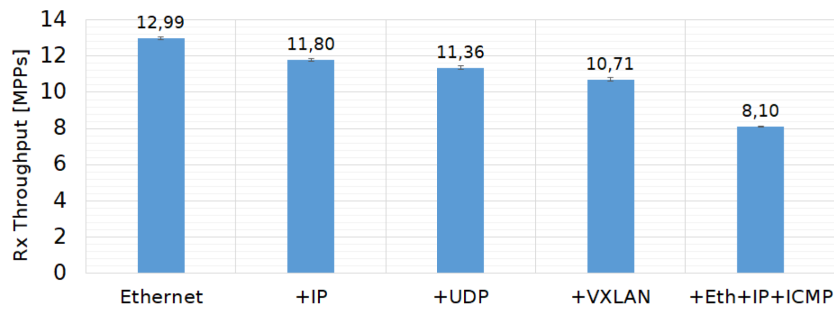


Figure 4.10: The impact of parser and deparser on the aggregated performance in millions of packets per second as more protocols are handled.

Parser and deparser performance. Figure 4.9 shows how CPU cycles per packet increase for both parser and deparser as the P4 program handles additional protocols. To parse only the Ethernet header, the parser consumes about 20 CPU cycles per packet, while the deparser consumes twice as many cycles, about 40 per packet. The deparsing process is more costly if there are a few protocols handled (up to 2). However, as the P4 program handles more protocols (layer 4 and above) the parser stage becomes more costly.

The cost of the parser for the protocol stack composed of seven protocol's headers is about 2.4 times greater than the cost of the deparser. It means that the performance results may be degraded for the P4 programs handling more complex protocol stacks. To show how much the performance of P4rt-OVS with a P4 program could be degraded due to the complexity of a parser and deparser, we measure the aggregated throughput for different protocol stacks (Figure 4.10). We can observe a strong correlation between the number of CPU cycles spent on packet processing and the overall throughput. If more protocols are handled by a P4 program, the throughput decreases.

Match-Action pipeline's performance. The packet processing pipeline is described in a control block of a P4 program. The control block may be composed of multiple Match-Action tables. As the P4-to-uBPF compiler generates the code that modifies packets in the *post-pipeline editing*, the cost of a write action (set or modify a field) is included in the cost of a deparser. Therefore, the main factors that impact the performance of the packet processing pipeline are operations on Match-Action tables.

Figure 4.11 shows how many CPU cycles are required for the Match-Action table's operations (lookup and update actions) as more tables are used to process a packet.

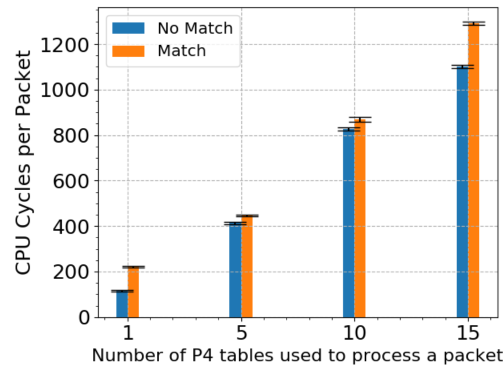


Figure 4.11: Performance in CPU cycles of the control block as more Match-Action tables are used to process a packet.

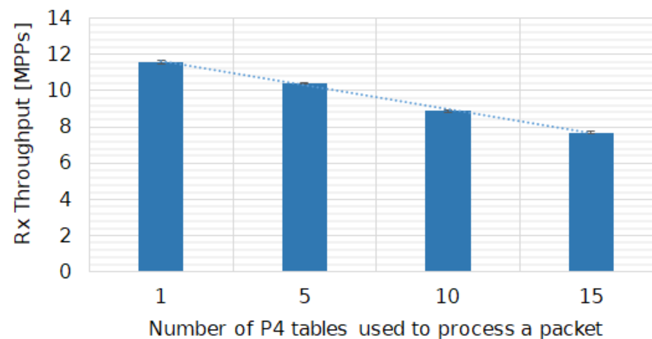


Figure 4.12: The impact of number of Match-Action tables in the control block on the aggregated performance in millions of packets per second.

As expected, the cost of the Match-Action table's operations grows almost linearly when the number of Match-Action tables is increased. If there is a match in the table lookup, appropriate action is invoked. However, the overhead of invoking an action is negligible due to *post-pipeline editing*.

Figure 4.12 shows the impact of the number of Match-Action tables on the overall throughput. As there is no significant difference between the Match and No Match scenarios, we measure throughput only for the latter case. The results confirm the observations in Figure 4.11. More Match-Action tables reduce performance. Therefore, we recommend to minimize the number of P4 tables to optimize the performance of a P4 program dedicated for P4rt-OVS.

We also measure how many CPU cycles per packet are required to perform table lookup and update (Figure 4.13). As expected, the cost of these operations is constant re-

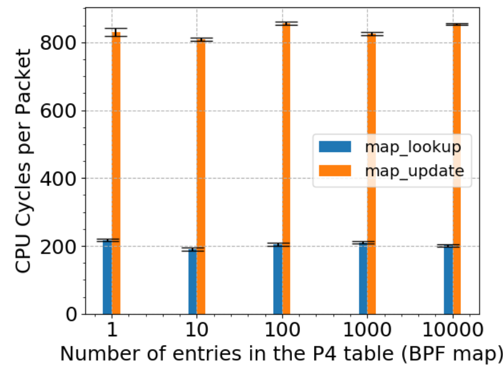


Figure 4.13: Performance of table operations as more table entries are added.

gardless of the number of entries stored in the P4 table. It is a consequence of using hash maps to implement P4 tables. However, we can observe that `ubpf_map_update()` uses many more CPU cycles per packet than the map lookup. This is due to the memory allocation required to add new entries to the hash map. In this case, we do not measure the overall throughput, because we expect the performance to be the same as the cost of table operations is constant.

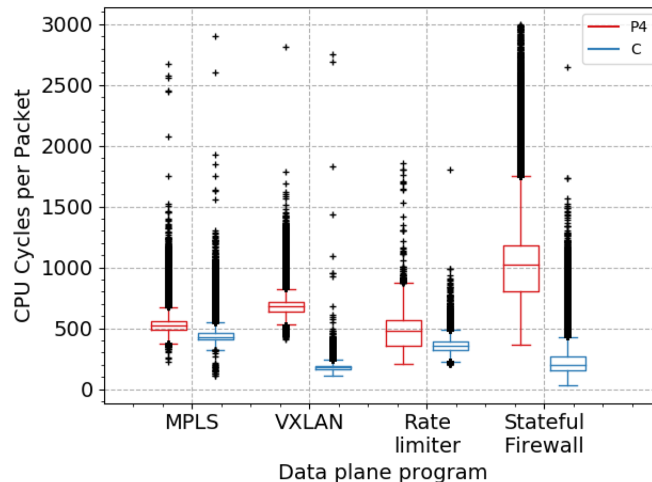


Figure 4.14: The performance of data plane programs written in P4 and C.

Overhead of the P4 programmability. The data plane programs for P4rt-OVS do not necessarily have to be developed in P4. A skillful programmer may also use the C language with standard userspace libraries to implement runtime extensions for P4rt-OVS. P4 provides an expressive, declarative, high-level language, but a protocol-independence and programmability come with the cost of a more complex program

structure and costly parsing and deparsing stages. In this experiment, we compare how many CPU cycles are consumed by the BPF program generated from P4 in comparison to the C-based program.

Figure 4.14 depicts the performance results for several programs written in either P4 or C. It shows that P4 introduces overhead in comparison to C programs, especially for VXLAN and stateful firewall. Based on these results, we can conclude that the performance of a data plane program performing MPLS tunneling is comparable for both C and P4. For this kind of program, a protocol stack is quite simple and, therefore, the cost of parsing and deparsing is low. However, we can observe significant overhead when comparing results for the VXLAN tunneling example. As observed in Figure 4.9, the cost of parsing and deparsing in P4 increases as more protocols are handled. This does not apply to the C program, as there is no need to perform a costly deparsing process.

We also measure the performance for two stateful programs, namely the rate limiter and the stateful firewall. The performance of C and P4 implementations of the former are comparable because there is no need to implement parser and deparser for simple rate limiting in P4. However, the performance of a stateful firewall written in C is higher than the corresponding P4 program. A stateful firewall tracks the state of the TCP connection, so the P4 program must parse headers up to layer 4. Again, due to the high cost of parsing and deparsing stage, the stateful firewall written in P4 performs worse.

Note also that the performance of a particular BPF program depends strongly on how the function is implemented and the results can vary considerably from one program to another. For instance, we have implemented a stateful firewall in C such that the packet processing is finished just after saving the state of the connection. On contrary, the program generated from P4 always reaches the deparser (as the P4 language does not provide an explicit keyword to drop packet and stop execution, such as `return` in C). This is also the reason why the C-based stateful firewall performs better.

In the case of measurements considered for the stateful firewall, there are many outliers. These are mainly values for `map_update()` operations as they are quite rare (an update of the state is done only when a session's state is changed); once the session

is open, `map_update()` is not invoked until the end of the session. To conclude, P4 introduces a notable performance overhead in comparison to C programs. It is mostly caused by the costly parsing and deparsing process performed in the BPF program generated from the P4 language. Based on the analysis of the results shown in Figure 4.12 and Figure 4.10, we expect similar throughput decreasing for scenarios with complex parsing and deparsing processes. Hence, there is a room for performance optimizations of the P4-to-uBPF compiler by generating a more efficient parser and deparser code.

4.3.4 Evaluation of an exemplary network function

In this section, the implementation of an exemplary network function for P4rt-OVS is presented. The Broadband Network Gateway (BNG) has been selected as a realistic and relatively complex network function to obtain performance results for the telecom operator use case. The PPP/PPPoE protocols used by BNG also represent the example of a domain-specific protocol. In order to take advantage of OVS built-in mechanisms and present the recommended way to implement new applications for P4rt-OVS, the BNG function has been implemented in a hybrid way: using both OpenFlow-based actions and P4 programming enabled by P4rt-OVS. Note that we evaluate only the data plane part and the example is not composed of both control and data plane components.

Figure 4.15 shows the design of the BNG's packet processing pipeline. It implements the most important functions of the BNG data plane, but some functions (such as traffic accounting) are missing. The BNG pipeline handles Ethernet, VLAN, PPPoE, PPP, and IP protocols, and performs VLAN encapsulation and decapsulation, routing table lookup based on the destination IP address, decrementation of TTL, and PPPoE/PPP encapsulation and decapsulation. In the upstream direction, the pipeline firstly decapsulates a packet from the VLAN header and passes the packet to the *ppp_decap* P4 program, which enhances P4rt-OVS with PPP decapsulation function. Then, the routing decision is made and the ACL table is applied. In the downstream direction, the reverse operations are performed with the use of the *ppp_encap* P4 program. The *ppp_encap* P4 program is presented in Listing 4.1. It is a dedicated P4 program responsible for only one task: performing PPP encapsulation based on the destination IP address. It is written for the uBPF architecture model and encapsulation is simply

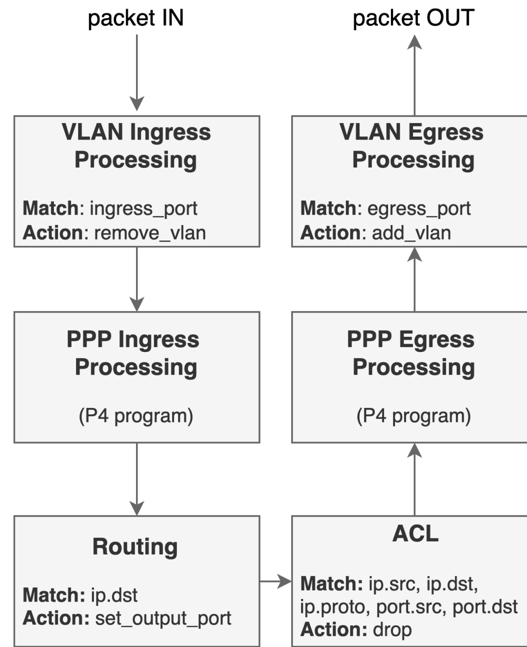


Figure 4.15: The packet processing pipeline of the BNG application

performed by validating the PPPoE header and filling its fields. In particular, a session identifier is provided as a flow rule installed by a control plane. Except for two P4 programs, all the other stages of the BNG pipeline are implemented as OpenFlow rules for OVS.

```
parser prs(packet_in packet, out Headers_t headers, inout metadata meta, inout
    standard_metadata std_meta) {
    state start {
        packet.extract(headers.ethernet);
        transition select(headers.ethernet.etherType) {
            16w0x800 : parse_ipv4;
            default : reject;
        }
    }
    state parse_ipv4 {
        packet.extract(headers.ipv4);
        transition accept;
    }
}
control pipe(inout Headers_t headers, inout metadata meta, inout
```

```
    standard_metadata std_meta) {

    action ppp_encap(bit<16> session_id) {
        headers.pppoes.setValid();
        headers.ethernet.etherType = 0x8864;
        headers.pppoes.version = 4w1;
        headers.pppoes.type_id = 4w1;
        headers.pppoes.code = 8w0; // 0 means session stage.
        headers.pppoes.session_id = session_id;
        headers.pppoes.length = headers.ipv4.totalLen + 16w2;
        headers.pppoes.protocol = 0x0021; // PPPoES IPv4 protocol
    }

    table encap_tbl {
        key = { headers.ipv4.dstAddr: exact; }
        actions = { ppp_encap; }
    }

    apply {
        encap_tbl.apply();
    }
}
control deparser(packet_out packet, in Headers_t headers) {
    apply {
        packet.emit(headers.ethernet);
        packet.emit(headers.pppoes);
        packet.emit(headers.ipv4);
    }
}
```

Listing 4.1: Pseudocode of the P4 program implementing PPP Egress Processing for P4rt-OVS. Pseudocode does not present the definition of protocols' headers.

We measure the performance of the BNG application by using the IXIA generator that emulates PPPoE traffic in the upstream and downstream directions. Figure 4.16

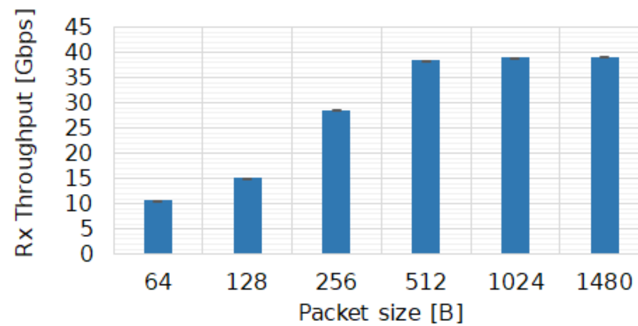


Figure 4.16: The performance of the BNG pipeline implementation for P4rt-OVS

shows the performance results measured for different lengths of packets. Depending on the packet size, the obtained throughput is between 10 Gbps and the line rate (40 Gbps) and based on that we can conclude that it is a satisfying level of performance for software switching solutions running on commodity hardware.

4.4 Conclusions

Even though OVS provides a high degree of programmability to the data center networking, it is still difficult to extend its packet processing pipeline to implement novel or domain-specific network protocols or stateful data plane programs. In this chapter, the design and implementation of P4rt-OVS, an original extension of OVS that allows for programming protocol-independent and stateful runtime extensions for OVS, has been presented. P4rt-OVS offers network engineers a flexible architecture to introduce new network features to OVS's forwarding pipeline dynamically and, therefore, to shorten the time to market for network protocols. The obtained performance evaluation results show that P4rt-OVS introduces a negligible overhead, unless a P4 program has complex parser and deparser or a high number of Match-Action tables is used. Moreover, the microbenchmarks provided guidelines on how to write efficient P4 programs for P4rt-OVS. Nevertheless, as microbenchmarks proved, there is still room for performance optimizations.

P4rt-OVS has been designed to allow for the extension of OVS's packet processing pipeline without the need for in-depth knowledge on OVS. Moreover, it enables the injection of those extensions at runtime, without the need for re-compilation. Anyone

can use P4rt-OVS's language, P4, which is a high-level and user-friendly way to describe the behavior of a network devices, and write their own network application with no need to use low-level languages such as C or to be familiar with the large codebase of OVS.

However, it is worth noticing that P4rt-OVS has some limitations. First, it is not a fully-featured P4 software switch. In fact, P4rt-OVS only implements a subset of P4 features and some complex operations (e.g., packet cloning or recirculation) cannot be described in the P4 architecture model for P4rt-OVS. Therefore, to implement more advanced network applications, users still have to rely on native OVS mechanisms. P4rt-OVS merely provides a way to extend the functionality of OVS at runtime (e.g., adding support for a new protocol). Second, P4rt-OVS was primarily motivated by the VNF offloading use cases. In this context, P4rt-OVS is also restricted by the limitations of the P4 language itself. In particular, even though some attempts were made to use P4 for NFV [131, 99, 165], it is sometimes not expressive enough to implement complex network functions, and it would need specialized P4 externs to support a wider range of the VNF offloading use cases.

To sum up, the P4rt-OVS solution does meet the requirements for performance and runtime programmability. However, as it uses DPDK, it does not fully meet the operability requirement. Moreover, since P4rt-OVS uses the restricted uBPF P4 architecture model as well as built-in OpenFlow mechanisms provided by OVS, it does not provide a sufficient level of programming abstraction and feature-richness. Therefore, in the next chapter, we come up with the NIKSS solution, a novel, kernel-based software datapath that meets all the design principles defined in Chapter 3.

Chapter 5

NIKSS: A novel programmable software datapath for Software-Defined Networking

This chapter introduces NIKSS (Native In-Kernel SDN Switch), a novel programmable software datapath for Software-Defined Networking, that enables the customization of end-host packet processing pipelines for specific use cases. NIKSS has been designed around the design principles that are mentioned in Chapter 3: High-level and feature-rich programming abstraction, Performance, Runtime programmability, and Operability. Combining all these design principles in a single solution is a key challenge addressed the NIKSS solution. In particular, an SDN software switch is expected to provide high performance, while keeping a high degree of programmability, flexibility and general-purpose usage. Therefore, the main goal of the research around NIKSS is to obtain enhanced programmability at runtime without sacrificing performance and operability.

All the aforementioned design principles together differentiates NIKSS from existing programmable software datapaths. P4-DPDK [137], a main alternative to NIKSS, may suffer from operability issues due to DPDK as discussed in [177]. Similarly, PISCES [160] is based on DPDK and, additionally, requires re-compilation every time the P4 program is changed. Furthermore, BMv2 [136] is not performant enough, while use cases of P4-eBPF [138] and P4-XDP [156] are very limited as they do not provide a fully-featured programming model. The former only provides packet filtering, while

the latter does not provide some P4 capabilities such as stateful packet processing or packet cloning. Open vSwitch (OVS) [148] suffers from OpenFlow limitations [24]. Finally, other solutions such as Click [96] or BESS [68] still require a domain-specific knowledge about a low-level platform design to implement custom packet processing modules.

NIKSS leverages eBPF [85] as a packet processing engine. eBPF allows to inject packet processing modules at runtime and, with eXpress Data Path (XDP) [75], provides a technology for flexible and high-speed packet processing on the Linux operating system. Moreover, it is highly operable since it is well-integrated with the Linux networking tools. Furthermore, NIKSS leverages P4 as a high-level, declarative data plane programming language that abstracts a restricted eBPF programming model, eBPF internals and low-level details about the Linux network stack and, thus, hides eBPF intricacies. Benefits of using P4 over eBPF for end-host network programming could be twofold. First and foremost, P4 provides an auto-generated control plane interface (P4Runtime [174]) and, thus, it provides interoperability with existing SDN controllers (such as ONOS [19]) and enables integrating software switches into the end-to-end programmable network. Second, P4 can lower a development complexity for network developers - they no longer need to satisfy the static eBPF in-kernel verifier, as it is a role of the P4 compiler to generate a safe and verifiable eBPF code. Moreover, as shown in [160], the P4 language can significantly simplify the development of new network applications by reducing the lines of code compared to an equivalent C code. The P4 processing model has also been selected due to its proven Match-Action pipeline that has been successfully adopted by OpenFlow [110]. Moreover, P4 has been selected as DSL over its main alternative - NPLang [122] - because P4 works in conjunction with the P4Runtime, a control plane API for P4-programmable devices. This is in contrary to NPLang that does not provide a unified control interface. Finally, the NIKSS prototype leverages a P4 Portable Switch Architecture (PSA) as an abstract forwarding model. PSA is a fully-featured switch architecture that provides a set of packet processing primitives that are necessary to implement a fully-functional SDN switch.

In summary, the main contributions of this chapter are as follows:

- The design of the NIKSS P4-programmable packet processing model. NIKSS supports two alternative designs of a packet processing model: a general-purpose

design able to implement *any* PSA program and a more limited, XDP-based design that provides better performance.

- The design and implementation of the PSA to eBPF compiler (Section 5.2), an original extension to the P4 compiler, that implements the PSA model for NIKSS. In particular, this chapter presents the design and implementation of a ternary packet classification based on eBPF primitives and a number of compiler's optimizations to maximize performance of generated eBPF code.
- A thorough performance evaluation of NIKSS[1] (Section 5.3), including microbenchmarks and a comparison with alternative software datapaths.
- A public, open-source implementation of the PSA-to-eBPF compiler [140] and NIKSS [129].

5.1 Design

In this section, we describe how the NIKSS packet processing pipeline is designed and mapped to the eBPF subsystem following the workflow shown in Figure 5.1. In particular, we provide two alternative designs of generated eBPF code: a general-purpose TC-based design and an XDP-based design. A user may choose between these two alternative designs, depending on the required capabilities and expected performance level.

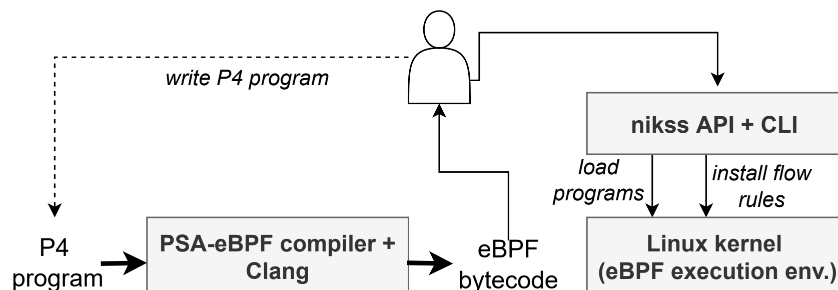


Figure 5.1: NIKSS workflow.

[1]Documentation and scripts to reproduce our results are available at *https://github.com/P4-Research/nikss-artifacts*

5.1.1 General-purpose design

To achieve the best performance, the entire packet processing should be offloaded to the XDP layer. However, during the design phase, several limitations of XDP had been encountered that prevent XDP from being used as a general-purpose target for NIKSS: (1) a lack of egress processing, (2) a lack of a BPF helper for packet cloning, and (3) no QoS mechanisms (e.g., traffic shaping or prioritization) due to a lack of a built-in buffering mechanism. We were able to provide a partial solution to (1) and (2), hence we present a limited, XDP-based design in section 5.1.2. Nonetheless, the limitation related to QoS is still valid, as QoS mechanisms are only available using *queueing disciplines (qdisc)*, which are executed on the egress path at the TC layer. Therefore, we decided to leverage TC as the main packet processing engine for NIKSS. This choice allows us to provide a generic solution that makes the implementation of any P4 program written for PSA architecture feasible but implies lower performance.

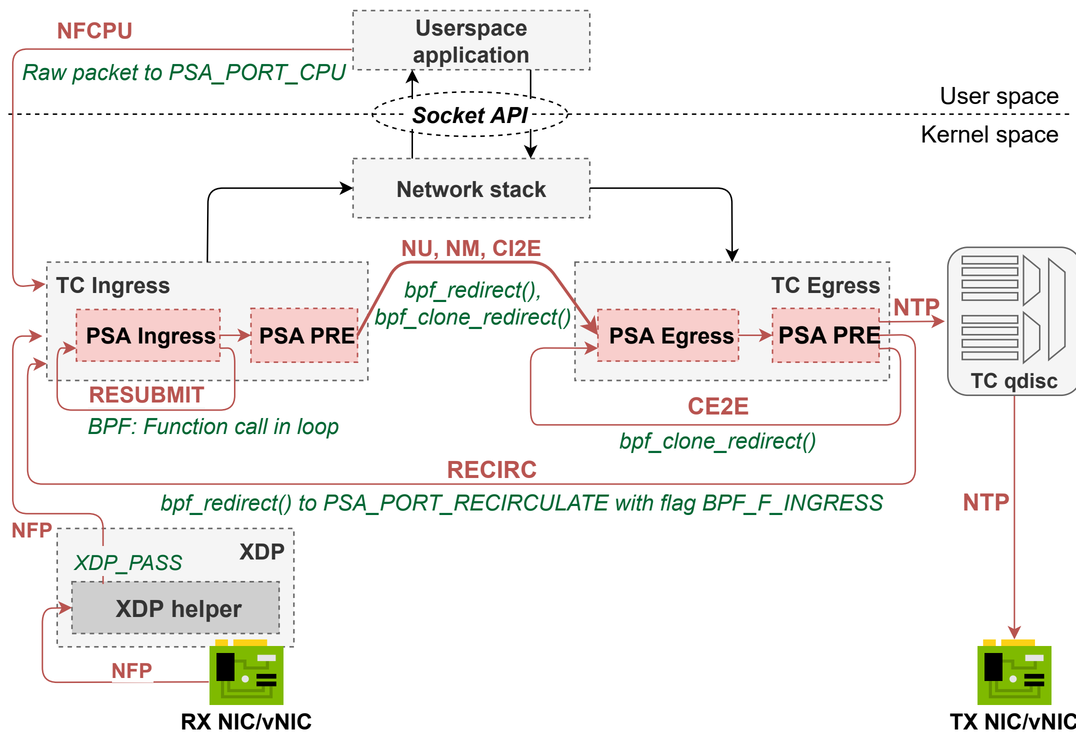


Figure 5.2: General-purpose TC-based NIKSS design (the red color depicts PSA constructs, while the green color represents mapping to eBPF primitives).

The TC-based NIKSS design (Figure 5.2) is composed of several eBPF programs generated by the PSA-eBPF compiler. First of all, a single P4 pipeline (composed of

a parser, a control block and a deparser) is implemented by a single eBPF program. Representing a single P4 pipeline by a single eBPF program makes sharing the packet processing state between the parser, control and deparser blocks straightforward. The PSA Ingress pipeline is represented as an eBPF program attached to a TC Ingress hook point, while the eBPF program implementing the PSA Egress pipeline is attached to the TC Egress hook point. To simplify the design, we decided to decompose the Packet Replication Engine (PRE) block into two independent functions placed at the end of both the ingress and egress eBPF programs. This design choice allows us to simplify implementation and avoid the performance overhead caused by jumping between eBPF programs. The decomposition of PRE was possible using the shared BPF map - both ingress and egress PREs have access to the same BPF maps and share multicast groups and clone sessions. Finally, we also provide the XDP helper program attached to the XDP hook point at the ingress interface, as we encountered a significant limitation for using the TC layer as the execution environment for P4 programs during the prototyping phase. It turns out that some BPF helpers available at the TC layer (e.g., `bpf_skb_adjust_room()`) are dependent on the IPv4/IPv6 EtherType and fail for different packet types (e.g., MPLS). As a P4 target must support protocol-independent packet processing, this limitation appears as a significant blocking point to enable all PSA mechanisms at the TC layer. To overcome this restriction, an XDP helper program is provided to make the TC layer protocol-independent by replacing the EtherType with a TC-compatible value.

The TC-based NIKSS design realizes all the packet paths that are defined by the PSA specification. The NFP path refers to the process of receiving a packet by the eBPF program attached to the TC Ingress. The XDP helper program is involved in the receiving process and sends a packet up to the TC stack. Once a packet leaves the PSA Ingress pipeline, it may be resubmitted back to the ingress pipeline. In eBPF, the RESUBMIT paths is implemented by calling the ingress packet processing function in a bounded loop. Next, there are three packet paths from the ingress to the egress pipeline. In the basic scenario, a packet is redirected as a unicast packet to the egress port by using the `bpf_redirect()` helper (the NU packet path). If a packet is cloned (the CI2E path) or sent to a multicast group (the NM packet path), the ingress eBPF program calls the `bpf_clone_redirect()` helper. We use the SKB control block (`skb->cb`) to share

per-packet metadata between ingress and egress. On the egress path, a packet is, firstly, intercepted by the PSA Egress pipeline. If the packet is sent directly towards the egress interface (the NTP path), the egress eBPF program accepts the packet. The packet is then sent to the *TC qdisc* to shape or prioritize traffic based on the Class of Service (`skb->priority`) assigned by the PSA Ingress pipeline and sent out to the output port. Note that the *TC qdisc* maps the Buffer Queuing Engine of the PSA. Furthermore, in the PSA Egress pipeline, a packet may be cloned and sent back to the PSA Egress pipeline (CE2E) or recirculated. In the former case, the `bpf_clone_redirect()` helper is used, similarly to packet cloning at ingress. In the latter case, the `bpf_redirect()` helper is called with the `BPF_F_INGRESS` flag to recirculate the packet and handle it in the PSA Ingress pipeline, again. However, the packet is not redirected back to the same ingress interface. We assume that there is at least one dedicated interface created as a `PSA_PORT_RECIRCULATE` port. Then, in the case of packet recirculation, one of these ports is used to handle the recirculated packet. Last but not least, a control plane application may also send a packet to the data plane (the NFCPU path). For this case, we designed another dedicated interface (called the `PSA_PORT_CPU`), which is used for communication between a user space application and the data plane.

5.1.2 Specialized XDP-based design

As mentioned before, XDP has three major limitations that prevented us from using it as the main BPF hook. We present a partial solution to the first two problems in this subsection.

Figure 5.3 shows the XDP-based NIKSS design. First of all, we offload the PSA Ingress pipeline to the ingress XDP hook. Then, we mimic egress processing by using the BPF `DEVMAP`, which enables attaching BPF programs to a map entry. When the ingress XDP program calls the `bpf_redirect_map()` to forward a packet, an attached program is executed. In the XDP-based design, the XDP program attached to the `DEVMAP` implements the PSA Egress pipeline. Furthermore, we solve the lack of packet cloning in XDP by using eBPF/TC programs to assist in the packet cloning process. For the XDP-based design, the PSA-eBPF compiler generates two more eBPF programs. The TC Ingress hook runs the PRE program, while the TC Egress program implements a mirror reflection of the PSA Egress pipeline attached to the `DEVMAP` and the egress

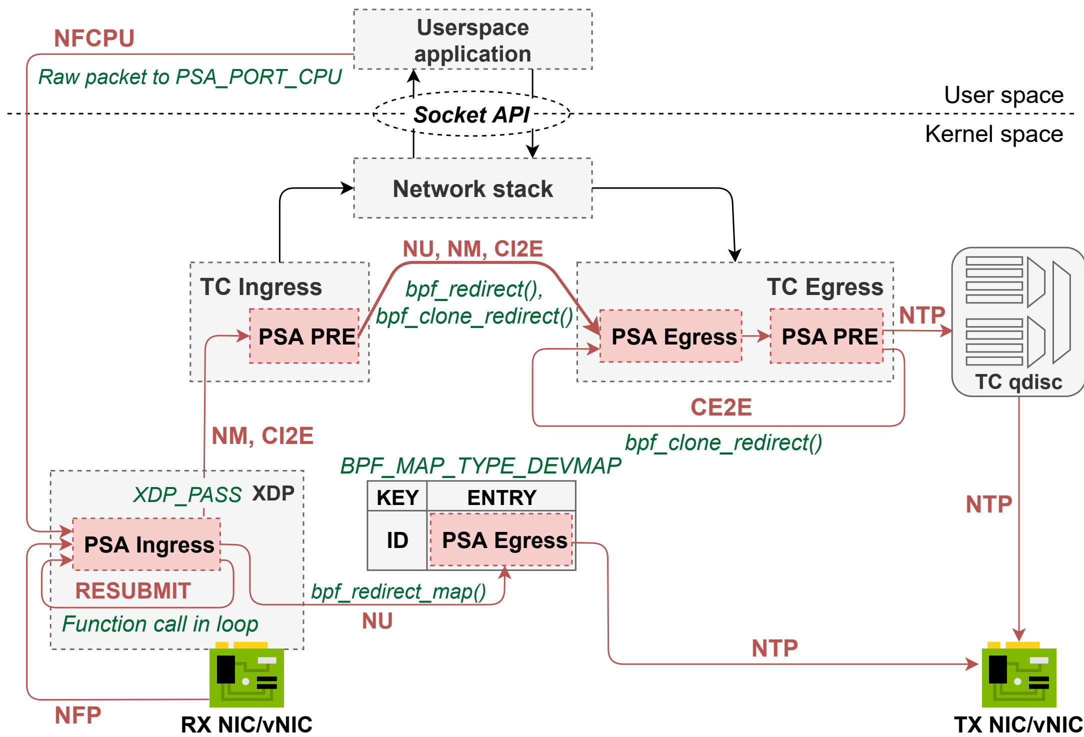


Figure 5.3: Specialized, XDP-based NIKSS design (the red color depicts PSA constructs, while the green color represents mapping to eBPF data types or primitives).

PRE. If the ingress pipeline performs packet cloning, it sends a packet up to the TC layer, including the per-packet state (e.g., parsed packet headers) that is injected to the packet using the `bpf_xdp_adjust_head()`. Next, the PRE program intercepts the packet and performs packet cloning. The packet is further handled by the TC Egress in the same way as for the TC-based design.

However, the XDP-based design implies other limitations. Due to the fact that the XDP program attached to the `DEVMAP` cannot take the control back to the ingress program, this approach does not enable packet recirculation. For the same reason, the CLONE_E2E is not supported as there is no way to pass a packet up to TC from an XDP program attached to the `DEVMAP`.

5.2 PSA-eBPF compiler

The eBPF backend supporting PSA for the P4 compiler is implemented in compliance with the $P4_{16}$ compiler's design [28]. We base it on the already-existing *ebpf_model.p4* backend for the P4 compiler [138] and, similarly, the PSA-eBPF compiler firstly gen-

erates the C code, which is further compiled down to the eBPF bytecode by *Clang*. In this section, we provide an overview of the PSA-eBPF compiler and describe technical details about the implementation of the PSA externs, ternary matching algorithm and compiler's optimizations.

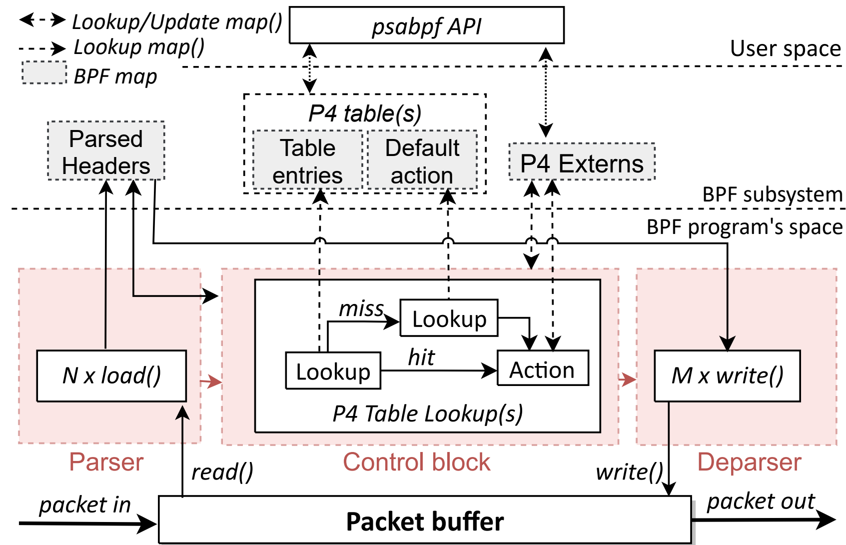


Figure 5.4: NIKSS: P4 pipeline to eBPF translation.

5.2.1 P4 pipeline to eBPF translation

Figure 5.4 shows the eBPF representation of a single PSA pipeline. A single PSA pipeline (Ingress/Egress) is translated to a single eBPF program[2]. A single eBPF program implements a parser, a control block and a deparser and uses a combination of BPF maps to implement more complex constructs such as P4 tables and PSA externs.

Parser. A parser is responsible for extracting packet headers from the packet buffer and copying them to the `Headers` structure (hereinafter referred to as `Headers`), which stores all the headers defined in a P4 program. Initially, the PSA-eBPF compiler was implemented to generate `Headers` as an on-stack variable. However, for more complex P4 programs, defining multiple packet headers and large user-defined metadata, we met compiler issues due to the exceeded BPF stack size (512 bytes). Hence, to free up space on the stack, we decided to use the per-CPU array map for storing `Headers`

[2]Latest kernels support up to 1M instructions per eBPF program, which is enough for most P4 programs. Larger P4 programs can be divided into several eBPF programs and integrated via BPF tail calls.

and user-defined metadata. When a new packet arrives on the interface, the parser reads each header field by field by loading byte words and shifting or masking bits, if necessary. The PSA-eBPF compiler supports standard P4 primitives for packet parsing such as `lookahead()` or `ValueSet`. The output of a parsing stage is the `Headers` structure filled with a packet headers' data. The `Headers` data is further used as an input to the control block's operations and the deparser.

Control block. A control block is composed of a set of Match-Action tables and actions implementing a packet processing algorithm. The PSA-eBPF compiler generates the C code that implements the control block's flow. Each P4 table is implemented as a BPF map, and each `apply()` operation on a table is translated to the BPF map lookup. For each instance of a P4 table, the compiler generates two BPF maps - the first map stores runtime-configurable table entries, while the second map only has a single element storing the default action in the BPF array map to provide the fastest possible access. The type of first map depends on the P4 match kind. If a P4 table only defines the *exact* match kind (we refer to such a table as an *exact* table), it is translated into the BPF hash map, where the hash key is a structure composed of all P4 table keys and the value stores the action identifier as well as action parameters provided by the control plane. If a P4 table defines at least one *LPM* match kind (*LPM* table), the entire P4 table is represented as `BPF_MAP_LPM_TRIE`, a BPF map implementing the Longest-Prefix Match (LPM) lookup. The PSA specification also defines *range* and *ternary* match kinds, for which there are no eBPF primitives available. We describe the implementation of the *ternary* table in section 5.2.5. The PSA-eBPF compiler's prototype does not cover the *range* match kind, but according to our study its implementation in eBPF is feasible (e.g., using range to prefix conversion [65]). Apart from P4 tables, a control block may make use of PSA externs. The PSA-eBPF compiler implements all PSA externs (see Section 5.2.4 for implementation details). Furthermore, we provide a *nikss* C library as well as a *nikss-ctl* command line tool that provide APIs to manage P4 tables and PSA externs.

Deparser. Its function is to prepare a packet buffer to be sent out from the pipeline. In particular, it is responsible for writing data from the `Headers` structure to the packet buffer. Before data is actually copied to the packet buffer, the size is adjusted by using the `bpf_skb_adjust_room()` (in TC) or `bpf_xdp_adjust_head()` (in XDP) helper that

sets the size of the outgoing packet. Next, the packet headers' fields are copied from the `Headers` structure to the packet buffer byte by byte.

5.2.2 XDP helper program

The role of the XDP helper program in the general-purpose TC-based NIKSS design is to make the TC layer protocol-independent by replacing an original EtherType by a TC-compatible value. To do that, the XDP helper program must read an original EtherType from a packet, save it, replace the original one with the IPv4 EtherType and then pass a packet up to the TC layer. By default, PSA-eBPF uses the `bpf_xdp_adjust_meta()` helper to append the original EtherType to the `skb`'s `data_meta` field, which is further read by the TC Ingress to restore the original format of the packet.

The way of passing metadata is determined by the `xdp2tc` mode. We have noticed that some NIC drivers does not support the `bpf_xdp_adjust_meta()` BPF helper and the default mode cannot be used. Therefore, we come up with a more generic mode called `head`, which uses `bpf_xdp_adjust_head()` instead to prepend a packet with metadata. In this mode, the helper must be invoked twice - in the XDP helper program to append the metadata and in the TC Ingress to strip the metadata out of a packet. We also introduce the third mode - `cpumap`, which can be used, if and only if, the Receive Packet Steering mechanism is not used. This assumes that the single CPU core handles a packet in the run-to-completion mode from XDP up to the TC layer (in other words, for a given packet, the CPU core running the TC program is the same as the one for XDP). If the above condition is met, the `cpumap` mode uses the per-CPU BPF array map to transfer metadata from XDP to TC.

Note that the XDP helper program introduces a constant but noticeable per-packet overhead. Though, it is necessary to implement P4 processing in the TC layer.

5.2.3 Packet Replication Engine

The Packet Replication Engine (PRE) is placed at the end of both the ingress and egress pipelines and is responsible for making copies of a packet (for simplicity, we describe the algorithm based on clone sessions, though the multicast process does not differ significantly). Once a packet is parsed and classified, a *clone* flag may be set. If so, PRE

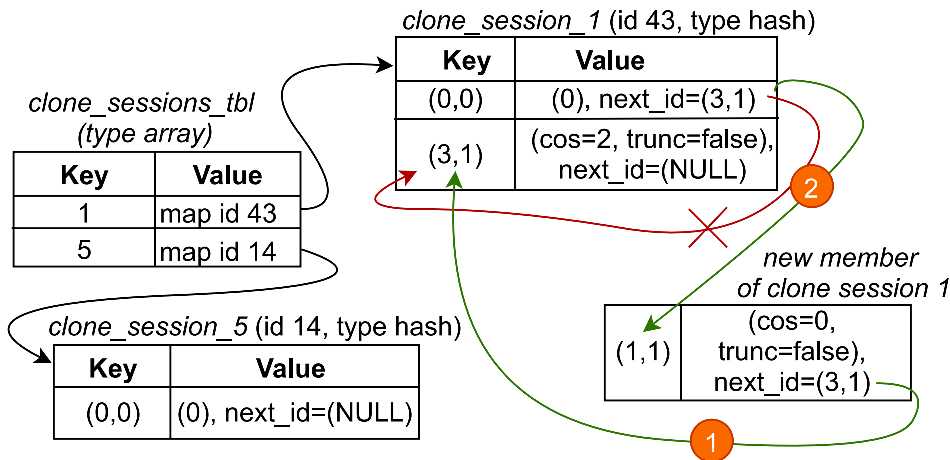


Figure 5.5: The memory organization of Packet Replication Engine implementation in NIKSS. It shows the example of inserting a new clone session member: 1) a new clone session member is allocated, and the next element is set as an element located in the front of the list; 2) the head of the list is updated to point to a new element.

must find a corresponding clone session identified by a *clone session identifier*. Then, if the clone session exists, PRE clones a packet to each clone session member. It is implemented by invoking the `bpf_clone_redirect()` helper N-times, where N is a number of clone session members. To implement the packet replication procedure, an eBPF program must operate on two maps: the outer map - indexed by the clone session identifier - stores pointers to an inner map, which contains clone session members. The PSA-eBPF compiler leverages the `BPF_MAP_TYPE_ARRAY_MAP_OF_MAPS` map to store clone sessions.

NIKSS is based on kernel v5.8, but this kernel version does not provide a built-in helper to iterate over a BPF map. Thus, an efficient way to iterate over clone session members becomes a challenge. One way is to visit every index in a map, check if a clone session member at a given index exists and, if so, execute the *clone()* action. This is a naive approach that results in a constant, but long time (dependent on the map size) to perform packet cloning (even if there are no clone session members). To solve the above-described problem, we suggest using a linked list abstraction for an eBPF map. The mechanism is depicted in Figure 5.5. First, as the PSA specification imposes the uniqueness of a clone session member (identified by a *(egress_port, instance)* pair), we use `BPF_MAP_TYPE_HASH` to store clone session members. Then, we treat the BPF

hash map as a memory space with fixed-size memory blocks. Each memory block has its address (hash map index *(egress_port, instance)*) and stores a value (clone session parameters). We assume that each value of the hash map not only contains clone session parameters, but also a "pointer" (a hash map index) to the next element (a memory block) in the list. Therefore, iterating over clone session members in the eBPF program is as simple as performing a lookup and jumping to the next element. Since we implement a clone session table as a linked list, we can easily iterate over its elements in the data plane. The complexity of a lookup to a clone session table is still linear but depends on the number of clone session members.

For online packet processing systems, what is as important as execution time is the time needed to update a table. As elements in a clone session table do not need to be ordered, we assume that a new element is added to the front of a linked list to optimize insertion time. Deleting a clone session member also becomes straightforward but requires iterating over a list to find a previous element. Then, the pointer to the next element of the previous element must be updated and the element can be deleted. To further optimize deletion complexity, implementation should be changed to a doubly linked list.

5.2.4 PSA externs

In this subsection, we briefly describe how PSA externs are implemented using eBPF data types and primitives to provide a fully-functional programming abstraction for NIKSS.

Parser Value Set. Technically, ValueSet is not defined as a PSA extern, but we find it useful to provide details about its implementation along with PSA externs. ValueSet is used to select the next parser state based on a comparison between a packet's field and a value provided by a control plane at runtime. The PSA-eBPF compiler generates a BPF hash map for each ValueSet instance. If a parser state invokes the `select()` operation on ValueSet, a lookup to the BPF map is performed to check if a given key exists in the map. The value of the BPF map's entry is ignored. If a match is found, the eBPF program jumps to a next parser state, according to the state machine defined in a P4 program.

Register. Register enables stateful packet processing. The PSA-eBPF compiler either

generates an array or a hash map for each Register instance. If Register defines an index shorter than 32 bits, the compiler generates the BPF array map for each Register instance. Otherwise, the BPF hash map is generated. Then, the `bpf_map_lookup_elem()` helper is used to read from Register, and the `bpf_map_update_elem()` helper is generated to write a value to a Register.

Counter and **Direct Counter.** The PSA-eBPF compiler generates a BPF array map for each instance of an indirect counter. This choice has consequences - the Counter size (number of indexes) is limited by the size of the BPF array map (a 32-bit unsigned integer). When Counter is updated, the packets counter and/or bytes counter are incremented using a built-in function called `__sync_fetch_and_add()` to provide atomicity of the operation. **Direct Counter** is implemented similarly but stores the packets and/or byte counters as a part of a table entry, within action data, instead of a dedicated BPF map.

Checksum, **InternetChecksum** and **Hash**. These externs do not need additional BPF helpers or maps to be implemented. The PSA-eBPF compiler generates a sequence of mathematical operations to calculate checksums or hash from a given portion of data.

Meter and **Direct Meter**. Each indirect Meter instance defined in a P4 program is translated by the compiler into a BPF hash map, where the map key refers to the meter index. The map value stores the Meter state and configuration (e.g., Peak Burst Size). The PSA-eBPF compiler generates a dedicated eBPF function performing metering each time the Meter instance is invoked. Each metering operation includes retrieving the current Meter state, performing a calculation to determine the packet color and updating the Meter state. An important aspect of Meter implementation is concurrency management between multiple CPU cores. Each time a packet is metered, our algorithm must perform a lookup and an update to a BPF map, which might simultaneously be used by some other eBPF program on another CPU core. This situation may cause some inaccuracy of the metering mechanism. To overcome concurrency issues and make operations on Meters atomic, we make use of the BPF spinlock [91]. The BPF spinlock introduces additional overhead, which is, unfortunately, necessary for correct implementation. Another factor affecting the accuracy of metering is the fact that `bpf_ktime_get_ns()` is used to get a current timestamp for a packet. It has

experimentally been proven that both BPF spinlocks and the BPF helper introduces a non-negligible overhead to the metering process in eBPF [146]. **Direct Meter** uses the same mechanisms as Meter, but it stores the Meter configuration and state as a part of an entry within a map representing a P4 table instead of a separate BPF hash map.

Packet Digest. Digests are intended to carry a small piece of user-defined data from the data plane to a control plane. The PSA-eBPF compiler translates each Digest instance into `BPF_MAP_TYPE_QUEUE` that implements the FIFO queue. If a deparser triggers the `pack()` method, data defined for a Digest is pushed into the BPF queue map. A control plane is responsible for performing periodic queries to this map to read a Digest message.

Action Profile. The Action Profile is a PSA extern that introduces a level of indirection to P4 tables. Our implementation does not differ from the implementation proposal in [175]. Hence, each P4 table using the Action Profile is translated to two BPF maps. The first map is a hash map matching on packet fields and storing references to actions, while the second one is an array map that is indexed by action references and stores action parameters.

Action Selector. The Action Selector is an extension of the Action Profile, enabling a dynamic selection of an action reference. The dynamic selection algorithm uses a hash calculated from `selector` fields to determine an action reference. The PSA-eBPF compiler generates a set of BPF maps to implement the Action Selector. During a lookup to a map, a so-called group reference is, firstly, retrieved from the hash map based on match fields (excluding `selector` fields). Then, a group (implemented as a BPF map of maps) is found based on the group reference. The first entry of the group stores an actual number of entries in the group. The number is taken as an input to the selection algorithm, which calculates a hash and picks an action reference from the group by performing the modulo operation of the number of entries in the group for a calculated hash. Thus, a lookup to the Action Selector table requires a lookup to three BPF maps and the calculation of a hash from arbitrary `selector` fields.

Random. This extern can be used to get a random number from within a P4 program. PSA-eBPF generates a random value by using the `bpf_get_prandom_u32()` helper and converts an obtained value to a requested range by performing a few mathematical operations.

5.2.5 Ternary matching algorithm

The eBPF subsystem does not provide a built-in BPF map implementing the wildcard (i.e., ternary) lookup. Hence, the PSA-eBPF compiler relies on basic eBPF primitives to implement the ternary matching algorithm. We analyzed several state-of-the-art algorithms [65, 190, 74, 178, 13]. However, either their implementation was not feasible in the restricted eBPF environment or their complexity made eBPF implementation inefficient. Finally, we decided to adapt the Tuple Space Search (TSS) algorithm [169]. We found TSS suitable for eBPF because (1) it relies on hash maps and, thus, its implementation is feasible in a restricted eBPF environment and provides reasonable classification time, (2) it is flexible enough to support arbitrary packet fields used for the P4 table's lookup, and (3) provides constant-time updates.

Concisely, TSS decomposes a ternary lookup into a sequence of lookups to hash tables (called *tuples*[3]). Each tuple stores all the table entries that share the same ternary mask (a bit mask defining the particular bits of packet fields that are used during a lookup to a P4 table). Thus, TSS groups table entries based on a ternary mask and creates a hash table for all rules that share the same unique ternary mask. All the match fields specified in a P4 table form the hash table's key. The *nikss* library is responsible for inserting a P4 table entry to a relevant tuple based on a ternary mask provided by a user. However, a P4 table may be composed of match fields of different types. Therefore, the *exact* field is always masked to use all the bits of a field. The ternary mask for the *lpm* field is created based on the prefix length, while the arbitrary mask for the *ternary* field is provided by a user. Then, masks of all the fields are concatenated to construct a single ternary mask. If there is no tuple created for a given ternary mask, *nikss* creates a new hash table. During the P4 table lookup, each tuple is analyzed to find a matching entry. Hence, for each tuple, relevant bits are extracted from a packet using a ternary mask associated to a tuple and then the masked bit array is used as a search key to a hash map storing a group of table entries. The TSS algorithm searches all the tuples to find a matching entry with the highest priority. This leads to the linear (*O(n)*) lookup time where n is the number of unique ternary masks created for a given P4 table.

[3]Properly, a tuple is the set of fields that form the hash table key, but the hash table itself is often called the tuple. This is a handy short-cut.

The foundation of TSS implementation in eBPF is the *BPF hash map of maps*. The outer map stores all the tuples and associated ternary masks created using *nikss-ctl* at runtime. Each inner map stores a group of table entries that share the same ternary mask. For each `apply()` operation on the P4 table containing a ternary match field, the PSA-eBPF compiler generates a piece of code that iterates over tuples (the outer map), masks header fields extracted from a packet and performs a lookup to the inner map using the masked hash key.

5.2.6 Compiler optimizations

In this subsection, we present the PSA-eBPF compiler's optimizations.

Table caching. Flow caching is a known technique for optimizing the packet classification time at an average case [95, 163, 193]. However, implementing the per-flow generic cache in a P4-programmable software switch is not straightforward since a P4 program may define a custom packet processing algorithm, including conditional statements and stateful processing. Therefore, the PSA-eBPF compiler implements a per-table cache - a hash map matching on all the fields defined for a P4 table - examined before the lookup to the P4 table is performed. The PSA-eBPF compiler generates a per-table cache in three cases. Firstly, if a P4 table defines the ternary match, the PSA-eBPF compiler generates a table cache in the front of such a table to avoid costly ternary classification for subsequent packets matching a table entry. Secondly, our benchmarking showed that the `LPM_TRIE` map suffers from inefficiency, especially if it contains more map entries. Hence, we also decided to generate a table cache for *LPM* tables. Finally, the PSA-eBPF compiler generates a cache for tables using the Action Selector extern. This is motivated by the fact that the dynamic, hash-based selection requires a significant amount of CPU cycles affecting the packet classification time. Considering the fact that the selection algorithm should be consistent (i.e., each packet with the same calculated hash value should get the same action specification), the PSA-eBPF compiler can safely apply the table caching technique to reduce CPU cycles spent for the lookup to Action Selector tables.

The PSA-eBPF compiler uses the LRU (Least-Recently Used) hash map to implement per-table cache. The LRU map is optimized for the cases in which the number of reads is much higher than the number of writes to the map. Additionally, it simplifies the

management of per-table caches. If a P4 table match is found for a ternary, LPM or an Action Selector table, the data plane inserts a new entry to a per-table cache and all subsequent packets matching the P4 table skip the costly classification. If a table cache is full, the LRU map itself removes the least-recently used entry and inserts a new one. The current prototype invalidates the entire cache if a user adds a new table entry.

Pipeline-aware optimization. The optimization is based on two principles: (1) at compile time, analyze a P4 program to generate a more efficient packet processing pipeline in eBPF (if possible), and (2) at run time, share `Headers` between the ingress and egress pipelines to avoid additional processing (e.g., the headers' extraction) in the egress pipeline. The PSA-eBPF compiler applies the rules below when generating code:

- If a P4 program does not define egress processing (all P4 programmable blocks of the egress pipeline are empty), the compiler does not generate an egress eBPF program.
- At runtime, if a packet header has already been extracted or added in the ingress pipeline, the egress parser skips extracting it to free CPU cycles and retrieves packet header fields from the shared `Headers` data instead.
- If a packet header is valid in the ingress pipeline and is parsed by the egress parser, but the egress deparser does not emit the header, the compiler skips emitting the header in the ingress deparser. This operation can free CPU cycles required to deparse the header. A common use case is bridged metadata that is typically appended to the front of a packet to transfer user-defined, per-packet metadata from the ingress to egress pipeline.
- If a packet header is placed at the same position in both ingress and egress deparsers and the packet header is unconditionally parsed by the egress parser, the compiler skips emitting the packet header in the ingress deparser to avoid deparsing a packet twice (it will only be emitted by the egress deparser). A common case is the Ethernet header, which is typically emitted by both deparsers at the first position.

The last three optimizations listed above are feasible only if `Headers` can be shared between the ingress and egress pipelines. For the XDP hook, `Headers` can be shared by

changing the way the compiler generates the XDP egress program - if pipeline-aware optimization is enabled, instead of using a BPF program attached to the `DEVMAP` entry, the ingress pipeline invokes the egress eBPF program by using the `bpf_tail_call()` helper that guarantees the program be run on the same CPU core. Therefore, the compiler uses a per-CPU array map to transfer `Headers`.

5.3 Performance evaluation

In this section, we evaluate the performance of NIKSS. Firstly, we show the packet forwarding rate for different test programs and provide an in-depth analysis of their performance. Next, we present microbenchmarks of the NIKSS itself and compare its performance with that of other software datapaths.

5.3.1 Environment setup

The Switch Under Test is installed on the HP ProLiant DL380 Gen9 server equipped with 2x 12-core Intel(R) Xeon(R) CPU E5-2690 v3 running at 2.60GHz and one dual-port Intel 82599ES 10GE NIC. We use TRex [80] as the traffic generator connected directly to the ports of the server. For all experiments, we use Linux kernel v5.11.3 and disable Turbo Boost and Hyper-Threading. We also turn off the `irqbalance`, set the maximum RX descriptor ring size for the NIC, set IRQ affinities to pin a single CPU core for the NIC receive queues and isolate CPU cores from the Linux scheduler. Furthermore, the PSA-eBPF compiler makes use of Clang v10.0 and the *v1* eBPF instruction set. In the spirit of benchmarking methodology outlined in [25], we measure the Partial-Drop Rate of a single core, assuming 0.1 % packet loss (we follow the settings defined by NFVBench [56]). Furthermore, we use the `bpftool prog profile` tool to count average CPU cycles per packet over all packets forwarded in an experiment run that lasts 30 seconds and generates traffic at a line-rate. For both throughput rates and CPU cycles we calculate the 95% confidence interval (CI) over 10 runs. We also use Linux `perf` profiling utility [126] to analyze the costs of underlying internal processes. Finally, we measure latency using TRex and report latency distribution by percentiles.

Name	Supported protocols	No. of P4 tables	Externs	Functionalities
L2FWD	Ethernet	1	-	L2 Forwarding
L2L3-ACL	Ethernet, VLAN, IPv4, UDP, TCP	7 (incl. 1 LPM)	ActionSelector, Counter, InternetChecksum, Digest, DirectCounter	L2 Forwarding, VLAN tagging, MAC learning, ECMP, L3 routing, port statistics, ACL
UPF	Ethernet, IP, UDP, GTP-U + inner IP, UDP, TCP, ICMP	9 (incl. 1 LPM, 1 ternary)	InternetChecksum	GTP-U tunneling, PFCP model
BNG	Ethernet, VLAN QinQ, PPPoE, MPLS, IPv4	10 (incl. 1 LPM, 3 ternary)	Meter (if encap), Counter, InternetChecksum	VLAN QinQ tagging, PPPoE tunnelling, traffic accounting, rate limiting

Table 5.1: Test programs used for NIKSS evaluation; UPF - User Plane Function, BNG - Broadband Network Gateway.

5.3.2 Packet forwarding rate

Firstly, we measure the packet forwarding rate of NIKSS for a set of representative test programs described in Table 5.1: L2FWD, L2L3-ACL, BNG and UPF. Figure 5.6 shows the results for both TC (hereinafter referred to as NIKSS-TC) and XDP (NIKSS-XDP) variants. First of all, we can clearly see the major difference in the throughput rate between TC and XDP. Depending on the test program, the throughput of programs executed by NIKSS-XDP is higher by 123% (BNG) up to 343% (L2FWD). This clearly shows the benefits of using the NIKSS-XDP mode, if possible. The throughput rate measured in packets per second for both NIKSS-TC and NIKSS-XDP is independent of the packet size (hence we only report throughput for 64B for other results), however a drop in the throughput rate is observed, if traffic achieves a line-rate. In the case of NIKSS-TC, we achieve a line-rate for L2FWD (for a packet size of 1024B and larger) and UPF (1512B) only, while NIKSS-XDP saturates the link for all programs.

Among test programs, L2FWD performs the best (1.33 MPPS for TC and 5.9 MPPS for XDP) because it is the least complex program (uses only a single P4 table) comparing to others. BNG experiences the worst performance (0.62/1.34 MPPS) for both design

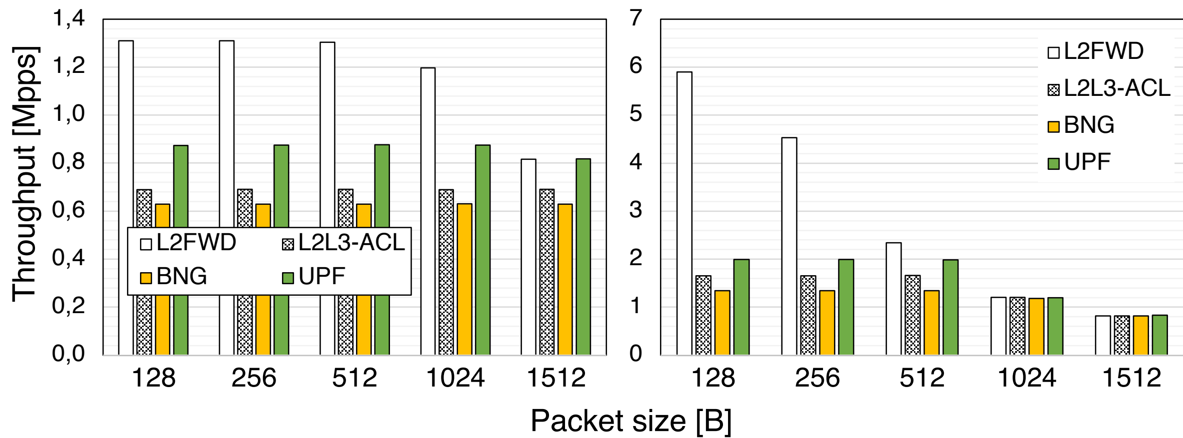


Figure 5.6: Packet forwarding rate of NIKSS-TC (left) and NIKSS-XDP (right) with only pipeline-aware optimization enabled for different packet sizes (in bytes). The 95% CI is less than 0.007 MPPS for all data points. The Y axis has a different scale in the left and right plots.

variants because it implements complex packet classification based on ternary matching and makes use of Meter that introduces significant overhead (see subsection 5.3.3). L2L3-ACL only runs slightly faster (0.69/1.65 MPPS) due to its complexity. Finally, the UPF program provides a reasonably high performance (0.87/1.99 MPPS) because it is a simplified version of 5G UPF that makes modest use of PSA externs.

Table 5.2 provides an in-depth analysis of test programs to better understand their performance and the impact of individual components on overall performance. First, we analyze non-optimized programs for TC and XDP. At the beginning, it is worth noting that the total number of CPU cycles for XDP-based programs is lower than for the corresponding TC-based program. Except for minor differences in how the PSA-eBPF compiler generates code for XDP and TC (visible for the parser and control blocks), we should notice that the deparser in XDP consumes less CPU cycles than the deparser in TC. With the help of `perf`, we observed that `bpf_skb_adjust_room()` used by TC introduces a noticeable overhead, while the cost of `bpf_xdp_adjust_head()` is almost negligible (about 0.5% of CPU time). We can also observe that the number of instructions for the XDP-based L2L3-ACL program is higher than for TC. It is due to the additional instructions required in XDP to pass packets marked to be cloned up to the TC layer, according to the NIKSS-XDP design.

Next, we analyze the per-block overhead based on the XDP flavor of L2L3-ACL,

Program	BPF hook	Optimization	Average number of CPU cycles per packet							Throughput	# of instructions	
			Ingress parser	Ingress control	Ingress deparser	Egress parser	Egress control	Egress deparser	Total	(Mpps)	Ingress	Egress
L2L3-ACL	TC	none	79	799	290	52	77	103	1400	0.75	1464	304
	XDP	none	86	796	172	48	73	50	1225	1.68	1808	259
		+ table caching	-	-121	-	-	-	-	-121	+0.21	+83	-
		+ pipeline opt.	-	-	-55	-17	-	+9	-63	+0.06	-74	+76
		All optimizations	86	675	117	31	73	59	1041	1.947	1817	335
BNG (encap)	TC	none	160	1051	340	115	19	272	1957	0.654	1977	1027
	XDP	none	173	1013	219	119	19	173	1716	1.304	1938	1020
		+ table caching	-	-353	-	-	-	-	-353	+0.259	+314	-
		+ pipeline opt.	-	-	-54	-42	-	+51	-45	+0.076	-78	+139
		All optimizations	173	660	165	77	19	223	1318	1.639	2174	1159
UPF (decap)	TC	none	131	1005	268	47	0	16	1467	0.7558	1889	57
	XDP	none	145	948	201	53	0	0	1347	1.606	1874	45
		+ table caching	-	-456	-	-	-	-	-456	+0.633	+202	-
		+ pipeline opt.	-	-	-	-53	-	-	-53	+0.161	-46	-45
		All optimizations	145	492	201	0	0	0	838	2.4	2122	0

Table 5.2: The in-depth performance analysis of test programs. The average number of CPU cycles per packet is approximated as described in Appendix A ("-" indicates no change compared to the row above). 95% CI of the throughput rate is less than 0.004 MPPS for all measurements.

which is the most realistic (hence representative) test program. On average, L2L3-ACL takes 1225 CPU cycles per packet in total, out of which 1054 are spent for ingress processing because it implements most of the functionality. Ingress Control consumes 65% of total CPU cycles, even though it is implemented using only 23.5% of all instructions. The same observation applies to other programs. The reason is that **BPF helpers used to implement Match-Action tables and PSA externs introduce a significant overhead**. To give a benchmark, the lookup to a single P4 table done by the Egress Control consumes 73 CPU cycles, while the Ingress Control block uses as many as six P4 tables. Together, the ingress parser and the ingress deparser consume 21% of total CPU cycles, but the deparser is usually more costly than the parser (twice in the case of L2L3-ACL). There are two reasons. Firstly, the parser reads chunks of data (e.g., 16/32/64 bits) at once, while the deparser writes to the packet buffer byte by byte. As a result, the deparser takes 974 instructions (47% of all instructions), while the parser only needs 347 instructions. We see room for improvement by minimizing the number of instructions generated for the deparser. The second reason is that the L2L3-ACL appends the VLAN tag to packets and, thus, `bpf_xdp_adjust_head()` is called, contributing to additional CPU cycles. Finally, the entire egress pipeline only takes 14% of total CPU cycles as it defines the less complex parser, control block and deparser.

When looking at the results for the BNG and UPF programs, we can make similar

observations. The Ingress Control of these two programs uses more tables (including ternary tables) and, therefore, takes more CPU cycles (59% and 70% of total CPU cycles for BNG and UPF, respectively). However, the difference between the parser and deparser is not so significant for UPF and BNG. In the case of UPF, this is because we measure a decapsulation scenario, in which the parser performs more operations (reads 6 packet headers in total), while the deparser only emits 3 headers. When it comes to BNG, the protocol stack is not so complex, but the high number of CPU cycles for the parser is caused by the fact that the cost of the parser also includes the cost of intrinsic variable initialization. In this case, this is the timestamp retrieved by `bpf_ktime_get_ns()` that is further used by the Meter extern. It is also worth noting that the Egress Control of BNG only consumes 19 CPU cycles as it only performs operations on headers, without any P4 table.

The impact of compiler optimizations. We analyze the impact of compiler optimizations based on XDP-based programs. Table caching optimization is in place for each test program. It is worth noting that we generate a single traffic flow (causing a very high cache hit ratio), so the performance gain due to table caching may be much lower in the case of more real-world traffic workload. However, the cache hit ratio depends on packet fields, which are matched using the ternary algorithm. For instance, the BNG program makes use of a table defining the ternary match kind for the Ethernet destination MAC and EtherType that are less likely to be changed for different flows. On the contrary, the UPF's classification table uses source and destination IP addresses that are typically unique per flow. For a single flow we can see that table caching reduces the average number of CPU cycles per packet by 15% for L2L3-ACL (due to the caching of the LPM table and Action Selector's decisions), 35% for BNG and 48% for UPF (due to the caching of ternary tables). Consequently, table caching noticeably improves throughput. Pipeline-aware optimization has a minor impact on the performance of L2L3-ACL and BNG. In fact, it only slightly optimizes the egress parser (due to a lack of need to parse bridged metadata and already-parsed packet headers) and moves the cost of the `bpf_redirect_map()` to the egress deparser. However, contrary to L2L3-ACL and BNG, the UPF program does not define egress processing and pipeline-aware optimization gets rid of the egress program. This causes a significant throughput improvement by 0.161 MPPS. Analysis using `perf` indicated

that the kernel spends additional CPU cycles to call the `dev_map_run_prog()`. It shows that even an empty egress program causes an overhead that decreases throughput.

5.3.3 Microbenchmarks

This subsection presents the evaluation of individual blocks contributing the packet processing overhead to the overall performance of PSA programs compiled to eBPF.

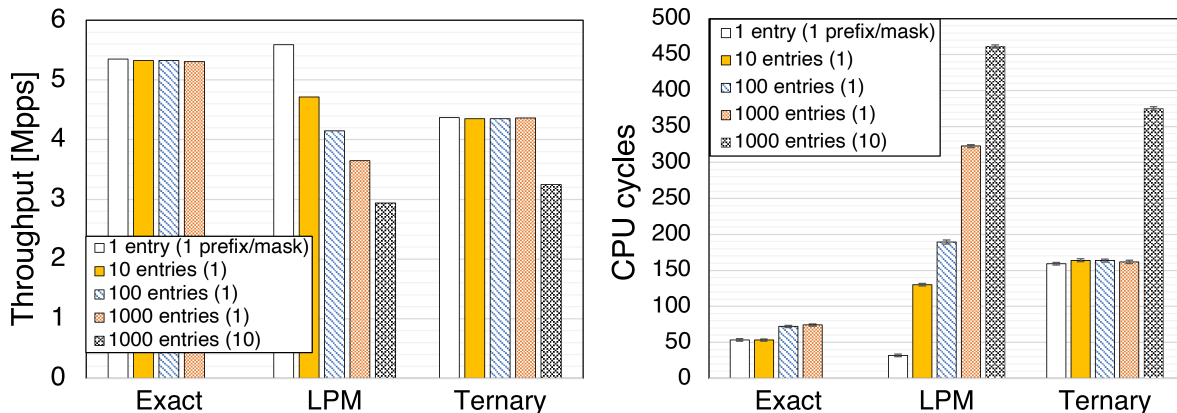


Figure 5.7: The cost of different P4 match kinds measured in the throughput rate with a 95% CI of less than 0.015 MPPS (left) and average CPU cycles per packet over a baseline with 95% CI of less than 3 cycles (right), depending on the number of table entries.

The cost of P4 table's lookup. Figure 5.7 depicts the cost of different P4 match kinds depending on the number of table entries. As expected, the `exact` match table provides the constant lookup complexity as it is implemented using the hash map. The impact of LPM and ternary match kinds is more significant. We can observe that the more entries or distinct prefixes in the LPM table, the less efficient the lookup is. On the other hand, the ternary table provides a constant complexity as long as a single ternary mask is used. Once a user adds more ternary masks, performance decreases. This behavior is the feature of the TSS algorithm, which we use for ternary tables. In this test case, we used the same flow ruleset for both *LPM* and *ternary* tables. Given the same flow ruleset for `LPM_TRIE` and TSS, we can notice that our TSS implementation performs better than the built-in `LPM_TRIE`, if the number of table entries is greater than 100. However, the performance of TSS may vary depending on the actual state. In the best case scenario, all rules are placed in the same tuple, providing the best lookup

performance. On the other hand, if each rule occupies a separate tuple (the worst case scenario), the lookup complexity will be linear.

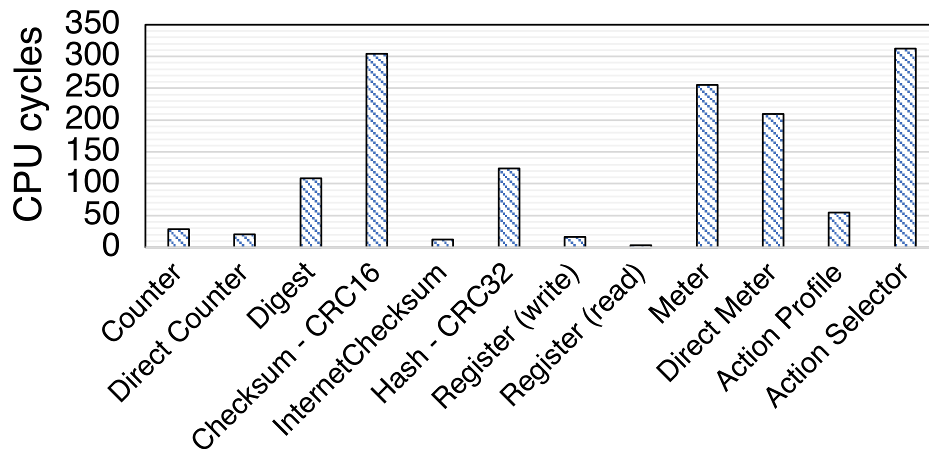


Figure 5.8: The cost of PSA externs measured in average CPU cycles per packet over a baseline program with 95% CI of less than 2 cycles for all data points.

The cost of PSA externs. Figure 5.8 proves that the performance of a PSA program differs depending on the set of PSA externs being used. We observe that the Action Selector contributes the greatest overhead (312 CPU cycles per packet). This justifies the need for applying table caching for Action Selector tables. We should also notice that the Checksum using the `CRC16` algorithm is as cost ineffective as the Action Selector. Furthermore, as we have already indicated in previous sections, Meter does also introduce significant cost due to the use of the BPF spinlock (for concurrency management in the multi-core environment) and the `bpf_ktime_get_ns()` helper. Let us note that it has been experimentally proven that both BPF spinlocks and the helper introduces a non-negligible overhead and inaccuracy to the metering process [146]. On the other hand, Counters and Registers have a negligible impact because they are based on BPF array maps. Let us note that, in this test case, Registers used indexes shorter than 32 bits and their overhead is expected to increase if a programmer defines wider indexes (due to a fallback to hash maps). The InternetChecksum is also very efficient (12 cycles only), as long as it only re-calculates a few fields, with its impact growing if more fields are re-calculated.

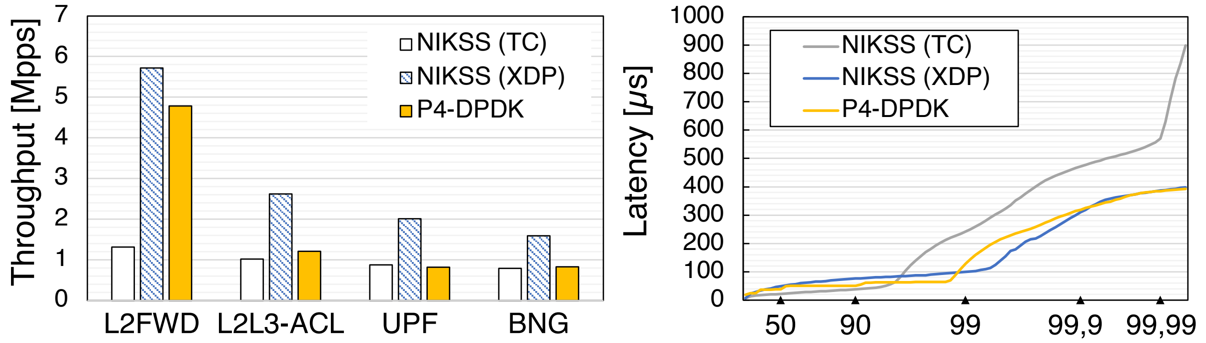


Figure 5.9: The throughput of test programs (left) and latency distribution by percentiles for L2L3-ACL and 0.8 MPPS of the offered load (right) for NIKSS and P4-DPDK.

5.3.4 Comparison with software datapaths

First, we compare the performance of NIKSS with other software PSA implementation - P4-DPDK[4] [137]. We decided not to compare NIKSS with BMv2 [136] as it is a reference model and is not meant to be a production-grade software switch. In fact, the authors claim the expected throughput rate is about 80 KPPS [135], which is several orders of magnitude lower than for NIKSS. Due to differences in supported features between NIKSS and P4-DPDK, we adjusted test programs to make them run on both targets. For instance, the current version of P4-DPDK does not support egress processing properly, hence we use ingress-only processing for all programs. In this test case, we disable table caching to avoid a high cache hit ratio as we generate a single flow. Figure 5.9 shows the obtained results. We can observe that NIKSS-XDP and P4-DPDK provides comparable throughput for the simplest program (L2FWD), outperforming NIKSS-TC. However, for more complex programs (L2L3-ACL, BNG, UPF), NIKSS-XDP outperforms both P4-DPDK and NIKSS-TC. In fact, P4-DPDK only provides a slightly higher throughput for L2L3-ACL (1.2 MPPS vs. 1.0 MPPS) than NIKSS-TC and comparable throughput for BNG and UPF. We can observe that the performance of P4-DPDK decreases and becomes comparable to NIKSS-TC if a P4 program uses ternary or LPM tables, which are implemented as scalar ACL-like tables [47]. Figure 5.9 also shows latency distribution by percentiles of the L2L3-ACL program for both NIKSS and P4-DPDK. NIKSS-TC provides the lowest latency (50-th/90-th/99-th percentiles of 21/38/242 µs) for most

[4]We use version v21.11.0 of DPDK SWX pipeline and the P4-DPDK compiler retrieved from https://github.com/p4lang/p4c on December 13, 2021

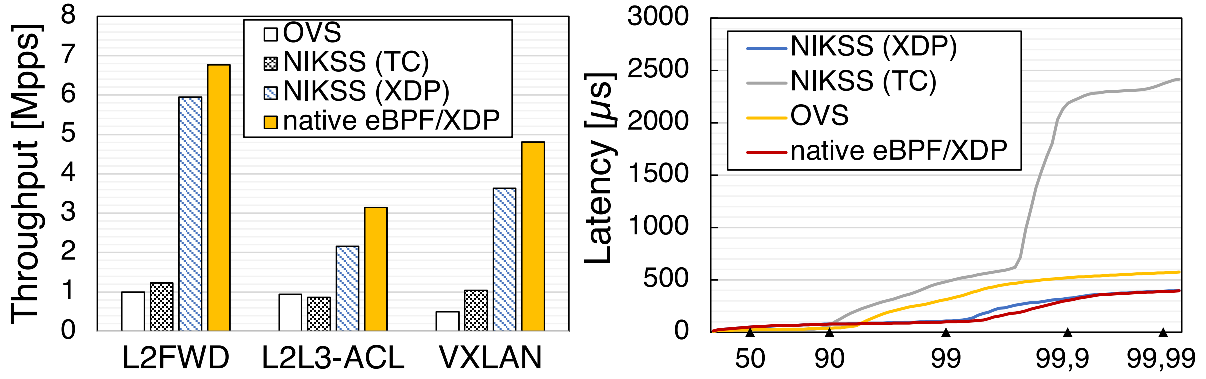


Figure 5.10: The throughput of test programs (left) and latency distribution by percentiles for L2L3-ACL and 0.8 MPPS of the offered load (right) for kernel datapaths.

(90%) packets, but experiences very high latency for minority traffic. P4-DPDK offers lower latency (38/51/128 μs) than NIKSS-XDP (48/76/100 μs) for most packets (exactly 98.8% of packets) and features a roughly comparable latency distribution for 1% of packets. Nevertheless, the P4-DPDK throughput and latency do not reach the same level as we might expect from DPDK [75], as it uses the interpreter mode to run P4 programs [48][5]. Moreover, NIKSS is more resource-efficient than P4-DPDK in terms of CPU usage [114].

Next, we compare the performance of NIKSS to OVS [148], which is a kernel datapath commonly used for network virtualizaton [182, 51]. We also show the overhead of P4 programmability, in comparison with native eBPF programming. Figure 5.10 shows a comparison between NIKSS, OVS and native eBPF/XDP programs written in C. We use L2FWD, L2L3-ACL, and VXLAN (as an example with a complex parser and deparser, feasible on all kernel datapaths) as test programs. First of all, XDP-based programs outperform programs running in TC or by OVS in terms of throughput. Nevertheless, we observed NIKSS-TC performing better than OVS for L2FWD (by 19%) and VXLAN (by 110%). On the other hand, the more complex program (L2L3-ACL) is about 9.5% faster on OVS. The difference between NIKSS-XDP and native eBPF/XDP programs shows the cost of P4 programmability. We observe lower throughput for NIKSS-XDP by 12%, 31% and 24% for L2FWD, L2L3-ACL and VXLAN, respectively.

[5] the version of P4-DPDK compiler translates a P4 program into assembly-like instructions that are interpreted by the virtual machine provided by the DPDK pipeline library. A patch is in progress to translate a P4 program into a C program like NIKSS

The performance difference is mainly caused by a more efficient parser and deparser in the case of native eBPF/XDP programs. Looking at latency distribution, we can again see that NIKSS-TC behaves well (21/66/481 µs), on average, but experiences very high tail latency. OVS provides the lowest latency in the average case (21/36/309 µs) but, similarly, a minority of traffic (1%) experiences high processing time (more than 309 µs). On the other hand, both NIKSS-XDP and native XDP programs have comparable latency distribution (48/77/106 µs and 48/75/97 µs, respectively) providing higher latency than OVS and NIKSS-TC for 90% of packets, keeping reasonably low latency (less than 106 and 97 µs, respectively) for 99% of packets.

5.4 Limitations

There are a few P4 and PSA features that NIKSS does not currently support, because they were not feasible in eBPF. First of all, we did not find a way to implement table entry timeout notification (i.e., `psa_idle_timeout`), as BPF maps do not provide an appropriate mechanism. Since the BPF spinlock cannot be used for the `LPM_TRIE` map and the BPF map-in-map, NIKSS does not support Direct Meter for tables using *lpm* or *ternary* match kinds. Moreover, the number of distinct tuples for *ternary* tables is constrained by BPF complexity issues (the 1 million instructions limit), and the actual limit depends on a P4 program (e.g., the BNG program can use up to 37 tuples per *ternary* table). Finally, the XDP hook has limitations that prevents it from being used as the PSA target for P4 programs leveraging the Buffer Queueing Engine, CLONE_E2E or packet recirculation.

We started the design and implementation of NIKSS, when the Linux kernel v5.11 was the most recent version[6]. Since that time, kernel developers have provided features that enable implementing missing or improving existing mechanisms, but they have not been incorporated into the NIKSS design yet. The BPF helper enabling iteration over BPF maps from within an eBPF program [191] would eliminate BPF complexity issues for ternary tables. BPF maps with timeouts [38] are required to implement timeout notifications for P4 table entries. The XDP multicast support [70] might enable implementing the Packet Replication Engine entirely in XDP. Furthermore, some extensions

[6]However, NIKSS is verified to work with kernel versions later than v5.8

of the PSA specification or eBPF subsystem would make NIKSS implementation more efficient. We believe that to improve performance, PSA should allow a P4 programmer to explicitly skip egress processing for certain packets, as Tofino Native Architecture does [82]. Another PSA extension - that has already been accepted, but not yet released [134] - is the *optional* match kind that will greatly improve the scalability of ternary tables. Moreover, support for XDP in the egress path [43] is a missing piece, causing the NIKSS to imitate egress processing in XDP. We also find early discussions on QoS support in XDP [119] needed from a NIKSS perspective. Finally, we believe that a built-in wildcard (ternary) BPF map would be a useful extension that gives programmers more flexibility in implementing packet processing algorithms.

5.5 Conclusions

This chapter presented NIKSS, a novel programmable software datapath for Software-Defined Networking. NIKSS applies a concept of programmable data planes to software switches, allowing for rapid development of new network features and tight integration with an end-to-end programmable network infrastructure. NIKSS has been designed around the following principles: high-level and feature-rich programming abstraction, performance, runtime programmability and operability. Hence, the main challenge addressed by the NIKSS solution is how to obtain enhanced programmability without sacrificing performance and operability.

NIKSS leverages P4 and eBPF to fulfil design principles. We presented the design of the NIKSS P4-programmable pipeline that leverages PSA as an abstract forwarding model and eBPF as an in-kernel packet processing engine. In particular, two alternative designs of NIKSS have been proposed. The TC-based NIKSS provides a general-purpose architecture that can implement any PSA program at the cost of performance. The XDP-based NIKSS implements a limited version of PSA but offers better performance. Moreover, the PSA-eBPF compiler, an original extension to the open-source P4 compiler, have been developed to generate efficient eBPF programs composing the NIKSS packet processing pipeline.

Enhanced programmability due to P4 does not come without a cost, though. We observed lower performance by 12-31% for the XDP-based NIKSS compared to native

XDP programs written in C. Additionally, microbenchmarks showed that the actual performance of eBPF programs used by NIKSS depends on the complexity of a P4 program (including the number of P4 tables or PSA externs used). Nevertheless, the performance evaluation proved that NIKSS might still be a viable alternative to kernel-based (e.g., OVS) or kernel-bypass solutions (e.g., P4-DPDK), however, there is a trade-off between a full set of PSA functionalities (TC-based NIKSS) and high performance (XDP-based NIKSS).

In future work, we plan to integrate NIKSS with a P4Runtime software stack, implement missing features for the PSA-eBPF compiler (e.g., the *range* match kind) and achieve parity with the eBPF capabilities of the latest Linux kernel. We would also like to investigate the hardware offload of NIKSS, possibly with hXDP [27]. Finally, the primary goal is to release NIKSS and interact with developers and protocol designers to make the community adapt NIKSS as the standard, high-performance P4 software switch.

Chapter 6

Summary

With the advent of SDN the approach to designing modern network systems such as 5G has been radically changed. SDN enables programmability in the network and significantly facilitates building scalable virtualized network infrastructure. However, the early OpenFlow-based SDN systems have been based on fixed-function data plane devices that were limited to a pre-defined set of supported network protocols and packet processing algorithms. With data plane programmability, a next step in the evolution of SDN, data plane devices became programmable too. Network operators can now define their own custom packet processing pipelines in a software program using high-level data plane programming language such as P4 and deploy them on white-box programmable switches.

Nevertheless, the data plane programmability has mainly been leveraged for hardware switches and NICs so far, with limited applications to software switches. At the same time, software switches have become a vital component of modern multi-tenant systems, playing a role of the primary provider of network services to virtualized end applications. With the emergence of 5G and NFV more and more network functions are moved from hardware appliances to software modules running in the virtualized data center, putting higher demands on software switches. The software switches must primarily meet the demands of software-defined networks for high throughput packet processing, but, at the same time, provide a high degree of programmability, so that network owners can smoothly introduce new features, fix network issues, perform software upgrades to the data plane devices or customize their network protocols to meet customers' requirements. Thus, network operators should be able to use a high-

level domain-specific language such as P4 to customize a packet processing pipeline without an intimate knowledge about underlying software switching platform. Moreover, programmable data planes for software switches is a missing piece to enable end-to-end programmable network, where each data plane device on the data path is programmable. Such the end-to-end programmable network enables new use cases that would have not been possible or would be difficult to implement in a timely manner without P4-programmable software switches.

In this dissertation, we present the design, implementation and evaluation of a programmable data plane for software switches in the virtualized network infrastructure. In particular, we present two novel systems, P4rt-OVS [132] and NIKSS [130], that apply data plane programmability to software switches and provide a unique contribution to the research area on programmable networks.

In Chapter 2, we introduced background on software-based packet processing, including characteristics of multi-core CPU environment, packet processing frameworks and packet classification algorithms in software, and performance metrics used to evaluate software switches. Next, we present two main technologies used for P4rt-OVS and NIKSS: P4 and eBPF. P4 is a high-level language for programming protocol-independent packet processing pipelines and is designed to be implementable on a variety of targets, including programmable NICs, FPGAs, software switches and hardware ASICs. On the other hand, eBPF, that originates from the Linux OS, is designed as a technology for commodity servers equipped with general-purpose CPUs and provides a generic framework to extend OS capabilities at runtime. In particular, eBPF enables attaching packet processing modules written in an eBPF assembly-like language to the NIC driver (XDP) or Linux network stack.

Both P4rt-OVS and NIKSS use the eBPF framework as a packet processing engine for end-host networking, but leverages P4 as a high-level language for describing packet processing pipelines. In this dissertation, we argued that the P4 language can bring significant benefits to software switches. First and foremost, P4 provides an auto-generated control plane interface (P4Runtime [174]) and, thus, it provides interoperability with existing SDN controllers (such as ONOS [19]). This enables integrating software switches into the end-to-end programmable network and can lead to novel network applications. Second, P4 can reduce a development complexity for network

developers - they no longer need to satisfy the eBPF verifier, as it is a role of the P4 compiler to generate a code that is compatible with the underlying packet processing framework. Moreover, as shown in [160], the P4 language can significantly simplify the development of new network applications by reducing the lines of code compared to an equivalent C code.

In Chapter 3, we motivated the need for a P4-programmable software switch for SDN by showing different use cases of the P4 programmability at the end hosts, including advanced network monitoring, stateful firewalls, heavy hitter detection, in-network DDoS mitigation and more. In particular, we presented two case studies: 1) end-to-end network visibility with In-Band Network Telemetry and 2) scalable large flows' detection, to prove that P4rt-OVS and NKSS can provide advantages over the state of the art. Furthermore, we performed analysis of existing solutions and defined the following requirements for programmable software switches: 1) high-level, fully-fledged programming abstraction, 2) high performance, 3) runtime programmability and 4) operability. NIKSS fulfills all of them, while P4rt-OVS lacks better programming abstraction and operability.

In Chapter 4, we presented P4rt-OVS [132], an original extension of Open vSwitch (OVS) that enables runtime programming of protocol-independent and stateful packet processing pipelines. It extends the packet forwarding model with userspace eBPF, bringing a new runtime extensibility mechanism to OVS. The P4rt-OVS design results in a hybrid approach that provides P4 programmability without sacrificing well-known features of OVS. The main advantage of P4rt-OVS is that anyone can use the P4 language to develop custom packet processing modules for OVS with no need to use low-level language such as C or to be familiar with the large codebase of OVS. P4rt-OVS makes also a number of unique contribution to the state of the art. We presented a new programming model and control plane API for OVS to enable extending the packet processing pipeline at runtime with P4. We designed and implemented a novel P4 to userspace eBPF (P4-uBPF) compiler that allows developers to write P4 programs and run them in userspace BPF VM on general-purpose CPUs. The P4-uBPF compiler has been publicly released [139] and can be used by the community to build other solutions. Furthermore, we conducted performance evaluation showing that P4rt-OVS does not introduce significant packet processing overhead comparing to the classic OVS

forwarding model, yet enables runtime protocol extensions and stateful processing. Finally, the P4rt-OVS solution is open-source [125] and available for researchers and system developers to build upon. Nevertheless, P4rt-OVS suffers from limitations such as a constrained P4 architecture model and high CPU usage due to DPDK. Therefore, we developed a novel solution based on learnings from the work on P4rt-OVS.

In Chapter 5, we presented NIKSS (Native In-Kernel SDN Software Switch) [130], a novel programmable software datapath for Software-Defined Networking. NIKSS leverages P4 as a data plane programming language providing high-level and feature-rich programming model along with the Portable Switch Architecture forwarding model. eBPF has been used for NIKSS as a packet processing engine to provide runtime programmability and operability. Performance has been achieved by combining the native eBPF environment with the PSA to eBPF compiler that generates efficient eBPF bytecode. The PSA to eBPF compiler [140] is an original extension to the P4 compiler that implements the PSA model. We implemented several compiler's optimizations to maximize NIKSS performance, including table caching and pipeline-aware optimizations. We also presented a unique ternary matching algorithm based on eBPF primitives. Last but not least, we designed and implemented the NIKSS P4-programmable packet processing pipeline leveraging either Traffic Control (TC) or eXpress Data Path (XDP) hooks in the Linux kernel. The TC-based design provides a general-purpose model able to implement any PSA program, while the XDP-based NIKSS design is more limited, but provides better performance. Our performance evaluation showed that the TC-based NIKSS offers comparable throughput and latency distribution to alternative software datapaths, while the XDP-based NIKSS outperforms evaluated alternatives.

According to our knowledge, NIKSS is the first kernel-based solution that implements P4 Portable Switch Architecture. Nevertheless, there are several features that are left for future work on NIKSS, including the P4Runtime support, design and implementation of missing features (e.g., range matching algorithm) and further performance optimizations. Moreover, to foster the adoption of NIKSS, we plan to provide a reference P4 program implementing most common features for end-host networking. Finally, the primary goal is to interact with network system developers and protocol designers to make the P4 community adapt NIKSS as the standard, high-performance P4

software switch. The source code of NIKSS has been publicly released under the open source Apache 2.0 license for academic and industry use [129]. Going forward, we believe that NIKSS can be used by researchers and system developers either as a software switch emulator to conduct new research or as a high-performance P4-programmable datapath to build production systems.

Finally, P4rt-OVS and NIKSS can be used wherever P4-programmable software switches are needed. They can be used for research experiments or real-world network applications, either as a standalone packet processor or integrated into the end-to-end programmable network. Both have already been used in some applications. In [145] we showed how P4rt-OVS can be used to enhance 5G SDN/NFV edge deployments with P4 and data plane programmability based on the P4-based implementation of the DDoS mitigation. We also researched the VNF offloading to programmable infrastructure [133, 131], where both P4rt-OVS and NIKSS can provide an offloading engine for network functions. Moreover, in [127] we showed a prototype of end-to-end network visibility with INT, where the NIKSS switch could play a role of a software switch providing telemetry information about how packets have been processed by end hosts.

6.1 Research outcomes

At the beginning of this dissertation we stated several research questions. Our research experiments have led to the following lessons learned:

- *Can programmable data plane and P4 be successfully used for software datapaths without sacrificing performance?*

  Yes. This thesis has proven by the examples of P4rt-OVS and NIKSS that, in general, the P4 language can be used to implement software-based packet processing pipelines. P4rt-OVS has shown almost no performance overhead over the fixed-function software switch, while the performance gap of NIKSS compared to native eBPF programming is not significant. There is still a trade-off between programmability and performance, but the benefits from using P4, such as auto-generated control plane interface, faster time to market for new network features, lower learning curve and development burden, makes the performance gap acceptable.

- *Given the hardware focus, is P4 flexible enough to express most common features of software switches?*

  Generally yes, but there are limitations. P4 as a language is flexible enough to express major data plane functions for software switches such as parsing, deparsing, encapsulation, decapsulation, packet filtering, stateful processing, load balancing and more. Some of features require support from P4 externs. However, although the P4 language can cover majority of data plane functions of a modern hypervisor switch or VNF, there are still some functionalities, which are typically implemented as software modules, but cannot be defined by the P4 language itself. For example, software switches may require to provide end-host encryption (e.g., with IPSec). Furthermore, some VNFs requires L7 processing to perform Deep Packet Inspection (DPI) or Intrusion Detection and Prevention Systems (IDS/IPS). These functions should be defined as new P4 externs that are dedicated for software switches.

- *Is the P4 Portable Switch Architecture (PSA) the right architecture for software datapaths? What is the price of portability?*

  Partially. PSA is a fully-featured forwarding model and, as proven by NIKSS, can be successfully used to implement common functions of a software switch. Nevertheless, as described in the previous bullet, P4 can describe majority of data plane functions in software, but not all of them. Similarly, PSA does not provide a complete list of P4 externs to cover all data plane functions that are typically implemented on general-purpose CPUs (e.g., IPSec encryption). This is a known cost of portability - the feature set is limited to those supported by most network devices. Otherwise, PSA programs would not be portable among different P4 targets. However, there is an ongoing standardization work on defining a Portable NIC Architecture (PNA) that targets programmable NICs and extends the PSA model with IPSec encryption and better support for stateful connection tracking. The capabilities of software switches running on general-purpose CPUs are typically more advanced than programmable ASICs and NICs. Therefore, we see the need for defining a Portable vSwitch Architecture (PvS) that would define missing P4 primitives for software switching such as in-built support for connection tracking exposed as a P4 extern, basic support for DPI or

packet buffering. In fact, the PvS architecture could be built on top of PSA, but extend it with more advanced packet processing capabilities.

6.2 Future work

We conclude this dissertation with a discussion of future research directions, open challenges and opportunities for improvement.

Need for a novel P4 architecture model for software switches. As summarized in the research outcomes, the existing P4 architecture models are not enough to cover all features of modern software switches. Therefore, we see a need for Portable vSwitch Architecture (PvS) that will extend the Portable Switch Architecture with packet processing primitives that are specific to software switching. Furthermore, we believe that, going forward, data plane programming frameworks should be flexible enough to express custom packet processing algorithms, but, on the other hand, provide as much as possible packet processing building blocks, which are re-usable components with a well-defined API ready to be used by data plane programmers. Since software switches typically work better for use cases requiring a lot of memory, the PvS architecture should at least provide building blocks to implement fine-grained connection tracking and NAT, packet buffering, security functions such as IPSec or TLS offload and Deep Packet Inspection. Finally, the PvS architecture should be extensible enough to allow developers define and integrate their own building blocks without the need to modify the architecture model definition. Such an extensible system to derive from is λ-NIC [36], a framework to run interactive workloads directly in the data plane.

Disaggregating network functions on heterogeneous network processors. Programmable networks can leverage a variety of network processors, including programmable switches, SmartNICs, FPGAs and software switches. P4 as a language can express packet processing pipelines for all of them. However, each of these platforms has different characteristics when it comes to throughput, latency, available memory or energy consumption. For instance, programmable switches can offer Tbps of aggregated throughput, but suffers from limited memory. On the other hand, software switches running on general-purpose CPUs provide a lot of memory resources, but cannot achieve the throughput equivalent to hardware-based packet processors. At the

same time, most advanced network functions require diverse functionalities and high performance. We believe that network functions should be disaggregated on heterogeneous packet processors to combine strengths of different programmable platforms. With the data plane dissegration, a single network function would be decomposed into several P4 programs, each running on a different packet processor, but the P4 programs would be coordinated to implement the network function. This approach requires new techniques such as Flightplan [171] to chain service functions running on different targets and orchestrate P4 programs.

Towards the end-to-end programmable network. It is worth emphasizing that the end-to-end programmable network leveraging a common data plane programming framework such as P4 and unified control plane could unleash unseen innovation in the networking systems. With a network that is full of programmable devices, nothing than creativity stems network researchers and system designers from implementing sophisticated yet efficient solutions. In the short-term, data plane programmability can provide significant improvements in some network domains such as IT data centers or NFV deployments as shown in Chapter 3. However, looking ahead, the end-to-end programmable network could become the enabler to replace the TCP/IP model or build a completely new network based on novel networking paradigms such as Information-Centric Networking (ICN) [9], ETSI Non-IP Networking (NIN) [6], NewIP [84] or RINA [44] as the interoperability with the existing TCP/IP systems is no longer a technology limitation. In future, the programmable network devices can be a foundation for deploying these or other novel network paradigms. Today, programmable switches, NICs and software switches can already be used to build experimental testbeds to conduct a research on novel networking techniques. Finally, it is also worth noting that the data plane programmability will not only be limited to traditional network devices such as programmable switches, NICs and software switches - the programmability domain can eventually span WiFi hotspots, mobile terminals, software-defined radio base stations and more.

Appendix A

Methodology to measure per-component CPU cycles for NIKSS

To count the average number of CPU cycles per packet we leverage the `bpftool prog profile` that provides a way to measure CPU cycles taken by the execution of an eBPF program. However, in our microbenchmarks we were interested in observing how much overhead individual building blocks contribute. Therefore, we came up with a slightly inaccurate, but only feasible way to measure CPU cycles consumed by a sub-component of an eBPF program. In summary, the methodology can be described as follows:

1. Perform a test run of N iterations of a pass through an eBPF program (return `XDP_PASS` for XDP and return `TC_ACT_OK` for TC). We call it a baseline program and mark CPU cycles measured for this scenario as $C_{baseline}$.

2. Perform a test run of N iterations of an eBPF program that uses a component (e.g., a PSA extern, or a P4 table).

3. Calculate the difference in average CPU cycles per packet between the two programs to get the cost of a component.

We use this methodology to measure the cost of PSA externs and matching algorithms for P4 tables. We also follow this methodology to measure an average number of CPU cycles per P4 block (a parser, control block or deparser), reported in Table 5.2. However, to measure CPU cycles consumed by a particular P4 block, we use hand-crafted eBPF programs (based on C programs generated by the PSA-eBPF compiler)

that only includes a parser (C_P), parser and control block ($C_{P,C}$) and the entire pipeline ($C_{P,C,D}$). Then, the average number of CPU cycles consumed by a parser (C_{Parser}) is calculated as follows:

$$C_{Parser} = C_P - C_{baseline}$$

The average number of CPU cycles consumed by a control block ($C_{Control}$) is calculated as:

$$C_{Control} = C_{P,C} - C_{baseline} - C_{Parser}$$

The average number of CPU cycles consumed by a deparser block
($C_{Deparser}$) is calculated as:

$$C_{Deparser} = C_{P,C,D} - C_{baseline} - C_{Parser} - C_{Control}$$

Then, the total number of CPU cycles per pipeline (C_{Total}) is calculated as:

$$C_{Total} = C_{Parser} + C_{Control} + C_{Deparser}$$

, which is roughly the same as:

$$C_{Total} = C_{P,C,D} - C_{baseline}$$

We follow exactly the same methodology to find the difference between non-optimized and optimized PSA-eBPF programs. However, this methodology suffers from slight inaccuracy for the following reasons:

- If we compile crafted eBPF programs down to an eBPF code, the *Clang* compiler optimizes the piece of code that is not used by a given block. For instance, for the handcrafted eBPF program consisting of just a parser, the instructions filling the PSA standard metadata (used further by a control block) is optimized out. However, this does not have a significant impact on the total number of CPU cycles. From our experience, the *Clang* compiler only optimizes a few instructions. This can lead to the inaccuracy of a few CPU cycles per packet.
- We observed that the number of CPU cycles measured for handcrafted eBPF programs might be different, depending on surrounding eBPF instructions. This is related to how a CPU executes a unique set of instructions. To give an example, when we enabled table caching, we retrieved different values of an average

number of CPU cycles (±5-20 cycles) for a parser or deparser, even though table caching neither modifies the parser or deparser.

Therefore, we made the following assumption. If a compiler's optimization does not modify a P4-programmable block (table caching just modifies a control block and pipeline-aware optimization only affects the ingress deparser, egress parser and egress deparser), we do not measure CPU cycles for the block again, but we take the base measurement for the block (measured for the non-optimized case). To sum up, this leads to a slight inaccuracy of results as they are approximated. However, the results are in line with what can be expected from given blocks and optimizations. It is worth outlining that, **from our experience, the approximation error is not more than ±10-20 CPU cycles**.

Appendix B

Online DDoS mitigator with P4rt-OVS

In [145], we ported the P4 DDoS mitigator program (around 100 lines of code) written for NetFPGA [144] to P4rt-OVS to show the P4 potentials in terms of performance in virtualized scenarios. The P4 DDoS mitigator exploits P4 stateful objects. With respect to stateless firewalls, the P4 switch processing is able to store and correlate protocol field values belonging to the same sessions, thus identifying suspected flows (e.g., port scan behavior) that may be dynamically blocked. However, it may not be sufficient to detect complex distributed attacks, simultaneously affecting multiple fog/edge nodes. Selected mirroring is then introduced to enable an external monitor to correlate packets traversing different edge nodes and potentially part of a more complex attack.

Listing B.1 shows an excerpt of the P4 code exploiting novel stateful objects to implement DDoS mitigator. For selected sessions, a register stores the last TCP port `port_r` and the number of related scan occurrences `scan_r`. The ingress control applies the `update_scan_param` action through the `m_table` table, storing the TCP port of the current session packet and retrieving the relative session stateful values as packet metadata. The scan condition (i.e., an incremental port pattern) determines a pipeline switch where, upon a given threshold, the packet is dropped, or subject to selected counter-actions, such as mirroring to a monitoring element in charge of intelligent correlations.

```
@name(".port_register") register<bit<16>>(32w16384) port_register;
@name(".scan_register") register<bit<32>>(32w16384) scan_register;

@name(".m_action") action update_scan_param(bit<32> register1) {
```

```
    port_register.read(meta.meta.previous_port, (bit<32>)register1);
    meta.meta.current_port = hdr.tcp.dstPort;
    scan_register.read(meta.meta.scan_occurrences, (bit<32>)register1);
}

control ingress {
  m_table.apply(); // run update_scan_param
  if (meta.previous_port == meta.current_port - 1) {
    attack.apply();
    m_filter.apply();
  } else{
    no_attack.apply();
    m_go.apply();
  }
}
```

Listing B.1: P4 code excerpt of DDoS port scan mitigator.

The evaluation of NetFPGA-based implementation has been shown in [144]. In [145], the P4 program was deployed on two platforms: Behavioral Model version 2 (BMv2) - the most common software switch employing the P4 language, and P4rt-OVS. We tested it over a standard Linux box server (Intel Xeon CPU E5-2620 6-core 2.10GHz, 16GB RAM). In the Main (M) configuration the P4 virtual switch runs directly on the bare-metal server, while in Docker (D), runs as a virtual container. The P4 virtual switch is loaded with 1000 flow entries and attached to 10 Gigabit Ethernet optical interfaces I1 and I2. TCP traffic (frame length 1500 byte) generated by the Spirent N2U Traffic Generator and Analyzer is injected in I1 and received by I2 crossing the P4 virtual switch.

We measured the BMv2 throughput and latency as a function of the number of utilized cores (i.e., 1, 2 and 4). Results show that, using four cores, the BMv2 switch is capable of sustaining more than 1Gb/s traffic while applying the DDoS mitigator, introducing an average latency below 150 s. The utilization of a reduced number of cores impacts the maximum throughput linearly (one core achieves around 200Mb/s, two cores around 500Mb/s) and the latency (reaching 105s and 140s approaching the

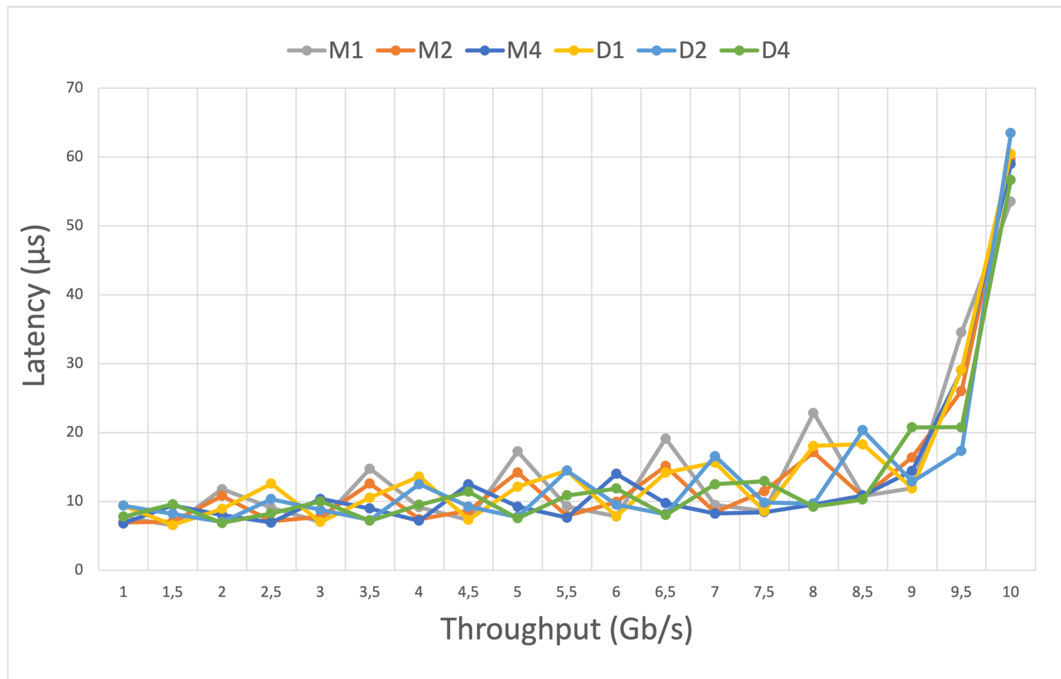


Figure B.1: P4rt-OVS DDoS mitigator latency and throughput: main (M) and Docker (D) instances.

maximum throughput, respectively). This is because the software switch always runs four parallel threads. A significant result is derived by comparing the main and the virtual container instances. In fact, the docker instance does not impact negatively on networking performance. Indeed, a slight improvement was observed due to a better thread scheduling among the available cores, which limits the 100 percent CPU load events that are at the basis of the BMV2 packet drop statistics. In light of this, the docker version achieves an improvement of 100-150 Mb/s of additional sustainable throughput. Moreover, docker assures no latency degradation, rather showing a slight improvement with a higher number of available cores serving BMV2 threads. Such effect was analyzed in the literature, confirming that containers are particularly suitable for virtualizing P4 stateful switches.

In order to prove that the virtualized P4 application can run at the 10G+ speed, the DDoS mitigator was also tested with P4rt-OVS. The test setup was the same as for BMv2, but the P4 program was adapted to make it compatible with the architecture model of P4rt-OVS. Figure B.1 shows the throughput and latency observed for P4rt-OVS. Because P4rt-OVS is based on DPDK to poll packets from an interface, the measured latency is comparable for every test scenario. The latency behavior is similar to results obtained

with FPGAs, with stable floor not exceeding 20s (using FPGA the floor was around 5us) under 90 percent interface nominal speed utilization [144]. However, similarly to BMv2, the latency observed for Docker was slightly lower than for physical machine, thus confirming the effectiveness of containerized solutions. It is worth noting that the test session successfully achieved the link speed (10G). Other tests confirmed wire speed performance at even 40G for flows having a packet size of 1500 bytes.

Bibliography

[1] Calico. https://projectcalico.docs.tigera.io/.

[2] Calico eBPF datapath. https://projectcalico.docs.tigera.io/maintenance/ebpf/.

[3] Cilium eBPF datapath. https://docs.cilium.io/en/stable/concepts/ebpf/.

[4] Flannel. https://github.com/flannel-io/flannel.

[5] Weave Net. https://github.com/weaveworks/weave.

[6] Non-IP Networking ETSI Industry Specification Group (ISG) . Non-IP Networking (NIN); Problem statement: networking with TCP/IP in the 2020s , 2021.

[7] Yehuda Afek, Anat Bremler-Barr, and Lior Shafir. Network anti-spoofing with SDN data plane. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pages 1–9, May 2017.

[8] Mamta Agiwal, Abhishek Roy, and Navrati Saxena. Next Generation 5G Wireless Networks: A Comprehensive Survey. *IEEE Communications Surveys Tutorials*, 18(3):1617–1655, 2016.

[9] Bengt Ahlgren, Christian Dannewitz, Claudio Imbrenda, Dirk Kutscher, and Borje Ohlman. A survey of information-centric networking. *IEEE Communications Magazine*, 50(7):26–36, 2012.

[10] Muhammad Bilal Anwer, Murtaza Motiwala, Mukarram bin Tariq, and Nick Feamster. Switchblade: A platform for rapid deployment of network protocols on programmable hardware. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 183–194, New York, NY, USA, 2010. Association for Computing Machinery.

[11] Arthur Fabre. L4Drop: XDP DDoS Mitigations. https://blog.cloudflare.com/l4drop-xdp-ebpf-based-ddos-mitigations, 2018. Accessed: 2021-11-27.

[12] Manikandan Arumugam, Deepak Bansal, Navdeep Bhatia, James Boerner, Simon Capper, Changhoon Kim, Sarah McClure, Neeraj Motwani, Ranga Narasimhan, Urvish Panchal, Tommaso Pimpo, Ariff Premji, Pranjal Shrivastava, and Rishabh Tewari. Bluebird: High-performance SDN for bare-metal cloud services. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 355–370, Renton, WA, April 2022. USENIX Association.

[13] Hirochika Asai. Palmtrie: A Ternary Key Matching Algorithm for IP Packet Filtering Rules. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '20, pages 323–335, New York, NY, USA, 2020. Association for Computing Machinery.

[14] Florin Baboescu and George Varghese. Scalable packet classification. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 199–210, New York, NY, USA, 2001. Association for Computing Machinery.

[15] David Barach, Leonardo Linguaglossa, Damjan Marion, Pierre Pfister, Salvatore Pontarelli, and Dario Rossi. High-Speed Software Data Plane via Vectorized Packet Processing. In *IEEE Communications Magazine*, volume 56, pages 97–103, 2018.

[16] Tom Barbette, Cyril Soldani, and Laurent Mathy. Fast userspace packet processing. In *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 5–16, 2015.

[17] Ran Ben Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. Optimal elephant flow detection. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pages 1–9, 2017.

[18] Cristian Hernandez Benet, Andreas J. Kassler, Theophilus Benson, and Gergely Pongracz. MP-HULA: Multipath Transport Aware Load Balancing Using Programmable Data Planes. In *Proceedings of the 2018 Morning Workshop on In-Network*

Computing, NetCompute '18, pages 7–13, New York, NY, USA, 2018. Association for Computing Machinery.

[19] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. ONOS: Towards an Open, Distributed SDN OS. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 1–6, New York, NY, USA, 2014. Association for Computing Machinery.

[20] Giuseppe Bianchi, Marco Bonola, Antonio Capone, and Carmelo Cascone. OpenState: Programming Platform-Independent Stateful Openflow Applications inside the Switch. In *SIGCOMM Comput. Commun. Rev.*, volume 44, pages 44–51, New York, NY, USA, April 2014. Association for Computing Machinery.

[21] Giuseppe Bianchi, Marco Bonola, Salvatore Pontarelli, Davide Sanvito, Antonio Capone, and Carmelo Cascone. Open Packet Processor: a programmable architecture for wire speed platform-independent stateful in-network processing. *CoRR*, abs/1605.01977, 2016.

[22] Nicola Bonelli, Stefano Giordano, and Gregorio Procissi. Network Traffic Processing With PFQ. In *IEEE Journal on Selected Areas in Communications*, volume 34, pages 1819–1833, 2016.

[23] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-Independent Packet Processors. In *SIGCOMM Comput. Commun. Rev.*, volume 44, pages 87–95, New York, NY, USA, July 2014. Association for Computing Machinery.

[24] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *SIGCOMM Comput. Commun. Rev.*, volume 43, pages 99–110, New York, NY, USA, August 2013. Association for Computing Machinery.

[25] Scott Bradner and Jim McQuaid. Benchmarking Methodology for Network Interconnect Devices. IETF RFC 2544, March 1999.

[26] Scott O. Bradner. Benchmarking Terminology for Network Interconnection Devices. IETF RFC 1242, July 1991.

[27] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. hXDP: Efficient Software Packet Processing on FPGA NICs. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 973–990. USENIX Association, November 2020.

[28] Mihai Budiu and Chris Dodd. The P4-16 Programming Language. In *SIGOPS Oper. Syst. Rev.*, volume 51, pages 5–14, New York, NY, USA, September 2017. ACM.

[29] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding Host Network Stack Overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, pages 65–77, New York, NY, USA, 2021. Association for Computing Machinery.

[30] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '07, pages 1–12, New York, NY, USA, 2007. Association for Computing Machinery.

[31] Carmelo Cascone, Nicola Bonelli, Luca Bianchi, Antonio Capone, and Brunilde Sansò. Towards approximate fair bandwidth sharing via dynamic priority queuing. In *2017 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, pages 1–6, 2017.

[32] Danilo Cerovic, Valentin Del Piccolo, Ahmed Amamou, Kamel Haddadou, and Guy Pujolle. Fast Packet Processing: A Survey. In *IEEE Communications Surveys Tutorials*, volume 20, pages 3645–3676, 2018.

[33] Paul Chaignon, Diane Adjavon, Kahina Lazri, Jérôme François, and Olivier Festor. Offloading Security Services to the Cloud Infrastructure. In *Proceedings of the 2018 Workshop on Security in Softwarized Networks: Prospects and Challenges*, SecSoN '18, pages 27–32, New York, NY, USA, 2018. ACM.

[34] Paul Chaignon, Kahina Lazri, Jérôme François, Thibault Delmas, and Olivier Festor. Oko: Extending Open vSwitch with Stateful Filters. In *Proceedings of the Symposium on SDN Research*, SOSR '18, pages 13:1–13:13, New York, NY, USA, 2018. ACM.

[35] Chien Chen, Hoa-Chuan Fang, and Muhammad Shahid Iqbal. QoSTCP: Provide Consistent Rate Guarantees to TCP flows in Software Defined Networks. In *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*, pages 1–6, 2020.

[36] Sean Choi, Muhammad Shahbaz, Balaji Prabhakar, and Mendel Rosenblum. λ-NIC: Interactive Serverless Compute on Programmable SmartNICs. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 67–77, 2020.

[37] Google Cloud. Dataplane V2. https://cloud.google.com/kubernetes-engine/docs/concepts/dataplane-v2, 2021.

[38] Cong Wang. bpf: introduce timeout hash map. https://lwn.net/Articles/843877/, 2021.

[39] Andrew R. Curtis, Wonho Kim, and Praveen Yalagandula. Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection. In *2011 Proceedings IEEE INFOCOM*, pages 1629–1637, 2011.

[40] Daly, James and Bruschi, Valerio and Linguaglossa, Leonardo and Pontarelli, Salvatore and Rossi, Dario and Tollet, Jerome and Torng, Eric and Yourtchenko, Andrew. TupleMerge: Fast Software Packet Processing for Online Packet Classification. In *IEEE/ACM Transactions on Networking*, volume 27, pages 1417–1431, 2019.

[41] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Noa Zilberman, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. P4xos: Con-

sensus as a Network Service. In *IEEE/ACM Transactions on Networking*, volume 28, pages 1726–1738, 2020.

[42] Rakesh Datta, Sean Choi, Anurag Chowdhary, and Younghee Park. P4Guard: Designing P4 Based Firewall. In *MILCOM 2018 - 2018 IEEE Military Communications Conference (MILCOM)*, pages 1–6, 2018.

[43] David Ahern. Add support for XDP in egress path. https://lwn.net/Articles/813406/, 2020.

[44] John Day. *Patterns in Network Architecture: A Return to Fundamentals*. 01 2008.

[45] Damu Ding, Marco Savi, Gianni Antichi, and Domenico Siracusa. An Incrementally-Deployable P4-Enabled Architecture for Network-Wide Heavy-Hitter Detection. In *IEEE Transactions on Network and Service Management*, volume 17, pages 75–88, 2020.

[46] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 15–28, New York, NY, USA, 2009. Association for Computing Machinery.

[47] DPDK project. DPDK Packet Classification and Access Control. https://doc.dpdk.org/guides/prog_guide/packet_classif_access_ctrl.html, 2021.

[48] DPDK project. The Software Switch (SWX) Pipeline. https://doc.dpdk.org/guides/prog_guide/packet_framework.html#the-software-switch-swx-pipeline, 2021.

[49] Cristian F. Dumitrescu. Design Patterns for Packet Processing Applications on Multi-core Intel® Architecture Processors. Technical report, Intel, 12 2008.

[50] Paul Emmerich, Maximilian Pudelko, Simon Bauer, and Georg Carle. User space network drivers. In *Proceedings of the Applied Networking Research Workshop*, ANRW '18, pages 91–93, New York, NY, USA, 2018. Association for Computing Machinery.

[51] Ericsson. Cloud SDN. https://www.ericsson.com/en/portfolio/digital-services/cloud-infrastructure/cloud-sdn, 2020.

[52] A. Feldman and S. Muthukrishnan. Tradeoffs for packet classification. In *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No.00CH37064)*, volume 3, pages 1193–1202 vol.3, 2000.

[53] Eder Leão Fernandes, Elisa Rojas, Joaquin Alvarez-Horcajo, Zoltàn Lajos Kis, Davide Sanvito, Nicola Bonelli, Carmelo Cascone, and Christian Esteve Rothenberg. The road to BOFUSS: The basic OpenFlow userspace software switch. *Journal of Network and Computer Applications*, page 102685, 2020.

[54] Daniel Firestone. VFP: A virtual switch platform for host SDN in the public cloud. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 315–328, Boston, MA, March 2017. USENIX Association.

[55] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, pages 51–64, USA, 2018. USENIX Association.

[56] Open Platform for NFV. NFVbench. https://docs.anuket.io/projects/nfvbench, July 2021.

[57] Nate Foster, Nick McKeown, Jennifer Rexford, Guru Parulkar, Larry Peterson, and Oguz Sunay. Using Deep Programmability to Put Network Owners in Control. In *SIGCOMM Comput. Commun. Rev.*, volume 50, pages 82–88, New York, NY, USA, October 2020. Association for Computing Machinery.

[58] Sebastian Gallenmüller, Paul Emmerich, Florian Wohlfart, Daniel Raumer, and Georg Carle. Comparison of frameworks for high-performance packet IO. In *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 29–38, 2015.

[59] Massimo Gallo and Rafael Laufer. ClickNF: a modular stack for custom network functions. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 745–757, Boston, MA, July 2018. USENIX Association.

[60] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. BMC: Accelerating memcached using safe in-kernel caching and pre-stack processing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 487–501. USENIX Association, April 2021.

[61] Glen Gibb, George Varghese, Mark Horowitz, and Nick McKeown. Design principles for packet parsers. In *Architectures for Networking and Communications Systems*, pages 13–24, 2013.

[62] Younghwan Go, Muhammad Asim Jamshed, YoungGyoun Moon, Changho Hwang, and KyoungSoo Park. APUNet: Revitalizing GPU as packet processing accelerator. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 83–96, Boston, MA, March 2017. USENIX Association.

[63] Luke Gorrie. Snabb: Simple and fast packet networking. https://github.com/snabbco/snabb. Accessed: 2022-04-05.

[64] Anton Gulenko, Marcel Wallschläger, and Odej Kao. A Practical Implementation of In-Band Network Telemetry in Open vSwitch. In *2018 IEEE 7th International Conference on Cloud Networking (CloudNet)*, pages 1–4, 2018.

[65] P. Gupta and N. McKeown. Algorithms for packet classification. In *IEEE Network*, volume 15, pages 24–32, 2001.

[66] Pankaj Gupta and Nick McKeown. Packet Classification using Hierarchical Intelligent Cuttings. *Proc. Hot Interconnects*, 20, 01 2000.

[67] Mosab Hamdan, Bushra Mohammed, Usman Humayun, Ahmed Abdelaziz, Suleman Khan, M. Akhtar Ali, Muhammad Imran, and M. N. Marsono. Flow-Aware Elephant Flow Detection for Software-Defined Networks. In *IEEE Access*, volume 8, pages 72585–72597, 2020.

[68] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. SoftNIC: A Software NIC to Augment Hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.

[69] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. Packetshader: A gpu-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 195–206, New York, NY, USA, 2010. Association for Computing Machinery.

[70] Hangbin Liu. xdp: add a new helper for dev map multicast support. https://lwn.net/Articles/845106/, 2021.

[71] Rob Harrison, Qizhe Cai, Arpit Gupta, and Jennifer Rexford. Network-Wide Heavy Hitter Detection with Commodity Switches. In *Proceedings of the Symposium on SDN Research*, SOSR '18, New York, NY, USA, 2018. Association for Computing Machinery.

[72] Frederik Hauser, Marco Häberle, Daniel Merling, Steffen Lindner, Vladimir Gurevich, Florian Zeiger, Reinhard Frank, and Michael Menth. A survey on data plane programming with P4: fundamentals, advances, and applied research. *CoRR*, abs/2101.10632, 2021.

[73] Mu He, Arsany Basta, Andreas Blenk, Nemanja Deric, and Wolfgang Kellerer. P4NFV: An NFV Architecture with Flexible Data Plane Reconfiguration. In *2018 14th International Conference on Network and Service Management (CNSM)*, pages 90–98, 2018.

[74] Peng He, Gaogang Xie, Kavé Salamatian, and Laurent Mathy. Meta-algorithms for Software-Based Packet Classification. In *2014 IEEE 22nd International Conference on Network Protocols*, pages 308–319, 2014.

[75] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The EXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, pages 54–66, New York, NY, USA, 2018. Association for Computing Machinery.

[76] Kuo-Feng Hsu, Praveen Tammana, Ryan Beckett, Ang Chen, Jennifer Rexford, and David Walker. Adaptive Weighted Traffic Splitting in Programmable Data Planes. In *Proceedings of the Symposium on SDN Research*, SOSR '20, pages 103–109, New York, NY, USA, 2020. Association for Computing Machinery.

[77] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. NetVM: High performance and flexible networking using virtualization on commodity platforms. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 445–458, Seattle, WA, April 2014. USENIX Association.

[78] Stephen Ibanez. P4 Language Tutorial. https://opennetworking.org/wp-content/uploads/2020/12/P4_D2_East_2018_01_basics.pdf, 2018. East Coast P4 Developer Day, Spring 2018.

[79] Stephen Ibanez, Gianni Antichi, Gordon Brebner, and Nick McKeown. Event-driven packet processing. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, HotNets '19, pages 133–140, New York, NY, USA, 2019. Association for Computing Machinery.

[80] Cisco Inc. Trex: Realistic traffic generator. https://trex-tgn.cisco.com/, 2021.

[81] Intel. TCP-INT: Lightweight In-band Network Telemetry for TCP. https://github.com/p4lang/p4app-TCP-INT.

[82] Intel. $P4_{16}$ Intel Tofino Native Architecture - Public Version. https://github.com/barefootnetworks/Open-Tofino, March 2021.

[83] Intel DPDK Project. Data Plane Development Kit. www.dpdk.org. Accessed: 2022-04-05.

[84] ITU-T TSAG. New IP, Shaping Future Network, 2019.

[85] J. Corbet. BPF: the universal in-kernel virtual machine. https://lwn.net/Articles/599755/, 2014.

[86] J. Corbet. Accelerating networking with AF_XDP. https://lwn.net/Articles/750845/, 2018.

[87] Ethan J. Jackson et al. SoftFlow: A Middlebox Architecture for Open vSwitch. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 15–28, Denver, CO, 2016. USENIX Association.

[88] Theo Jepsen, Ali Fattaholmanan, Masoud Moshref, Nate Foster, Antonio Carzaniga, and Robert Soulé. Forwarding and routing with packet subscriptions. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '20, pages 282–294, New York, NY, USA, 2020. Association for Computing Machinery.

[89] Wei Jiang, Bin Han, Mohammad Asif Habibi, and Hans Dieter Schotten. The Road Towards 6G: A Comprehensive Survey. In *IEEE Open Journal of the Communications Society*, volume 2, pages 334–366, 2021.

[90] Lennart Johnsson and G Netzer. The impact of Moore's Law and loss of Dennard scaling: Are DSP SoCs an energy efficient alternative to x86 SoCs? In *Journal of Physics: Conference Series*, volume 762, page 012022, 10 2016.

[91] Jonathan Corbet. Concurrency management in BPF. https://lwn.net/Articles/779120/, 2019.

[92] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The Nature of Data Center Traffic: Measurements & Analysis. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*, IMC '09, pages 202–208, New York, NY, USA, 2009. Association for Computing Machinery.

[93] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. HULA: Scalable Load Balancing Using Programmable Data Planes. In *Proceedings of the Symposium on SDN Research*, SOSR '16, New York, NY, USA, 2016. Association for Computing Machinery.

[94] Peyman Kazemian, George Varghese, and Nick McKeown. Header Space Analysis: Static Checking for Networks. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 113–126, San Jose, CA, April 2012. USENIX Association.

[95] Changhoon Kim, Matthew Caesar, Alexandre Gerber, and Jennifer Rexford. Revisiting Route Caching: The World Should Be Flat. In Sue B. Moon, Renata Teixeira, and Steve Uhlig, editors, *Passive and Active Network Measurement*, pages 3–12, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[96] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click Modular Router. In *ACM Trans. Comput. Syst.*, volume 18, pages 263–297, New York, NY, USA, August 2000. Association for Computing Machinery.

[97] Teemu Koponen, Keith Amidon, Peter Balland, Martín Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Natasha Gude, Paul Ingram, Ethan Jackson, Andrew Lambeth, Romain Lenglet, Shih-Hao Li, Amar Padmanabhan, Justin Pettit, Ben Pfaff, Rajiv Ramanathan, Scott Shenker, Alan Shieh, Jeremy Stribling, Pankaj Thakkar, Dan Wendlandt, Alexander Yip, and Ronghua Zhang. Network Virtualization in Multi-tenant Datacenters. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 203–216, Seattle, WA, 2014. USENIX Association.

[98] Diego Kreutz, Fernando M. V. Ramos, Paulo Esteves Veríssimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-Defined Networking: A Comprehensive Survey. In *Proceedings of the IEEE*, volume 103, pages 14–76, 2015.

[99] Ralf Kundel, Leonhard Nobach, Jeremias Blendin, Wilfried Maas, Andreas Zimber, Hans-Joerg Kolbe, Georg Schyguda, Vladimir Gurevich, Rhaban Hark, Boris Koldehofe, and Ralf Steinmetz. OpenBNG: Central office network functions on programmable data plane hardware. In *International Journal of Network Management*, volume 31, page e2134, 2021.

[100] T. V. Lakshman and D. Stiliadis. High-Speed Policy-Based Packet Forwarding Using Efficient Multi-Dimensional Range Matching. In *SIGCOMM Comput. Com-*

mun. Rev., volume 28, pages 203–214, New York, NY, USA, oct 1998. Association for Computing Machinery.

[101] B. Lampson, V. Srinivasan, and G. Varghese. IP lookups using multiway and multicolumn search. In *Proceedings. IEEE INFOCOM '98, the Conference on Computer Communications. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Gateway to the 21st Century (Cat. No.98*, volume 3, pages 1248–1256 vol.3, 1998.

[102] Rich Lane. Userspace eBPF VM. https://github.com/iovisor/ubpf, 2015. Retrieved Sep. 24, 2022.

[103] Abir Laraba, Jérôme François, Isabelle Chrisment, Shihabur Rahman Chowdhury, and Raouf Boutaba. Defeating Protocol Abuse with P4: Application to Explicit Congestion Notification. In *2020 IFIP Networking Conference (Networking)*, pages 431–439, 2020.

[104] Benjamin Lewis, Lyndon Fawcett, Matthew Broadbent, and Nicholas Race. Using P4 to Enable Scalable Intents in Software Defined Networks. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, pages 442–443, 2018.

[105] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 1–14, New York, NY, USA, 2016. Association for Computing Machinery.

[106] Athanasios Liatifis, Panagiotis Sarigiannidis, Vasileios Argyriou, and Thomas Lagkas. Advancing SDN from OpenFlow to P4: A Survey. *ACM Comput. Surv.*, 55(9), January 2023.

[107] Robert MacDavid, Carmelo Cascone, Pingping Lin, Badhrinath Padmanabhan, Ajay ThakuR, Larry Peterson, Jennifer Rexford, and Oguz Sunay. A P4-Based 5G User Plane Function. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*, SOSR '21, pages 162–168, New York, NY, USA, 2021. Association for Computing Machinery.

[108] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. ClickOS and the Art of Network Function Virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 459–473, USA, 2014. USENIX Association.

[109] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-Level Packet Capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX'93, page 2, USA, 1993. USENIX Association.

[110] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. In *SIGCOMM Comput. Commun. Rev.*, volume 38, pages 69–74, New York, NY, USA, March 2008. Association for Computing Machinery.

[111] Hesham Mekky, Fang Hao, Sarit Mukherjee, T. V. Lakshman, and Zhi-Li Zhang. Network function virtualization enablement within SDN data plane. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pages 1–9, May 2017.

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

[113] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Mauricio Vásquez Bernal, Yunsong Lu, and Jianwen Pi. Securing Linux with a Faster and Scalable Iptables. In *SIGCOMM Comput. Commun. Rev.*, volume 49, pages 2–17, New York, NY, USA, 2019. Association for Computing Machinery.

[114] Sebastiano Miano, Fulvio Risso, Mauricio Vásquez Bernal, Matteo Bertrone, and Yunsong Lu. A Framework for eBPF-Based Network Functions in an Era of Microservices. In *IEEE Transactions on Network and Service Management*, volume 18, pages 133–151, 2021.

[115] Oliver Michel, Roberto Bifulco, Gábor Rétvári, and Stefan Schmid. The Programmable Data Plane: Abstractions, Architectures, Algorithms, and Applications. In *ACM Comput. Surv.*, volume 54, New York, NY, USA, May 2021. Association for Computing Machinery.

[116] Rui Miguel, Salvatore Signorello, and Fernando M. V. Ramos. Named Data Networking with Programmable Switches. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, pages 400–405, 2018.

[117] Daniele Moro, Giacomo Verticale, and Antonio Capone. A Framework for Network Function Decomposition and Deployment. In *2020 16th International Conference on the Design of Reliable Communication Networks DRCN 2020*, pages 1–6, 2020.

[118] Jad Naous, Glen Gibb, Sara Bolouki, and Nick McKeown. NetFPGA: reusable router architecture for experimental research. In *In PRESTO '08: Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, pages 1–7. ACM, 2008.

[119] Netdev conference 0x15. XDP General Workshop. https://netdevconf.info/0x15/session.html?XDP-General-Workshop, 2021.

[120] Van-Giang Nguyen, Anna Brunstrom, Karl-Johan Grinnemo, and Javid Taheri. SDN/NFV-Based Mobile Packet Core Network Architectures: A Survey. In *IEEE Communications Surveys Tutorials*, volume 19, pages 1567–1602, 2017.

[121] Nikita V. Shirokov. XDP: 1.5 years in production. Evolution and lessons learned. http://vger.kernel.org/lpc_net2018_talks/LPC_XDP_Shirokov_paper_v1.pdf, 2018.

[122] NPLang.org. NPL – Network Programming Language. https://nplang.org/.

[123] ntop. PF_RING ZC (Zero Copy). https://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zc-zero-copy/. Accessed: 2022-04-05.

[124] Open Networking Foundation (ONF). SD-Fabric. https://opennetworking.org/sd-fabric/.

[125] Orange. Programming runtime extensions for Open vSwitch with P4. https://github.com/Orange-OpenSource/p4rt-ovs, 2020. Accessed: 2023-01-17.

[126] The Linux Kernel Organization. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org, 2020.

[127] Tomasz Osiński and Carmelo Cascone. Achieving End-to-End Network Visibility with Host-INT. In *Proceedings of the Symposium on Architectures for Networking and Communications Systems*, ANCS '21, pages 140–143, New York, NY, USA, 2021. Association for Computing Machinery.

[128] Tomasz Osiński, Mateusz Kossakowski, Mateusz Pawlik, Jan Palimąka, Michał Sala, and Halina Tarasiuk. Unleashing the Performance of Virtual BNG by Offloading Data Plane to a Programmable ASIC. In *Proceedings of the 3rd P4 Workshop in Europe*, EuroP4'20, pages 54–55, New York, NY, USA, 2020. Association for Computing Machinery.

[129] Tomasz Osiński and Jan Palimąka. Native In-Kernel P4-programmable Software Switch for Software-Defined Networking. https://github.com/NIKSS-vSwitch/nikss, 2023. Accessed: 2023-01-17.

[130] Tomasz Osiński, Jan Palimąka, Mateusz Kossakowski, Frédéric Dang Tran, El-Fadel Bonfoh, and Halina Tarasiuk. A Novel Programmable Software Datapath for Software-Defined Networking. In *Proceedings of the 18th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '22, pages 245–260, New York, NY, USA, 2022. Association for Computing Machinery.

[131] Tomasz Osiński, Mateusz Kossakowski, Halina Tarasiuk, and Roland Picard. Offloading data plane functions to the multi-tenant Cloud Infrastructure using P4. In *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 1–6, 2019.

[132] Tomasz Osiński, Halina Tarasiuk, Paul Chaignon, and Mateusz Kossakowski. P4rt-OVS: Programming Protocol-Independent, Runtime Extensions for Open vSwitch with P4. In *2020 IFIP Networking Conference (Networking)*, pages 413–421, 2020.

[133] Tomasz Osiński, Halina Tarasiuk, Lukasz Rajewski, and Emil Kowalczyk. DPPx: A P4-based Data Plane Programmability and Exposure framework to enhance NFV services. In *2019 IEEE Conference on Network Softwarization (NetSoft)*, pages 296–300, 2019.

[134] P4.org. New match kind: optional. https://github.com/p4lang/p4-spec/issues/794, 2019. Accessed: 2023-01-17.

[135] P4.org. Performance of BMv2. https://github.com/p4lang/behavioral-model/blob/main/docs/performance.md, 2019.

[136] P4.org. Behavioral Model version 2 (BMv2). https://github.com/p4lang/behavioral-model, 2021.

[137] P4.org. DPDK backend for the P4 compiler, 2021.

[138] P4.org. eBPF Backend for the P4 compiler. https://github.com/p4lang/p4c/tree/main/backends/ebpf, 2021.

[139] P4.org. p4c-ubpf: a new back-end for the P4 Compiler. https://github.com/p4lang/p4c/tree/main/backends/ubpf, 2022. Accessed: 2023-01-17.

[140] P4.org. PSA implementation for eBPF backend. https://github.com/p4lang/p4c/tree/main/backends/ebpf/psa, 2022. Accessed: 2023-01-17.

[141] P4.org. In-Band Network Telemetry, December 2019.

[142] P4.org. P4_16 language specification. https://github.com/p4lang/p4-spec/tree/master/p4-16/spec, December 2019.

[143] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the V out of NFV. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 203–216, Savannah, GA, November 2016. USENIX Association.

[144] F. Paolucci, F. Civerchia, A. Sgambelluri, A. Giorgetti, F. Cugini, and P. Castoldi. P4 edge node enabling stateful traffic engineering and cyber security. In *Journal of Optical Communications and Networking*, volume 11, pages A84–A95, 2019.

[145] Francesco Paolucci, Filippo Cugini, Piero Castoldi, and Tomasz Osiński. Enhancing 5G SDN/NFV Edge with P4 Data Plane Programmability. In *IEEE Network*, volume 35, pages 154–160, 2021.

[146] Federico Parola, Sebastiano Miano, and Fulvio Risso. A Proof-of-Concept 5G Mobile Gateway with eBPF. In *Proceedings of the ACM SIGCOMM 2020 Conference on Posters and Demos*, SIGCOMM '20. Association for Computing Machinery, 2020.

[147] Federico Parola, Roberto Procopio, and Fulvio Risso. Assessing the Performance of XDP and AF_XDP Based NFs in Edge Data Center Scenarios. In *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '21, pages 481–482, New York, NY, USA, 2021. Association for Computing Machinery.

[148] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. The Design and Implementation of Open vSwitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 117–130, Oakland, CA, May 2015. USENIX Association.

[149] Benoît Pit-Claudel, Yoann Desmouceaux, Pierre Pfister, Mark Townsley, and Thomas Clausen. Stateless Load-Aware Load Balancing in P4. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, pages 418–423, 2018.

[150] Marcelo Pizzutti and Alberto Egon Schaeffer-Filho. An Efficient Multipath Mechanism Based on the Flowlet Abstraction and P4. In *2018 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, 2018.

[151] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, Felipe Huici, and Giuseppe Siracusano. FlowBlaze:

Stateful Packet Processing in Hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 531–548, Boston, MA, February 2019. USENIX Association.

[152] Reza Rahimi, M. Veeraraghavan, Y. Nakajima, H. Takahashi, Y. Nakajima, S. Okamoto, and N. Yamanaka. A high-performance OpenFlow software switch. In *2016 IEEE 17th International Conference on High Performance Switching and Routing (HPSR)*, pages 93–99, 2016.

[153] Ruben Ricart-Sanchez, Pedro Malagon, Jose M. Alcaraz-Calero, and Qi Wang. NetFPGA-Based Firewall Solution for 5G Multi-Tenant Architectures. In *2019 IEEE International Conference on Edge Computing (EDGE)*, pages 132–136, 2019.

[154] Luigi Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 101–112, Boston, MA, June 2012. USENIX Association.

[155] Luigi Rizzo and Giuseppe Lettieri. VALE, a Switched Ethernet for Virtual Machines. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 61–72, New York, NY, USA, 2012. Association for Computing Machinery.

[156] Fabian Ruffy et al. P4C-XDP: Programming the Linux Kernel Forwarding Plane Using P4. In *Linux Plumbers Conference*, Vancouver, 2018.

[157] R.R. Schaller. Moore's law: past, present and future. In *IEEE Spectrum*, volume 34, pages 52–59, 1997.

[158] Robert Sedgewick and Kevin Wayne. *Algorithms, 4th Edition*. Addison-Wesley, 2011.

[159] Rinku Shah, Vikas Kumar, Mythili Vutukuru, and Purushottam Kulkarni. TurboEPC: Leveraging Dataplane Programmability to Accelerate the Mobile Packet Core. In *Proceedings of the Symposium on SDN Research*, SOSR '20, pages 83–95, New York, NY, USA, 2020. Association for Computing Machinery.

[160] Muhammad Shahbaz, Sean Choi, Ben Pfaff, Changhoon Kim, Nick Feamster, Nick McKeown, and Jennifer Rexford. PISCES: A Programmable, Protocol-Independent Software Switch. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 525–538, New York, NY, USA, 2016. ACM.

[161] Muhammad Shahbaz, Lalith Suresh, Jennifer Rexford, Nick Feamster, Ori Rottenstreich, and Mukesh Hira. Elmo: Source Routed Multicast for Public Clouds. In *IEEE/ACM Transactions on Networking*, volume 28, pages 2587–2600, 2020.

[162] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating Fair Queueing on Reconfigurable Switches. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 1–16, Renton, WA, April 2018. USENIX Association.

[163] Nick Shelly, Ethan J. Jackson, Teemu Koponen, Nick McKeown, and Jarno Rajahalme. Flow Caching for High Entropy Packet Fields. In *SIGCOMM Comput. Commun. Rev.*, volume 44, New York, NY, USA, August 2014. Association for Computing Machinery.

[164] Sumeet Singh, Florin Baboescu, George Varghese, and Jia Wang. Packet Classification Using Multidimensional Cutting. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '03, pages 213–224, New York, NY, USA, 2003. Association for Computing Machinery.

[165] Suneet Kumar Singh, Christian Esteve Rothenberg, Gyanesh Patra, and Gergely Pongracz. Offloading Virtual Evolved Packet Gateway User Plane Functions to a Programmable ASIC. In *Proceedings of the 1st ACM CoNEXT Workshop on Emerging In-Network Computing Paradigms*, ENCP '19, pages 9–14, New York, NY, USA, 2019. Association for Computing Machinery.

[166] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. Heavy-Hitter Detection Entirely in the Data Plane. In *Proceedings of the Symposium on SDN Research*, SOSR '17, pages 164–176, New York, NY, USA, 2017. Association for Computing Machinery.

[167] 6Wind Virtualized Networking Software. Virtual Service Router (VSR). https://www.6wind.com/products/solutions/data-center-networking, 2022.

[168] John Sonchack, Adam J. Aviv, Eric Keller, and Jonathan M. Smith. Turboflow: Information Rich Flow Record Generation on Commodity Switches. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[169] V. Srinivasan, S. Suri, and G. Varghese. Packet classification using Tuple Space Search. In *Proc. ACM SIGCOMM*, 1999.

[170] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast and scalable layer four switching. In *Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '98, pages 191–202, New York, NY, USA, 1998. Association for Computing Machinery.

[171] Nik Sultana, John Sonchack, Hans Giesen, Isaac Pedisich, Zhaoyang Han, Nishanth Shyamkumar, Shivani Burad, André DeHon, and Boon Thau Loo. Flightplan: Dataplane Disaggregation and Placement for P4 Programs. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 571–592. USENIX Association, April 2021.

[172] Weibin Sun and Robert Ricci. Fast and flexible: Parallel packet processing with GPUs and click. In *Architectures for Networking and Communications Systems*, pages 25–35, 2013.

[173] Lu Tang, Qun Huang, and Patrick P. C. Lee. SpreadSketch: Toward Invertible and Network-Wide Detection of Superspreaders. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*, pages 1608–1617, 2020.

[174] The P4.org API Working Group. P4Runtime Specification, version 1.3.0. https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html, July 2021.

[175] The P4.org Architecture Working Group. P4_16 Portable Switch Architecture (PSA). https://p4lang.github.io/p4-spec/docs/PSA.pdf, April 2021.

[176] Cheng-Chun Tu, Joe Stringer, and Justin Pettit. Building an Extensible Open VSwitch Datapath. In *SIGOPS Oper. Syst. Rev.*, volume 51, pages 72–77, New York, NY, USA, 2017. Association for Computing Machinery.

[177] William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. Revisiting the Open VSwitch Dataplane Ten Years Later. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, pages 245–257, New York, NY, USA, 2021. Association for Computing Machinery.

[178] Balajee Vamanan, Gwendolyn Voskuilen, and T. N. Vijaykumar. Efficuts: Optimizing packet classification for memory and throughput. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 207–218, New York, NY, USA, 2010. Association for Computing Machinery.

[179] G. Varghese. *Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices*. Morgan Kaufmann series in networking. Elsevier/Morgan Kaufmann, 2005.

[180] Giorgos Vasiliadis, Lazaros Koromilas, Michalis Polychronakis, and Sotiris Ioannidis. Design and Implementation of a Stateful Network Packet Processing Framework for GPUs. In *IEEE/ACM Transactions on Networking*, volume 25, pages 610–623, 2017.

[181] Marcos A. M. Vieira, Matheus S. Castanho, Racyus D. G. Pacífico, Elerson R. S. Santos, Eduardo P. M. Câmara Júnior, and Luiz F. M. Vieira. Fast Packet Processing with EBPF and XDP: Concepts, Code, Challenges, and Applications. In *ACM Comput. Surv.*, volume 53, New York, NY, USA, February 2020. Association for Computing Machinery.

[182] VMware. VMware NSX Data Center. https://www.vmware.com/products/nsx.html, 2020.

[183] Péter Vörös, Dániel Horpácsi, Róbert Kitlei, Dániel Leskó, Máté Tejfel, and Sándor Laki. T4P4S: A Target-independent Compiler for Protocol-independent Packet Processors. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–8, 2018.

[184] Yanshu Wang, Dan Li, Yuanwei Lu, Jianping Wu, Hua Shao, and Yutian Wang. Elixir: A High-performance and Low-cost Approach to Managing Hardware/-Software Hybrid Flow Tables Considering Flow Burstiness. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 535–550, Renton, WA, April 2022. USENIX Association.

[185] Wenfeng Xia, Yonggang Wen, Chuan Heng Foh, Dusit Niyato, and Haiyong Xie. A Survey on Software-Defined Networking. In *IEEE Communications Surveys Tutorials*, volume 17, pages 27–51, 2015.

[186] Peng Xiao, Wenyu Qu, Heng Qi, Yujie Xu, and Zhiyang Li. An efficient elephant flow detection with cost-sensitive in SDN. In *2015 1st International Conference on Industrial Networks and Intelligent Systems (INISCom)*, pages 24–28, 2015.

[187] Zhaoqi Xiong and Noa Zilberman. Do Switches Dream of Machine Learning? Toward In-Network Classification. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, HotNets '19, pages 25–33, New York, NY, USA, 2019. Association for Computing Machinery.

[188] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic Sketch: Adaptive and Fast Network-Wide Measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 561–575, New York, NY, USA, 2018. Association for Computing Machinery.

[189] Jin-Li Ye, Chien Chen, and Yu Huang Chu. A Weighted ECMP Load Balancing Scheme for Data Centers Using P4 Switches. In *2018 IEEE 7th International Conference on Cloud Networking (CloudNet)*, pages 1–4, 2018.

[190] Sorrachai Yingchareonthawornchai, James Daly, Alex X. Liu, and Eric Torng. A Sorted-Partitioning Approach to Fast and Scalable Dynamic Packet Classification. In *IEEE/ACM Transactions on Networking*, volume 26, pages 1907–1920, 2018.

[191] Yonghong Song. bpf: add bpf_for_each_map_elem() helper. https://lwn.net/Articles/846504/, 2021.

[192] Faqir Zarrar Yousaf, Michael Bredel, Sibylle Schaller, and Fabian Schneider. NFV and SDN - Key Technology Enablers for 5G Networks. In *IEEE Journal on Selected Areas in Communications*, volume 35, pages 2468–2478, 2017.

[193] Minlan Yu, Jennifer Rexford, Michael J. Freedman, and Jia Wang. Scalable Flow-Based Networking with DIFANE. In *SIGCOMM Comput. Commun. Rev.*, volume 40, pages 351–362, New York, NY, USA, August 2010. Association for Computing Machinery.

[194] Tianzhu Zhang, Leonardo Linguaglossa, Massimo Gallo, Paolo Giaccone, Luigi Iannone, and James Roberts. Comparing the Performance of State-of-the-Art Software Switches for NFV. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, CoNEXT '19, pages 68–81, New York, NY, USA, 2019. Association for Computing Machinery.

[195] Dong Zhou, Bin Fan, Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, pages 97–108, New York, NY, USA, 2013. Association for Computing Machinery.

List of Tables

3.1 Comparison of the most relevant state-of-the-art programmable software switches 63

5.1 Test programs used for evaluation of NIKSS. 105

5.2 NIKSS: The in-depth performance analysis of test programs. 107

List of Figures

1.1 The role of software switches in the end-to-end 5G/6G mobile network architecture. 13

2.1 Functional blocks of packet processing pipelines. 20
2.2 The end-to-end packet path between two NICs within a single Linux server. 22
2.3 Packet processing frameworks. 27
2.4 Typical performance evaluation framework for software switches with measurement points marked by red circles. 34
2.5 The P4 workflow. 38
2.6 P4 information flow in the PISA model. 39
2.7 Portable Switch Architecture (PSA) for the P4 language 44
2.8 The eBPF subsystem. 46

3.1 Vision of end-to-end programmable network. 53

4.1 The concept of offloading packet processing tasks from the Virtual Network Function (VNF) to a virtual switch. 65
4.2 Open vSwitch's two-level caching architecture. 67
4.3 The overall architecture of P4rt-OVS 68
4.4 The BPF subsystem and forwarding model of OVS 69
4.5 The forwarding model of the BPF program generated from the P4 language 72
4.6 The test topology for P4rt-OVS 74
4.7 L2 Forwarding performance in Mpps for input traffic of 40 Gbps with and without the microflow. 75

4.8 Comparison of OVS, P4rt-OVS, and PISCES's throughputs with three network functions: DNAT, NAPT, and MPLS LER. 76
4.9 The performance of parser and deparser as more protocols are handled. 77
4.10 The impact of parser and deparser on the aggregated performance in millions of packets per second as more protocols are handled. 78
4.11 Performance in CPU cycles of the control block as more Match-Action tables are used to process a packet. 79
4.12 The impact of number of Match-Action tables in the control block on the aggregated performance in millions of packets per second. 79
4.13 Performance of table operations as more table entries are added. 80
4.14 The performance of data plane programs written in P4 and C. 80
4.15 The packet processing pipeline of the BNG application 83
4.16 The performance of the BNG pipeline implementation for P4rt-OVS . . . 85

5.1 NIKSS workflow. 89
5.2 NIKSS: General-purpose TC-based design. 90
5.3 NIKSS: Specialized, XDP-based design. 93
5.4 NIKSS: P4 pipeline to eBPF translation. 94
5.5 NIKSS: The memory organization of Packet Replication Engine implementation. 97
5.6 Packet forwarding rate of NIKSS-TC and NIKSS-XDP with only pipeline-aware optimization enabled for different packet sizes (in bytes). 106
5.7 NIKSS: The cost of different P4 match kinds measured in the throughput rate with a 95% CI of less than 0.015 MPPS and average CPU cycles per packet over a baseline with 95% CI of less than 3 cycles, depending on the number of table entries. 109
5.8 NIKSS: The cost of PSA externs measured in average CPU cycles per packet over a baseline program with 95% CI of less than 2 cycles for all data points. 110
5.9 The throughput of test programs and latency distribution by percentiles for L2L3-ACL and 0.8 MPPS of the offered load for NIKSS and P4-DPDK. 111

5.10 The throughput of test programs and latency distribution by percentiles for L2L3-ACL and 0.8 MPPS of the offered load for kernel datapaths (NIKSS, OVS, eBPF/TC, eBPF/XDP). 112

B.1 P4rt-OVS DDoS mitigator latency and throughput: main (M) and Docker (D) instances. 129

Listings

2.1 Sample declaration of packet headers supported by a P4 program. 40

2.2 Sample Parser implementation extracting Ethernet, MPLS, IPv4, UDP and TCP headers. 40

2.3 Sample Control block implementing basic L2/L3 processing. 41

2.4 Sample Deparser implementation emitting a combination of Ethernet, MPLS, VLAN, IPv4, TCP and UDP headers. 42

2.5 Snippet of a BPF program implementing simple packet forwarding based on destination IPv4 address. 48

2.6 BPF bytecode for the C program in Listing 2.5. 49

4.1 Pseudocode of the P4 program implementing PPP Egress Processing for P4rt-OVS. Pseudocode does not present the definition of protocols' headers. 83

B.1 P4 code excerpt of DDoS port scan mitigator. 127

List of Acronyms

ACL Access Control List.

API Application Programming Interface.

ASIC Application-Specific Integrated Circuit.

BGP Border Gateway Protocol.

BNG Broadband Network Gateway.

CPU Central Processing Unit.

DDoS Distributed Denial of Service.

DPDK Data Plane Development Kit.

FPGA Field-Programmable Gate Array.

IP Internet Protocol.

IPSec Internet Protocol Security.

LPM Longest-Prefix Match.

MPLS Multi Protocol Label Switching.

NAT Network Address Translation.

NFV Network Functions Virtualization.

NIC Network Interface Card.

NIKSS Native In-Kernel SDN Software Switch.

NPU Network Processing Unit.

OS Operating System.

OSPF Open Shortest Path First.

OVS Open vSwitch.

QoS Quality of Service.

SDN Software-Defined Networking.

TCP Transport Control Protocol.

UDP User Datagram Protocol.

UPF User Plane Function.

VM Virtual Machine.

VNF Virtual Network Function.

VXLAN Virtual Extensible Local Area Network.

XDP eXpress Data Path.